O'REILLY®

Head First

Изучаем

Go

Научитесь создавать простой и понятный код

Избегайте досадных ошибок с типами

Решайте упражнения!

Руководство для начинающих программистов на Go

Сделайте работу более результа-тивной

Запускайте конкурентные функции с помощью горутин

Джей Макгаврен

Head First
Go

Jay McGavren

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Head First
Go

Джей Макгаврен

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2020

ББК 32.973.2-018.1
УДК 004.43
М15

Макгаврен Джей

М15 Head First. Изучаем Go. — СПб.: Питер, 2020. — 544 с.: ил. — (Серия «Head First O'Reilly»).
ISBN 978-5-4461-1395-8

Go упрощает построение простых, надежных и эффективных программ. А эта книга сделает его доступным для обычных программистов. Основная задача Go — эффективная работа с сетевыми коммуникациями и многопроцессорной обработкой, но код на этом языке пишется и читается не сложнее чем на Python и JavaScript. Простые примеры позволят познакомиться с языком в действии и сразу приступить к программированию на Go. Так что вы быстро освоите общепринятые правила и приемы, которые позволят вам называть себя гофером.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1491969557 англ.

ISBN 978-5-4461-1395-8

Содержание (сводка)

Содержание (настоящее)

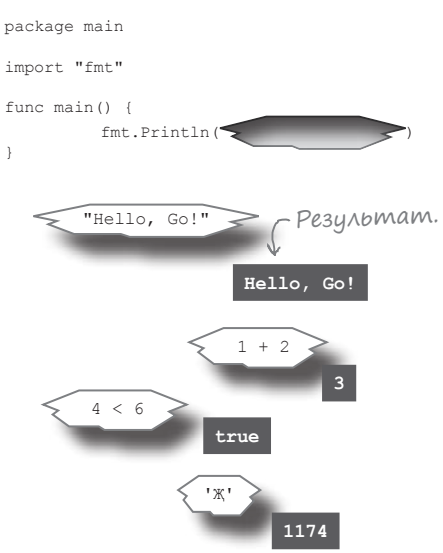
Введение

Ваш мозг и Go. Вы сидите за книгой и пытаетесь что-нибудь выучить, но ваш мозг считает, что вся эта писанина не нужна. Ваш мозг говорит: «Выгляни в окно! На свете есть более важные вещи, например сноуборд». Как заставить мозг изучить программирование на Go?

1 Знакомство с Go

Основы синтаксиса

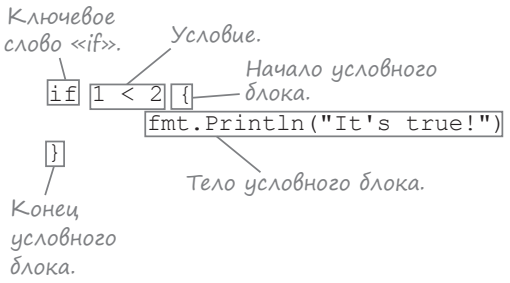
Готовы поднять свой код на новый уровень? Нужен **простой** язык программирования, который **быстро компилируется** и **быстро выполняется**? Язык, с которым вы сможете **легко и удобно распространять** свое ПО среди пользователей? Тогда знакомьтесь: **Go** — язык программирования, ориентированный на **простоту** и **скорость**. Он проще других языков, и поэтому вы быстрее освоите его. Кроме того, Go эффективно использует мощь современных многоядерных процессоров, а значит, программы будут выполняться быстрее. В этой главе представлены возможности Go, которые упростят **вам работу** и наверняка придутся по вкусу **пользователям**.



2 Какой код будет выполняться?

Условные команды и циклы

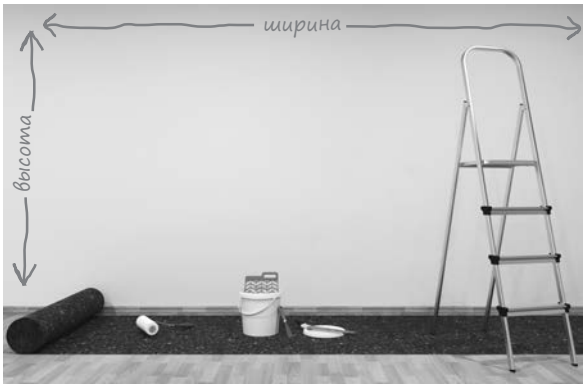
В каждой программе есть части, которые должны выполняться только в определенных ситуациях. «Этот код должен выполняться, *если* произошла ошибка. А если нет — должен выполняться этот код». Почти в каждой программе присутствует код, который должен выполняться только в том случае, если некоторое *условие* истинно. Почти в каждом языке программирования существуют **условные команды,** которые позволяют определить, нужно ли выполнять те или иные сегменты кода. Язык Go не исключение. Возможно, какие-то части кода должны выполняться *многократно*. Как и многие языки, Go поддерживает **циклы** для многократного выполнения блоков кода. В этой главе вы научитесь применять как условные команды, так и циклы!



3 Вызовы функций

Функции

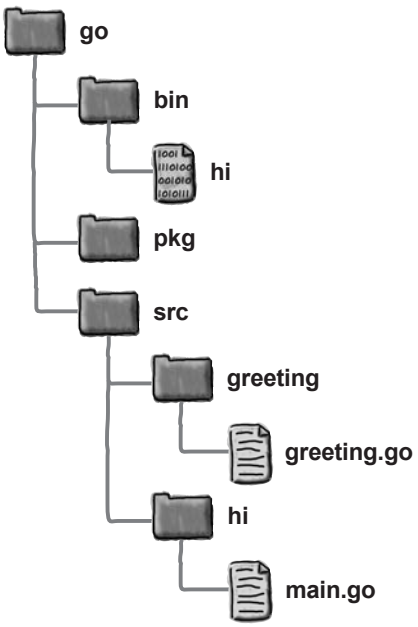
И все же чего-то не хватало. Вы вызывали функции как настоящий профи. Но могли вызывать только те функции, которые были определены для вас в Go. Настала ваша очередь. В этой главе мы покажем, как создавать собственные функции. Вы научитесь объявлять функции с параметрами и без. Сначала вы узнаете, как объявлять функции, которые возвращают одно значение, а потом мы перейдем к возвращению нескольких значений, чтобы функция могла сигнализировать об ошибке. А еще в этой главе рассматриваются **указатели**, которые повышают эффективность вызовов функций по затратам памяти.



4 Упаковка кода

Пакеты

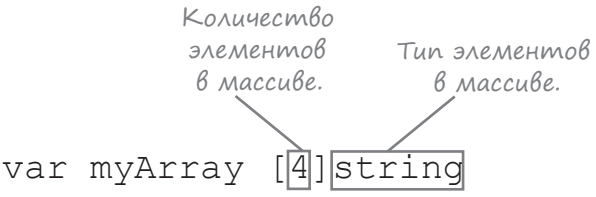
Время навести порядок! До сих пор мы сваливали весь свой код в один файл. Но чем больше и сложнее будут становиться наши программы, тем скорее такой подход приведет к хаосу. В этой главе мы научим вас создавать **пакеты** для хранения взаимосвязанного кода в одном месте. Но пакеты нужны не только для организации кода. Пакеты предоставляют простые средства для *повторного использования кода в разных программах*. А еще это простой способ *распространения кода среди разработчиков*.

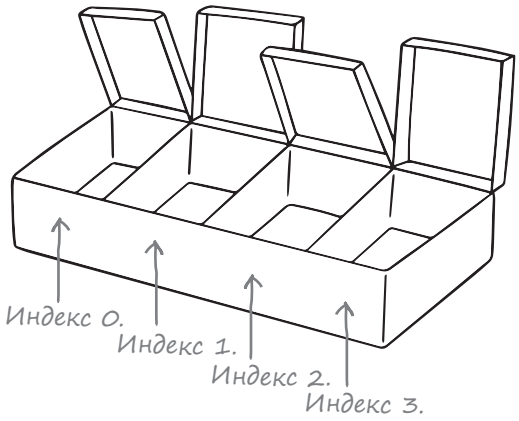


И далее по списку

5 Массивы

Многие программы работают со списками. Списки адресов. Списки телефонных номеров. Списки товаров. В Go существуют *два* встроенных способа хранения списков. В этой главе мы разберем первый способ: **массивы**. Вы научитесь создавать массивы, заполнять их данными и извлекать сохраненные данные. Далее рассмотрим способы обработки всех элементов в массиве: сначала *сложный* вариант с циклами `for`, а затем *простой* — с циклами `for...range`.





6 Сегменты

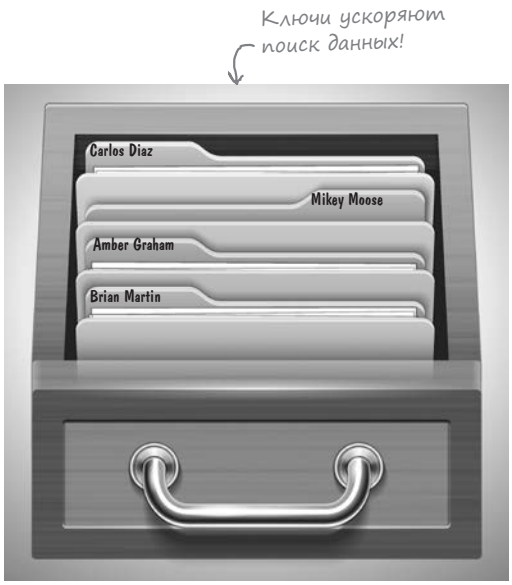
Проблема с присоединением

Вы уже знаете, что в массив нельзя добавить новые элементы. В нашей программе это создает настоящие проблемы, потому что количество значений данных в файле неизвестно заранее. На помощь приходят **сегменты** Go. Сегменты — разновидность коллекций, которые могут расширяться для хранения дополнительных элементов; а это как раз то, что нужно! Вы также увидите, как использовать сегменты для простой передачи данных *любым* программам и как с их помощью пишутся функции, которые удобно вызывать.

7 Значения и метки

Карты

Сваливать все в одну кучу удобно до тех пор, пока не потребуется что-нибудь найти. Вы уже видели, как создавать списки значений в *массивах* и *сегментах*. Вы знаете, как применить одну операцию *к каждому значению* в массиве или сегменте. Но что, если требуется поработать с *конкретным* значением? Чтобы найти его, придется начать с начала массива или сегмента и *просмотреть. Каждое. Существующее. Значение.* А если бы существовала разновидность коллекций, в которой каждое значение снабжается специальной меткой? Тогда нужное значение можно было бы быстро найти по метке! Эта глава посвящена **картам**, которые предназначены именно для этого.

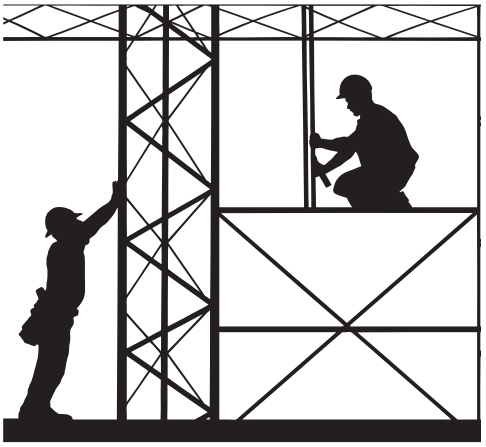


Карта

Совместное хранение

8 Структуры

Иногда требуется хранить вместе несколько типов данных. Сначала вы познакомились с сегментами, предназначенными для хранения списков. Затем были рассмотрены карты, связывающие список ключей со списком значений. Но обе структуры данных позволяют хранить значения только *одного* типа, а в некоторых ситуациях требуется сгруппировать значения *нескольких* типов. Например, в почтовых адресах названия улиц (строки) группируются с почтовыми индексами (целые числа). Или в информации о студентах имена (строки) объединяются со средними оценками (вещественные числа). Сегменты и карты не позволяют смешивать разные типы. Тем не менее это *возможно* при использовании другого типа данных, называемого **структурой**. В этой главе мы подробно изучим структуры.



9 Ты — мой тип!

Определяемые типы

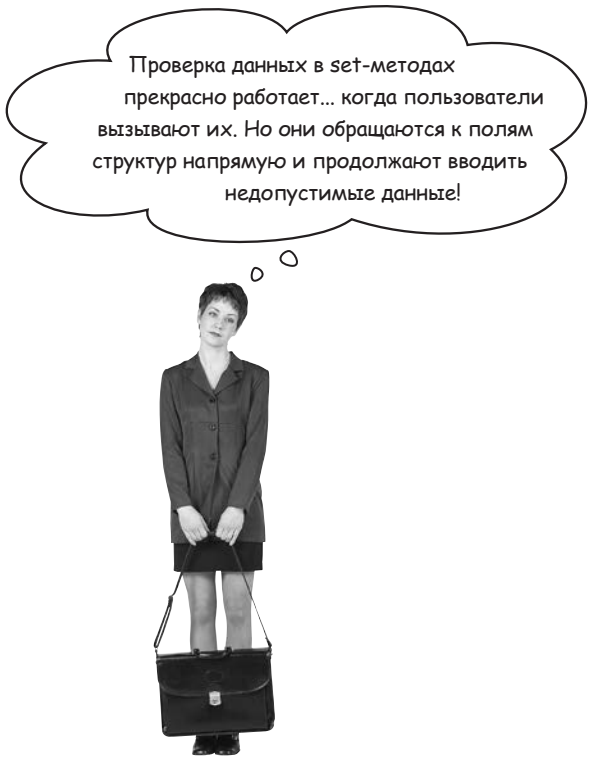
Мы еще не все рассказали об определяемых типах. В предыдущей главе вы узнали, как определяются типы, у которых базовым типом является тип структуры. Но при этом мы *не сказали*, что в качестве базового может использоваться *любой* тип.

Помните о методах — особой разновидности функций, связываемой со значениями определенного типа? В книге неоднократно встречались примеры вызова методов для разных значений, но *собственные* методы вы еще не умеете определять. В этой главе мы исправим это. А теперь за дело!

10 Все при себе

Инкапсуляция и встраивание

Ошибки случаются. Иногда программа получает недействительные данные от пользователя, из файла или другого источника. В этой главе рассматривается **инкапсуляция**: механизм защиты полей структурного типа от недействительных данных. И будьте уверены — теперь с данными полей можно безопасно работать! Мы также покажем, как **встраивать** другие типы в структуры. Если вашему типу структуры потребуются методы, которые уже существуют у другого типа, вам не придется копировать и вставлять код метода. Вы можете встроить другой тип в свой тип структуры, а затем воспользоваться методами встроенного типа так, если бы они были определены для вашего собственного типа!



11 Что можно сделать?

Интерфейсы

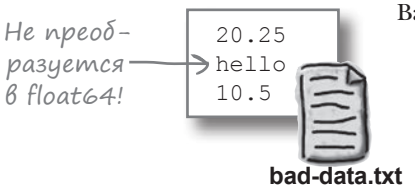
Иногда конкретный тип значения не важен. Вас не интересует, с чем вы работаете. Вы просто хотите быть уверены в том, что оно может делать то, что нужно вам. Тогда вы сможете вызывать для значения *определенные методы*. Неважно, с каким значением вы работаете — `Pen` или `Pencil`; вам просто нужно нечто, содержащее метод `Draw`. Именно эту задачу решают **интерфейсы** в языке Go. Они позволяют определять переменные и параметры функций, которые могут хранить *любой* тип при условии, что этот тип определяет некоторые методы.

12

Снова на ногах

Восстановление после сбоев

Любая программа сталкивается с ошибками. Учтите их при планировании. Иногда обработка ошибки сводится к простому выводу сообщения и завершению программы. Другие ошибки требуют дополнительных действий: например, закрытия открытых файлов или сетевых подключений или иного освобождения ресурсов, чтобы после вашей программы оставался порядок. В этой главе мы покажем, как **отложить** завершающие действия, чтобы они выполнились даже в случае ошибки. Также вы узнаете, как поднять **панику** в тех (редких) ситуациях, когда это уместно, и как **восстановиться** после нее.



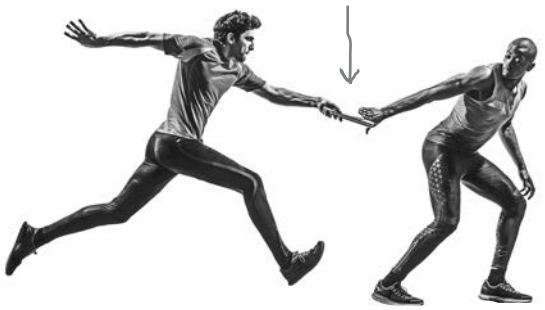
13

Совместное выполнение

Горутины и каналы

Одновременная работа над одной задачей не всегда быстрее всего приводит к цели. Некоторые большие задачи можно разбить на мелкие. **Горутины (goroutines)** позволяют программам работать над несколькими задачами одновременно. Они координируют свою работу при помощи **каналов**, по которым могут отправлять данные друг другу *и* синхронизировать выполнение, чтобы одна горутина не опережала другую. Горутины позволяют использовать всю мощь многопроцессорных компьютеров, чтобы программы выполнялись как можно быстрее!

Получающая горутина ожидает, когда другая горутина отправит значение.



14 Контроль качества кода

Автоматизация тестирования

А вы уверены, что ваша программа работает правильно? Точно уверены? Прежде чем рассылать новую версию пользователям, вы, скорее всего, опробовали ее новые возможности и убедились, что все работает. Но опробовали ли вы *старые* возможности? А вдруг что-нибудь сломалось в процессе доработки? *Все* старые возможности? Если от этого вопроса вам стало слегка не по себе, значит, вашим программам необходимо **автоматизированное тестирование**. Автоматизированные тесты гарантируют, что компоненты программы работают правильно даже после того, как вы внесете изменения в код. Средства тестирования Go и команда `go test` упрощают написание автотестов.

Проходит.

☑ For `[]slice{"apple", "orange", "pear"}`, `JoinWithCommas` should return `"apple, orange, and pear"`.

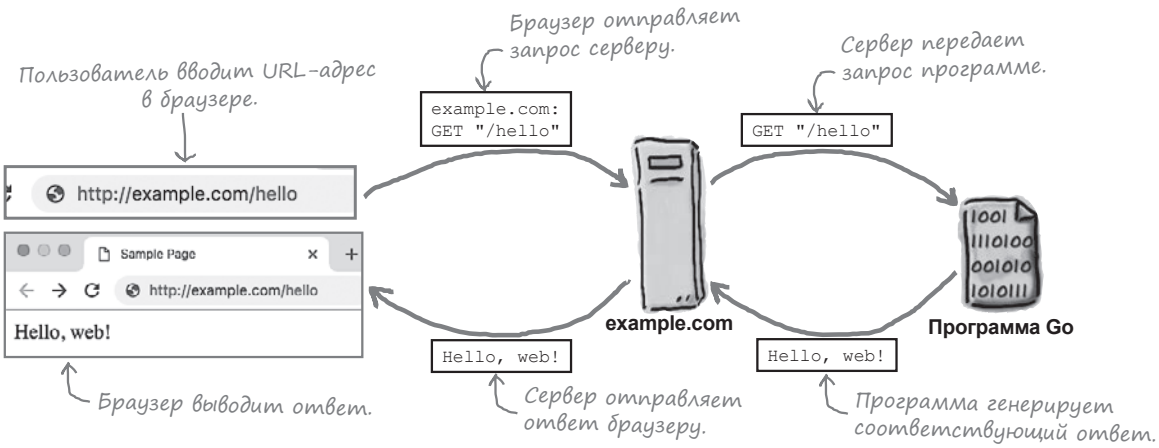
Ошибка!

☒ For `[]slice{"apple", "orange"}`, `JoinWithCommas` should return `"apple and orange"`.

15 Запросы и ответы

Веб-приложения

Мы живем в XXI веке. Пользователям нужны веб-приложения. Go поможет и в этом! Стандартная библиотека Go включает пакеты, при помощи которых можно размещать собственные веб-приложения и делать их доступными для любого веб-браузера. Последние две главы этой книги будут посвящены построению веб-приложений. Первое, что необходимо веб-приложению — способность отвечать на запросы, отправляемые браузером. В этой главе вы узнаете, как реализовать эту функцию с использованием пакета `net/http`.



16 Пример для подражания

Шаблон HTML

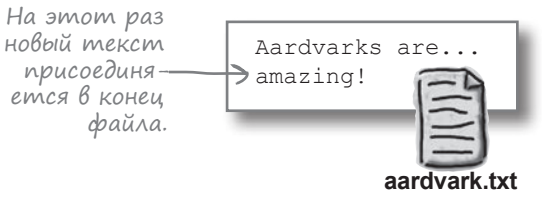
Веб-приложение должно выдавать ответ с разметкой HTML, а не с простым текстом. Для сообщений электронной почты и постов в соцсетях достаточно простого текста. Тем не менее ваши страницы должны быть отформатированы с выделением заголовков и разбиением на абзацы. На них должны быть формы, в которых пользователь сможет ввести данные для приложения. Для решения таких задач вам понадобится разметка HTML.

Рано или поздно в разметку HTML потребуется вставлять данные. Для этого в Go существует пакет `html/template` — мощный инструмент для вставки данных в HTML-ответы приложения. Шаблоны играют ключевую роль при построении более масштабных и качественных веб-приложений, и в последней главе книги мы научим вас ими пользоваться!

Функция os.OpenFile

Приложение A. Открытие файлов

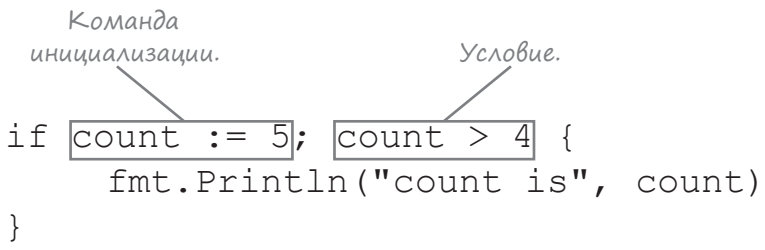
Некоторые программы не только читают данные, но и записывают их в файлы. Когда в этой книге мы собирались поработать с файлами, приходилось создавать их в текстовом редакторе, чтобы программа могла прочитать данные. Но некоторые программы *генерируют* данные, и когда это происходит, программа должна иметь возможность *записать* данные в файл. Функция `os.OpenFile` уже использовалась для открытия файла для записи. Но у нас не было возможности объяснить, как она работает. В этом приложении вы узнаете все, что необходимо знать для эффективного использования `os.OpenFile`!

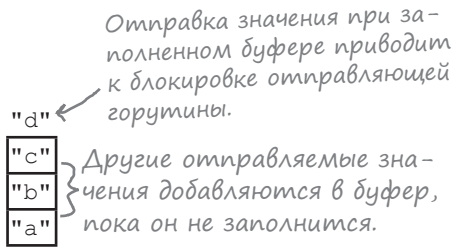


Еще шесть тем

Приложение Б. Напоследок

Позади большой путь, и книга почти подошла к концу. Мы будем скучать, но, пожалуй, было бы неправильно отпускать вас в свободное плавание без *небольшой* дополнительной подготовки. Мы приберегли шесть важных тем для этого приложения.





Посвящается моей неизменно терпеливой Кристин

Автор Head First Go

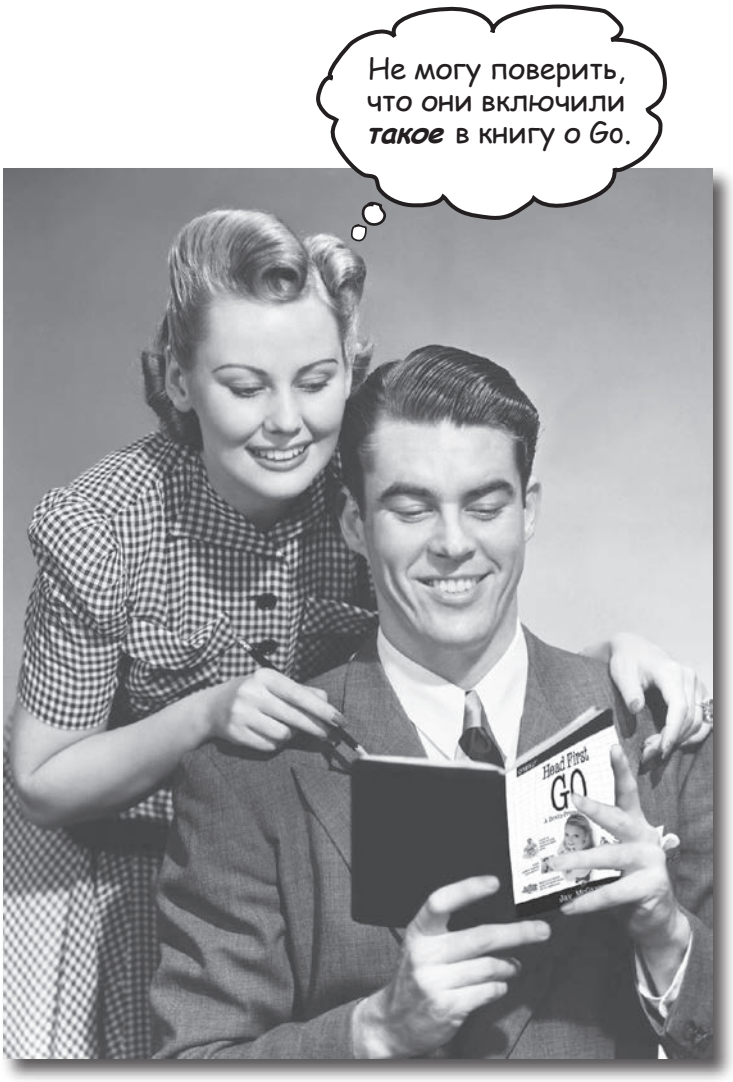
Джей Макгаврен

Джей Макгаврен – автор книг *Head First Ruby* и *Head First Go*, опубликованных издательством O'Reilly. Преподает программирование в компании Treehouse.

Вместе с ним в пригороде Финикса проживают его очаровательная жена и непостоянное количество детей и собак.

Персональный веб-сайт Джея находится по адресу *http://jay.mcgavren.com*.

Введение



В этом разделе мы ответим на важный вопрос: «Так почему они включили ТАКОЕ в книгу о Go?»

Для кого написана эта книга?

Если вы ответите «да» на все следующие вопросы:

1. У вас есть доступ к компьютеру с текстовым редактором?
2. Вы хотите изучить язык программирования, который делает разработку **быстрой** и **производительной**?
3. Вы предпочитаете застольные беседы сухим, скучным академическим лекциям?

тогда эта книга для вас.

Кому эта книга не подойдет?

Если вы ответите «да» хотя бы *на один* из следующих вопросов:

1. **У вас нет никакого опыта работы с компьютером?**

   (Быть экспертом не обязательно, но нужно понимать, что такое файлы и папки, как открыть приложение-терминал и как пользоваться простым текстовым редактором.)

2. Вы — опытный разработчик, которому нужен ***справочник***?
3. Вы **боитесь попробовать что-нибудь новое**? Скорее пойдете к зубному врачу, чем наденете полосатое с клетчатым? Вы считаете, что техническая книга, в которой полно неудачных каламбуров, хорошей быть не может?

...эта книга *не* для вас.

[Замечание от отдела продаж: вообще-то эта книга для любого, у кого есть деньги.]

Мы знаем, о чем вы думаете

«Разве серьезные книги по программированию на Go *такие?*»

«И почему здесь столько рисунков?»

«Можно ли так чему-нибудь *научиться?*»

И мы знаем, о чем думает ваш *мозг*

Мозг жаждет новых впечатлений. Он постоянно ищет, анализирует, *ожидает* чего-то необычного. Он так устроен, и это помогает нам выжить.

Как наш мозг поступает со всеми обычными, повседневными вещами? Он *всеми силами* пытается оградиться от них, чтобы они не мешали его *настоящей* работе — сохранению того, что *действительно важно*. Мозг не считает нужным сохранять скучную информацию. Она не проходит фильтр, отсекающий «очевидно несущественное».

Но как же мозг *узнает*, что важно? Представьте, что вы выехали на прогулку и вдруг прямо перед вами появляется тигр. Что происходит в вашей голове и теле?

Активизируются нейроны. Вспыхивают эмоции. *Происходят химические реакции*. И тогда ваш мозг понимает...

Конечно, это важно! Не забывать!

А теперь представьте, что вы находитесь дома или в библиотеке — в теплом, уютном месте, где тигры не водятся. Вы учитесь — готовитесь к экзамену. Или пытаетесь освоить сложную техническую тему, на которую вам выделили неделю... максимум десять дней.

И тут возникает проблема: ваш мозг пытается оказать вам услугу. Он старается сделать так, чтобы на эту *очевидно* несущественную информацию не тратились драгоценные ресурсы. Их лучше потратить на что-нибудь важное. На тигров, например. Или на то, что к огню лучше не прикасаться. Или что те фотки с вечеринки не стоило публиковать на странице Facebook. Нет простого способа сказать своему мозгу: «Послушай, мозг, я тебе, конечно, благодарен, но какой бы скучной ни была эта книга и пусть мой датчик эмоций сейчас на нуле, я *хочу* запомнить то, что здесь написано».

Эта книга для тех, кто хочет учиться.

Как мы что-то узнаем? Сначала нужно это «что-то» *понять*, а потом *не забыть*. Затолкать в голову побольше фактов недостаточно. Согласно новейшим исследованиям в области когнитивистики, нейробиологии и психологии обучения, для *усвоения материала* требуется что-то большее, чем простой текст на странице. Мы знаем, как заставить ваш мозг работать.

Основные принципы серии «Head First»

Наглядность. Графика запоминается лучше, чем обычный текст, и значительно повышает эффективность восприятия информации (до 89 % по данным исследований). Кроме того, материал становится более понятным. **Текст размещается на рисунках**, к которым он относится, а не под ними или на соседней странице — и вероятность успешного решения задач, относящихся к материалу, повышается *вдвое*.

Разговорный стиль изложения. Недавние исследования показали, что при разговорном стиле изложения материала (вместо формальных лекций) улучшение результатов на итоговом тестировании достигает 40 %. Рассказывайте историю, вместо того чтобы читать лекцию. Не относитесь к себе слишком серьезно. Что привлечет ваше внимание: занимательная беседа за столом или лекция?

Активное участие читателя. Пока вы не начнете напрягать извилины, в вашей голове ничего не произойдет. Читатель должен быть заинтересован в результате; он должен решать задачи, формулировать выводы и овладевать новыми знаниями. А для этого необходимы упражнения и каверзные вопросы, в решении которых задействованы оба полушария мозга и разные чувства.

Привлечение (и сохранение) внимания читателя. Ситуация, знакомая каждому: «Я очень хочу изучить это, но засыпаю на первой странице». Мозг обращает внимание на интересное, странное, притягательное, неожиданное. Изучение сложной технической темы не обязано быть скучным. Интересное узнается намного быстрее.

Обращение к эмоциям. Известно, что наша способность запоминать в значительной мере зависит от эмоционального сопереживания. Мы запоминаем то, что нам небезразлично. Мы запоминаем, когда что-то *чувствуем*. Нет, сантименты здесь ни при чем: речь идет о таких эмоциях, как удивление, любопытство, интерес и чувство «Да я крут!» при решении задачи, которую окружающие считают сложной — или когда вы понимаете, что разбираетесь в теме *лучше*, чем всезнайка Боб из технического отдела.

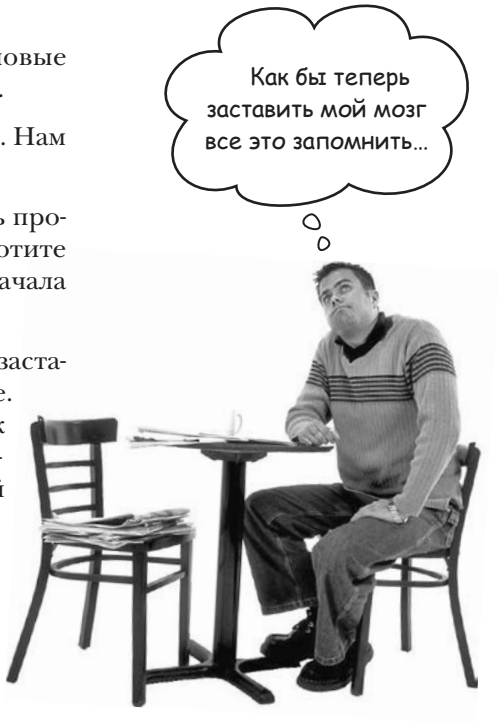
Метапознание: наука о мышлении

Если вы действительно хотите быстрее и глубже усваивать новые знания — задумайтесь над тем, как вы думаете. Учитесь учиться.

Мало кто из нас изучает теорию метапознания во время учебы. Нам *положено* учиться, но нас редко этому *учат*.

Но раз вы читаете эту книгу, то, вероятно, вы хотите освоить программирование для Android, и по возможности быстрее. Вы хотите *запомнить* прочитанное, а для этого абсолютно необходимо сначала *понять* прочитанное.

Чтобы извлечь максимум пользы из учебного процесса, нужно заставить ваш мозг воспринимать новый материал как Нечто Важное. Критичное для вашего существования. Такое же важное, как тигр. Иначе вам предстоит бесконечная борьба с вашим мозгом, который всеми силами уклоняется от запоминания новой информации.



Как же УБЕДИТЬ мозг, что программирование не менее важно, чем голодный тигр?

Есть способ медленный и скучный, а есть быстрый и эффективный. Первый основан на тупом повторении. Всем известно, что даже самую скучную информацию *можно* запомнить, если повторять ее снова и снова. При достаточном количестве повторений ваш мозг прикидывает: «*Вроде бы* несущественно, но раз одно и то же повторяется *столько* раз... Ладно, уговорил».

Быстрый способ основан на **повышении активности мозга** и особенно на сочетании разных ее *видов*. Доказано, что все факторы, перечисленные на предыдущей странице, помогают вашему мозгу работать на вас. Например, исследования показали, что размещение слов *внутри* рисунков (а не в подписях, в основном тексте и т. д.) заставляет мозг анализировать связи между текстом и графикой, а это приводит к активизации большего количества нейронов. Больше нейронов — выше вероятность того, что информация будет сочтена важной и достойной запоминания.

Разговорный стиль тоже важен: обычно люди проявляют больше внимания, когда они участвуют в разговоре, так как им приходится следить за ходом беседы и высказывать свое мнение. Причем мозг совершенно не интересует, что вы «разговариваете» с книгой! С другой стороны, если текст сух и формален, то мозг чувствует то же, что чувствуете вы на скучной лекции в роли пассивного участника. Его клонит в сон.

Но рисунки и разговорный стиль — это только начало.

Вот что сделали МЫ

Мы использовали **рисунки**, потому что мозг лучше приспособлен для восприятия графики, чем текста. С точки зрения мозга рисунок стоит 1024 слов. А когда текст комбинируется с графикой, мы внедряем текст прямо в рисунки, потому что мозг при этом работает эффективнее.

Мы используем **избыточность**: повторяем одно и то же несколько раз, применяя разные средства передачи информации, обращаемся к разным чувствам — и все для повышения вероятности того, что материал будет закодирован в нескольких областях вашего мозга.

Мы используем концепции и рисунки несколько **неожиданным** образом, потому что мозг лучше воспринимает новую информацию. Кроме того, рисунки и идеи обычно имеют ***эмоциональное содержание***, потому что мозг обращает внимание на биохимию эмоций. То, что заставляет нас ***чувствовать***, лучше запоминается — будь то ***шутка***, ***удивление*** или ***интерес***.

Мы используем ***разговорный стиль***, потому что мозг лучше воспринимает информацию, когда вы участвуете в разговоре, а не пассивно слушаете лекцию. Это происходит и при *чтении*.

В книгу включены многочисленные упражнения, потому что мозг лучше запоминает, когда вы что-то делаете. Мы постарались сделать их непростыми, но интересными — то, что предпочитает большинство читателей.

Мы совместили ***несколько стилей обучения***, потому что одни читатели предпочитают пошаговые описания, другие стремятся сначала представить «общую картину», а третьим хватает фрагмента кода. Независимо от ваших личных предпочтений полезно видеть несколько вариантов представления одного материала.

Мы постарались задействовать ***оба полушария вашего мозга***; это повышает вероятность усвоения материала. Пока одна сторона мозга работает, другая часто имеет возможность отдохнуть; это повышает эффективность обучения в течение продолжительного времени.

А еще в книгу включены ***истории*** и упражнения, отражающие другие точки зрения. Мозг глубже усваивает информацию, когда ему приходится оценивать и выносить суждения.

В книге часто встречаются ***вопросы***, на которые не всегда можно дать простой ответ, потому что мозг быстрее учится и запоминает, когда ему приходится что-то делать. Невозможно накачать ***мышцы***, наблюдая за тем, как занимаются ***другие***. Однако мы позаботились о том, чтобы усилия читателей были приложены в ***верном*** направлении. Вам не придется ломать голову над невразумительными примерами или разбираться в сложном, перенасыщенном техническим жаргоном или слишком лаконичном тексте.

В историях, примерах, на картинках используются ***люди*** — потому что вы тоже ***человек***. И ваш мозг обращает на людей больше внимания, чем на неодушевленные ***предметы***.

Вырежьте и прикрепите на холодильник.

Что можете сделать ВЫ, чтобы заставить свой мозг повиноваться

Мы свое дело сделали. Остальное за вами. Эти советы станут отправной точкой; прислушайтесь к своему мозгу и определите, что вам подходит, а что не подходит. Пробуйте новое.

1 Не торопитесь. Чем больше вы поймете, тем меньше придется запоминать.

Просто читать недостаточно. Когда книга задает вам вопрос, не переходите к ответу. Представьте, что кто-то *действительно* задает вам вопрос. Чем глубже ваш мозг будет мыслить, тем скорее вы поймете и запомните материал.

2 Выполняйте упражнения, делайте заметки.

Мы включили упражнения в книгу, но выполнять их за вас не собираемся. И не *разглядывайте* упражнения. **Берите карандаш и пишите**. Физические действия *во время* учения повышают его эффективность.

3 Читайте врезки.

Это значит: читайте все. ***Врезки — часть основного материала!*** Не пропускайте их.

4 Не читайте другие книги после этой перед сном.

Часть обучения (особенно перенос информации в долгосрочную память) происходит *после* того, как вы откладываете книгу. Ваш мозг не сразу усваивает информацию. Если во время обработки поступит новая информация, часть того, что вы узнали ранее, может быть потеряна.

5 Говорите вслух.

Речь активизирует другие участки мозга. Если вы пытаетесь что-то понять или получше запомнить, произнесите вслух. А еще лучше — попробуйте объяснить кому-нибудь другому. Вы будете быстрее усваивать материал и, возможно, откроете для себя что-то новое.

6 Пейте воду. И побольше.

Мозг лучше всего работает в условиях высокой влажности. Дегидратация (которая может наступить еще до того, как вы почувствуете жажду) снижает когнитивные функции.

7 Прислушивайтесь к своему мозгу.

Следите за тем, когда ваш мозг станет уставать. Если вы начинаете поверхностно воспринимать материал или забываете только что прочитанное, пора сделать перерыв.

8 Чувствуйте!

Ваш мозг должен знать, что материал книги действительно *важен*. Переживайте за героев наших историй. Придумывайте собственные подписи к фотографиям. Поморщиться над неудачной шуткой все равно *лучше*, чем не почувствовать ничего.

9 Пишите побольше кода!

Освоить программирование на Go можно только одним способом: **писать побольше кода**. Именно этим мы и будем заниматься в книге. Программирование — искусство, и добиться мастерства в нем можно только практикой. Для этого у вас будут все возможности: в каждой главе приведены упражнения, в которых вам придется решать задачи. Не пропускайте их — работа над упражнениями является важной частью процесса обучения. К каждому упражнению приводится решение — не бойтесь **заглянуть** в него, если окажетесь в тупике! (Споткнуться можно даже о маленький камешек.) По крайней мере постарайтесь решить задачу, прежде чем заглядывать в решение. Обязательно добейтесь того, чтобы ваше решение заработало, прежде чем переходить к следующей части книги.

Примите к сведению

Это учебник, а не справочник. Мы намеренно убрали из книги все, что могло бы помешать изучению материала, над которым вы работаете. И при первом чтении книги начинать следует с самого начала, потому что книга предполагает наличие у читателя определенных знаний и опыта.

Предполагается, что у вас уже есть опыт программирования на другом языке.

Большинство разработчиков приходит в Go *после* знакомства с другим языком программирования (причем нередко они спасаются бегством от него). Мы рассмотрим основы в достаточной степени, чтобы материал был понятен даже начинающему, но не будем подробно рассказывать, что такое переменная и как работает конструкция `if`. Читателю будет проще, если у него есть *хотя бы* небольшой опыт программирования.

Мы не описываем каждый существующий тип, функцию и пакет.

Go поставляется с ***огромным*** количеством встроенных пакетов. Безусловно, все они представляют интерес, но мы не смогли бы описать их даже в книге ***вдвое большего*** размера. Мы сосредоточились на основных типах и функциях, которые *особенно важны* для вас, то есть для новичка. Мы стремились к тому, чтобы вы глубоко поняли эти темы и четко представляли, когда и как их следует применять.

Упражнения ОБЯЗАТЕЛЬНЫ.

Упражнения являются частью основного материала книги. Одни упражнения способствуют запоминанию материала, другие помогают лучше понять его, третьи ориентированы на его практическое применение. ***Не пропускайте упражнения***.

Повторение применяется намеренно.

У книг этой серии есть одна принципиальная особенность: мы хотим, чтобы вы ***действительно хорошо*** усвоили материал. И чтобы вы запомнили все, что узнали. Большинство справочников не ставит своей целью успешное запоминание, но это не справочник, а ***учебник***, поэтому некоторые концепции излагаются в книге по нескольку раз.

Мы постарались сделать примеры по возможности компактными.

Никому не захочется продираться через 200 строк кода в поисках двух строк, которые важно понять. Большинство примеров в книге приводится в минимальном возможном контексте, чтобы та часть, которую вы изучаете, была по возможности простой и ясной. Не стоит ожидать, что код примеров защищен от ошибок или хотя бы полон. Этими аспектами займетесь *вы* после прочтения книги. Книга написана именно для *обучения*, поэтому приводимый код не всегда является полнофункциональным.

Все файлы примеров доступны в интернете. Их можно загрузить по адресу *http://headfirstgo.com/*.

Благодарности

Основателям серии

Огромное спасибо основателям серии Head First **Кэти Сьерра** и **Берту Бэйтсу**. Эта серия книг понравилась мне еще более десяти лет назад, когда я впервые столкнулся с ней, но я никогда не думал, что буду писать для нее. Спасибо за создание этого замечательного стиля обучения!

Издательству O'Reilly

Спасибо всем сотрудникам O'Reilly, которые принимали участие в работе над книгой, особенно редактору **Джеффу Блейлю**, а также **Кристер Браун**, **Рэчел Монахан** и другим участникам производственной группы.

Научным редакторам

Все мы совершаем ошибки. К счастью, научные редакторы **Тим Хекман**, **Эдвард Ю Шун Вон** и **Стефан Покман** постарались отыскать все мои ошибки. Вы даже не представляете, сколько проблем они нашли, потому что я быстро уничтожил все доказательства. Но их помощь и обратная связь были безусловно необходимы, и я им вечно благодарен!

И многим другим

Спасибо **Лео Ричардсону** за дополнительную корректуру.

И пожалуй, самое важное: спасибо **Кристин**, **Кортни**, **Брайану**, **Ленни** и **Джереми** за их терпение и поддержку (уже за две книги!).

Основы синтаксиса

Готовы поднять свой код на новый уровень? Нужен **простой** язык программирования, который **быстро компилируется** и **быстро выполняется**? Язык, с которым вы сможете **легко и удобно распространять** свое ПО среди пользователей? Тогда знакомьтесь: **Go** — язык программирования, ориентированный на **простоту** и **скорость**. Он проще других языков, и поэтому вы быстрее освоите его. Кроме того, Go эффективно использует мощь современных многоядерных процессоров, а значит, программы будут выполняться быстрее. В этой главе представлены возможности Go, которые упростят **вам работу** и наверняка придутся по вкусу **пользователям**.

На старт... Внимание... Go!

В 2007 году у поисковой системы Google возникли проблемы. Разработчикам приходилось заниматься сопровождением кодовой базы, состоящей из миллионов строк кода. Прежде чем тестировать новые изменения, они должны были откомпилировать код в исполняемую форму, а процесс занимал в лучшем случае около часа. Не стоит и говорить, что это плохо сказывалось на эффективности разработки.

Так инженеры Google Роберт Гризмер (Robert Griesemer), Роб Пайк (Rob Pike) и Кен Томпсон (Ken Thompson) наметили следующие цели для создания нового языка.

- Быстрая компиляция.
- Компактный код.
- Автоматическое освобождение неиспользуемой памяти (уборка мусора).
- Простота написания программ, способных одновременно выполнять несколько операций (параллелизм).
- Качественная поддержка многоядерных процессоров.

Через пару лет работы Google создала Go: язык, на котором код пишется быстро, а полученные программы легко компилируются и выполняются. В 2009 году проект перешел на лицензию с открытым исходным кодом. Сейчас этот язык могут свободно использовать все желающие. И поверьте, он того стоит! Go стремительно набирает популярность из-за своей простоты и мощи.

Если вы пишете программы командной строки, Go сможет строить исполняемые файлы для Windows, macOS и Linux из одного исходного кода. Если вы пишете веб-сервер, Go поможет организовать обслуживание одновременного подключения многих пользователей. Словом, *что бы* вы ни писали, язык Go упростит сопровождение и расширение вашего кода.

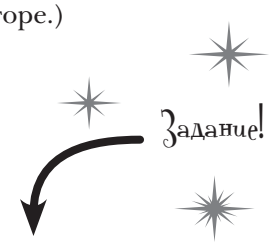
Готовы узнать больше? Тогда вперед!

Интерактивная среда Go Playground

Чтобы поэкспериментировать с Go, проще всего открыть страницу *https://play.golang.org* в браузере. Команда разработчиков Go создала простой редактор, в котором вы можете ввести код и выполнить его на внешнем сервере. Результат выводится прямо в браузере.

(Конечно, этот способ работает только при наличии хорошего подключения к интернету. Если этот вариант вам не подходит, на с. 60 рассказано о том, как загрузить и запустить компилятор Go прямо на вашем компьютере. В этом случае приведенные примеры следует запускать в компиляторе.)

Давайте опробуем Go в деле!



1. Откройте страницу *https://play.golang.org* в браузере. (Не беспокойтесь, если то, что вы увидите, отличается от приведенного скриншота; это говорит лишь о том, что с момента издания книги сайт был улучшен!)

2. Очистите область редактирования и введите следующий код:

```
package main

import "fmt"

func main() {
     fmt.Println("Hello, Go!")
}
```

Спокойно! Мы объясним, что все это значит, на следующей странице!

3. Щелкните на кнопке Format. Ваш код автоматически переформатируется в соответствии с общепринятыми соглашениями Go.

4. Щелкните на кнопке Run.

В нижней части экрана должно появиться сообщение «Hello, Go!». Поздравляем, вы только что выполнили свою первую программу на языке Go!

Результат. → `Hello, Go!`

Переверните страницу, и мы подробно всё объясним...

Что это все означает?

Вы только что выполнили свою первую программу на языке Go! А теперь рассмотрим код и разберемся, что же это все означает...

Каждый файл Go начинается с директивы `package`. **Пакет** представляет собой набор блоков кода, выполняющих похожие операции — например, форматирование строк или построение графических изображений. Директива `package` задает имя пакета, частью которого станет код этого файла. В нашем случае используется специальный пакет `main`; это необходимо для того, чтобы код можно было запускать напрямую (чаще всего в терминале).

Файлы Go почти всегда содержат одну или несколько директив `import`. Каждый файл должен **импортировать** другие пакеты, чтобы использовать код, содержащийся в этих пакетах. Но если бы в оперативную память загружался весь код Go, который только найдется на компьютере, программы стали бы слишком громоздкими и медленными, поэтому вы указываете только те пакеты, которые вам действительно нужны, — то есть импортируете эти пакеты.

```
package main

import "fmt"

func main() {
        fmt.Println("Hello, Go!")
}
```

Эта строка сообщает, что остальной код этого файла относится к пакету «main».

Означает, что мы будем использовать код форматирования текста из пакета «fmt».

Функция «main» играет особую роль — именно она выполняется при запуске программы.

Для этого она вызывает функцию «Println» из пакета «fmt».

Эта строка выводит сообщение «Hello, Go!» в терминале (или браузере, если вы используете среду Go Playground).

Далее в каждом файле Go следует код, который часто разбивается на одну или несколько функций. **Функция** представляет собой набор строк кода, которые могут **вызываться** (выполняться) из других мест программы. При запуске программа Go ищет функцию с именем `main` и выполняет ее в первую очередь; поэтому-то мы и присвоили своей функции имя `main`.

Если что-то показалось непонятным — не огорчайтесь!

Все эти темы будут более подробно рассмотрены на нескольких ближайших страницах.

Структура типичного файла Go

Вы быстро привыкнете к тому, что эти три части (именно в таком порядке) встречаются практически во всех файлах Go, с которыми вы будете работать:

1. Директива `package`.
2. Директива `import`.
3. Собственно код программы.

Директива package.
```
package main
```
Директива import.
```
import "fmt"
```
Собственно код программы.
```
func main() {
        fmt.Println("Hello, Go!")
}
```

Поговорка гласит: «Всему свое место и все хорошо на своем месте». Go — чрезвычайно *последовательный* язык. И это хорошо: часто вы просто *знаете, где* в вашем проекте находится тот или иной код, и вам даже не приходится над этим задумываться!

Часто Задаваемые Вопросы

В: Другие языки программирования требуют, чтобы каждая команда завершалась символом «точка с запятой» (;). А в Go есть такое требование?

О: В Go команды *можно* разделять символом ";", но это необязательно (и более того, обычно считается нежелательным).

В: Что это за кнопка Format? Почему мы нажимали ее перед запуском своего кода?

О: Компилятор Go содержит стандартное средство форматирования — команду `go fmt`. Кнопка Format представляет собой веб-версию команды `go fmt`.

Когда вы распространяете свой код, другие разработчики ожидают, что он будет оформлен в стандартном формате Go. Это означает стандартное форматирование отступов, пробелов и т. д., чтобы другим людям было проще читать ваш код. Если в других языках программирования разработчикам приходилось вручную переформатировать свой код, чтобы он соответствовал стилевому руководству, в Go достаточно выполнить команду `go fmt`, которая автоматически сделает все за вас.

Мы использовали автоматическое форматирование во всех примерах, созданных для этой книги. И вам тоже стоит применять его во всем вашем коде!

А если что-то пойдет не так?

Программы Go должны соблюдать определенные правила, чтобы не сбивать с толку компилятор. Если вы нарушите одно из таких правил, компилятор выдаст сообщение об ошибке.

Допустим, вы забыли поставить круглые скобки при вызове функции `Println` в строке 6.

Попытавшись запустить эту версию программы, вы получите сообщение об ошибке:

```
Строка
     1 package main
     2
     3 import "fmt"
     4
     5 func main() {
     6         fmt.Println "Hello, Go!"
     7 }
```

Допустим, вы забыли поставить здесь круглые скобки...

```
prog.go:6:14: syntax error: unexpected literal "Hello, Go!" at end of statement
```

Имя файла, используемое Go Playground.

Номер строки, в которой произошла ошибка.

Описание ошибки.

Позиция в строке, в которой произошла ошибка.

Компилятор сообщает, в каком исходном файле и в какой строке следует искать ошибку. (Среда Go Playground сохраняет ваш код во временном файле, прежде чем выполнять его; отсюда имя файла *prog.go*.) Далее следует описание ошибки. В нашем случае из-за отсутствия круглых скобок компилятор не осознает, что мы пытаемся вызвать функцию `Println`, и поэтому не понимает, что делает сообщение `"Hello, Go"` в конце строки 6.

Сломай и изучи!

Чтобы получить представление о правилах, которые должны соблюдаться в программах Go, попробуем намеренно сломать нашу программу разными способами. Возьмите этот пример кода, внесите одно из указанных изменений и запустите программу, затем отмените изменение и переходите к следующему. Посмотрите, что из этого выйдет!

```
package main

import "fmt"

func main() {
        fmt.Println("Hello, Go!")
}
```

Попробуйте сломать наш пример кода и посмотрите, что получится!

Если...	*...программа не будет работать, потому что...*
Удалить директиву `package`... ~~`package main`~~	Каждый файл Go должен начинаться с директивы `package`
Удалить инструкцию `import`... ~~`import "fmt"`~~	Каждый файл Go должен импортировать все пакеты, которые в нем используются
Импортировать второй (неиспользуемый) пакет... `import "fmt"` `import "strings"`	Файлы Go должны импортировать *только* те пакеты, которые в них используются. (Это ускоряет компиляцию кода!)
Переименовать функцию `main`... `func` ~~`main`~~`hello`	Компилятор Go ищет функцию с именем `main`, чтобы выполнить ее при запуске программы
Преобразовать имя `Println` к нижнему регистру... `fmt.`~~`P`~~`println("Hello, Go!")`	В Go учитывается регистр символов. `fmt.Println` — действительное имя, но имени `fmt.println` не существует
Удалить имя пакета перед `Println`... ~~`fmt.`~~`Println("Hello, Go!")`	Функция `Println` не принадлежит пакету `main`, поэтому перед вызовом функции необходимо указать имя пакета

Давайте вместе проведем первый эксперимент.

Удалите директиву package...

```
import "fmt"

func main() {
        fmt.Println("Hello, Go!")
}
```

... и получите сообщение об ошибке!

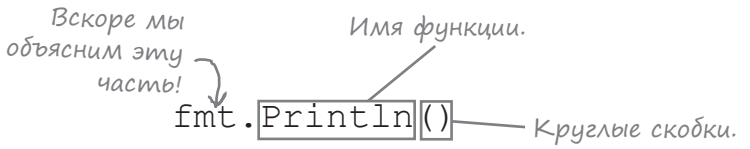
```
can't load package: package main:
prog.go:1:1: expected 'package', found 'import'
```

Вызов функций

В нашем примере вызывается функция `Println` из пакета `fmt`. Чтобы вызвать функцию, введите имя функции (в данном случае `Println`) и пару круглых скобок.

```
package main

import "fmt"

func main() {
        fmt.Println("Hello, Go!")
}
```

Вызов функции Println.



Как и многие функции, `Println` может получать один или несколько **аргументов** — значений, с которыми должна работать функция. Аргументы перечисляются в круглых скобках после имени функции.

В круглых скобках перечисляются аргументы, разделенные запятыми.

```
fmt.Println("First argument", "Second argument")
```

Результат.

```
First argument Second argument
```

Хотя функции `Println` и можно передать несколько аргументов, она может вызываться без них. Когда мы будем заниматься другими функциями, вы заметите, что они обычно должны получать строго определенное количество аргументов. Если аргументов будет слишком мало или слишком много, то вы получите сообщение об ошибке, в котором будет указано ожидаемое количество аргументов. В этом случае программу нужно будет исправить.

Функция Println

При помощи функции `Println` можно узнать, как идет выполнение программы. Все аргументы, переданные этой функции, выводятся в терминале (а их значения разделяются пробелами).

После вывода всех аргументов `Println` переходит на следующую строку в терминале. (Отсюда суффикс «ln» — сокращение от «line» — в конце имени.)

```
fmt.Println("First argument", "Second argument")
fmt.Println("Another line")
```

Результат.

```
First argument Second argument
Another line
```

Использование функций из других пакетов

Весь код нашей первой программы является частью пакета `main`, но функция `Println` принадлежит пакету `fmt` (сокращение от «format»). Чтобы в программе можно было вызвать функцию `Println`, необходимо сначала импортировать пакет, содержащий эту функцию.

```
package main

import "fmt"

func main() {
       fmt.Println("Hello, Go!")
}
```

Пакет «fmt» необходимо импортировать, чтобы вызвать его функцию Println.

Сообщает, что вызываемая функция является частью пакета «fmt».

После того как пакет будет импортирован, вы сможете вызывать функции из этого пакета. Для этого укажите имя пакета, поставьте точку (.) и введите имя нужной функции.

Следующий пример демонстрирует вызов функций из двух других пакетов. Так как мы собираемся импортировать несколько пакетов, то переходим на альтернативный формат инструкции, который позволяет перечислять в круглых скобках сразу несколько пакетов, по одному имени пакета в строке.

```
package main

import (
       "math"
       "strings"
)

func main() {
       math.Floor(2.75)
       strings.Title("head first go")
}
```

Альтернативный формат инструкции «import» позволяет импортировать сразу несколько пакетов.

Импортируем пакет «math», чтобы использовать функцию math.Floor.

Импортируем пакет «strings», чтобы использовать функцию strings.Title.

Вызываем функцию Floor из пакета «math».

Вызываем функцию Title из пакета «strings».

Программа ничего не выводит. (Сейчас мы объясним почему!)

После того как пакеты `math` и `strings` будут импортированы, вы сможете вызвать функцию `Floor` из пакета `math` в форме `math.Floor`, а также функцию `Title` из пакета `strings` в форме `strings.Title`.

Вероятно, вы заметили, что несмотря на включение двух вызовов функций в код программы, этот пример не выводит никакого результата. Сейчас мы покажем, как решить эту проблему.

Возвращаемые значения функций

В приведенном примере мы вызываем функции `math.Floor` и `strings.Title`, но эти функции ничего не выводят:

```
package main

import (
       "math"
       "strings"
)

func main() {
       math.Floor(2.75)
       strings.Title("head first go")
}
```

Программа ничего не выводит!

При вызове функции `fmt.Println` вам не нужно обмениваться с ней дополнительной информацией. Вы передаете `Println` одно или несколько выводимых значений и ожидаете вывод. Но иногда программа должна вызвать функцию и получить от нее дополнительные данные. Из-за этого в большинстве языков программирования функции имеют **возвращаемые значения**, которые вычисляются функциями и возвращаются на сторону вызова.

Функции `math.Floor` и `strings.Title` относятся к числу функций, имеющих возвращаемое значение. Функция `math.Floor` получает число с плавающей точкой, округляет его до ближайшего меньшего целого и возвращает полученное число. А функция `strings.Title` получает строку, преобразует первую букву каждого слова к верхнему регистру и возвращает полученную строку.

Чтобы увидеть результаты этих вызовов функций, необходимо взять возвращаемые значения и передать их `fmt.Println`:

```
package main

import (
       "fmt"
       "math"
       "strings"
)

func main() {
       fmt.Println(math.Floor(2.75))
       fmt.Println(strings.Title("head first go"))
}
```

Также импортируется пакет «fmt».

Функция fmt.Println вызывается для возвращаемого значения функции math.Floor.

Получает число, округляет его в меньшую сторону и возвращает полученное значение.

Результат.

```
2
Head First Go
```

Функция fmt.Println вызывается для возвращаемого значения функции strings.Title.

Получает строку и возвращает новую строку, в которой все слова начинаются с буквы верхнего регистра.

После внесения этого изменения возвращаемые значения выводятся в терминале, где вы сможете ознакомиться с результатами.

У бассейна

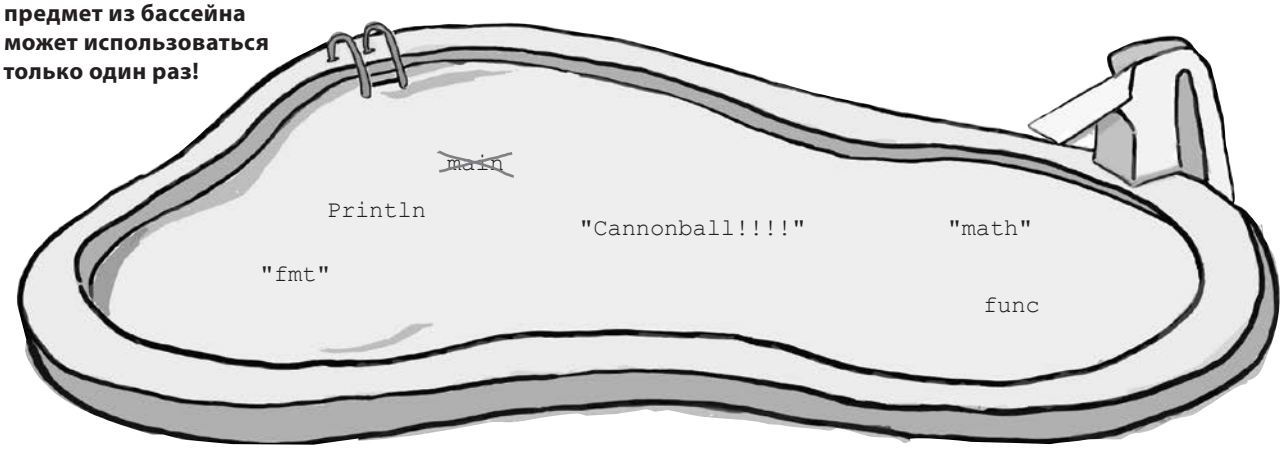
Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты необязательно. Ваша **задача**: построить код, который будет успешно выполняться и выведет показанный результат.

```
package main        <- Мы вставили первый фрагмент за вас!

import (
        _____
)

_____ main() {
        fmt.Println(_____________)
}
```

Результат.

```
Cannonball!!!!
```

Примечание: каждый предмет из бассейна может использоваться только один раз!



Ответ на с. 63.

Шаблон программы Go

Ниже приводятся примеры фрагментов кода. Просто представьте, что они вставляются в следующую завершенную программу Go.

А еще лучше — введите эту программу в среде Go Playground, а потом вставляйте фрагменты по одному и наблюдайте за тем, что они делают!

```
package main

import "fmt"

func main() {
        fmt.Println(
}
```

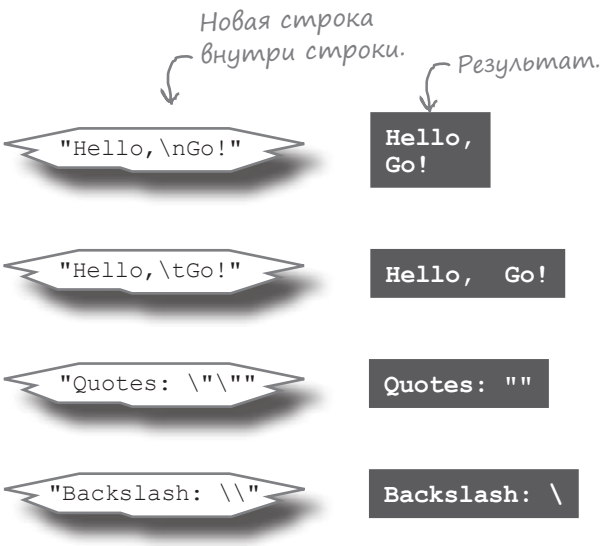
Вставьте сюда свой код!

Строки

В аргументах `Println` передавались **строки**. Строка представляет собой последовательность байтов, которые обычно представляют символы текста. Строки можно определять прямо в программе в виде **строковых литералов**: компилятор Go интерпретирует текст, заключенный в двойные кавычки, как строку.

Некоторые управляющие символы, которые неудобно вводить с клавиатуры (символы новой строки, табуляции и т. д.), внутри строк могут представляться в виде **служебных последовательностей**: символа «обратный слеш», за которым следует другой символ (или символы).

Последовательность sequence	Значение
\n	Символ новой строки
\t	Символ табуляции
\"	Двойная кавычка
\\	Обратный слеш



Руны

```
package main

import "fmt"

func main() {
        fmt.Println(          )
}
```

И снова наш шаблон...

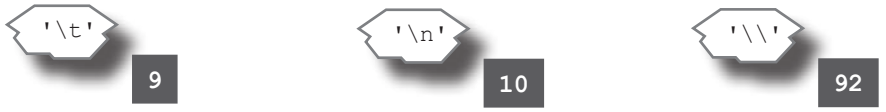
Вставьте сюда свой код!

Если строки обычно используются для представления последовательностей символов, то **руны** в языке Go представляют отдельные символы.

Строковые литералы заключаются в двойные кавычки (`"`), а **рунные литералы** записываются в одиночных кавычках (`'`).

В программах Go могут использоваться практически любые символы любых мировых языков, потому что в Go для хранения рун используется стандарт Юникод. Руны хранятся в виде числовых кодов, а не в виде символов; если передать руну функции `fmt.Println`, то выведется числовой код, а не исходный символ.

В рунных литералах (как и в строковых) можно использовать служебные последовательности для представления символов, которые неудобно вводить с клавиатуры для включения в программу:



Логические значения

Логические величины принимают всего два возможных значения: `true` или `false`. Они особенно удобны в условных командах, в которых выполнение блока кода зависит от того, истинно или ложно некоторое условие. (Условные команды рассматриваются в следующей главе.)

Числа

Числа тоже можно определять прямо в программном коде. Это еще проще, чем определять строковые литералы: просто введите нужное число.

```
package main

import "fmt"

func main() {
        fmt.Println(          )
}
```

И снова наш шаблон...

Вставьте сюда свой код!

```
42
```
Целое число.

42 Результат.

```
3.1415
```
Число с плавающей точкой.

3.1415

Как вы вскоре увидите, в языке Go целые числа и числа с плавающей точкой интерпретируются как разные типы. Помните, что целое число можно отличить от числа с плавающей точкой по разделителю дробной части — точке.

Математические операции и сравнения

Основные математические операторы Go работают так же, как и в большинстве других языков. Оператор + выполняет сложение, оператор - выполняет вычитание, оператор * — умножение, и оператор / — деление.

```
1 + 2
```
3

```
5.4 - 2.2
```
3.2

```
3 * 4
```
12

```
7.5 / 5
```
1.5

При помощи операторов < и > можно сравнить два значения и проверить, какое из них больше другого. Оператор == (*два* знака равенства) проверяет, что два значения равны, а оператор != проверяет, что два значения не равны, <= проверяет, что второе значение меньше *или* равно первому, а оператор >= проверяет, что второе значение больше или равно первому.

Результатом сравнения является логическое значение (`true` или `false`).

```
4 < 6
```
true

```
4 > 6
```
false

```
2+2 == 5
```
false

```
2+2 != 5
```
true

```
4 <= 6
```
true

```
4 >= 4
```
true

Типы

В предыдущем примере кода использовалась функция `math.Floor`, округляющая число с плавающей точкой с уменьшением, и функция `strings.Title`, преобразующая первую букву каждого слова в строке к верхнему регистру. Логично ожидать, что в аргументе функции `Floor` передается число, а в аргументе функции `Title` передается строка. Но что произойдет, если передать функции `Floor` строку, а функции `Title` число?

```
package main

import (
       "fmt"
       "math"
       "strings"
)

func main() {
       fmt.Println(math.Floor("head first go"))
       fmt.Println(strings.Title(2.75))
}
```

Обычно получает число с плавающей точкой!

Обычно получает строку!

Ошибки.

```
cannot use "head first go" (type string) as type float64 in argument to math.Floor
cannot use 2.75 (type float64) as type string in argument to strings.Title
```

Go выводит два сообщения об ошибках — по одному для каждого вызова функции, а программа даже не запускается!

Объекты в окружающем мире часто можно разделить на типы в зависимости от того, для чего они используются. Машину или грузовик нельзя съесть на завтрак, а на омлете или чашке с кукурузными хлопьями не поедешь на работу — они предназначены для другого.

Аналогичным образом значения в Go делятся на разные **типы**, которые определяют, для чего они могут использоваться. Целые числа могут использоваться в математических операциях, а в строке не могут. В строках можно преобразовать регистр символов, а в числах нельзя... И так далее.

В языке Go используется **статическая типизация** — это означает, что типы всех значений известны еще до запуска программы. Функции ожидают, что их аргументы относятся к конкретным типам, а их возвращаемые значения тоже имеют типы (которые могут совпадать или не совпадать с типами аргументов). Если случайно использовать неправильный тип значения в неподходящем месте, компилятор Go выдаст сообщение об ошибке. И это очень хорошо: вы узнаете о существовании проблемы до того, как о ней узнают пользователи!

Go — язык со статической типизацией. Если вы используете неправильный тип значения в неподходящем месте, Go сообщит вам об этом.

Типы (продолжение)

Чтобы узнать тип любого значения, передайте его функции `TypeOf` из пакета `reflect`. Давайте узнаем типы некоторых значений, которые уже встречались в примерах программ:

```
package main

import (
       "fmt"
       "reflect"
)

func main() {
       fmt.Println(reflect.TypeOf(42))
       fmt.Println(reflect.TypeOf(3.1415))
       fmt.Println(reflect.TypeOf(true))
       fmt.Println(reflect.TypeOf("Hello, Go!"))
}
```

Импортируем пакет «reflect», чтобы использовать его функцию TypeOf.

Возвращает тип своего аргумента.

Результат.

```
int
float64
bool
string
```

Эти типы предназначены для следующих целей:

Тип	Описание
`int`	Целое число (не имеющее дробной части)
`float64`	Число с плавающей точкой. Тип используется для хранения чисел, имеющих дробную часть. (Для хранения числа используются 64 бита данных, отсюда суффикс 64 в имени. Значения типа float64 обеспечивают очень высокую, хотя и не бесконечную точность.)
`bool`	Логическое значение (`true` или `false`)
`string`	Строка — последовательность данных, которые обычно представляют символы текста

Упражнение

Соедините каждый фрагмент кода с правильным типом.

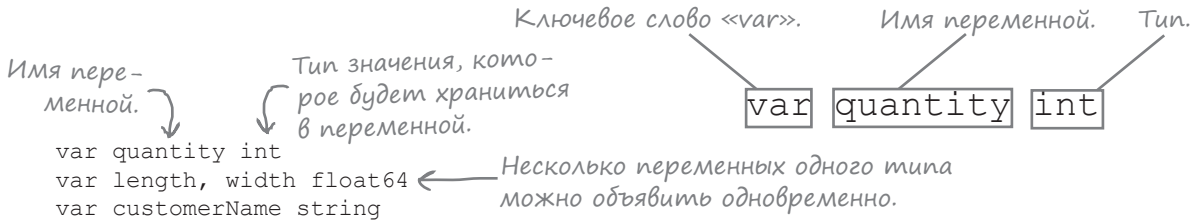
Некоторым типам могут соответствовать несколько фрагментов.

`reflect.TypeOf(25)`	`int`
`reflect.TypeOf(true)`	
`reflect.TypeOf(5.2)`	`float64`
`reflect.TypeOf(1)`	
`reflect.TypeOf(false)`	`bool`
`reflect.TypeOf(1.0)`	
`reflect.TypeOf("hello")`	`string`

Ответы на с. 63.

Объявление переменных

В языке Go **переменная** представляет собой область памяти, в которой хранится значение. Чтобы к переменной можно было обращаться по имени, используйте **объявление переменной**. Объявление состоит из ключевого слова `var`, за которым следует имя и тип значений, которые будут храниться в переменной.



Имя переменной.

Тип значения, которое будет храниться в переменной.

```
var quantity int
var length, width float64
var customerName string
```

Несколько переменных одного типа можно объявить одновременно.

После того как переменная будет объявлена, ей можно будет присвоить любое значение этого типа оператором = (*один* знак равенства):

```
quantity = 2
customerName = "Damon Cole"
```

В одной команде можно присвоить значения сразу нескольким переменным. Для этого перечислите имена переменных слева от = и такое же количество значений в правой части, разделяя их запятыми.

```
length, width = 1.2, 2.4
```

Одновременное присваивание значений нескольким переменным.

После того как переменной будет присвоено значение, вы сможете использовать ее в любом контексте, где может использоваться исходное значение:

```
package main

import "fmt"

func main() {
    var quantity int
    var length, width float64
    var customerName string

    quantity = 4
    length, width = 1.2, 2.4
    customerName = "Damon Cole"

    fmt.Println(customerName)
    fmt.Println("has ordered", quantity, "sheets")
    fmt.Println("each with an area of")
    fmt.Println(length*width, "square meters")
}
```

Объявление переменных.

Присваивание значений переменным.

Использование переменных.

```
Damon Cole
has ordered 4 sheets
each with an area of
2.88 square meters
```

Объявление переменных (продолжение)

Если значение переменной известно заранее, можно объявить переменную и присвоить ей значение в одной строке:

Просто добавьте присваивание в конец команды.

Объявление переменных и присваивание значений

```
var quantity int = 4
var length, width float64 = 1.2, 2.4
var customerName string = "Damon Cole"
```

Если вы объявляете несколько переменных, укажите несколько значений.

Существующим переменным можно присваивать новые значения, но эти значения должны относиться к тому же типу. Статическая типизация в Go гарантирует, что переменной не будет случайно присвоено значение неподходящего типа.

Типы присваиваемых значений не соответствуют объявленным типам!

```
quantity = "Damon Cole"
customerName = 4
```

Ошибки.

```
cannot use "Damon Cole" (type string) as type int in assignment
cannot use 4 (type int) as type string in assignment
```

Если значение переменной присваивается одновременно с ее объявлением, тип переменной в объявлении обычно не указывают. Тип значения, присвоенного переменной, будет считаться типом этой переменной.

Тип переменной не указан.

```
var quantity = 4
var length, width = 1.2, 2.4
var customerName = "Damon Cole"
fmt.Println(reflect.TypeOf(quantity))
fmt.Println(reflect.TypeOf(length))
fmt.Println(reflect.TypeOf(width))
fmt.Println(reflect.TypeOf(customerName))
```

```
int
float64
float64
string
```

Нулевые значения

Если переменная объявляется без присваивания значения, то она будет содержать **нулевое значение** для этого типа. Для числовых типов нулевое значение равно `0`:

```
var myInt int
var myFloat float64
fmt.Println(myInt, myFloat)
```

Нулевое значение для переменных «int» — 0.

```
0 0
```

Нулевое значение для переменных «float64» — 0.

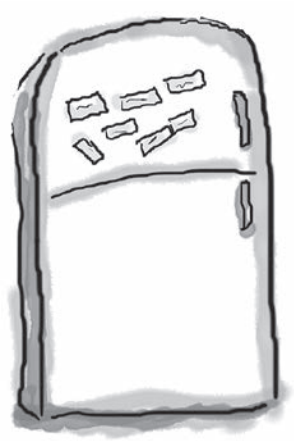
Но для других типов значение `0` будет недействительным, поэтому нулевое значение для этого типа будет отличаться. Скажем, для строковых переменных нулевым значением является пустая строка, а для переменных `bool` — значение `false`.

```
var myString string
var myBool bool
fmt.Println(myString, myBool)
```

Нулевое значение для переменных «string» — пустая строка.

```
false
```

Нулевое значение для переменных «bool» — false.



Развлечения с магнитами

На холодильнике была выложена программа Go. К сожалению, некоторые магниты упали на пол. Удастся ли вам расставить фрагменты кода по местам и создать работоспособную программу, которая будет выводить нужный результат?

Результат.

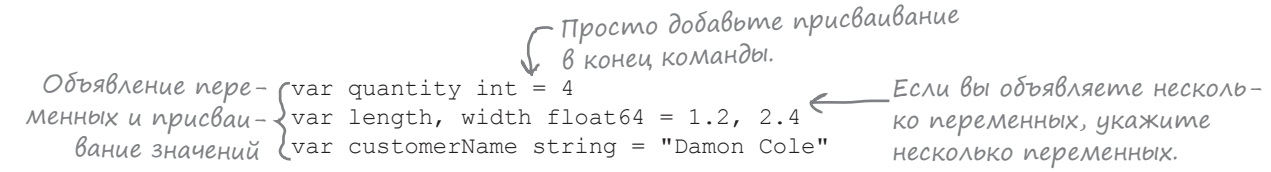
```
I started with 10 apples.
Some jerk ate 4 apples.
There are 6 apples left.
```

```
, "apples.")
, "apples.")
, "apples left.")
var
var
int
originalCount
func main() {
}
int
originalCount
fmt.Println("I started with",
eatenCount
fmt.Println("Some jerk ate",
=
=
fmt.Println("There are",
10
4
eatenCount
import (
    "fmt"
)
package main
originalCount-eatenCount
```

Ответ на с. 64.

Короткие объявления переменных

Ранее мы уже упоминали о том, что объявление переменной и присваивание ей значения можно совместить в одной строке:



Но если вы знаете исходное значение переменной на момент ее объявления, то можно применить **короткое объявление переменной**. Вместо того чтобы явно объявлять тип переменной и позднее присваивать ей значение оператором =, вы совмещаете эти две операции с помощью синтаксиса :=.

Давайте изменим предыдущий пример, чтобы в нем использовались короткие объявления переменных:

```
package main

import "fmt"

func main() {
    quantity := 4
    length, width := 1.2, 2.4
    customerName := "Damon Cole"

    fmt.Println(customerName)
    fmt.Println("has ordered", quantity, "sheets")
    fmt.Println("each with an area of")
    fmt.Println(length*width, "square meters")
}
```

Объявление переменных и присваивание значений

```
Damon Cole
has ordered 4 sheets
each with an area of
2.88 square meters
```

Явно объявлять тип переменной не обязательно: тип значения, присвоенного переменной, становится типом этой переменной.

Поскольку короткие объявления переменных очень удобны и компактны, они используются чаще обычных объявлений. Впрочем, время от времени вам будут встречаться обе формы, поэтому важно знать их.

Сломай и изучи!

Возьмите программу, в которой используются переменные, внесите одно из указанных изменений и запустите программу; затем отмените изменение и переходите к следующему. Посмотрите, что из этого выйдет!

```
package main

import "fmt"

func main() {
       quantity := 4
       length, width := 1.2, 2.4
       customerName := "Damon Cole"

       fmt.Println(customerName)
       fmt.Println("has ordered", quantity, "sheets")
       fmt.Println("each with an area of")
       fmt.Println(length*width, "square meters")
}
```

```
Damon Cole
has ordered 4 sheets
each with an area of
2.88 square meters
```

Если...	*...программа не будет работать, потому что...*
Добавить второе объявление для той же переменной `quantity := 4` `quantity := 4`	Переменную можно объявить только один раз. (Хотя при желании ей можно сколько угодно раз присваивать новые значения. Также можно объявлять другие переменные с тем же именем, но для этого они должны принадлежать другой области видимости. Тема областей видимости рассматривается в следующей главе.)
Убрать символ «:» из короткого объявления переменной `quantity = 4`	Если вы забудете поставить двоеточие, конструкция рассматривается как присваивание, а не как объявление, а переменной, которая не была объявлена, нельзя будет присвоить значение

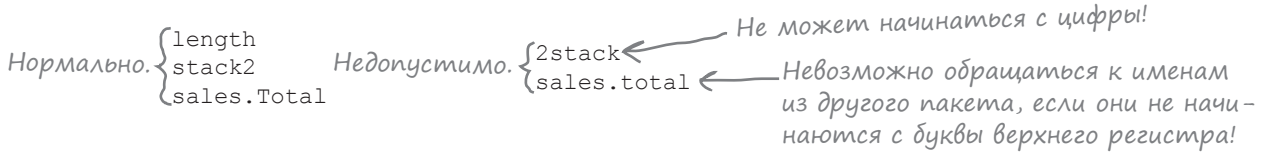
Присвоить строковое значение переменной `int`
`quantity := 4`
`quantity = "a"`
	Переменным могут присваиваться только значения того же типа

Количества переменных и значений не совпадают `length, width := 1.2`	Вы должны предоставить значение для каждой переменной и переменную для каждого значения
Удалить код, в котором используется переменная ~~`fmt.Println(customerName)`~~	Все объявленные переменные должны использоваться в программе. Если вы удаляете код, в котором используется переменная, необходимо также удалить и объявление

Правила выбора имен

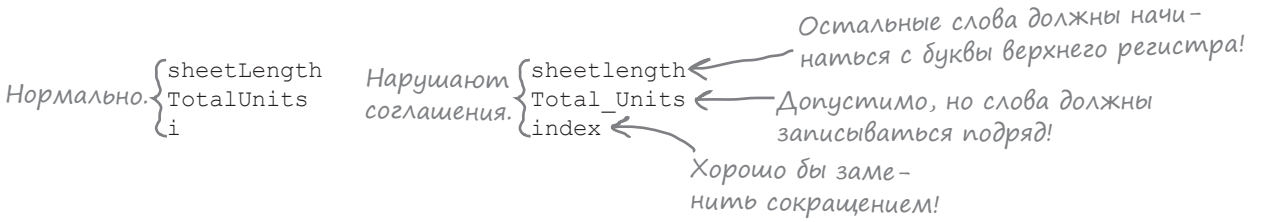
В Go существует один простой набор правил, применяемых к именам переменных, функций и типов:

- Имя должно начинаться с буквы и может содержать любое количество дополнительных букв и цифр.
- Если имя переменной, функции или типа начинается с буквы верхнего регистра, оно считается **экспортируемым** и может использоваться в других пакетах, кроме текущего. (Именно поэтому буква `P` в `fmt.Println` имеет верхний регистр: это нужно для того, чтобы его можно было использовать в `main` или любом другом пакете.) Если имя переменной/функции/типа начинается с буквы нижнего регистра, оно считается **неэкспортируемым**. Такие имена доступны только в текущем пакете.



Эти правила должны обязательно выполняться на уровне языка. Но сообщество Go также соблюдает ряд дополнительных соглашений:

- Если имя состоит из нескольких слов, каждое слово после первого должно начинаться с буквы верхнего регистра, и они должны следовать друг за другом без разделения пробелами: `topPrice`, `RetryConnection` и т. д. (Первая буква имени имеет верхний регистр только в том случае, если оно должно экспортироваться из пакета.) Этот стиль записи часто называется *верблюжьим регистром,* потому что буквы верхнего регистра напоминают горбы у верблюда.
- Если смысл имени очевиден по контексту, в сообществе Go принято сокращать его: использовать `i` вместо `index`, `max` вместо `maximum` и т. д. (Однако мы в Head First считаем, что во время изучения нового языка ничего очевидного нет, и поэтому *не будем* придерживаться этого соглашения в книге.)



Только переменные, функции и типы, имена которых начинаются с буквы верхнего регистра, считаются экспортируемыми, то есть доступными за пределами текущего пакета.

Преобразования

При выполнении математических операций и операций сравнения в Go значения должны относиться к одному типу. Если же типы различаются, то при попытке выполнения кода вы получите сообщение об ошибке.

Создаем переменную типа float64.

Создаем переменную типа int.

```
var length float64 = 1.2
var width int = 2
fmt.Println("Area is", length*width)
fmt.Println("length > width?", length > width)
```

Если использовать float64 и int в математической операции...

...или при сравнении...

...вы получите сообщение об ошибке!

Ошибки.

```
invalid operation: length * width (mismatched types float64 and int)
invalid operation: length > width (mismatched types float64 and int)
```

Этот принцип действует и при присваивании новых значений переменным. Если тип присваиваемого значения не соответствует объявленному типу переменной, вы получите сообщение об ошибке.

Создаем переменную типа float64.

Создаем переменную типа int.

```
var length float64 = 1.2
var width int = 2
length = width
fmt.Println(length)
```

Если присвоить значение int переменной float64...

...вы получите сообщение об ошибке!

Ошибка.

```
cannot use width (type int) as type float64 in assignment
```

Проблема решается **преобразованием** значений одного типа к другому типу. Для этого следует указать тип, к которому должно быть преобразовано значение, а за ним преобразуемое значение в скобках.

```
var myInt int = 2
float64(myInt)
```

Тип, к которому преобразуется значение.

Преобразуемое значение.

В результате преобразования вы получите новое значение нужного типа. В следующем примере выводится результат вызова `TypeOf` для значения в целочисленной переменной, а потом результат вызова для того же значения после преобразования к типу `float64`:

Без преобразования...

```
var myInt int = 2
fmt.Println(reflect.TypeOf(myInt))
fmt.Println(reflect.TypeOf(float64(myInt)))
```

```
int
float64
```

Тип изменен.

После преобразования...

Преобразования (продолжение)

Давайте исправим наш пример неработоспособного кода и преобразуем значение `int` к типу `float64` перед его использованием в математических операциях или сравнениях с другими значениями `float64`.

```
var length float64 = 1.2
var width int = 2
fmt.Println("Area is", length*float64(width))
fmt.Println("length > width?", length > float64(width))
```

Преобразование int к типу float64 перед умножением на другое значение float64.

Преобразование int к типу float64 перед сравнением с другим значением float64.

```
Area is 2.4
length > width? false
```

Теперь математические операции и операции сравнения работают правильно!

Попробуем преобразовать значение `int` к типу `float64` перед присваиванием переменной `float64`:

```
var length float64 = 1.2
var width int = 2
length = float64(width)
fmt.Println(length)
```

Преобразование int к типу float64 перед присваиванием переменной float64.

```
2
```

И снова после преобразования присваивание проходит успешно.

Всегда держите в голове, как преобразования изменяют выходные значения. Например, переменные `float64` могут хранить дробные значения, а переменные `int` — нет. Когда вы преобразовываете `float64` в `int`, дробная часть просто отбрасывается! Это может внести путаницу в любые операции, выполняемые с результирующим значением.

```
var length float64 = 3.75
var width int = 5
width = int(length)
fmt.Println(width)
```

В результате этого преобразования дробная часть теряется!

```
3
```

Полученное значение стало на 0,75 меньше!

Но если действовать внимательно, вы поймете, что преобразования исключительно важны для работы с Go. С ними вы сможете совместно использовать несовместимые типы.

Упражнение

Ниже приведен код Go, который вычисляет общую цену с учетом налога и определяет, хватит ли имеющихся денег для покупки. Но если вы попытаетесь включить его в полноценную программу, компилятор выдаст сообщения об ошибках!

```
var price int = 100
fmt.Println("Price is", price, "dollars.")

var taxRate float64 = 0.08
var tax float64 = price * taxRate
fmt.Println("Tax is", tax, "dollars.")

var total float64 = price + tax
fmt.Println("Total cost is", total, "dollars.")

var availableFunds int = 120
fmt.Println(availableFunds, "dollars available.")
fmt.Println("Within budget?", total <= availableFunds)
```

Ошибки.

```
invalid operation: price * taxRate (mismatched types int and float64)
invalid operation: price + tax (mismatched types int and float64)
invalid operation: total <= availableFunds (mismatched types float64 and int)
```

Обновите код, заполнив пропуски. Исправьте ошибки, чтобы программа выдавала желаемый результат. (Подсказка: перед выполнением математических операций или сравнений необходимо провести преобразования, чтобы обеспечить совместимость типов.)

```
var price int = 100
fmt.Println("Price is", price, "dollars.")

var taxRate float64 = 0.08
var tax float64 = _____________________________
fmt.Println("Tax is", tax, "dollars.")

var total float64 = _____________________
fmt.Println("Total cost is", total, "dollars.")

var availableFunds int = 120
fmt.Println(availableFunds, "dollars available.")
fmt.Println("Within budget?", _____________________________________)
```

Ожидаемый результат.

```
Price is 100 dollars.
Tax is 8 dollars.
Total cost is 108 dollars.
120 dollars available.
Within budget? true
```

Ответ на с. 64.

Установка Go на вашем компьютере

Интерактивная среда Go Playground прекрасно подходит для того, чтобы опробовать новый язык. Тем не менее возможности ее практического применения ограничены. Например, она не может использоваться для работы с файлами. А еще в ней не предусмотрены средства получения пользовательского ввода в терминале, что понадобится нам в следующей программе.

В завершающей части этой главы мы загрузим и установим Go на вашем компьютере. Не беспокойтесь, команда Go постаралась, чтобы это было просто! Во многих операционных системах для этого достаточно запустить программу установки.

1. Откройте страницу *https://golang.org* в браузере.
2. Щелкните на ссылке Download.
3. Выберите установочный пакет для своей операционной системы (ОС). Загрузка должна начаться автоматически.
4. Откройте страницу с инструкциями по установке для вашей ОС (возможно, эта страница откроется автоматически после начала загрузки). Выполните инструкции, приведенные на странице.
5. Откройте новое окно терминала или командной строки.
6. Чтобы убедиться в том, что установка Go прошла успешно, введите команду **`go version`** в приглашении и нажмите клавишу Return (или Enter). На экране должно появиться сообщение с установленной версией Go.

Веб-сайты постоянно изменяются.

Может оказаться, что сайт golang.org или программа установки Go изменились после выхода книги, и инструкции окажутся в чем-то неточными. В таком случае зайдите на сайт

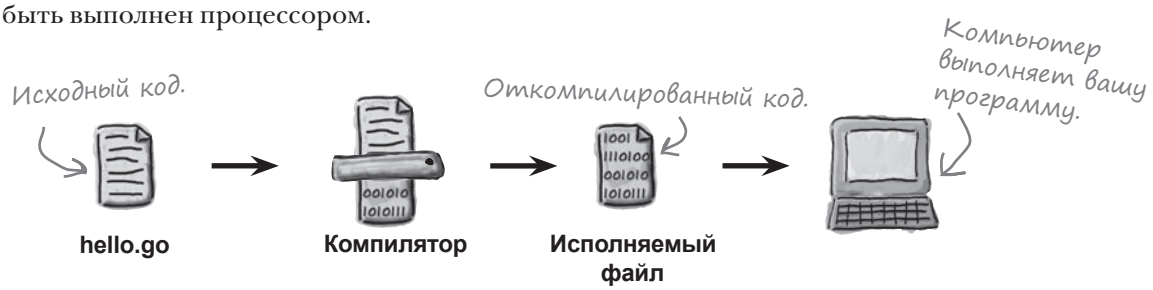
http://headfirstgo.com

за технической поддержкой и советами по решению возможных проблем!

Компиляция кода Go

До настоящего момента ваше взаимодействие с Go Playground было довольно примитивным: вы набирали код, а он каким-то таинственным образом исполнялся. После того как вы установили компилятор Go на своем компьютере, стоит поближе присмотреться к тому, как все работает.

На самом деле компьютер не способен выполнить код Go напрямую. Чтобы это произошло, необходимо взять файл с исходным кодом и **откомпилировать** его: преобразовать в двоичный формат, который может быть выполнен процессором.



А теперь воспользуемся установленной копией Go для компиляции и запуска примера «Hello, Go!», приводившегося ранее.

1. Запустите свой любимый текстовый редактор, введите код программы «Hello, Go!» и сохраните его в текстовом файле с именем *hello.go*.
2. Откройте новое окно терминала или командной строки.
3. В терминале перейдите в каталог, в котором был сохранен файл *hello.go*.
4. Выполните команду **`go fmt hello.go`**, чтобы отформатировать код. (Этот шаг не является строго обязательным, но мы все равно рекомендуем это сделать.)
5. Выполните команду **`go build hello.go`**, чтобы откомпилировать исходный код. В текущем каталоге появляется исполняемый файл. В macOS и Linux исполняемый файл будет называться просто *hello*. В Windows ему будет присвоено имя *hello.exe*.
6. Запустите исполняемый файл. В macOS и Linux для этого следует ввести команду **`./hello`** (что означает «запустить программу с именем `hello` из текущего каталога»). В системе Windows введите команду **`hello.exe`**.

Сохраните этот код в файле.


```
package main

import "fmt"

func main() {
        fmt.Println("Hello, Go!")
}
```

Перейдите в каталог, в котором был сохранен файл hello.go.
Форматирование кода.
Компиляция кода.
Запуск исполняемого файла.

```
Shell  Edit  View  Window  Help
$ cd try_go
$ go fmt hello.go
$ go build hello.go
$ ./hello
Hello, Go!
$
```

Компиляция и запуск программы *hello.go* в macOS и Linux

Перейдите в каталог, в котором был сохранен файл hello.go.
Форматирование кода.
Компиляция кода.
Запуск исполняемого файла.

```
Command Prompt
>cd try_go
>go fmt hello.go
>go build hello.go
>hello.exe
Hello, Go!
>
```

Компиляция и запуск программы *hello.go* в Windows

Инструменты Go

При установке Go в окружение командной строки добавляется исполняемый файл с именем *go*. Исполняемый файл *go* предоставляет в ваше распоряжение различные команды, в числе которых:

Команда	Описание
`go build`	Компилирует файлы с исходным кодом в двоичные файлы
`go run`	Компилирует и запускает программу без сохранения в исполняемом файле
`go fmt`	Переформатирует исходные файлы с использованием стандартного форматирования Go
`go version`	Выводит текущую версию Go

Мы выполнили команду `go fmt`, которая переформатирует код по стандартам Go. Она делает то же, что и кнопка Format на сайте Go Playground. Мы рекомендуем выполнять команду `go fmt` для каждого созданного вами исходного файла.

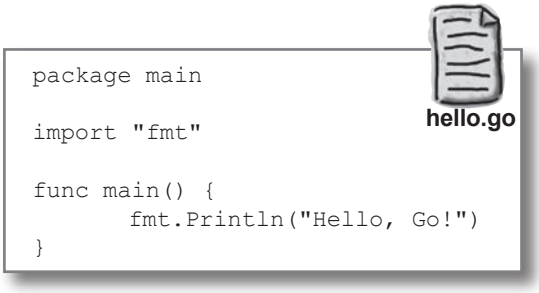
Многие редакторы можно настроить так, чтобы они автоматически выполняли команду go fmt при каждом сохранении файла! См. https://blog.golang.org/go-fmt-your-code.

Также мы использовали команду `go build` для компиляции кода в исполняемый файл. Такие исполняемые файлы могут распространяться среди пользователей, причем пользователи смогут запускать их, даже если на их компьютерах не установлен компилятор Go.

Но мы еще не опробовали команду `go run`. Давайте сделаем это!

Быстрый запуск кода командой «go run»

Команда `go run` компилирует и запускает исходный файл без сохранения исполняемого файла в текущем каталоге. Она прекрасно подходит для быстрой проверки простых программ. Воспользуемся ею для выполнения примера *hello.go*.



1. Откройте новое окно терминала или командной строки.
2. В терминале перейдите в каталог, в котором был сохранен файл *hello.go*.
3. Введите команду **`go run hello.go`** и нажмите Enter/Return. (Эта команда выглядит одинаково во всех операционных системах.)

Перейдите в каталог, в котором был сохранен файл hello.go.

Запуск файла с исходным кодом.

```
Shell Edit View Window Help
$ cd try_go
$ go run hello.go
Hello, Go!
$
```

Выполнение *hello.go* командой `go run` (в любой ОС)

Результаты работы программы немедленно появляются на экране. Если вы внесли изменения в исходный код, выполнять отдельный шаг компиляции не нужно; просто запустите свой код командой `go run` и сразу получите результат. При работе с небольшими программами команда `go run` очень удобна!

Ваш инструментарий Go

Глава 1 подошла к концу. В ней ваш инструментарий пополнился вызовами функций и типами.

Вызовы функций

Функция представляет собой блок кода, который может вызываться в других местах программы.

При вызове функции ей могут передаваться данные в аргументах.

Типы

У всех значений в Go имеется тип, который определяет, для чего могут использоваться эти значения. Математические операции и сравнения с разными типами запрещены, хотя при необходимости значение можно преобразовать к другому типу.

В переменных Go могут храниться значения только того типа, с которым они были объявлены.

КЛЮЧЕВЫЕ МОМЕНТЫ

- **Пакет** представляет собой группу взаимосвязанных функций и других блоков кода.
- Прежде чем использовать функции из пакета в файле Go, необходимо **импортировать** этот пакет.
- Строка — последовательность байтов, обычно представляющих символы текста.
- Руна представляет отдельный символ текста.
- Два самых распространенных числовых типа Go — `int` (для хранения целых чисел) и `float64` (для хранения чисел с плавающей точкой).
- Тип `bool` используется для хранения логических значений (`true` или `false`).
- **Переменная** представляет собой блок памяти для хранения значения заданного типа.
- Если переменной не присвоено значение, то она содержит **нулевое значение** для своего типа. Примеры нулевых значений: `0` для переменных `int` или `float64`, `""` для строковых переменных.
- Объявление переменной можно совместить с присваиванием ей значения при помощи **короткого объявления переменной** `:=`.
- К переменной, функции или типу можно обращаться из кода других пакетов только в том случае, если ее имя начинается с буквы верхнего регистра.
- Команда `go fmt` автоматически переформатирует исходные файлы по стандартам Go. Всегда выполняйте команду `go fmt` для любого кода, который вы собираетесь передавать другим разработчикам.
- Команда `go build` **компилирует исходный код** Go в двоичный формат, который может выполняться компьютером.
- Команда `go run` компилирует и выполняет программу без сохранения в исполняемом файле в текущем каталоге.

У бассейна. Решение

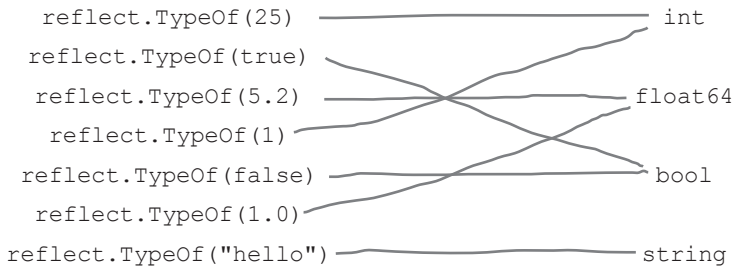
```
package main

import (
        "fmt"
)

func main() {
        fmt.Println("Cannonball!!!!")
}
```

Результат.

```
Cannonball!!!!
```

Упражнение
Решение

Соедините каждый фрагмент кода с правильным типом.
Некоторым типам могут соответствовать несколько фрагментов.



Развлечения с магнитами. Решение

```
package main

import (
        "fmt"
)

func main() {

        var originalCount int = 10
        fmt.Println("I started with", originalCount, "apples.")
        var eatenCount int = 4
        fmt.Println("Some jerk ate", eatenCount, "apples.")
        fmt.Println("There are", originalcount-eatenCount, "apples left.")
}
```

Результат.

```
I started with 10 apples.
Some jerk ate 4 apples.
There are 6 apples left.
```

Упражнение
Решение

Обновите код, заполнив пропуски в следующем коде. Исправьте ошибки, чтобы программа выдавала желаемый результат. (Подсказка: перед выполнением математических операций или сравнений необходимо провести преобразование, чтобы обеспечить совместимость типов.)

```
var price int = 100
fmt.Println("Price is", price, "dollars.")

var taxRate float64 = 0.08
var tax float64 = float64(price) * taxRate
fmt.Println("Tax is", tax, "dollars.")

var total float64 = float64(price) + tax
fmt.Println("Total cost is", total, "dollars.")

var availableFunds int = 120
fmt.Println(availableFunds, "dollars available.")
fmt.Println("Within budget?", total <= float64(availableFunds))
```

Ожидаемый результат.

```
Price is 100 dollars.
Tax is 8 dollars.
Total cost is 108 dollars.
120 dollars available.
Within budget? true
```

Условные команды и циклы

В каждой программе есть части, которые должны выполняться только в определенных ситуациях. «Этот код должен выполняться, *если* произошла ошибка. А если нет — должен выполняться этот код». Почти в каждой программе присутствует код, который должен выполняться только в том случае, если некоторое *условие* истинно. Почти в каждом языке программирования существуют **условные команды,** которые позволяют определить, нужно ли выполнять те или иные сегменты кода. Язык Go не исключение. Возможно, какие-то части кода должны выполняться *многократно*. Как и многие языки, Go поддерживает **циклы** для многократного выполнения блоков кода. В этой главе вы научитесь применять как условные команды, так и циклы!

Вызов методов

В языке Go можно определять **методы**: функции, связанные со значениями определенного типа. Методы Go похожи на связываемые с «объектами» методы других языков программирования, но в Go все проще.

О том, как работают методы, будет подробно рассказано в главе 9. Тем не менее для материала этой главы нам потребуется пара методов, поэтому сейчас рассмотрим несколько коротких примеров вызова методов.

В пакете `time` определен тип `Time`, представляющий дату (год, месяц и день) и время (час, минуты, секунды и т. д.). Каждое значение `time.Time` содержит метод `Year`, который возвращает год. Приведенный ниже код использует этот метод для вывода текущего года:

```
package main

import (
       "fmt"
       "time"
)

func main() {
       var now time.Time = time.Now()
       var year int = now.Year()
       fmt.Println(year)
}
```

Необходимо импортировать пакет «time», чтобы использовать тип time.Time.

Метод time.Now возвращает значение time.Time, представляющее текущую дату и время.

У значений time.Time имеется метод Year, который возвращает текущий год.

```
2019
```

(Или год, выставленный на системных часах вашего компьютера.)

Функция `time.Now` возвращает новое значение `Time` для текущей даты и времени; это значение сохраняется в переменной `now`. Затем мы вызываем метод `Year` для значения, ссылка на которое хранится в `now`:

Содержит значение time.Time

Вызывает метод Year для значения time.Time.

```
now.Year()
```

Метод `Year` возвращает целое значение года, которое выводится программой.

Методы — это функции, связанные со значениями конкретного типа.

Вызов методов (продолжение)

Пакет `strings` содержит тип `Replacer`, который ищет подстроку в строке и заменяет каждое вхождение этой подстроки в другой строке. Следующий код заменяет каждый символ # в строке буквой o:

```
package main

import (
        "fmt"
        "strings"
)

func main() {
        broken := "G# r#cks!"
        replacer := strings.NewReplacer("#", "o")
        fixed := replacer.Replace(broken)
        fmt.Println(fixed)
}
```

```
Go rocks!
```

Импортируем пакеты, использованные в функции «main».

Возвращает значение strings.Replacer, настроенное для замены всех «#» в строке на «o».

Вызываем метод Replace для strings.Replacer и передаем ему строку, в которой должны осуществляться замены.

Вывод строки, возвращенной методом Replace.

Функция `strings.NewReplacer` получает аргументы — заменяемую строку (`"#"`) и заменяющую строку (`"o"`) — и возвращает значение `strings.Replacer`. Когда мы передаем строку методу `Replace` значения `Replacer`, то метод возвращает строку, в которой выполнена указанная замена.

Синтаксис вызова методов очень похож на синтаксис вызова функций других пакетов. Они как-то связаны между собой?

Точка означает, что имя в правой части принадлежит тому, что находится в левой части.

Если функции, упоминавшиеся ранее, принадлежали *пакету*, методы принадлежат конкретному *значению*. Это значение указывается слева от точки.

Значение. *Имя метода.*

```
now.Year()
replacer.Replace(broken)
```

Значение. *Имя метода.*

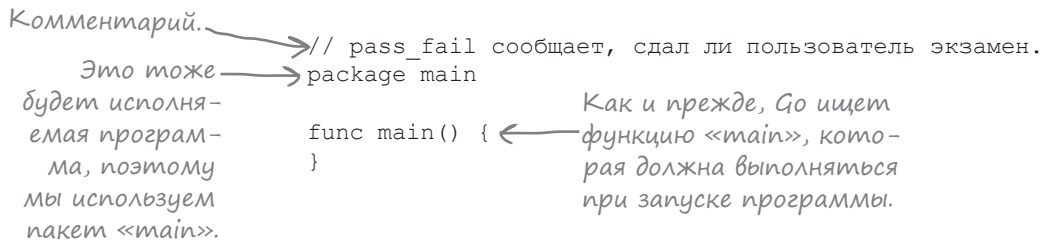
Проверка результата

В этой главе рассматриваются средства Go для принятия решений о том, нужно ли выполнять блок кода или нет, в зависимости от некоторого условия. Рассмотрим ситуацию, в которой может понадобиться такая возможность...

Необходимо написать программу, в которой студент вводит свой процент правильных ответов и узнает, прошел он экзамен или нет. Результат определяется простой формулой: при значении 60% и выше экзамен успешно сдан, а при значении ниже 60% — провален. Итак, наша программа должна выдавать один ответ, если введенное пользователем значение равно 60 и выше, и другой ответ в противном случае.

Комментарии

Давайте создадим новый файл *pass_fail.go* для хранения программы. Мы позаботимся об одной подробности, которая была упущена в предыдущей программе, и добавим в начало описание того, что делает программа.



В исходный код многих программ Go включается описание того, что они делают. Эти описания предназначены для людей, которые будут заниматься сопровождением кода. Компилятор **комментарии** игнорирует.

Самая распространенная форма комментариев обозначается двумя слешами (//). Все символы от // до конца строки рассматриваются как часть комментария. Комментарий // может занимать всю строку или следовать после кода.

```
// Общее количество виджетов в системе.
var TotalCount int // Должно быть целым числом.
```

Более редкая форма комментариев занимает несколько строк. **Блочные комментарии** начинаются с /* и заканчиваются */, а весь текст между этими маркерами (включая символы новой строки) является частью комментария.

```
/*
Пакет widget включает все функции,
используемые для работы с виджетами.
*/
```

Получение значения от пользователя

Теперь добавим в программу *pass_fail.go* полезный код. Прежде всего программа должна получить от пользователя процент. Мы хотим, чтобы пользователь ввел число и нажал Enter, а введенное число было сохранено в переменной. Добавим код для решения этой задачи. *(Внимание: этот код не будет компилироваться в том виде, в котором он здесь приведен; вскоре мы объясним почему!)*

```
// pass_fail сообщает, сдал ли пользователь экзамен.
package main

import (
        "bufio"
        "fmt"
        "os"
)

func main() {
        fmt.Print("Enter a grade: ")
        reader := bufio.NewReader(os.Stdin)
        input := reader.ReadString('\n')
        fmt.Println(input)
}
```

Импортируем пакеты, используемые в функции «main».

Запросить у пользователя значение.

Создать «буферизованное средство чтения» для получения текста с клавиатуры.

Возвращает текст, введенный пользователем до нажатия клавиши Enter.

Вывод введенных данных.

Сначала нужно запросить данные у пользователя, и для вывода приглашения используется функция `fmt.Print`. (В отличие от `Println`, функция `Print` не переходит на новую строку в терминале после вывода сообщения; таким образом, запрос и введенные данные размещаются в одной строке.)

Затем нам понадобится механизм чтения (получения и хранения) ввода из *стандартного ввода* программы, в который поступает весь ввод с клавиатуры. Строка `reader := bufio.NewReader(os.Stdin)` сохраняет `bufio.Reader` в переменной `reader`, которая сделает это за вас.

Возвращает новое значение bufio.Reader.

```
reader := bufio.NewReader(os.Stdin)
```

Reader читает данные из стандартного ввода (с клавиатуры).

Чтобы получить введенные данные от пользователя, мы вызываем метод `ReadString` для `Reader`. Методу `ReadString` требуется аргумент с руной (символом), отмечающей конец ввода. Мы хотим прочитать весь текст, введенный пользователем до нажатия Enter, поэтому `ReadString` передается руна новой строки.

Возвращает данные, введенные пользователем, в виде строки.

```
input := reader.ReadString('\n')
```

Будет прочитан весь текст до руны новой строки.

После того как от пользователя будут получены данные, мы просто выводим их.

В теории. Но при попытке откомпилировать или запустить эту программу происходит ошибка:

Ошибка.

```
multiple-value
reader.ReadString()
in single-value context
```

Не пытайтесь разобраться во всех тонкостях работы *bufio.Reader*.

Пока достаточно знать, что это средство позволяет читать текст, введенный с клавиатуры.

Множественные возвращаемые значения функций или методов

Мы пытаемся прочитать ввод с клавиатуры, но получаем сообщение об ошибке. Компилятор сообщает о проблеме в следующей строке кода:

```
input := reader.ReadString('\n')
```

Ошибка. → `multiple-value reader.ReadString() in single-value context`

Проблема в том, что метод `ReadString` пытается вернуть два значения, а мы предоставили только *одну* переменную для присваивания.

В большинстве языков программирования функции и методы могут возвращать только одно значение, но в Go функция может возвращать сколько угодно значений. Чаще всего множественные возвращаемые значения в Go используются для возвращения дополнительного значения ошибки, по которому можно определить, не возникли ли проблемы во время выполнения функции или метода. Несколько примеров:

```
bool, err := strconv.ParseBool("true")
file, err := os.Open("myfile.txt")
response, err := http.Get("http://golang.org")
```

Возвращает ошибку, если строку не удается преобразовать в логическое значение.

Возвращает ошибку, если файл не удалось открыть.

Возвращает ошибку, если страницу не удалось загрузить.

И в чем проблема? Просто добавьте переменную для хранения ошибки и не используйте ее!

Go требует, чтобы каждая *объявленная* переменная также *использовалась* где-то в программе. Если вы добавите переменную `err` и не проверите ее, программа не будет компилироваться. Неиспользуемые переменные часто указывают на ошибки в программе; это один из примеров того, как Go помогает вам выявлять и исправлять ошибки!

Go не позволит объявить переменную, если она не используется в программе.

```
// pass_fail сообщает, сдал ли...
package main

import (
        "bufio"
        "fmt"
        "os"
)

func main() {
        fmt.Print("Enter a grade: ")
        reader := bufio.NewReader(os.Stdin)
        input, err := reader.ReadString('\n')
        fmt.Println(input)
}
```

Если просто добавить переменную и не использовать ее...

...вы получите сообщение об ошибке!

Ошибка. → `err declared and not used`

Вариант 1. Игнорировать возвращаемое значение ошибки

Метод `ReadString` возвращает второе значение вместе с введенным текстом, и со вторым значением нужно что-то сделать. Мы только что попробовали добавить второе значение и игнорировать его, но код все равно не компилируется.

```
input, err := reader.ReadString('\n')
```

Ошибка. → `err declared and not used`

Если имеется значение, которое должно быть присвоено переменной, но вы не собираетесь его использовать, можно воспользоваться **пустым идентификатором** Go. Присваивание значения пустому идентификатору фактически приводит к тому, что оно теряется (а другим читателям вашего кода становится очевидно, что вы поступаете так намеренно). Чтобы использовать пустой идентификатор, просто введите символ подчеркивания (_) в команде присваивания, где должно использоваться имя переменной.

Попробуем использовать пустой идентификатор вместо обычной переменной `err`:

```
// pass_fail сообщает, сдал ли пользователь экзамен.
package main

import (
        "bufio"
        "fmt"
        "os"
)

func main() {
        fmt.Print("Enter a grade: ")
        reader := bufio.NewReader(os.Stdin)
        input, _ := reader.ReadString('\n')
        fmt.Println(input)
}
```

Пустой идентификатор используется как «заполнитель» для кода ошибки.

Теперь опробуем изменение на практике. В терминале перейдите в каталог, где был сохранен файл *pass_fail.go*, и запустите программу следующей командой:

```
go run pass_fail.go
```

Запускаем pass_fail.go.

Введите число и нажмите Enter.

Ваше число будет выведено в ответном сообщении.

```
Shell Edit View Window Help
$ go run pass_fail.go
Enter a grade: 100
100

$
```

Если ввести значение (или любую другую строку) в приглашении и нажать Enter, оно будет повторено в следующей строке. Программа работает!

Вариант 2. Обработка ошибки

Верно. Если в ходе выполнения возникнет ошибка, программа не скажет вам о ней!

Если метод `ReadString` вернет значение ошибки, с пустым идентификатором ошибка будет проигнорирована, а программа продолжит выполнение, возможно, с недействительными данными.

```
func main() {
        fmt.Print("Enter a grade: ")
        reader := bufio.NewReader(os.Stdin)
        input, _ := reader.ReadString('\n')
        fmt.Println(input)
}
```

Возвращаемый признак ошибки игнорируется!

Выводит значение, которое может быть недействительным!

В данном случае при возникновении ошибки правильнее было бы предупредить пользователя и прервать выполнение программы.

Пакет `log` содержит функцию `Fatal`, которая выполняет обе операции одновременно: вывод сообщения в терминале *и* остановку программы. (В этом контексте «фатальной» называется ошибка, «смертельная» для вашей программы.)

Давайте избавимся от пустого идентификатора и заменим его переменной `err`, чтобы ошибка снова сохранялась в программе. Затем воспользуемся функцией `Fatal` для вывода сообщения об ошибке и прерывания работы программы.

```
// pass_fail сообщает, сдал ли пользователь экзамен.
package main

import (
        "bufio"
        "fmt"
        "log"
        "os"
)

func main() {
        fmt.Print("Enter a grade: ")
        reader := bufio.NewReader(os.Stdin)
        input, err := reader.ReadString('\n')
        log.Fatal(err)
        fmt.Println(input)
}
```

Добавляем пакет «log».

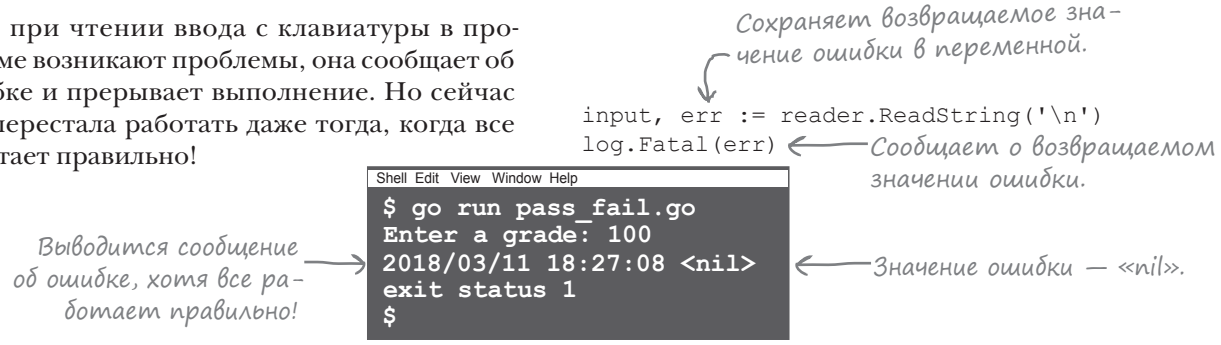
Возвращенный код ошибки снова сохраняется в переменной.

Сообщить об ошибке и остановить программу.

Но при попытке запустить обновленную программу обнаруживается новая проблема...

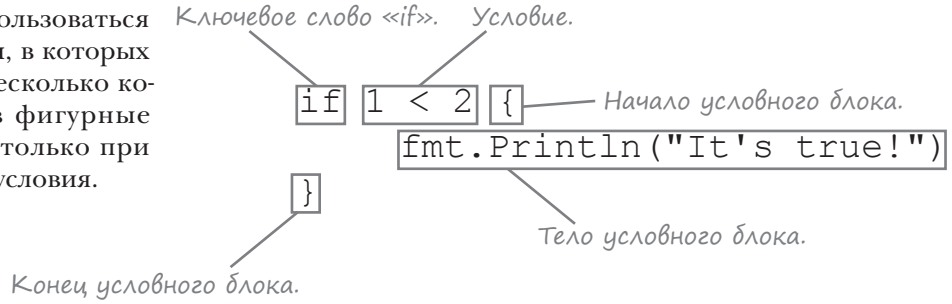
Условные команды

Если при чтении ввода с клавиатуры в программе возникают проблемы, она сообщает об ошибке и прерывает выполнение. Но сейчас она перестала работать даже тогда, когда все работает правильно!



Такие функции и методы, как `ReadString`, возвращают значение ошибки **nil**, что по сути означает «здесь ничего нет». Другими словами, если переменная `err` равна `nil`, значит, ошибки не было. Но наша программа написана так, что она просто сообщает об ошибке `nil`! Что же *нужно* сделать, чтобы программа завершалась только *в том случае*, если значение переменной не равно `nil`?

Для этого можно воспользоваться **условными командами**, в которых блок кода (одна или несколько команд, заключенных в фигурные скобки) выполняется только при истинности заданного условия.



Выражение вычисляется, и если полученный результат равен `true`, то выполняется код в теле условного блока. Если же результат равен `false`, условный блок пропускается.

```
if true {
        fmt.Println("I'll be printed!")
}
```

```
if false {
        fmt.Println("I won't!")
}
```

Как и многие другие языки, Go поддерживает множественное ветвление в условных командах. Такие команды записываются в форме `if...else if...else`.

```
if grade == 100 {
        fmt.Println("Perfect!")
} else if grade >= 60 {
        fmt.Println("You pass.")
} else {
        fmt.Println("You fail!")
}
```

Условные команды (продолжение)

Условные команды используют логическое выражение (результат которого равен `true` или `false`), чтобы определить, должен ли выполняться содержащийся в них код.

```
if 1 == 1 {
       fmt.Println("I'll be printed!")
}

if 1 > 2 {
       fmt.Println("I won't!")
}

if 1 < 2 {
       fmt.Println("I'll be printed!")
}

if 1 >= 2 {
       fmt.Println("I won't!")
}

if 2 <= 2 {
       fmt.Println("I'll be printed!")
}

if 2 != 2 {
       fmt.Println("I won't!")
}
```

Если код должен выполняться только в том случае, когда условие дает результат *false*, используйте `!` — оператор логического отрицания. Этот оператор берет значение `true` и превращает его в `false` или же берет значение `false` и превращает его в `true`.

```
if !true {
       fmt.Println("I won't be printed!")
}

if !false {
       fmt.Println("I will!")
}
```

Если код должен выполняться только в том случае, когда истинны *оба* условия, используйте оператор `&&` («и»). А если он должен выполняться лишь тогда, когда истинно *хотя бы одно* из двух условий, используйте оператор `||` («или»).

```
if true && true {
       fmt.Println("I'll be printed!")
}

if true && false {
       fmt.Println("I won't!")
}

if false || true {
       fmt.Println("I'll be printed!")
}

if false || false {
       fmt.Println("I won't!")
}
```

Часто Задаваемые Вопросы

В: Другой язык программирования требует, чтобы условие команды `if` заключалось в круглые скобки. В Go это не обязательно?

О: Нет. Более того, команда `go fmt` удалит все круглые скобки, добавленные вами, если только они не используются для определения порядка операций.

Упражнение

Эти вызовы `Println` заключены в условные блоки, лишь часть из них будет выполнена. Напишите, какой результат выведет программа.

(Мы вписали первые два результата за вас.)

```
if true {
       fmt.Println("true")
}
if false {
       fmt.Println("false")
}
if !false {
       fmt.Println("!false")
}
if true {
       fmt.Println("if true")
} else {
       fmt.Println("else")
}
if false {
       fmt.Println("if false")
} else if true {
       fmt.Println("else if true")
}
if 12 == 12 {
       fmt.Println("12 == 12")
}
if 12 != 12 {
       fmt.Println("12 != 12")
}
if 12 > 12 {
       fmt.Println("12 > 12")
}
if 12 >= 12 {
       fmt.Println("12 >= 12")
}
if 12 == 12 && 5.9 == 5.9 {
       fmt.Println("12 == 12 && 5.9 == 5.9")
}
if 12 == 12 && 5.9 == 6.4 {
       fmt.Println("12 == 12 && 5.9 == 6.4")
}
if 12 == 12 || 5.9 == 6.4 {
       fmt.Println("12 == 12 || 5.9 == 6.4")
}
```

Результат:

true

!false

Ответ на с. 109.

Условная выдача фатальной ошибки

Наша программа сообщает об ошибке и аварийно завершается, хотя данные с клавиатуры были успешно прочитаны.

```
input, err := reader.ReadString('\n')
log.Fatal(err)
```

Сохраняет возвращаемое значение ошибки в переменной.

Сообщает о возвращаемом значении ошибки.

```
Shell Edit View Window Help
$ go run pass_fail.go
Enter a grade: 100
2018/03/11 18:27:08 <nil>
exit status 1
$
```

Ошибка выдается даже в том случае, если все работает правильно!

Значение ошибки равно «nil».

Мы знаем, что если значение в переменной `err` равно `nil`, это говорит о том, что данные с клавиатуры были прочитаны успешно. Теперь, познакомившись с командами `if`, попробуем обновить код, чтобы сообщение об ошибке и завершение программы происходило только в том случае, если значение `err` *не равно* `nil`.

```
// pass_fail сообщает, сдал ли пользователь экзамен.
package main

import (
        "bufio"
        "fmt"
        "log"
        "os"
)

func main() {
        fmt.Print("Enter a grade: ")
        reader := bufio.NewReader(os.Stdin)
        input, err := reader.ReadString('\n')
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(input)
}
```

Если «ошибка» не равна nil...

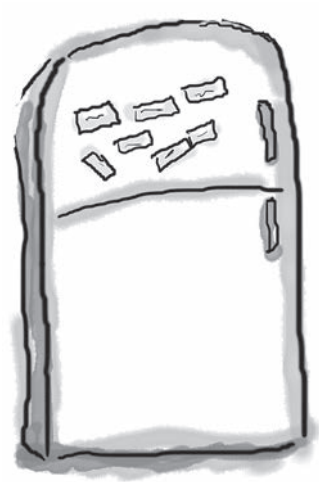
Сообщить об ошибке и прервать выполнение программы.

Запустите программу: вы увидите, что в новой версии она успешно работает. А если при чтении введенных данных возникнут ошибки, вы увидите и их!

Запускаем pass_fail.go.

Программа выводит введенное число.

```
Shell Edit View Window Help
$ go run pass_fail.go
Enter a grade: 100
100

$
```



Развлечения с магнитами

На холодильнике была выложена программа Go, которая выводит размер файла. Для этого программа вызывает функцию `os.Stat`, которая возвращает значение `os.FileInfo` и, возможно, значение ошибки. Затем она вызывает метод `Size` для значения `FileInfo`, чтобы получить размер файла.

Однако исходная программа использует пустой идентификатор `_`, чтобы проигнорировать значение ошибки из `os.Stat`. Если в программе произойдет ошибка (например, если файл не существует), программа будет работать некорректно.

Используйте дополнительные фрагменты кода и создайте программу, которая в основном работает так же, как исходная, но еще проверяет значение ошибки, полученной от `os.Stat`. Если ошибка от `os.Stat` не равна `nil`, программа должна сообщить об ошибке и аварийно завершиться. Снимите магнит с пустым идентификатором `_`; он не будет использоваться в программе.

```
package main
```

Эта программа работает! Но она игнорирует все ошибки, которые могут возникнуть при выполнении....

```
import (
        "fmt"
        "log"
        "os"
)
```

Пустой идентификатор игнорирует все значения ошибок. Снимите этот магнит и замените его одним из изображенных внизу!

Получает значение FileInfo с данными, относящимися к файлу my.txt.

```
func main() {
```

Содержит размер файла, дату его изменения и т. д.

```
fileInfo,   _   :=   os.Stat("my.txt")
```

Добавьте сюда свой код. Если код ошибки не равен nil, передайте его log.Fatal.

```
fmt.Println(fileInfo.Size())
}
```

Возвращает размер файла.

Это дополнительные магниты. Добавьте их в программу!

```
{   !=   }   nil   err   err   if   log.Fatal(err)
```

Ответ на с. 110.

Избегайте замещения имен

Меня еще кое-что беспокоит. Ранее вы говорили, что стараетесь избегать сокращений в этой книге. А здесь переменной присваивается имя `err` вместо `error`!

```
fmt.Print("Enter a grade: ")
reader := bufio.NewReader(os.Stdin)
input, err := reader.ReadString('\n')
if err != nil {
       log.Fatal(err)
}
```

Называть переменную `error` не рекомендуется, потому что это приведет к замещению типа с именем `error`.

Объявляя переменную, проследите за тем, чтобы ее имя не совпадало с именами существующих функций, пакетов, типов или других переменных. Если такое имя уже существует во внешней области видимости (вскоре мы поговорим об областях видимости), ваша переменная **заместит** его, то есть будет перехватывать все обращения к нему. И часто это нежелательно.

В следующем фрагменте объявляется переменная с именем `int`, которая замещает имя типа, переменная с именем `append`, которая замещает имя встроенной функции (функция `append` будет представлена в главе 6), а также переменная с именем `fmt`, которая замещает имя импортированного пакета. Эти имена создают путаницу, но сами по себе не порождают ошибок...

```
package main

import "fmt"

func main() {
       var int int = 12
       var append string = "minutes of bonus footage"
       var fmt string = "DVD"
}
```

Переменная с именем «int» замещает имя встроенного типа «int»!

Переменная с именем «append» замещает имя встроенной функции «append»!

Переменная с именем «fmt» замещает имя импортированного пакета «fmt»!

Избегайте замещения имен (продолжение)

...Но если вы попытаетесь обратиться к типу, функции или пакету, имя которых замещается переменной, то вы получите значение из переменной. В следующем примере это приводит к ошибке компиляции:

```
func main() {
        var int int = 12
        var append string = "minutes of bonus footage"
        var fmt string = "DVD"
        var count int
        var languages = append([]string{}, "Español")
        fmt.Println(int, append, "on", fmt, languages)
}
```

«int» теперь относится к переменной, объявленной выше, а не к числовому типу!

Имя «append» теперь обозначает переменную, а не функцию!

Имя «fmt» теперь обозначает переменную, а не пакет!

Ошибки компиляции

```
imported and not used: "fmt"
int is not a type
cannot call non-function append (type string), declared at prog.go:7:6
fmt.Println undefined (type string has no field or method Println)
```

Чтобы не путаться самому и не путать коллег, старайтесь по возможности избегать замещения имен. В данном случае проблема решается простым выбором неконфликтующих имен для переменных:

```
func main() {
        var count int = 12
        var suffix string = "minutes of bonus footage"
        var format string = "DVD"
        var languages = append([]string{}, "Español")
        fmt.Println(count, suffix, "on", format, languages)
}
```

Переименуем переменную «int».

Переименуем переменную «append».

Переименуем переменную «fmt».

```
12 minutes of bonus footage on DVD [Español]
```

Как было показано в главе 3, в Go существует встроенный тип с именем `error`. Вот почему при объявлении переменных, предназначенных для хранения ошибок, мы присваивали им имя `err` вместо `error` — мы хотели предотвратить замещение имени типа ошибки именем переменной.

«err», а не «error»!

```
fmt.Print("Enter a grade: ")
reader := bufio.NewReader(os.Stdin)
input, err := reader.ReadString('\n')
if err != nil {
        log.Fatal(err)
}
```

Если вы все же *назовете* свою переменную `error`, *возможно*, ваш код будет работать. По крайней мере *до того момента*, как вы забудете, что имя типа ошибки было замещено, попробуете воспользоваться типом и получите вместо него переменную. Лучше не рисковать; используйте имя `err` для своих переменных со значениями ошибок!

Преобразование строк в числа

Условные команды также могут быть использованы для проверки введенного значения. Давайте воспользуемся командой `if/else` для определения того, прошел ли пользователь экзамен или нет. Если введенный процент правильных ответов равен 60 и выше, переменной `status` присваивается строка `"passing"`. В противном случае переменной будет присвоена строка `"failing"`.

```
// Директивы package и import пропущены
func main() {
        fmt.Print("Enter a grade: ")
        reader := bufio.NewReader(os.Stdin)
        input, err := reader.ReadString('\n')
        if err != nil {
                log.Fatal(err)
        }

        if input >= 60 {
                status := "passing"
        } else {
                status := "failing"
        }
}
```

Однако в текущем варианте этой программы выдается ошибка компиляции.

Ошибка. →
```
cannot convert 60 to type string
invalid operation: input >= 60 (mismatched types string and int)
```

Дело в том, что ввод с клавиатуры читается как строка. Go может сравнивать числа только с другими числами; сравнить число со строкой не удастся. А прямого преобразования типа из строки в число не существует:

```
float64("2.6")
```

Ошибка. →
```
cannot convert "2.6" (type string) to type float64
```

Мы должны решить две задачи:

- В конце строки `input` находится символ новой строки, появившийся в результате нажатия клавиши Enter в процессе ввода. Его необходимо удалить.
- Остальные символы строки необходимо преобразовать в число с плавающей точкой.

Преобразование строк в числа (продолжение)

Удалить символ новой строки из конца входного текста несложно. Пакет `strings` содержит функцию `TrimSpace`, которая удаляет все символы-пропуски (символы новой строки, табуляции и обычные пробелы) в начале и в конце строки.

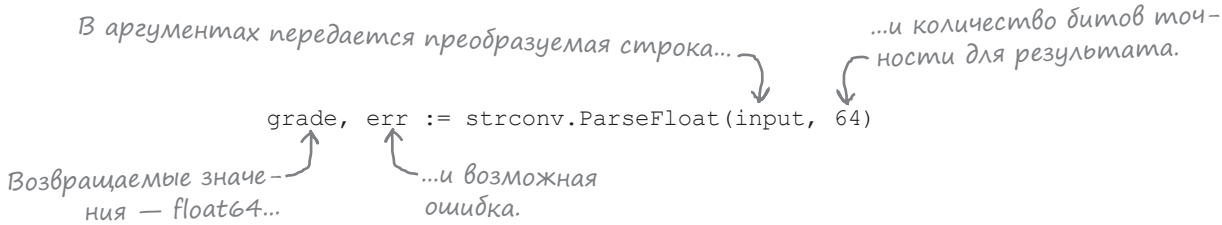
```
s := "\t formerly surrounded by space \n"
fmt.Println(strings.TrimSpace(s))
```

```
formerly surrounded by space
```

Таким образом, чтобы убрать символ новой строки из входной строки, следует передать ее `TrimSpace`, а затем снова присвоить возвращенное значение переменной `input`.

```
input = strings.TrimSpace(input)
```

После этого в строке `input` должно остаться только число, введенное пользователем. Мы воспользуемся функцией `ParseFloat` из пакета `strconv`, чтобы преобразовать его в значение `float64`.



Функции `ParseFloat` передается строка, которую необходимо преобразовать в число, а также количество битов точности для результата. Поскольку строка преобразуется в значение `float64`, мы передаем число `64`. (Кроме `float64`, в Go также поддерживается менее точный тип `float32`, однако им лучше не пользоваться, если только у вас нет на это веских причин.)

Функция `ParseFloat` преобразует число в строку и возвращает его в форме `float64`. Как и `ReadString`, она также имеет второе возвращаемое значение — значение ошибки. Оно должно быть равно `nil`, если только в ходе преобразования строки не возникли какие-то проблемы. (Например, переданная строка *не может* быть преобразована в число. Да и какой может быть числовой эквивалент у строки `"hello"`...)

Все эти «биты точности» сейчас не так уж важны.

По сути это просто размер компьютерной памяти, используемой для хранения числа с плавающей точкой. Если вы уверены в том, что вам нужно число `float64`, всегда передавайте 64 во втором аргументе `ParseFloat`, и все будет хорошо.

Преобразование строк в числа (продолжение)

Дополним программу *pass_fail.go* вызовами `TrimSpace` и `ParseFloat`:

```
// pass_fail сообщает, сдал ли пользователь экзамен.
package main

import (
        "bufio"
        "fmt"
        "log"
        "os"
        "strconv"
        "strings"
)

func main() {
        fmt.Print("Enter a grade: ")
        reader := bufio.NewReader(os.Stdin)
        input, err := reader.ReadString('\n')
        if err != nil {
                log.Fatal(err)
        }

        input = strings.TrimSpace(input)
        grade, err := strconv.ParseFloat(input, 64)
        if err != nil {
                log.Fatal(err)
        }

        if grade >= 60 {
                status := "passing"
        } else {
                status := "failing"
        }
}
```

Добавляем «strconv», чтобы использовать в программе ParseFloat.

Добавляем «strings», чтобы использовать в программе функцию TrimSpace.

Удаляем символ новой строки из входной строки.

Преобразовать строку в значение float64.

Как и в случае с ReadString, сообщить о любых ошибках при преобразовании.

Сравниваем с float64 в переменной «grade», а не со строкой в переменной «input».

Сначала нужные пакеты включаются в директиву `import`. Мы добавляем код удаления символа новой строки из входного текста, после чего передаем ввод функции `ParseFloat` и сохраняем полученное значение `float64` в новой переменной `grade`.

Как и в случае с `ReadString`, мы проверяем, возвращает ли `ParseFloat` значение ошибки. В этом случае программа сообщает об ошибке и аварийно завершается.

Наконец, мы обновляем условную команду, чтобы она проверяла число в `grade`, а не строку в `input`. На этом все ошибки, происходящие от сравнения строки с числом, должны быть исправлены.

Ошибки.

```
status declared
and not used
status declared
and not used
```

При запуске обновленной программы мы уже не получаем сообщение о несовпадении типов `string` и `int`. На первый взгляд проблема решена, однако в программе осталась еще пара ошибок. Сейчас мы займемся их устранением.

Блоки

Мы преобразовали значение, введенное пользователем, в значение `float64`, и включили его в условную команду, чтобы узнать результат экзамена. Тем не менее мы снова получаем пару ошибок компиляции.

```
if grade >= 60 {
      status := "passing"
} else {
      status := "failing"
}
```

Ошибки.

```
status declared
and not used
status declared
and not used
```

Как было показано ранее, объявление переменной — такой, как `status`, — без ее последующего использования в программе Go считается ошибкой. Немного странно, что ошибка повторяется дважды, но пока не будем обращать на это внимания. Добавим вызов `Println` для вывода введенного значения и значения `status`.

```
func main() {
      // Часть кода пропущена...
      if grade >= 60 {
            status := "passing"
      } else {
            status := "failing"
      }
      fmt.Println("A grade of", grade, "is", status)
}
```

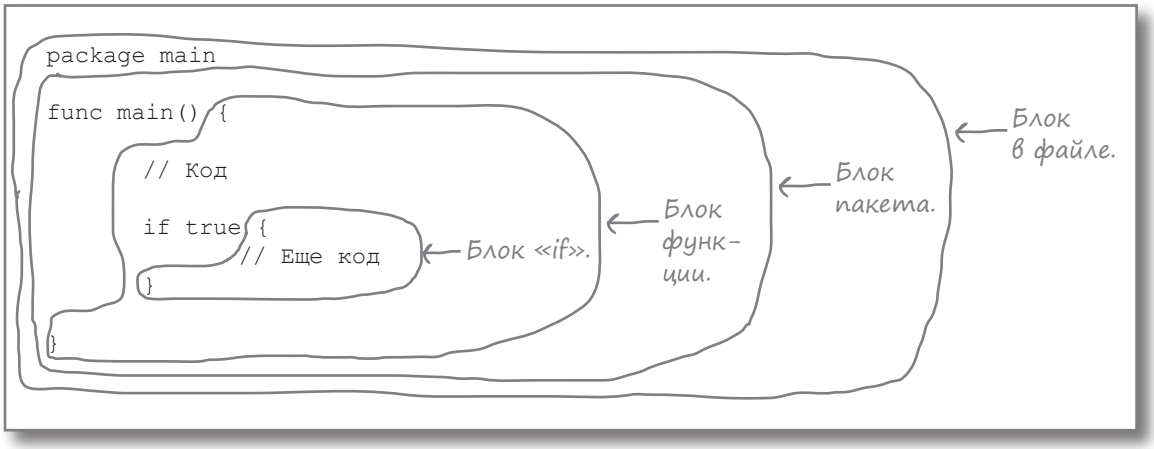
Выводим переменную status.

Ошибка.

```
undefined: status
```

Но теперь появляется *новая* ошибка — в сообщении говорится, что переменная `status` не определена при попытке использовать ее в команде `Println`! Что происходит?

Код Go делится на **блоки** (сегменты). Блоки обычно заключаются в фигурные скобки (`{}`) и могут существовать как на уровне файлов с исходным кодом, так и на уровне пакетов. Блоки могут вкладываться друг в друга.



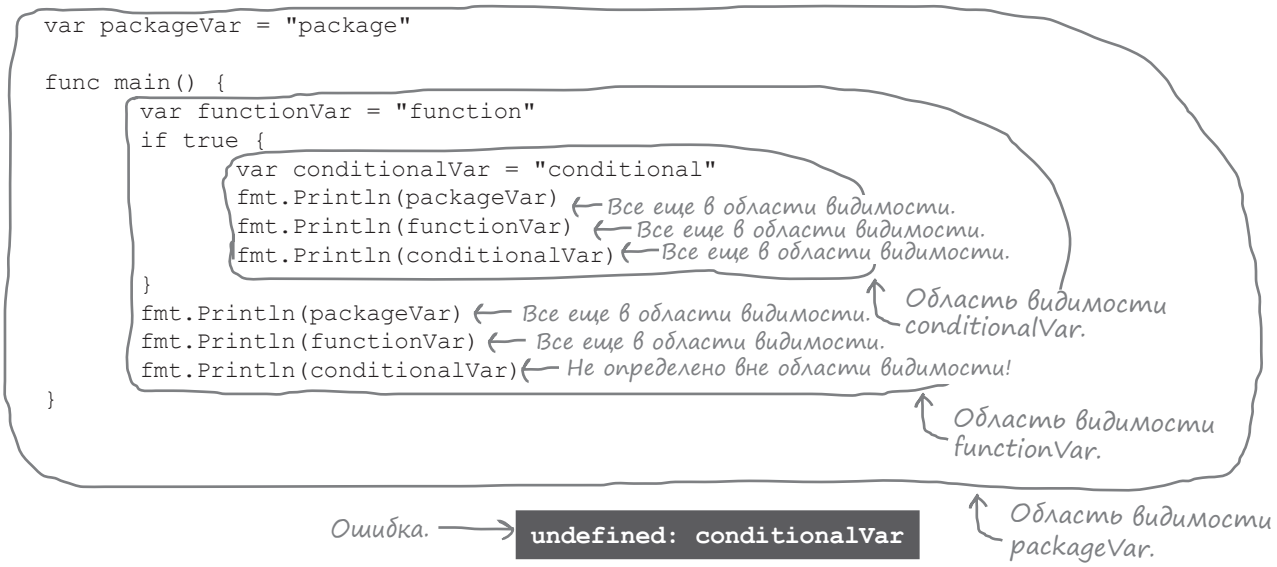
Тела функций и условных команд тоже являются блоками. Понимание этого — ключ к решению нашей проблемы с переменной `status`...

Блоки и область видимости переменной

Каждая объявленная переменная обладает **областью видимости**: частью кода, в которой она «видна». К объявленной переменной можно обращаться в любой точке ее области видимости, однако при попытке обратиться к ней за пределами этой области видимости вы получите сообщение об ошибке.

Область видимости переменной состоит из блока, в котором она была объявлена, и всех блоков, вложенных в этот блок.

```
package main

import "fmt"

var packageVar = "package"

func main() {
        var functionVar = "function"
        if true {
                var conditionalVar = "conditional"
                fmt.Println(packageVar)
                fmt.Println(functionVar)
                fmt.Println(conditionalVar)
        }
        fmt.Println(packageVar)
        fmt.Println(functionVar)
        fmt.Println(conditionalVar)
}
```

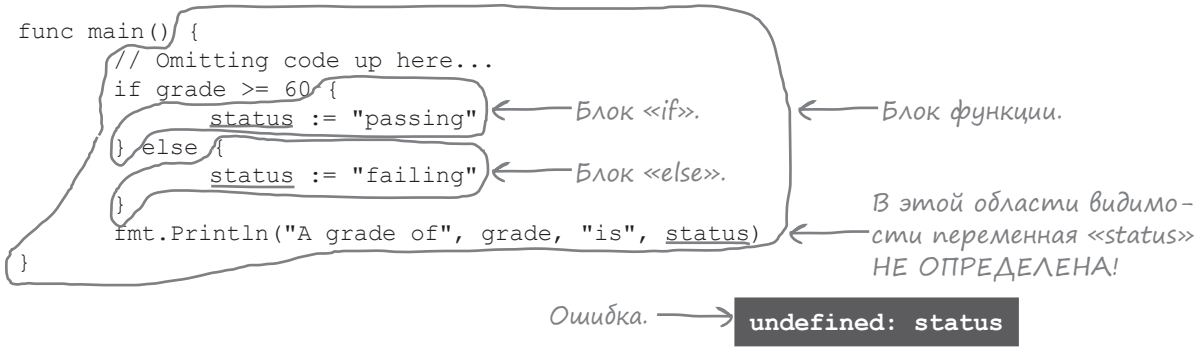


Области видимости переменных в приведенном выше коде:

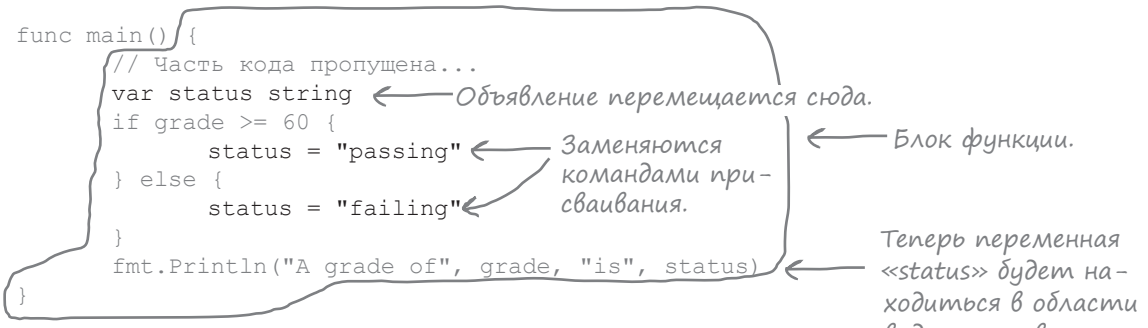
- Областью видимости `packageVar` является весь пакет `main`. К `packageVar` можно обращаться в любой точке любой функции, определенной в пакете.
- Областью видимости `functionVar` является вся функция, в которой объявлена переменная, включая блок `if`, вложенный в эту функцию.
- Область видимости `conditionalVar` ограничивается блоком `if`. При попытке обратиться к `conditionalVar` *после* закрывающей фигурной скобки `}` блока `if` вы получите сообщение об ошибке, в котором говорится, что переменная `conditionalVar` не определена!

Блоки и область видимости переменной (продолжение)

Теперь, когда вы понимаете смысл областей видимости переменных, мы можем объяснить, почему переменная `status` не была определена в программе. Мы объявили переменную `status` в условных блоках. (Собственно, она была объявлена дважды, поскольку программа содержит два разных блока. Именно поэтому мы получили две ошибки «переменная `status` объявлена, но не использована».) Но затем мы попытались обратиться к `status` *за пределами* этих блоков — в месте, не входившем в область видимости этой переменной.



Проблема решается перемещением объявления переменной `status` из блоков условных команд в блок функции. После этого переменная `status` будет находиться в области видимости как во вложенных условных блоках, так *и* в конце блока функции.



Будьте осторожны!

Не забудьте заменить короткие объявления переменных во вложенных блоках командами присваивания!

Если не заменить оба вхождения `:=` на `=`, вы случайно создадите новые переменные с именем `status` внутри вложенных условных блоков, и эти переменные не будут находиться в области видимости внешнего блока функции!

Работа над программой завершена!

Вот и все! Программа *pass_fail.go* готова к работе. Давайте еще раз взглянем на ее полный код:

```
// pass_fail сообщает, сдал ли пользователь экзамен.
package main

import (
        "bufio"
        "fmt"
        "log"
        "os"
        "strings"
        "strconv"
)

func main() {
        fmt.Print("Enter a grade: ")
        reader := bufio.NewReader(os.Stdin)
        input, err := reader.ReadString('\n')
        if err != nil {
                log.Fatal(err)
        }

        input = strings.TrimSpace(input)
        grade, err := strconv.ParseFloat(input, 64)
        if err != nil {
                log.Fatal(err)
        }

        var status string
        if grade >= 60 {
                status = "passing"
        } else {
                status = "failing"
        }
        fmt.Println("A grade of", grade, "is", status)
}
```

Функция «main» вызывается при запуске программы.

Запрашиваем у пользователя значение.

Создаем bufio.Reader для чтения ввода с клавиатуры.

Читает данные, вводимые пользователем до нажатия клавиши Enter.

Если произошла ошибка, вывести сообщение и прервать работу программы.

Удалить символ новой строки из введенных данных.

Преобразовать введенную строку в значение float64 (число).

Если произошла ошибка, вывести сообщение и прервать выполнение программы.

Переменная «status» объявляется здесь, чтобы она находилась в области видимости в границах функции.

Если значение grade равно 60 и более, переменной status присваивается строка «passing». В противном случае переменной присваивается строка «failing».

Выводим введенное значение...

...и результат сдачи экзамена.

Вы можете запускать программу сколько угодно раз. Введите значение меньше 60, и получите сообщение о том, что экзамен не сдан. Введите значение больше 60, и программа сообщит, что экзамен сдан успешно. Кажется, все работает!

```
Shell Edit View Window Help
$ go run pass_fail.go
Enter a grade: 56
A grade of 56 is failing
$ go run pass_fail.go
Enter a grade: 84.5
A grade of 84.5 is passing
$
```

Упражнение

Некоторые строки этого кода приводят к ошибкам компиляции, потому что обращаются к переменной, находящейся вне области видимости. Вычеркните строки, содержащие ошибки.

```
package main

import (
       "fmt"
)

var a = "a"

func main() {
       a = "a"
       b := "b"
       if true {
              c := "c"
              if true {
                     d := "d"
                     fmt.Println(a)
                     fmt.Println(b)
                     fmt.Println(c)
                     fmt.Println(d)
              }
              fmt.Println(a)
              fmt.Println(b)
              fmt.Println(c)
              fmt.Println(d)
       }
       fmt.Println(a)
       fmt.Println(b)
       fmt.Println(c)
       fmt.Println(d)
}
```

Ответ на с. 111.

Только одна переменная в коротком объявлении должна быть новой

И еще один момент! В этом коде есть странность. В главе 1 вы сказали, что переменную нельзя объявить дважды. Тем не менее переменная `err` встречается в двух разных коротких объявлениях!

Переменная «err» объявляется здесь.

```
input, err := reader.ReadString('\n')
// ...
grade, err := strconv.ParseFloat(input, 64)
```

Но похоже, мы объявляем переменную «err» во второй раз!

Действительно, если дважды объявить одно имя переменной в одной области видимости, компилятор выдаст сообщение об ошибке:

Попытаемся снова объявить «a».

```
a := 1
a := 2
```

Ошибка компиляции.

```
no new variables on left side of :=
```

Но если хотя бы одно имя переменной в коротком объявлении является новым, такая запись допустима. Новые имена переменных интерпретируются как объявление, а существующие — как присваивание.

```
a := 1                    // Объявляем «a».
b, a := 2, 3              // Объявляем «b», присваиваем «a».
a, c := 4, 5              // Присваиваем «a», объявляем «c».
fmt.Println(a, b, c)
```

```
4 2 5
```

У этого специального подхода есть причина: многие функции Go возвращают несколько значений. Было бы неприятно объявлять отдельно все переменные только потому, что вы захотели повторно использовать одну из них.

Вариант с отдельными объявлениями всех переменных работает, но, к счастью, поступать так не обязательно...

```
var a, b float64
var err error
a, err = strconv.ParseFloat("1.23", 64)
b, err = strconv.ParseFloat("4.56", 64)
```

Вместо этого Go позволяет использовать короткие объявления переменных, даже если для одной из переменных в действительности выполняется присваивание.

...Можно просто воспользоваться синтаксисом короткого объявления переменных.

```
a, err := strconv.ParseFloat("1.23", 64)   // Объявляем «a» и «err».
b, err := strconv.ParseFloat("4.56", 64)   // Объявляем «b» и присваиваем «err».
fmt.Println(a, b, err)
```

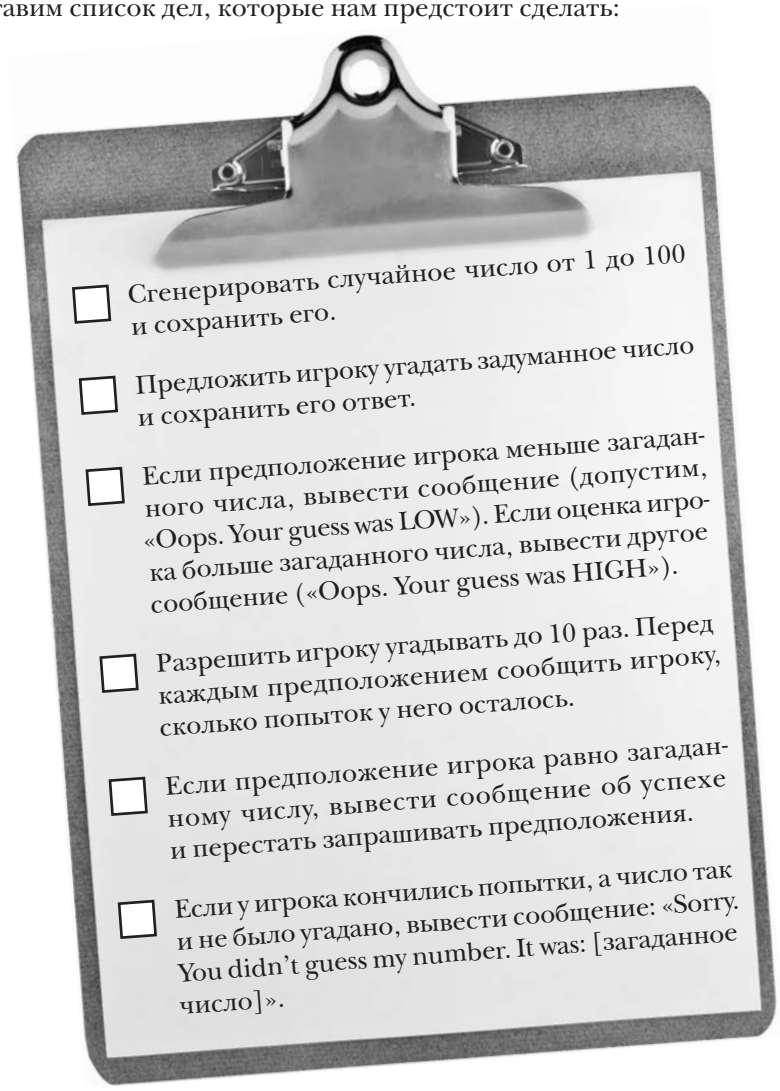
```
1.23 4.56 <nil>
```

Создание игры

Этот пример впервые появился в книге Head First Ruby. (Еще одна хорошая книга, которую также стоит купить!) Он оказался настолько удачным, что мы снова используем его здесь.

Завершим эту главу построением простой игры. Если это звучит слишком внушительно, не беспокойтесь: вы уже знаете большую часть того, что нужно! А попутно мы узнаем о *циклах*, которые позволят игроку повторять свои ходы.

Составим список дел, которые нам предстоит сделать:



Гэри Ришардо, гейм-дизайнер

Создадим новый файл с исходным кодом, который будет называться *guess.go*.

Похоже, начать нужно с генерирования случайного числа. За дело!

Имена пакетов и пути импортирования

Пакет `math/rand` содержит функцию `Intn`, которая сгенерирует случайное число за нас, поэтому в программу следует импортировать `math/rand`. После этого можно будет вызвать функцию `rand.Intn` для генерирования случайного числа.

```
package main

import (
        "fmt"
        "math/rand"
)

func main() {
        target := rand.Intn(100) + 1
        fmt.Println(target)
}
```

Импортируем пакет «math/rand».

Вызываем функцию rand.Intn, которая генерирует целое число.

Постойте-ка! Вы только что сказали, что `Intn` входит в пакет `math/rand`. Тогда почему вы ввели только `rand.Intn`, а не `math/rand.Intn`?

Потому, что первое — путь импортирования пакета, а второе — имя пакета.

Упоминая `math/rand`, мы имеем в виду *путь импортирования* пакета, а не его *имя*. **Путь импортирования** — всего лишь уникальная строка, которая идентифицирует пакет и используется в директиве `import`. После того как пакет будет импортирован, к нему можно обращаться по имени пакета.

Для всех пакетов, которые использовались до сих пор, путь импортирования совпадал с именем пакета. Несколько примеров:

Путь импортирования	*Имя пакета*
`"fmt"`	`fmt`
`"log"`	`log`
`"strings"`	`strings`

Однако путь импортирования и имя пакета могут различаться. Многие пакеты Go классифицируются по категориям — например, «сжатие» или «комплексные вычисления». По этой причине они часто группируются по префиксам пути импортирования (например, `"archive/"` или `"math/"`). (Их можно рассматривать как аналоги путей каталогов на жестком диске.)

Путь импортирования	*Имя пакета*
`"archive"`	`archive`
`"archive/tar"`	`tar`
`"archive/zip"`	`zip`
`"math"`	`math`
`"math/cmplx"`	`cmplx`
`"math/rand"`	`rand`

Имена пакетов и пути импортирования (продолжение)

Язык Go не требует, чтобы имя пакета было как-то связано с путем импортирования. Но по соглашению последний (или единственный) сегмент пути импортирования также используется в качестве имени пакета. Таким образом, для пути импортирования `"archive"` именем пакета также будет `archive`, а для пути импортирования `"archive/zip"` будет использоваться имя пакета `zip`.

Путь импортирования	Имя пакета
`"archive"`	`archive`
`"archive/tar"`	`tar`
`"archive/zip"`	`zip`
`"math"`	`math`
`"math/cmplx"`	`cmplx`
`"math/rand"`	`rand`

Именно по этой причине в директиве `import` используется путь `"math/rand"`, а в функции `main` имя пакета — `rand`.

```
package main

import (
        "fmt"
        "math/rand"
)

func main() {
        target := rand.Intn(100) + 1
        fmt.Println(target)
}
```

Используем полный путь импортирования для «math/rand».

Используем имя пакета: «rand».

Генерирование случайных чисел

Передайте число функции `rand.Intn`, и функция вернет случайное число в диапазоне от `0` до переданного числа. Другими словами, если передать аргумент `100`, будет получено случайное число в диапазоне 0–99. Так как нам требуется число в диапазоне 1–100, остается прибавить `1` к полученному случайному значению. Результат сохраняется в переменной `target`. Пока мы ограничимся простым выводом переменной `target`.

```
package main

import (
        "fmt"
        "math/rand"
)

func main() {
        target := rand.Intn(100) + 1
        fmt.Println(target)
}
```

Генерируем случайное число от 0 до 99.

Прибавить 1, чтобы целое число лежало в диапазоне от 1 до 100.

Попытавшись запустить программу в таком виде, вы получите случайное число. Однако вы будете получать *одно и то же* случайное число раз за разом! Дело в том, что случайные числа, генерируемые компьютерами, на самом деле не настолько уж случайны. Тем не менее их можно сделать более случайными...

```
Shell Edit View Window Help
$ go run guess.go
82
$ go run guess.go
82
$ go run guess.go
82
$
```

Получаем одно и то же случайное число при каждом запуске программы!

Генерирование случайных чисел (продолжение)

Чтобы получать разные случайные числа, необходимо передать значение функции `rand.Seed`. Тем самым вы «инициализируйте» генератор случайных чисел, то есть предоставляете значение, которое будет использоваться для генерирования других случайных чисел. Но если передавать одно и то же значение инициализации, то и случайные значения будут теми же и мы снова вернемся к тому, с чего начинали.

Ранее было показано, что функция `time.Now` выдает значение `Time`, представляющее текущую дату и время. Его можно использовать для того, чтобы получать разное значение инициализации при каждом запуске программы.

```
package main

import (
        "fmt"
        "math/rand"
        "time"
)

func main() {
        seconds := time.Now().Unix()
        rand.Seed(seconds)
        target := rand.Intn(100) + 1
        fmt.Println("I've chosen a random number between 1 and 100.")
        fmt.Println("Can you guess it?")
        fmt.Println(target)
}
```

Также импортируем пакет «time».

Получаем текущую дату и время в формате целого числа.

Функция генератора случайных чисел.

Теперь генерируемые числа будут разными при каждом запуске!

Сообщаем игроку о том, что число выбрано.

Функция `rand.Seed` ожидает получить целое число, поэтому передать ей значение `Time` напрямую не удастся. Вместо этого для `Time` следует вызвать метод `Unix`, который преобразует его в целое число. (А конкретно значение будет преобразовано в формат времени Unix — целое количество секунд, прошедших с 1 января 1970 года. Впрочем, запоминать это не нужно.) Это число передается `rand.Seed`.

Мы также добавим пару вызовов `Println`, чтобы уведомить пользователя о выборе случайного числа. Но помимо этого, оставшуюся часть кода, включая вызовы `rand.Intn`, можно оставить как есть. Инициализация генератора должна стать единственным внесенным изменением.

Теперь при каждом запуске программы вы будете видеть сообщение со случайным числом. Похоже, изменения были успешными!

Разные числа при каждом запуске программы!

```
Shell Edit View Window Help
$ go run guess.go
I've chosen a random number between 1 and 100.
Can you guess it?
73
$ go run guess.go
I've chosen a random number between 1 and 100.
Can you guess it?
18
$
```

Получение целого числа с клавиатуры

Первое требование выполнено! Теперь необходимо получить предположение пользователя, введенное на клавиатуре.

Все это должно работать практически так же, как при чтении входного значения в программе проверки результатов экзамена.

Есть только одно отличие: вместо того, чтобы преобразовывать входное значение в `float64`, мы преобразуем его в `int` (поскольку в игре используются только целые числа). По этой причине строка, прочитанная с клавиатуры, передается функции `Atoi` пакета `strconv` — вместо функции `ParseFloat`. Функция `Atoi` также возвращает целое число. (Как и `ParseFloat`, `Atoi` может вернуть ошибку, если преобразовать строку не удастся. В таком случае программа снова сообщает об ошибке и завершается.)

```
package main

import (
        "bufio"
        "fmt"
        "log"
        "math/rand"
        "os"
        "strconv"
        "strings"
        "time"
)

func main() {
        seconds := time.Now().Unix()
        rand.Seed(seconds)
        target := rand.Intn(100) + 1
        fmt.Println("I've chosen a random number between 1 and 100.")
        fmt.Println("Can you guess it?")
        fmt.Println(target)

        reader := bufio.NewReader(os.Stdin)

        fmt.Print("Make a guess: ")
        input, err := reader.ReadString('\n')
        if err != nil {
                log.Fatal(err)
        }
        input = strings.TrimSpace(input)
        guess, err := strconv.Atoi(input)
        if err != nil {
                log.Fatal(err)
        }
}
```

Импортировать дополнительные пакеты. (Все эти пакеты использовались в программе проверки результатов экзамена!)

Создаем bufio.Reader для чтения ввода с клавиатуры.

Запросить число.

Прочитать данные, введенные пользователем до нажатия Enter.

Если произошла ошибка, программа выводит сообщение и завершается.

Удаление символа новой строки.

Входная строка преобразуется в целое число.

Если произошла ошибка, программа выводит сообщение и завершается.

Сравнение предположения с загаданным числом

Еще одно требование выполнено. Да и со следующим должно быть просто... Нужно сравнить предположение пользователя со сгенерированным числом и сообщить, было ли предположение больше или меньше загаданного числа.

- [x] Предложить игроку угадать задуманное число и сохранить его ответ.
- [ ] Если предположение игрока меньше загаданного числа, вывести сообщение (допустим, «Oops. Your guess was LOW»). Если оценка игрока больше загаданного числа, вывести другое сообщение («Oops. Your guess was HIGH»).

Если предположение `guess` меньше `target`, необходимо вывести соответствующее сообщение. *В противном случае, если* предположение `guess` больше `target`, следует вывести сообщение об этом. Похоже, здесь будет уместна команда `if...else`. Мы добавим ее под остальным кодом функции `main`.

```
// Директивы package и import не изменились

func main() {
        // Предшествующий код тоже не изменился

        if guess < target {
                fmt.Println("Oops. Your guess was LOW.")
        } else if guess > target {
                fmt.Println("Oops. Your guess was HIGH.")
        }
}
```

Если предположение игрока меньше загаданного числа, сообщить об этом.

Если предположение игрока больше загаданного числа, сообщить об этом.

Теперь попробуем запустить обновленную программу в терминале. Программа все еще выводит `target` при каждом запуске, что может быть полезно для отладки. Введите меньшее число — и программа сообщит об этом. Если запустить программу повторно, вы получите новое значение `target`. Введите большее число, и получите другое сообщение.

```
Shell  Edit  View  Window  Help
$ go run guess.go
81
I've chosen a random number between 1 and 100.
Can you guess it?
Make a guess: 1
Oops. Your guess was LOW.
$ go run guess.go
54
I've chosen a random number between 1 and 100.
Can you guess it?
Make a guess: 100
Oops. Your guess was HIGH.
$
```

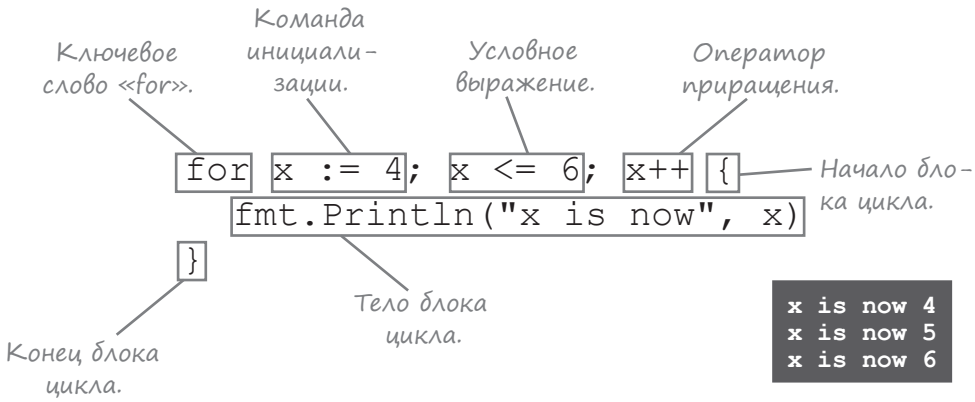
Циклы

И еще одно требование выполнено! Переходим к следующему.

Пока у игрока есть только одна попытка, но мы хотим, чтобы игрок мог угадывать до 10 раз.

Код для вывода запроса уже готов. Остается лишь организовать его *повторное выполнение*. Мы можем воспользоваться **циклом** для многократного выполнения блока кода. Если у вас есть опыт работы на других языках программирования, вероятно, вы уже сталкивались с циклами. Если вы хотите, чтобы одна или несколько команд выполнялись снова и снова, разместите их в цикле.

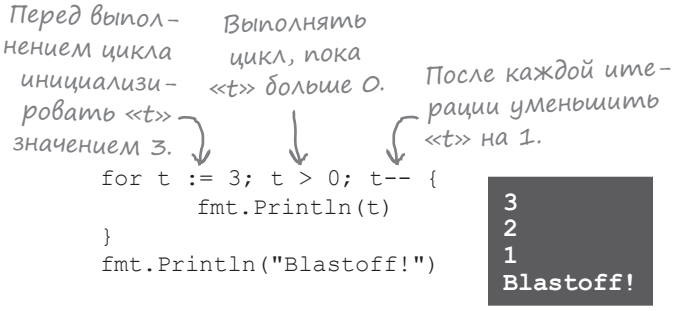
- [x] Если предположение игрока меньше загаданного числа, вывести сообщение (допустим, «Oops. Your guess was LOW»). Если оценка игрока больше загаданного числа, вывести другое сообщение («Oops. Your guess was HIGH»).
- [ ] Разрешить игроку угадывать до 10 раз. Перед каждым предположением сообщить игроку, сколько попыток у него осталось.



Циклы всегда начинаются с ключевого слова `for`. В одной из стандартных разновидностей циклов за `for` следуют три сегмента кода, которые управляют циклом:

- Команда инициализации, обычно используемая для инициализации переменной.
- Условное выражение, которое определяет, когда следует прервать выполнение цикла.
- Оператор приращения, который выполняется после каждой итерации цикла.

Команда инициализации часто используется для инициализации переменной; условное выражение обеспечивает выполнение цикла до того, как переменная достигнет определенного значения, и оператор приращения обновляет значение этой переменной. Например, в приведенном фрагменте переменная `t` инициализируется значением `3`, условие обеспечивает выполнение цикла, пока `t > 0`, а оператор приращения уменьшает `t` на 1 при каждом выполнении цикла. В конечном итоге `t` уменьшается до `0` и цикл завершается.



Циклы (продолжение)

Операторы ++ и -- часто встречаются в командах приращения циклов. ++ увеличивает значение переменной на 1, а -- уменьшает его на 1.

```
x := 0
x++
fmt.Println(x)
x++
fmt.Println(x)
x--
fmt.Println(x)
```

```
1
2
1
```

В циклах ++ и -- удобны для выполнения прямого или обратного отсчета.

```
for x := 1; x <= 3; x++ {
        fmt.Println(x)
}
```

```
1
2
3
```

```
for x := 3; x >= 1; x-- {
        fmt.Println(x)
}
```

```
3
2
1
```

В языке Go также поддерживаются операторы присваивания += и -=. Они получают значение в переменной, добавляют или вычитают другое значение, а затем присваивают результат той же переменной.

```
x := 0
x += 2
fmt.Println(x)
x += 5
fmt.Println(x)
x -= 3
fmt.Println(x)
```

```
2
7
4
```

Операторы += и -= также могут использоваться в циклах для изменения переменной на величину, отличную от 1.

```
for x := 1; x <= 5; x += 2 {
        fmt.Println(x)
}
```

```
1
3
5
```

```
for x := 15; x >= 5; x -= 5 {
        fmt.Println(x)
}
```

```
15
10
5
```

Когда цикл завершается, выполнение программы продолжится с команды, следующей за блоком цикла. При этом цикл продолжает выполняться, пока условное выражение остается истинным. Этот факт может иметь нежелательные последствия; ниже приведены примеры циклов, которые выполняются бесконечно или не выполняются ни одного раза:

Бесконечный цикл!

```
for x := 1; true; x++ {
        fmt.Println(x)
}
```

Цикл не выполняется ни разу!

```
for x := 1; false; x++ {
        fmt.Println(x)
}
```

Будьте осторожны!

Цикл может выполняться бесконечно; в этом случае ваша программа никогда не остановится сама.

Если это случится, в активном окне терминала нажмите клавишу Control одновременно с клавишей C, чтобы прервать выполнение программы.

Операторы инициализации и приращения необязательны

При желании операторы инициализации и приращения в заголовке цикла `for` можно опустить, оставив только условное выражение (хотя вы должны проследить за тем, чтобы условие в какой-то момент становилось ложным, иначе в программе возникнет бесконечный цикл).

```
x := 1
for x <= 3 {
        fmt.Println(x)
        x++
}
```

x объявляется в отдельной команде.

Используется только условное выражение.

x увеличивается в отдельной команде.

```
1
2
3
```

```
x := 3
for x >= 1 {
        fmt.Println(x)
        x--
}
```

x объявляется в отдельной команде.

Используется только условное выражение.

x уменьшается в отдельной команде.

```
3
2
1
```

Циклы и области видимости

Как и в случае с условными командами, область видимости любых переменных, объявленных в блоке цикла, ограничивается этим блоком (хотя команда инициализации, условное выражение и оператор приращения также могут считаться частью этой области видимости).

```
for x := 1; x <= 3; x++ {
        y := x + 1
        fmt.Println(y)
}
fmt.Println(y)
```

Остается в области видимости...

Ошибка: вне области видимости!

```
undefined: y
```

Ошибка.

```
for x := 1; x <= 3; x++ {
        fmt.Println(x)
}
fmt.Println(x)
```

Остается в области видимости...

Ошибка: вне области видимости!

```
undefined: x
```

Ошибка.

Как и в случае с условными командами, любые переменные, объявленные *до* начала цикла, находятся в области видимости в заголовке и блоке цикла и *остаются* в области видимости после выхода из цикла.

```
var x int
for x = 1; x <= 3; x++ {
        fmt.Println(x)
}
fmt.Println(x)
```

Объявлять x здесь не нужно, просто присвойте значение!

Объявляется за пределами цикла...

Остается в области видимости.

Остается в области видимости.

```
1
2
3
4
```

Сломай и изучи!

Эта программа считает до трех в цикле. Возьмите этот пример, внесите одно из указанных изменений и запустите программу; затем отмените изменение и переходите к следующему. Посмотрите, что из этого выйдет!

```
package main

import "fmt"

func main() {
       for x := 1; x <= 3; x++ {
              fmt.Println(x)
       }
}
```

```
1
2
3
```

Если...	*...программа не будет работать, потому что...*
Добавить круглые скобки после ключевого слова `for` `for (x := 1; x <= 3; x++)`	Другие языки требуют, чтобы управляющая часть цикла `for` заключалась в круглые скобки. Однако язык Go не только этого не требует, но и *запрещает*
Удалить `:` из команды инициализации `x = 1`	Если только вы не присваиваете значение переменной, уже объявленной во внешней области видимости (а это бывает довольно редко), команда инициализации должна быть объявлением, а не присваиванием
Удалить `=` из условного выражения `x < 3`	Выражение `x<3` становится ложным , когда значение `x` становится равным `3` (тогда как выражение `x<=3` все еще остается истинным). Таким образом, цикл будет вести отсчет только до `2`
Использовать противоположный оператор сравнения в условном выражении `x >= 3`	Так как условие будет ложным уже в самом начале цикла (`x` инициализируется `1`, что меньше `3`), цикл не будет выполнен ни одного раза
Заменить оператор приращения `x++` на `x--` `x--`	Переменная `x` начинает отсчет с `1` (`1`, `0`, `-1`, `-2` и т. д.). Так как она никогда не станет больше `3`, цикл будет выполняться бесконечно
Переместить команду `fmt.Println(x)` за пределы блока цикла	Переменные, объявленные в команде инициализации или в блоке цикла, остаются в области видимости только в пределах блока цикла

Упражнение

Внимательно рассмотрите команду инициализации, условное выражение и оператор приращения в каждом из этих циклов. Затем напишите, какой результат, по вашему мнению, выведет каждый цикл.

(Мы выполнили первое упражнение за вас.)

```
for x := 1; x <= 3; x++ {
       fmt.Print(x)
}
```

123

```
for x := 3; x >= 1; x-- {
       fmt.Print(x)
}
```

.....

```
for x := 2; x <= 3; x++ {
       fmt.Print(x)
}
```

.....

```
for x := 1; x < 3; x++ {
       fmt.Print(x)
}
```

.....

```
for x := 1; x <= 3; x+= 2 {
       fmt.Print(x)
}
```

.....

```
for x := 1; x >= 3; x++ {
       fmt.Print(x)
}
```

.....

Ответ на с. 112.

Использование цикла в игре

Наша игра по-прежнему запрашивает значение у пользователя только один раз. Заключим в цикл код, который запрашивает значение у пользователя, а потом сообщает, было ли оно больше или меньше загаданного числа. Пользователю предоставляются 10 попыток.

Количество попыток, сделанных пользователем, будет храниться в переменной `int` с именем `guesses`. В команде инициализации нашего цикла переменная `guesses` инициализируется значением `0`. В каждой итерации цикла значение `guesses` будет увеличиваться на 1, а когда оно достигнет `10`, цикл завершится.

Мы также добавим команду `Println` в начало блока цикла. Эта команда сообщает пользователю, сколько попыток у него осталось.

```
// Директивы package и import не изменились

func main() {
        seconds := time.Now().Unix()
        rand.Seed(seconds)
        target := rand.Intn(100) + 1
        fmt.Println("I've chosen a random number between 1 and 100.")
        fmt.Println("Can you guess it?")
        fmt.Println(target)

        reader := bufio.NewReader(os.Stdin)

        for guesses := 0; guesses < 10; guesses++ {
                fmt.Println("You have", 10-guesses, "guesses left.")

                fmt.Print("Make a guess: ")
                input, err := reader.ReadString('\n')
                if err != nil {
                        log.Fatal(err)
                }
                input = strings.TrimSpace(input)
                guess, err := strconv.Atoi(input)
                if err != nil {
                        log.Fatal(err)
                }

                if guess < target {
                        fmt.Println("Oops. Your guess was LOW.")
                } else if guess > target {
                        fmt.Println("Oops. Your guess was HIGH.")
                }
        }
}
```

В переменной «guesses» хранится количество сделанных попыток.

Чтобы вычислить количество оставшихся попыток, значение guesses вычитается из 10.

Уже написанный код, который запрашивает у пользователя значение и сообщает, больше оно или меньше загаданного; будет выполняться 10 раз.

Конец цикла for.

Использование цикла в игре (продолжение)

Если снова запустить программу после включения в нее цикла, программа 10 раз предложит угадать число!

Программа все еще выводит загаданное число при запуске.

Внутри цикла программа выводит количество оставшихся попыток, запрашивает у пользователя значение и сообщает, больше оно или меньше загаданного числа.

В текущей версии программа не сообщает игроку, что число было отгадано, и цикл не останавливается.

```
Shell Edit View Window Help
$ go run guess.go
68
I've chosen a random number between 1 and 100.
Can you guess it?
You have 10 guesses left.
Make a guess: 50
Oops. Your guess was LOW.
You have 9 guesses left.
Make a guess: 75
Oops. Your guess was HIGH.
You have 8 guesses left.
Make a guess: 68
You have 7 guesses left.
Make a guess:
```

Так как код, который запрашивает значение и проверяет его, заключен в цикл, он выполняется многократно. После 10 попыток цикл (и игра) заканчивается.

Но цикл всегда выполняется 10 раз, даже если игрок угадал число! Проблему нужно решить, и это станет нашим следующим требованием.

Пропуск частей цикла командами continue и break

Самое трудное позади! Остается реализовать только пару требований.

На данный момент цикл, который запрашивает у пользователя предположение, всегда выполняется 10 раз. Даже если игрок ввел правильное число, мы об этом не сообщаем, и цикл не останавливается. Сейчас мы исправим этот недостаток.

- [x] Разрешить игроку угадывать до 10 раз. Перед каждым предположением сообщить игроку, сколько предположений у него осталось.
- [ ] Если предположение игрока равно загаданному числу, вывести сообщение об успехе и перестать запрашивать предположения.

В Go предусмотрены два ключевых слова для управления циклом. Первое — `continue` — осуществляет немедленный переход к следующей итерации цикла; при этом дальнейший код текущей итерации в блоке цикла пропускается.

Переходит сразу к началу цикла.

```
for x := 1; x <= 3; x++ {
        fmt.Println("before continue")
        continue
        fmt.Println("after continue")
}
```

```
before continue
before continue
before continue
```

В приведенном выше примере строка `"after continue"` никогда не выводится, потому что ключевое слово `continue` всегда выполняет переход к началу цикла — до того, как отработает второй вызов `Println`.

Второе ключевое слово `break` приводит к немедленному выходу из цикла. Дальнейший код в блоке цикла не отрабатывается, другие итерации цикла не выполняются. Управление передается первой команде, следующей за циклом.

Цикл ДОЛЖЕН отработать три раза, если бы не команда break.

Немедленно выйти из цикла.

```
for x := 1; x <= 3; x++ {
        fmt.Println("before break")
        break
        fmt.Println("after break")
}
fmt.Println("after loop")
```

```
before break
after loop
```

Здесь при первой итерации цикла выводится сообщение `"before break"`, после чего команда `break` немедленно прерывает цикл; сообщение `"after break"` не выводится, и цикл не выполняется повторно (хотя без `break` он бы выполнился еще два раза). Управление передается команде, следующей за циклом.

Похоже, ключевое слово `break` именно то, что нам нужно: при вводе правильного значения цикл должен прерваться. Попробуем воспользоваться им в игре...

Выход из цикла

Мы используем условную команду `if...else`, чтобы сообщить игроку результат очередной попытки. Если игрок ввел слишком большое или слишком маленькое число, программа выводит соответствующее сообщение.

Если введенное значение не является *ни* слишком большим, *ни* слишком маленьким, значит, оно правильное. Добавим в условную команду ветвь `else`, которая будет выполняться для правильно угаданного значения. Внутри блока ветви `else` мы сообщим игроку, что он угадал, после чего прерываем цикл попыток при помощи команды `break`.

```
// Директивы package и import не изменились

func main() {
        // Предыдущий код не изменился

        for guesses := 0; guesses < 10; guesses++ {
                // No changes to previous code; omitting

                if guess < target {
                        fmt.Println("Oops. Your guess was LOW.")
                } else if guess > target {
                        fmt.Println("Oops. Your guess was HIGH.")
                } else {
                        fmt.Println("Good job! You guessed it!")
                        break
                }
        }
}
```

Поздравляем игрока. → `fmt.Println("Good job! You guessed it!")`

`break` ← *Выйти из цикла.*

Если пользователь угадал правильно, программа выдает поздравление, а цикл прерывается без 10-кратного повторения.

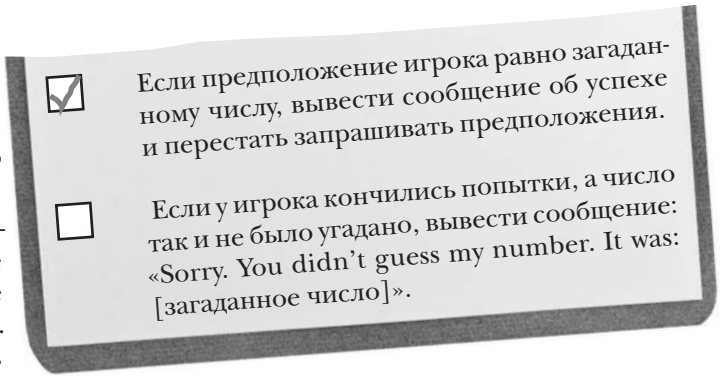
Загаданное число; мы сжульничаем и сразу введем правильное значение. →

Пользователь получил поздравление, цикл завершается! →

```
Shell Edit View Window Help
$ go run guess.go
48
I've chosen a random number between 1 and 100.
Can you guess it?
You have 10 guesses left.
Make a guess: 48
Good job! You guessed it!
$
```

Еще одно требование выполнено!

Вывод загаданного числа



Мы уже *почти* у цели! Осталось всего одно требование!

Если игрок делает 10 попыток, но так и не угадывает число, цикл завершается. В этом случае необходимо вывести сообщение о проигрыше и показать, какое же число было загадано.

Но при правильно отгаданном числе мы *тоже* выходим из цикла. Программа не должна выводить сообщение о проигрыше, если игрок уже победил!

Итак, перед циклом отгадывания мы объявим переменную `success` для хранения логического значения. (Переменная должна быть объявлена *перед* циклом, чтобы она оставалась в области видимости после завершения цикла.) Переменная `success` инициализируется значением по умолчанию `false`. Если пользователь отгадывает число, переменной `success` присваивается значение `true`, которое означает, что выводить сообщение о проигрыше не нужно.

```
// Директивы package и import не изменились

func main() {
        // Предыдущий код не изменился

        success := false
        for guesses := 0; guesses < 10; guesses++ {
                // Предыдущий код не изменился

                if guess < target {
                        fmt.Println("Oops. Your guess was LOW.")
                } else if guess > target {
                        fmt.Println("Oops. Your guess was HIGH.")
                } else {
                        success = true
                        fmt.Println("Good job! You guessed it!")
                        break
                }
        }

        if !success {
                fmt.Println("Sorry, you didn't guess my number. It was:", target)
        }
}
```

Объявляем переменную «success» до цикла, чтобы она оставалась в области видимости после выхода из цикла.

Если игрок угадал правильно, значит, выводить сообщение о проигрыше не нужно.

Если игрок НЕ угадал (если переменная «success» содержит false)...

...вывести сообщение о проигрыше.

После цикла добавляется блок `if` для вывода сообщения о проигрыше. Но блок `if` выполняется только в том случае, если условие равно `true`, а мы хотим, чтобы сообщение о проигрыше выводилось только для переменной `success`, равной `false`. По этой причине мы добавляем оператор логического отрицания (`!`). Как было показано ранее, этот оператор преобразует `true` в `false`, а `false` в `true`.

В результате сообщение о проигрыше будет выводиться, если переменная `success` равна `false`, и не будет, если переменная `success` равна `true`.

Последние штрихи

- ☑ Если у игрока кончились попытки, а число так и не было угадано, вывести сообщение: «Sorry. You didn't guess my number. It was: [загаданное число]».

Поздравляем, это было финальное требование!

Исправим еще пару недочетов в этом коде, а затем опробуем игру!

Как упоминалось ранее, в начало программы обычно вставляется комментарий с описанием того, что она делает. Давайте добавим такой комментарий в свою программу.

```
// guess - игра, в которой игрок должен угадать случайное число.
package main
...
```

Добавим комментарий с описанием программы, над директивой package.

Наша программа также дает игроку подсказку, выводя загаданное число в начале каждой игры. Удалим вызов `Println`, который это делает.

```
fmt.Println("I've chosen a random number between 1 and 100.")
fmt.Println("Can you guess it?")
~~fmt.Println(target)~~
```

Не сообщать загаданное число в начале каждой игры.

Наконец-то все готово, и мы сможем запустить готовый код игры!

Для начала намеренно исчерпаем все попытки, чтобы программа вывела загаданное число...

Другие неправильные попытки...

Если попытки кончились, программа выводит правильное число.

```
Shell Edit View Window Help
$ go run guess.go
I've chosen a random number between 1 and 100.
Can you guess it?
You have 10 guesses left.
Make a guess: 10
Oops. Your guess was LOW.
You have 9 guesses left.
Make a guess: 20
Oops. Your guess was LOW.
...
You have 1 guesses left.
Make a guess: 62
Oops. Your guess was LOW.
Sorry, you didn't guess my number. It was: 63
```

А теперь попробуем угадать число.

Наша игра прекрасно работает!

Если число названо правильно, выводится сообщение о выигрыше!

```
Shell Edit View Window Help Cheats
$ go run guess.go
I've chosen a random number between 1 and 100.
Can you guess it?
You have 10 guesses left.
Make a guess: 50
Oops. Your guess was HIGH.
You have 9 guesses left.
Make a guess: 40
Oops. Your guess was LOW.
You have 8 guesses left.
Make a guess: 45
Good job! You guessed it!
```

Поздравляем, игра готова!

Вы написали игру на языке Go с использованием условных команд и циклов! Налейте себе лимонада — вы это заслужили!

Полный исходный код guess.go!

```
// guess - игра, в которой игрок должен угадать случайное число.
package main

import (
        "bufio"
        "fmt"
        "log"
        "math/rand"
        "os"
        "strconv"
        "strings"
        "time"
)

func main() {
        seconds := time.Now().Unix()
        rand.Seed(seconds)
        target := rand.Intn(100) + 1
        fmt.Println("I've chosen a random number between 1 and 100.")
        fmt.Println("Can you guess it?")

        reader := bufio.NewReader(os.Stdin)
        success := false
        for guesses := 0; guesses < 10; guesses++ {
                fmt.Println("You have", 10-guesses, "guesses left.")
                fmt.Print("Make a guess: ")
                input, err := reader.ReadString('\n')
                if err != nil {
                        log.Fatal(err)
                }
                input = strings.TrimSpace(input)
                guess, err := strconv.Atoi(input)
                if err != nil {
                        log.Fatal(err)
                }

                if guess < target {
                        fmt.Println("Oops. Your guess was LOW.")
                } else if guess > target {
                        fmt.Println("Oops. Your guess was HIGH.")
                } else {
                        success = true
                        fmt.Println("Good job! You guessed it!")
                        break
                }
        }

        if !success {
                fmt.Println("Sorry, you didn't guess my number. It was:", target)
        }
}
```

Импортируем все пакеты, которые будут использоваться в коде.

Получаем текущую дату и время в виде целого числа.

Инициализируем генератор случайных чисел.

Генерируем целое число от 1 до 100.

Создаем bufio.Reader для чтения ввода с клавиатуры.

Настроить программу, чтобы по умолчанию выводилось сообщение о проигрыше.

Запрашиваем число.

Прочитать данные, введенные пользователем до нажатия Enter.

Если произошла ошибка, программа выводит сообщение и завершается.

Удаляем символ новой строки.

Введенная строка преобразуется в целое число.

Если произошла ошибка, программа выводит сообщение и завершается.

Если введенное значение меньше загаданного, сообщить об этом.

Если введенное значение больше загаданного, сообщить об этом.

В противном случае введенное значение должно быть правильным...

Предотвращает вывод сообщения о проигрыше.

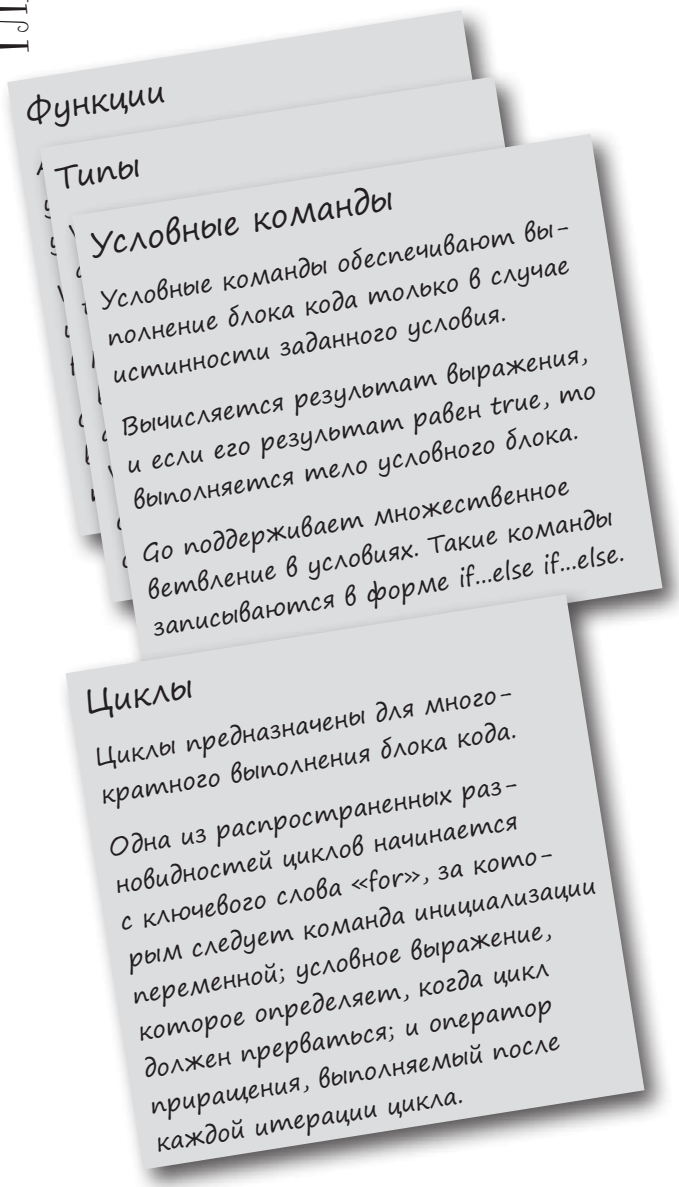
Выход из цикла.

Если переменная «success» равна false, сообщить игроку загаданное число.

Ваш инструментарий Go

Глава 2 подошла к концу! В ней ваш инструментарий пополнился условными командами и циклами.

ГЛАВА 2



КЛЮЧЕВЫЕ МОМЕНТЫ

- **Метод** — разновидность функций, связываемых со значениями конкретного типа.
- Go интерпретирует все символы от `//` до конца строки как **комментарий** и игнорирует их.
- Многострочные комментарии начинаются с `/*` и завершаются `*/`. Все символы между этими маркерами, включая символы новой строки, игнорируются.
- Традиционно в начало любой программы включается комментарий, который объясняет, что делает программа.
- В отличие от большинства языков программирования, Go допускает множественные возвращаемые значения из вызова функции или метода.
- Одно из стандартных применений множественных возвращаемых значений — возвращение основного результата функции и второго значения, которое сообщает, произошла ли ошибка при вызове.
- Чтобы проигнорировать значение без реального использования в программе, воспользуйтесь **пустым идентификатором** `_`. Пустой идентификатор может использоваться вместо любой переменной в любой команде присваивания.
- Постарайтесь не присваивать переменным имена, совпадающие с именами типов, функций или пакетов; это приведет к **замещению** (переопределению) элемента с тем же именем.
- Функции, условные команды и циклы содержат **блоки** кода, заключенные в фигурные скобки `{}`.
- Файлы и пакеты также образуют блоки, хотя содержащийся в них код и не заключается в фигурные скобки `{}`.
- **Область видимости** переменной ограничивается блоком, в котором она определяется, а также всеми блоками, вложенными в этот блок.
- Кроме имени, пакет может иметь **путь импортирования**, который должен указываться при импортировании.
- Ключевое слово `continue` осуществляет переход к следующей итерации цикла.
- Ключевое слово `break` полностью прерывает выполнение цикла.

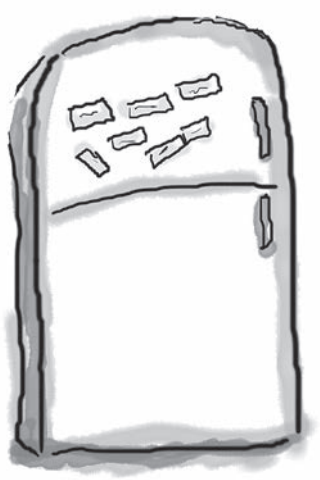
Упражнение
Решение

Эти вызовы `Println` заключены в условные блоки, лишь часть из них будет выполнена. Напишите, какой результат выведет программа.

```
if true {                          // Блоки «if» выполняются в том случае, если условие при вычислении
                                   // дает результат true (или СОДЕРЖИТ true).
       fmt.Println("true")
}
if false {                         // Если условие содержит false, блок не выполняется.
       fmt.Println("false")
}
if !false {                        // Оператор логического отрицания превращает false в true.
       fmt.Println("!false")
}
if true {                          // Выполняется ветвь «if»...
       fmt.Println("if true")
} else {                           // ...поэтому ветвь «else» не выполняется.
       fmt.Println("else")
}
if false {                         // Ветвь «if» не выполняется...
       fmt.Println("if false")
} else if true {                   // ...поэтому МОЖЕТ выполняться
       fmt.Println("else if true") //                ветвь «else if».
}
if 12 == 12 {                      // Условие 12 == 12 дает true.
       fmt.Println("12 == 12")
}                                  // Значения РАВНЫ, поэтому
if 12 != 12 {                      // условие равно false.
       fmt.Println("12 != 12")
}
if 12 > 12 {                       // Число 12 НЕ больше самого себя...
       fmt.Println("12 > 12")
}
if 12 >= 12 {                      // ...Но число 12 РАВНО самому себе.
       fmt.Println("12 >= 12")
}
if 12 == 12 && 5.9 == 5.9 {        // Оператор && дает результат true, если
                                   // истинны ОБА выражения.
       fmt.Println("12 == 12 && 5.9 == 5.9")
}
if 12 == 12 && 5.9 == 6.4 {        // Одно выражение ложно.
       fmt.Println("12 == 12 && 5.9 == 6.4")
}
if 12 == 12 || 5.9 == 6.4 {        // Оператор || дает результат true, если истинно
       fmt.Println("12 == 12 || 5.9 == 6.4")
}                                  //                ХОТЯ БЫ ОДНО выражение.
```

Результат:

```
true
!false
if true
else if true
12 == 12
12 >= 12
12 == 12 && 5.9 == 5.9
12 == 12 || 5.9 == 6.4
```



Развлечения с магнитами. Решение

На холодильнике была выложена программа Go, которая выводит размер файла. Для этого программа вызывает функцию `os.Stat`, которая возвращает значение `os.FileInfo` и, возможно, значение ошибки. Затем она вызывает метод `Size` для значения `FileInfo`, чтобы получить размер файла.

Однако исходная программа использует пустой идентификатор `_`, чтобы проигнорировать значение ошибки из `os.Stat`. Если в программе произойдет ошибка (например, если файл не существует), программа будет работать некорректно.

Используйте дополнительные фрагменты кода и создайте программу, которая в основном работает так же, как исходная, но еще проверяет значение ошибки, полученное от `os.Stat`. Если ошибка от `os.Stat` не равна `nil`, программа должна сообщить об ошибке и аварийно завершиться.

```
package main

import (
        "fmt"
        "log"
        "os"
)

func main() {
    fileInfo, err := os.Stat("my.txt")
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(fileInfo.Size())
}

_
```

Сохраняем ошибку, которая может быть получена от os.Stat.

Получает значение FileInfo с данными, относящимися к файлу my.txt.

Содержит размер файла, дату его изменения и т. д.

Если ошибка не равна nil, передать ее log.Fatal.

Возвращает размер файла.

Этот магнит можно убрать; пустой идентификатор больше не нужен!

Некоторые строки этого кода приводят к ошибкам компиляции, потому что обращаются к переменной, находящейся вне области видимости. Вычеркните строки, содержащие ошибки.

```
package main

import (
       "fmt"
)

var a = "a"

func main() {
       a = "a"
       b := "b"
       if true {
              c := "c"
              if true {
                     d := "d"
                     fmt.Println(a)
                     fmt.Println(b)
                     fmt.Println(c)
                     fmt.Println(d)
              }
              fmt.Println(a)
              fmt.Println(b)
              fmt.Println(c)
             ~~fmt.Println(d)~~
       }
       fmt.Println(a)
       fmt.Println(b)
      ~~fmt.Println(c)~~
      ~~fmt.Println(d)~~
}
```

Упражнение
Решение

Внимательно рассмотрите команду инициализации, условное выражение и оператор приращения в каждом из этих циклов. Затем напишите, какой результат по вашему мнению выведет каждый цикл.

Начинает с 1. Останавливается на 3. По возрастанию.

```
for x := 1; x <= 3; x++ {
        fmt.Print(x)
}
```

123

Начинает с 3. Останавливается на 1. По убыванию.

```
for x := 3; x >= 1; x-- {
        fmt.Print(x)
}
```

321

Начинает с 2. Останавливается на 3. По возрастанию.

```
for x := 2; x <= 3; x++ {
        fmt.Print(x)
}
```

23

Начинает с 1. Останавливается на 3. По возрастанию с шагом 2.

```
for x := 1; x <= 3; x+= 2 {
        fmt.Print(x)
}
```

13

Начинает с 1. Останавливается на 3. По возрастанию.

```
for x := 1; x < 3; x++ {
        fmt.Print(x)
}
```

12

Начинает с 1. Останавливается, когда x < 3 (то есть немедленно). Не выполняется ни разу!

```
for x := 1; x >= 3; x++ {
        fmt.Print(x)
}
```

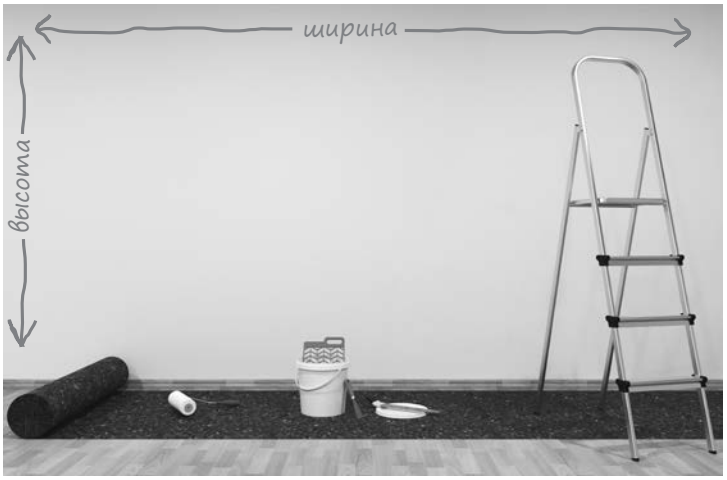
Результатов нет; цикл не выполняется ни разу!

Функции

И все же чего-то не хватало. Вы вызывали функции как настоящий профи. Но могли вызывать только те функции, которые были определены для вас в Go. Настала ваша очередь. В этой главе мы покажем, как создавать собственные функции. Вы научитесь объявлять функции с параметрами и без. Сначала вы узнаете, как объявлять функции, которые возвращают одно значение, а потом мы перейдем к возвращению нескольких значений, чтобы функция могла сигнализировать об ошибке. А еще в этой главе рассматриваются **указатели**, которые повышают эффективность вызовов функций по затратам памяти.

Повторяющийся код

Допустим, вы хотите вычислить, сколько краски потребуется для покрытия нескольких стен. Производитель указывает, что одного литра краски хватает на 10 квадратных метров. Следовательно, для определения количества литров необходимо вычислить площадь каждой стены, умножив ее ширину (в метрах) на высоту, а затем разделить результат на 10.




```
// Директивы package и imports пропущены
func main() {
        var width, height, area float64
        width = 4.2
        height = 3.0
        area = width * height
        fmt.Println(area/10.0, "liters needed")
        width = 5.2
        height = 3.5
        area = width * height
        fmt.Println(area/10.0, "liters needed")
}
```

Вычислить расход краски для первой стены.

Вычисляем площадь стены.

Вычисляем, сколько краски понадобится для этой площади.

Сделать то же самое для второй стены.

Вычисляем площадь стены.

Вычисляем, сколько краски понадобится для этой площади.

```
1.2600000000000002 liters needed
1.8199999999999998 liters needed
```

Такое решение работает, но у него есть пара недостатков:

- Похоже, произведение вычисляется с небольшой погрешностью, поэтому выводимые значения выглядят немного странно. Хватило бы пары цифр в дробной части.
- В этой версии присутствует большое количество повторяющегося кода. Если мы добавим новые стены, все станет еще хуже.

Для устранения обоих недостатков потребуются некоторые объяснения, поэтому начнем с первого пункта...

Вычисления содержат погрешность, потому что обычные операции с плавающей точкой на компьютерах не обладают идеальной точностью (обычно до нескольких квадриллионных). Причины слишком сложны, чтобы углубляться в них, однако эта проблема существует не только в Go.

Но если округлить число до разумной точности перед выводом, все должно быть нормально. Давайте ненадолго отвлечемся от основной темы и познакомимся с функцией для решения этой задачи.

Форматирование вывода функциями Printf и Sprintf

Числа с плавающей точкой в языке Go хранятся с высокой степенью точности. Если вам потребуется вывести такое число, результат может быть громоздким и неудобным:

```
fmt.Println("About one-third:", 1.0/3.0)
```

```
About one-third: 0.3333333333333333
```

Слишком много цифр!

Для решения подобных проблем форматирования в пакете `fmt` имеется функция `Printf` (сокращение от «**print**, with **f**ormatting», то есть «вывод с форматированием»). Функция получает строку и вставляет в нее одно или несколько значений, отформатированных заданным способом. После этого функция выводит полученную строку.

```
fmt.Printf("About one-third: %0.2f\n", 1.0/3.0)
```

```
About one-third: 0.33
```

Уже более читаемо!

Функция `Sprintf` (также из пакета `fmt`) в целом похожа на `Printf`, но она возвращает отформатированную строку, а не выводит ее.

```
resultString := fmt.Sprintf("About one-third: %0.2f\n", 1.0/3.0)
fmt.Printf(resultString)
```

```
About one-third: 0.33
```

Похоже, функции `Printf` и `Sprintf` *могут* помочь с выводом значений до нужного количества разрядов. Вопрос в том, *как* это сделать? Прежде всего, для эффективного использования функции `Printf` необходимо познакомиться с двумя ее возможностями:

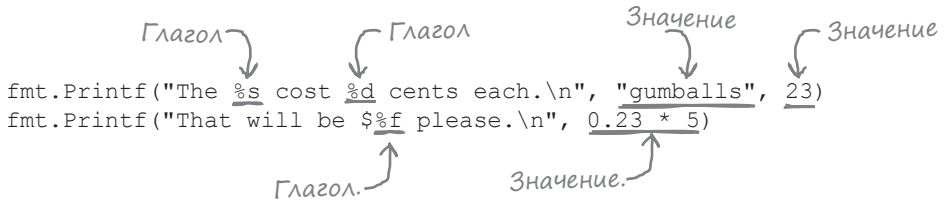
- «Глаголы» форматирования (такие, как `%0.2f` в приведенных строках).
- Ширина значений (`0.2` в середине глагола).

На нескольких ближайших страницах мы подробно объясним смысл этих аргументов `Printf`.

Да, мы понимаем, что эти вызовы выглядят немного странно. Мы приведем множество примеров, которые помогут прояснить ситуацию.

Глаголы форматирования

В первом аргументе `Printf` передается строка, которая будет использоваться для форматирования вывода. Большая часть вывода форматируется так, как он выглядит в строке. Однако знаки процента (`%`) рассматриваются как начало **глагола форматирования** — части строки, которая будет заменена значением в определенном формате. Остальные аргументы определяют значения, которые будут форматироваться с этими глаголами.




```
The gumballs cost 23 cents each.
That will be $1.150000 please.
```

Вскоре мы покажем, как решить эту проблему.

Буква, следующая за знаком процента, определяет тип глагола. Самые распространенные глаголы:

Глагол	Вывод
`%f`	Число с плавающей точкой
`%d`	Десятичное целое число
`%s`	Строка
`%t`	Логическое значение (`true` или `false`)
`%v`	Произвольное значение (подходящий формат выбирается на основании типа передаваемого значения)
`%#v`	Произвольное значение, отформатированное в том виде, в котором оно отображается в коде Go
`%T`	Тип переданного значения (`int`, `string` и т. п.)
`%%`	Знак процента (литерал)

```
fmt.Printf("A float: %f\n", 3.1415)
fmt.Printf("An integer: %d\n", 15)
fmt.Printf("A string: %s\n", "hello")
fmt.Printf("A boolean: %t\n", false)
fmt.Printf("Values: %v %v %v\n", 1.2, "\t", true)
fmt.Printf("Values: %#v %#v %#v\n", 1.2, "\t", true)
fmt.Printf("Types: %T %T %T\n", 1.2, "\t", true)
fmt.Printf("Percent sign: %%\n")
```

```
A float: 3.141500
An integer: 15
A string: hello
A boolean: false
Values: 1.2     true
Values: 1.2 "\t" true
Types: float64 string bool
Percent sign: %
```

Обратите внимание: в конец каждой форматной строки включается символ новой строки в виде служебной последовательности `\n`. Дело в том, что в отличие от `Println`, `Printf` не добавляет символ новой строки автоматически.

Глаголы форматирования (продолжение)

Особого внимания заслуживает глагол формата `%#v`. Так как он выводит значения в том виде, в котором они отображаются в коде Go, `%#v` позволяет выводить значения, которые обычно остаются скрытыми в выводе. Например, в следующем примере `%#v` показывает пустую строку, символ табуляции и символ новой строки — все эти символы остаются невидимыми при выводе с `%v`. Глагол `%#v` еще встретится вам в книге!

```
fmt.Printf("%v %v %v", "", "\t", "\n")
fmt.Printf("%#v %#v %#v", "", "\t", "\n")
```

%v выводит все значения...

```
"" "\t" "\n"
```

...но только с %#v вы можете фактически их увидеть!

Форматирование значений ширины

Глагол форматирования `%f` предназначен для чисел с плавающей точкой. Используем его для форматирования количества краски.

Сюда вставляется значение с плавающей точкой.

Одно из значений, ранее вычисленных нашей программой.

```
fmt.Printf("%f liters needed\n", 1.8199999999999998)
```

```
1.820000 liters needed
```

Округленное, но еще большое значение!

На этот раз значение округляется до разумной точности, но в дробной части по-прежнему выводятся шесть цифр — для наших целей этого слишком много.

В подобных ситуациях глаголы форматирования позволяют задать *ширину* отформатированного значения.

Предположим, вы хотите отформатировать данные в форме текстовой таблицы. Необходимо позаботиться о том, чтобы отформатированное значение занимало минимальное количество столбцов, а столбцы правильно выравнивались.

Минимальная ширина может задаваться после знака процента в глаголе форматирования. Если аргумент, соответствующий этому глаголу, короче минимальной длины, значение дополняется пробелами до достижения минимальной ширины.

Первое поле должно иметь минимальную ширину 12 символов.

Для второго поля минимальная ширина не определена.

Выводим заголовки столбцов.

```
fmt.Printf("%12s | %s\n", "Product", "Cost in Cents")
fmt.Println("----------------------------")
fmt.Printf("%12s | %2d\n", "Stamps", 50)
fmt.Printf("%12s | %2d\n", "Paper Clips", 5)
fmt.Printf("%12s | %2d\n", "Tape", 99)
```

Выводим разделительную черту заголовка.

Снова минимальная ширина равна 12.

Минимальная ширина 2.

Дополняется пробелами!

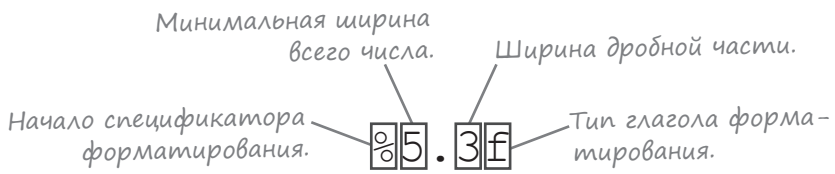
```
     Product | Cost in Cents
----------------------------
      Stamps | 50
 Paper Clips |  5
        Tape | 99
```

Без дополнения; значение уже заполняет минимальную ширину.

С дополнением!

Форматирование с дробными значениями ширины

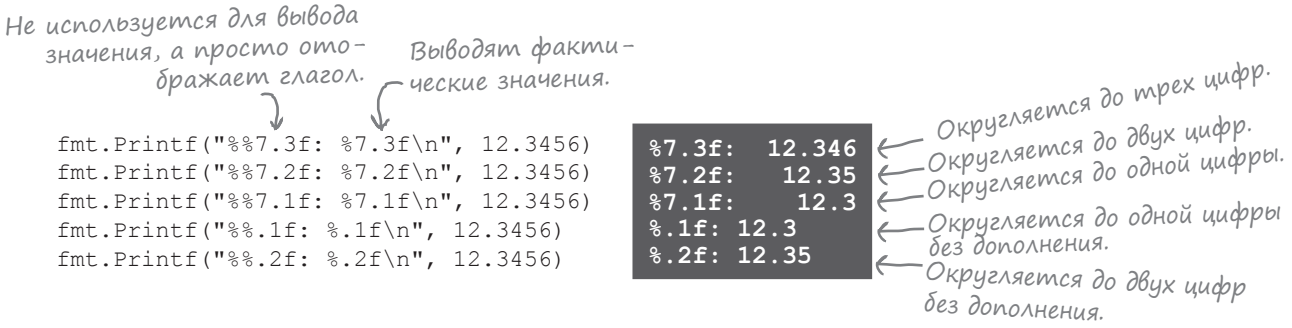
А теперь перейдем к той части, которая актуальна для нашей текущей задачи: ширина также может определять точность (количество выводимых цифр) в числах с плавающей точкой. Формат значения выглядит так:



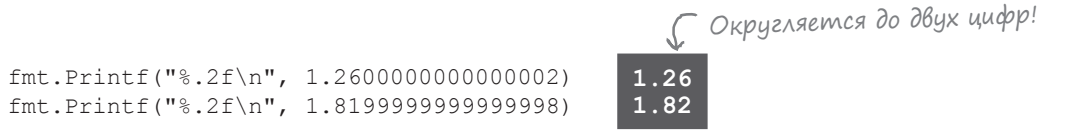
Минимальная ширина всего числа включает дробную часть и точку-разделитель. Если она указана, то более короткие числа будут дополняться пробелами в начале до достижения указанной ширины. Если она не указана, то пробелы не добавляются.

Ширина после точки определяет количество цифр в дробной части. Если выводимое число имеет больше разрядов, оно округляется (в большую или меньшую сторону) до заданного количества разрядов.

Вот как могут выглядеть различные значения ширины:



Последний формат `"%.2f"` позволяет взять число с плавающей точкой с произвольной точностью и округлить его до двух цифр в дробной части. (Также при этом число не дополняется лишними пробелами.) Попробуем применить его к излишне точным значениям в программе для расчета расхода краски.



Такой вывод воспринимается гораздо лучше. Похоже, функция `Printf` неплохо справляется с форматированием. Вернемся к программе для расчета расхода краски и применим в ней то, что вы узнали.

Использование Printf в программе

Глагол `Printf "%.2f"` округляет число с плавающей точкой до двух цифр в дробной части. Обновим программу для расчета расхода краски и используем в ней новую возможность.

```
// Директивы package и imports пропущены
func main() {
        var width, height, area float64
        width = 4.2
        height = 3.0
        area = width * height
        fmt.Printf("%.2f liters needed\n", area/10.0)
        width = 5.2
        height = 3.5
        area = width * height
        fmt.Printf("%.2f liters needed\n", area/10.0)
}
```

Значение форматируется и вставляется в строку.

Здесь происходит то же самое!

```
1.26 liters needed
1.82 liters needed
```

Округляется до двух цифр.

Наконец-то мы получили результат, который нормально выглядит! Мелкие неточности, появившиеся из-за использования вычислений с плавающей точкой, были исправлены при округлении.

Как-то неприятно обновлять код в двух местах. Если приходится вносить изменения, нужно помнить о том, чтобы обновить обе строки. И что произойдет, если появятся новые стены?

Верно подмечено. Go позволяет нам объявлять собственные функции, так что возможно, этот код следует вынести в функцию.

Как упоминалось в начале главы 1, функция представляет собой одну или несколько строк кода, которые могут вызываться из других мест программы. А в нашей программе встречаются две группы строк, которые почти не отличаются друг от друга:

Вычислить расход краски для первой стены.

```
var width, height, area float64
width = 4.2
height = 3.0
area = width * height
fmt.Printf("%.2f liters needed\n", area/10.0)
```

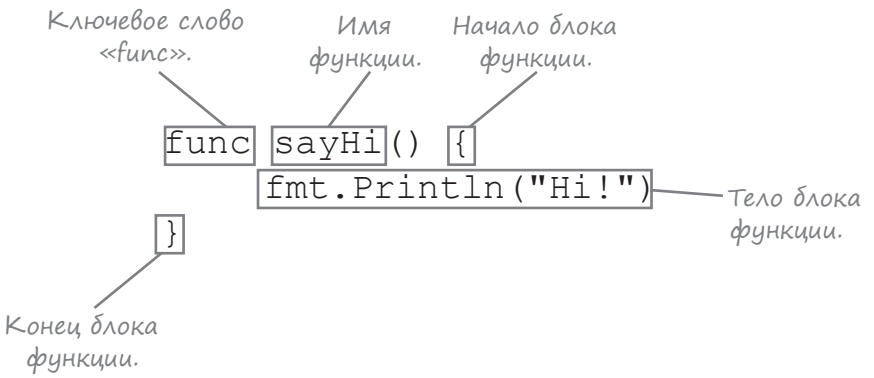
Вычислить расход краски для второй стены.

```
width = 5.2
height = 3.5
area = width * height
fmt.Printf("%.2f liters needed\n", area/10.0)
```

Нельзя ли преобразовать эти два фрагмента кода в одну функцию?

Объявление функций

Простое объявление функции может выглядеть так:



Объявление начинается с ключевого слова `func`, за которым следует имя функции, пара круглых скобок `()` и блок, содержащий код функции.

После того как функция будет объявлена, вы сможете вызывать ее в пакете — для этого достаточно указать ее имя и пару круглых скобок. При этом будет выполнен код функции.

Обратите внимание: при вызове `sayHi` перед именем функции не ставится имя пакета и точка. При вызове функции, определенной в текущем пакете, указывать имя пакета не нужно. (А вызов `main.sayHi()` приведет к ошибке компиляции.)

```
package main

import "fmt"

func sayHi() {
        fmt.Println("Hi!")
}

func main() {
        sayHi()
}
```

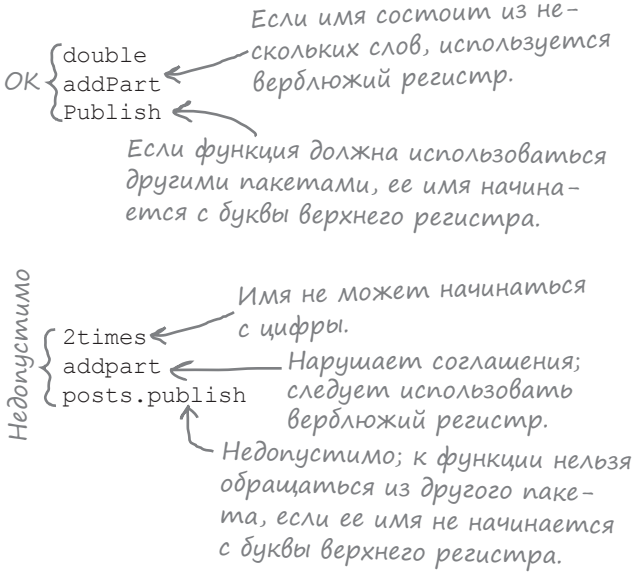
```
Hi!
```

Объявление функции «sayHi».

Вызываем «sayHi».

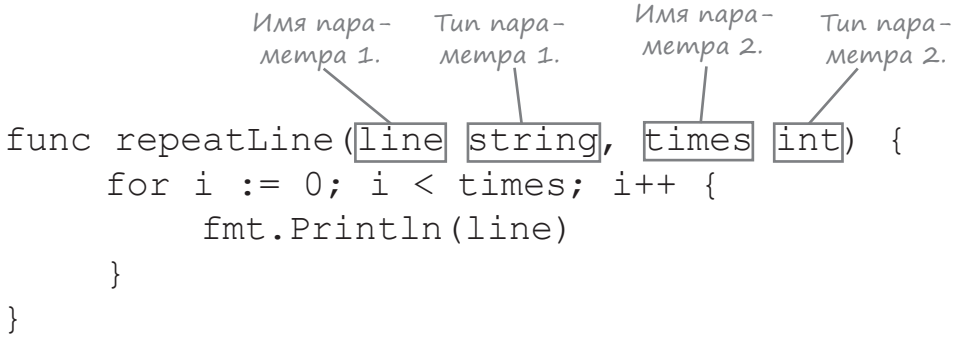
Имена функций подчиняются тем же правилам, что и имена переменных:

- Имя должно начинаться с буквы, за которой следует произвольное количество букв и цифр. (При нарушении этого правила выдается ошибка компиляции.)
- Функция, имя которой начинается с буквы верхнего регистра, *экспортируется* и может использоваться вне текущего пакета. Если функция должна использоваться только внутри текущего пакета, начните ее имя с буквы нижнего регистра.
- Имена, состоящие из нескольких слов, должны записываться в верблюжьем регистре.



Объявление параметров функции

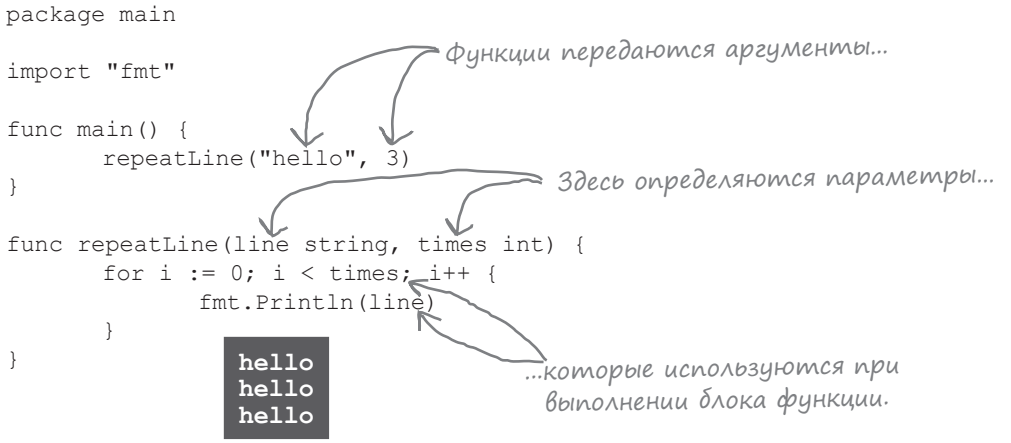
Если вы хотите, чтобы при вызове ваших функций передавались аргументы, необходимо объявить один или несколько параметров. **Параметром** называется переменная, локальная по отношению к функции, значение которой задается при вызове функции.



В объявлении функций в круглых скобках можно объявить один или несколько параметров, разделенных запятыми. Как и с любыми другими переменными, для каждого объявляемого параметра должно быть указано имя, за которым следует тип (`float64`, `bool` и т. д.).

Если функция имеет определенные параметры, нужно будет передать соответствующий набор аргументов при ее вызове. Когда функция запускается, каждому параметру присваивается копия значения в соответствующем аргументе. Эти значения параметров затем используются в функциональном блоке кода.

Параметр — переменная, локальная по отношению к функции, значение которой задается при вызове функции.



Использование функций в программе

Теперь вы знаете, как объявляются функции. Давайте посмотрим, как с помощью функций избавиться от повторений в нашей программе.

```
// Директивы package и imports пропущены
func main() {
    var width, height, area float64
    width = 4.2
    height = 3.0
    area = width * height
    fmt.Printf("%.2f liters needed\n", area/10.0)
    width = 5.2
    height = 3.5
    area = width * height
    fmt.Printf("%.2f liters needed\n", area/10.0)
}
```

Повторяющийся код!

Повторяющийся код!

```
1.26 liters needed
1.82 liters needed
```

Код для вычисления расхода краски выделяется в функцию с именем `paintNeeded`. Мы избавимся от двух наборов переменных `width` и `height` и оформим их в виде параметров функций. Затем в функции `main` функция `paintNeeded` будет вызываться для каждой стены, которую потребуется покрасить.

```
package main

import "fmt"

func paintNeeded(width float64, height float64) {
    area := width * height
    fmt.Printf("%.2f liters needed\n", area/10.0)
}

func main() {
    paintNeeded(4.2, 3.0)
    paintNeeded(5.2, 3.5)
    paintNeeded(5.0, 3.3)
}
```

Объявление функции с именем «paintNeeded».

Ширина стены передается в параметре.

Высота стены передается в другом параметре.

Ширина умножается на высоту, как и прежде.

Выводится расход краски, как и прежде.

Передается ширина.

Передается высота.

Вызываем новую функцию.

Понадобилось покрасить еще несколько стен? Добавьте еще несколько вызовов!

```
1.26 liters needed
1.82 liters needed
1.65 liters needed
```

Повторения кода исчезли из программы. А если мы захотим вычислить расход краски, необходимой для покраски еще нескольких стен, достаточно добавить еще несколько вызовов `paintNeeded`. Код стал чище и элегантнее!

Упражнение

Внизу приведен код программы с объявлением нескольких функций и последующим вызовом этих функций из `main`. Напишите, какой результат должна выводить программа.

(Мы записали первую строку за вас.)

```
package main

import "fmt"

func functionA(a int, b int) {
       fmt.Println(a + b)
}
func functionB(a int, b int) {
       fmt.Println(a * b)
}
func functionC(a bool) {
       fmt.Println(!a)
}
func functionD(a string, b int) {
       for i := 0; i < b; i++ {
              fmt.Print(a)
       }
       fmt.Println()
}

func main() {
       functionA(2, 3)
       functionB(2, 3)
       functionC(true)
       functionD("$", 4)
       functionA(5, 6)
       functionB(5, 6)
       functionC(false)
       functionD("ha", 3)
}
```

Результат:

5

Ответ на с. 145.

Функции и области видимости переменных

Функция `paintNeeded` объявляет переменную `area` в блоке функции:

```
func paintNeeded(width float64, height float64) {
    area := width * height
    fmt.Printf("%.2f liters needed\n", area/10.0)
}
```

Объявляем переменную «area».

Обращение к переменной.

Как и в случае с блоками условных команд и циклов, переменные, объявленные в блоке функции, остаются в области видимости только в границах блока функции. Таким образом, если вы попытаетесь обратиться к переменной `area` вне функции `paintNeeded`, компилятор сообщит об ошибке:

```
func paintNeeded(width float64, height float64) {
    area := width * height
    fmt.Printf("%.2f liters needed\n", area/10.0)
}

func main() {
    paintNeeded(4.2, 3.0)
    fmt.Println(area)
}
```

Ошибка. → **`undefined: area`**

Вне области видимости!

Но как и в случае с блоками условных команд и циклов переменные, объявленные *за пределами* функции, будут находиться в области видимости внутри этого блока. Это означает, что переменную можно объявить на уровне пакета и обращаться к ней из любой функции в этом пакете.

```
package main

import "fmt"

var metersPerLiter float64

func paintNeeded(width, height float64) float64 {
    area := width * height
    return area / metersPerLiter
}

func main() {
    metersPerLiter = 10.0
    fmt.Printf("%.2f", paintNeeded(4.2, 3.0))
}
```

Если переменная объявляется на уровне пакета...

...здесь она остается в области видимости.

...и здесь остается в области видимости.

`1.26`

Возвращаемые значения функций

Допустим, мы хотим вывести количество краски, необходимой для покрытия всех стен, которые мы собираемся покрасить. Сделать это с текущей версией функции `paintNeeded` не удастся; она лишь выводит результат, а потом отбрасывает его!

```
func paintNeeded(width float64, height float64) {
      area := width * height
      fmt.Printf("%.2f liters needed\n", area/10.0)
}
```

Выводит расход краски, но вычисленное значение будет потеряно!

Итак, пересмотрим функцию `paintNeeded`, чтобы она возвращала значение. Тогда при ее вызове можно будет вывести полученное значение, провести дополнительные вычисления или сделать что-то еще.

Функция всегда возвращает значение конкретного типа (и только этого типа). Чтобы объявить, что функция возвращает значение, добавьте тип возвращаемого значения после параметров в объявлении функции. Затем добавьте в блок функции ключевое слово `return`, за которым следует возвращаемое значение.

Тип возвращаемого значения.

```
func double(number float64) float64 {
      return number * 2
}
```

Ключевое слово return.

Возвращаемое значение.

После этого результат вызова функции можно присвоить переменной, передать другой функции или сделать что-то еще.

```
package main

import "fmt"

func double(number float64) float64 {
      return number * 2
}

func main() {
      dozen := double(6.0)
      fmt.Println(dozen)
      fmt.Println(double(4.2))
}
```

Возвращаемое значение присваивается переменной.

```
12
8.4
```

Возвращаемое значение передается другой функции.

Возвращаемые значения функций (продолжение)

При выполнении команды `return` функция немедленно возвращает управление, а следующий за ней код не выполняется. Ее можно использовать в сочетании с командой `if` для выхода из функции в том случае, если выполнение остального кода стало бессмысленным (из-за ошибки или другого условия).

```
func status(grade float64) string {
       if grade < 60.0 {
              return "failing"
       }
       return "passing"
}

func main() {
       fmt.Println(status(60.1))
       fmt.Println(status(59))
}
```

Если экзамен провален, немедленно вернуть.

Выполняется только в том случае, если grade >= 60.

```
passing
failing
```

Это означает, что при включении команды `return`, не являющейся частью блока `if`, какой-то код может не выполняться ни при каких условиях. Такая ситуация почти наверняка свидетельствует об ошибке в коде, поэтому Go помогает обнаруживать подобные ситуации: компилятор требует, чтобы любая функция с объявленным возвращаемым типом завершалась командой `return`. Если функция завершается любой другой командой, это приведет к ошибке компиляции.

```
func double(number float64) float64 {
       return number * 2
       fmt.Println(number * 2)
}
```

Функция всегда должна завершаться здесь...

А эта строка никогда не должна выполняться!

Ошибка →

```
missing return at end of function
```

Ошибка компиляции произойдет и в том случае, если тип возвращаемого значения не соответствует объявленному возвращаемому типу.

Должно возвращаться число с плавающей точкой...

```
func double(number float64) float64 {
       return int(number * 2)
}
```

...а возвращается целое число!

Ошибка →

```
cannot use int(number * 2) (type int)
as type float64 in return argument
```

Использование возвращаемого значения в программе

Итак, мы рассмотрели возможности использования возвращаемых значений функций. Давайте обновим нашу программу, чтобы она выводила общий объем необходимой краски вместе с расходом для каждой стены.

Изменим функцию `paintNeeded` так, чтобы она возвращала необходимый объем. Возвращаемое значение будет использоваться в функции `main` как для вывода расхода краски для текущей стены, так и для обновления переменной `total`, в которой накапливается общий расход краски.

```
package main

import "fmt"

func paintNeeded(width float64, height float64) float64 {
        area := width * height
        return area / 10.0
}

func main() {
        var amount, total float64
        amount = paintNeeded(4.2, 3.0)
        fmt.Printf("%0.2f liters needed\n", amount)
        total += amount
        amount = paintNeeded(5.2, 3.5)
        fmt.Printf("%0.2f liters needed\n", amount)
        total += amount
        fmt.Printf("Total: %0.2f liters\n", total)
}
```

Объявляем, что paintNeeded возвращает число с плавающей точкой.

Функция возвращает расход краски вместо того, чтобы выводить его.

Объявляем переменные для хранения расхода краски для текущей стены, а также для общего расхода по всем стенам.

Вызываем paintNeeded и сохраняем возвращаемое значение.

Выводим расход для первой стены.

Прибавляем расход для текущий стены к total.

Эти действия повторяются для второй стены.

Выводим общий расход по всем стенам.

```
1.26 liters needed
1.82 liters needed
Total: 3.08 liters
```

Работает! Возвращение значения позволяет функции `main` самостоятельно решить, что делать с вычисленной величиной; она не полагается на то, что значение будет выведено функцией `paintNeeded`.

Сломай и изучи!

Ниже приведена обновленная версия функции `paintNeeded` с возвращаемым значением. Внесите одно из указанных изменений в программу и попробуйте ее откомпилировать; затем отмените изменение и переходите к следующему. Посмотрите, что из этого выйдет!

```
func paintNeeded(width float64, height float64) float64 {
       area := width * height
       return area / 10.0
}
```

Если...	...программа не будет работать, потому что...

Удалить команду `return`:
`func paintNeeded(width float64, height float64) float64 {`
`  area := width * height`
`  ~~return area / 10.0~~`
`}`
	Если функция объявляет возвращаемый тип, Go требует, чтобы она содержала команду `return`

Добавить строку *после* команды `return`:
`func paintNeeded(width float64, height float64) float64 {`
`  area := width * height`
`  return area / 10.0`
`  fmt.Println(area / 10.0)`
`}`
	Если функция объявляет возвращаемый тип, Go требует, чтобы ее последней командой была команда `return`

Удалить объявление возвращаемого типа:
`func paintNeeded(width float64, height float64) ~~float64~~ {`
`  area := width * height`
`  return area / 10.0`
`}`
	Go не позволяет вернуть значение, тип которого не был объявлен

Изменить тип возвращаемого значения:
`func paintNeeded(width float64, height float64) float64 {`
`  area := width * height`
`  return int(area / 10.0)`
`}`
	Go требует, чтобы тип возвращаемого значения соответствовал объявленному типу

Функции paintNeeded нужна обработка ошибок

Ваша функция `paintNeeded` отлично работает... чаще всего. Но один из пользователей недавно случайно передал ей отрицательное число и получил **отрицательный** расход краски!

Если случайно передать отрицательное число...

```
func main() {
        amount := paintNeeded(4.2, -3.0)
        fmt.Printf("%0.2f liters needed\n", amount)
}

func paintNeeded(width float64, height float64) float64 {
        area := width * height
        return area / 10.0
}
```

*4.2 * -3.0 равно -12.6!*

-12.6 / 10.0 равно -1.26!

```
-1.26 liters needed
```

Похоже, функция `paintNeeded` и не подозревает, что переданный ей аргумент недействителен. Она просто идет напролом, использует этот аргумент в своих вычислениях и возвращает недействительный результат. Это создает проблемы — даже если вы знаете, где можно приобрести отрицательный объем краски, захочется ли вам использовать такую краску у себя дома? Необходимо обнаруживать недопустимые значения аргументов и сообщать об ошибке.

В главе 2 были продемонстрированы функции, которые в дополнение к основному возвращаемому значению также возвращают второе значение — признак ошибки. Например, функция `strconv.Atoi` пытается преобразовать строку в целое число. Если преобразование прошло успешно, возвращается значение ошибки `nil`, означающее, что программа может продолжать работу. Но если значение ошибки *не равно* `nil`, значит, строку невозможно преобразовать в число. В таком случае мы решили вывести значение ошибки и прервать выполнение программы.

Если обнаружена ошибка, вывести сообщение и прервать работу программы.

```
guess, err := strconv.Atoi(input)
if err != nil {
        log.Fatal(err)
}
```

Входная строка преобразуется в целое число.

Если мы хотим, чтобы при вызове функции `paintNeeded` происходило то же самое, необходимо реализовать две возможности:

- Создание значения, представляющего ошибку.
- Возвращение дополнительного значения из функции `paintNeeded`.

А теперь разберемся, как это делается!

Значения ошибок

Прежде чем вернуть значение ошибки из функции `paintNeeded`, необходимо его как-то получить. Значение ошибки представляет собой любое значение с методом `Error`, который возвращает строку. Чтобы создать такое значение, проще всего передать строку функции `New` из пакета `errors` — функция создаст и вернет новое значение ошибки. Если вызвать для полученного значения метод `Error`, вы получите строку, переданную `errors.New`.

```
package main

import (
       "errors"
       "fmt"
)

func main() {
       err := errors.New("height can't be negative")
       fmt.Println(err.Error())
}
```

Создаем новое значение ошибки.

Возвращает сообщение об ошибке.

```
height can't be negative
```

Но если значение ошибки передается функции из пакета `fmt` или `log`, скорее всего, вам не нужно будет вызвать его метод `Error`. Функции пакетов `fmt` и `log` были написаны так, что они проверяют, содержит ли переданное им значение метод `Error`, и если содержит — выводят возвращаемое значение `Error`.

```
err := errors.New("height can't be negative")
fmt.Println(err)
log.Fatal(err)
```

Выводит сообщение об ошибке.

Тоже выводит сообщение об ошибке, после чего завершает программу.

```
height can't be negative
2018/03/12 19:49:27 height can't be negative
```

Если вам потребуется отформатировать числа или другие значения для использования в сообщениях об ошибках, воспользуйтесь функцией `fmt.Errorf`. Функция вставляет значения в форматную строку так же, как это делает `fmt.Printf` или `fmt.Sprintf`, но вместо вывода или возвращения строки она возвращает значение ошибки.

Вставляется число с плавающей точкой, округленное до двух цифр в дробной части.

Возвращает значение ошибки.

Выводит сообщение об ошибке.

Также выводит сообщение об ошибке.

```
err := fmt.Errorf("a height of %0.2f is invalid", -2.33333)
fmt.Println(err.Error())
fmt.Println(err)
```

```
a height of -2.33 is invalid
a height of -2.33 is invalid
```

Объявление нескольких возвращаемых значений

Затем необходимо как-то объявить, что функция `paintNeeded` наряду с расходом краски будет возвращать значение ошибки.

Чтобы объявить функцию с несколькими возвращаемыми значениями, заключите типы возвращаемых значений во *второй* набор круглых скобок в объявлении функции (после круглых скобок с параметрами), разделяя их запятыми. (Круглые скобки вокруг возвращаемых значений можно опустить, если функция возвращает всего одно значение, но с несколькими возвращаемыми значениями они обязательны.)

После этого при вызове функции следует помнить о дополнительных возвращаемых значениях — обычно они присваиваются дополнительным переменным.

```
package main

import "fmt"

func manyReturns() (int, bool, string) {
       return 1, true, "hello"
}

func main() {
       myInt, myBool, myString := manyReturns()
       fmt.Println(myInt, myBool, myString)
}
```

Функция возвращает целое число, логическое значение и строку.

Каждое возвращаемое значение сохраняется в переменной.

```
1 true hello
```

Вы также можете задать имя для каждого возвращаемого значения (по аналогии с именами параметров), если это поможет лучше понять их предназначение. Именованные возвращаемые значения в основном служат документацией для программистов, читающих код.

```
package main

import (
       "fmt"
       "math"
)

func floatParts(number float64) (integerPart int, fractionalPart float64) {
       wholeNumber := math.Floor(number)
       return int(wholeNumber), number - wholeNumber
}

func main() {
       cans, remainder := floatParts(1.26)
       fmt.Println(cans, remainder)
}
```

Имя для первого возвращаемого значения.

Имя для второго возвращаемого значения.

```
1 0.26
```

Использование множественных возвращаемых значений с функцией paintNeeded

Как упоминалось на предыдущей странице, функция может возвращать несколько значений любых типов. Впрочем, чаще всего эта возможность используется для возвращения основного значения, за которым следует дополнительное значение, указывающее, обнаружила ли функция ошибку. Дополнительному значению обычно присваивается `nil`, если выполнение прошло без проблем, или значение ошибки, если возникла ошибка.

Мы также будем следовать этим соглашениям в своей функции `paintNeeded`. В объявлении функции будет указано, что она возвращает два значения, `float64` и `error`. (Значения ошибок имеют тип `error`.) В блоке функции мы прежде всего проверяем параметры. Если хотя бы один из параметров `width` или `height` меньше 0, функция возвращает расход краски `0` (это значение бессмысленно, но что-то надо вернуть) и значение ошибки, сгенерированное вызовом `fmt.Errorf`. Проверка ошибок в начале функции позволяет легко пропустить остальной код функции вызовом `return` при возникновении проблем.

Если значения параметров допустимы, мы переходим к вычислению и возвращению расхода краски, как и прежде. В коде функции встречается только одно отличие: наряду с расходом краски возвращается второе значение `nil`, указывающее на отсутствие ошибок.

```
package main

import "fmt"

func paintNeeded(width float64, height float64) (float64, error) {
        if width < 0 {
                return 0, fmt.Errorf("a width of %0.2f is invalid", width)
        }
        if height < 0 {
                return 0, fmt.Errorf("a height of %0.2f is invalid", height)
        }
        area := width * height
        return area / 10.0, nil
}

func main() {
        amount, err := paintNeeded(4.2, -3.0)
        fmt.Println(err)
        fmt.Printf("%0.2f liters needed\n", amount)
}
```

Возвращаемое значение с расходом краски, как и прежде.

Второе возвращаемое значение сообщает, возникли ли ошибки при выполнении.

Если ширина имеет недопустимое значение, вернуть 0 и признак ошибки.

Если высота имеет недопустимое значение, вернуть 0 и признак ошибки.

Возвращает расход краски и значение «nil», которое указывает на отсутствие ошибок.

Добавляем вторую переменную для второго возвращаемого значения.

Выводим значение ошибки (или «nil», если ошибки не было).

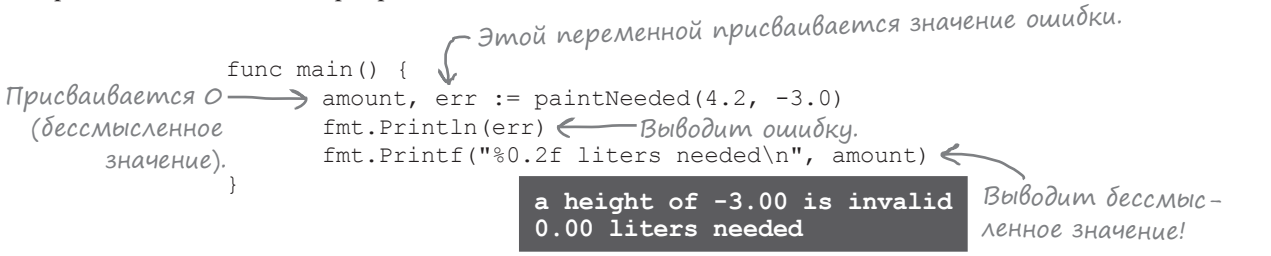
```
a height of -3.00 is invalid
0.00 liters needed
```

В функцию `main` была добавлена вторая переменная для хранения значения ошибки из `paintNeeded`. Программа выводит ошибку (если она есть), а затем расход краски.

Если передать `paintNeeded` недействительный аргумент, функция вернет значение ошибки, которое будет выведено программой. При этом также будет возвращен расход краски, равный 0. (Как упоминалось выше, при наличии ошибки это значение бессмысленно, но надо же *что-то* передать в первом возвращаемом значении.) В итоге программа выведет сообщение «0.00 liters needed»! С этим нужно что-то сделать...

Всегда обрабатывайте ошибки!

При передаче `paintNeeded` недопустимого аргумента вы получаете значение ошибки, которое выводится для просмотра пользователем. Но при этом мы также получаем (недействительный) расход краски, который тоже выводится программой!



Наряду со значением ошибки функция также обычно должна возвращать основное значение. Но любые другие значения, возвращаемые вместе со значением ошибки, следует считать ненадежными и игнорировать. Когда вы вызываете функцию, возвращающую значение ошибки, прежде всего необходимо убедиться в том, что это значение равно `nil`. Если значение отлично от `nil`, значит, в программе возникла ошибка, которую необходимо обработать.

Как именно должна обрабатываться ошибка — зависит от ситуации. Возможно, в случае с функцией `paintNeeded` лучше всего будет пропустить текущее вычисление и продолжить выполнение программы:

```
func main() {
        amount, err := paintNeeded(4.2, -3.0)
        if err != nil {
                fmt.Println(err)
        } else {
                fmt.Printf("%0.2f liters needed\n", amount)
        }
        // Дальнейшие вычисления...
}
```

Если значение ошибки не равно nil, в программе возникли проблемы...
...поэтому выводим ошибку.
В противном случае значение ошибки будет равно nil...
...и можно спокойно вывести полученный расход краски.

```
a height of -3.00 is invalid
```

Но в такой короткой программе вместо этого можно вызвать `log.Fatal`, чтобы вывести сообщение об ошибке и прервать выполнение.

```
func main() {
        amount, err := paintNeeded(4.2, -3.0)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Printf("%0.2f liters needed\n", amount)
}
```

Если значение ошибки не равно nil, в программе возникли проблемы...
...выводим ошибку и прерываем выполнение программы.
Этот код никогда не будет выполняться при возникновении ошибки.

```
2018/03/12 19:49:27 a height of -3.00 is invalid
```

Важно помнить, что возвращаемое значение всегда следует проверять, чтобы знать, *произошла ли* ошибка. А что делать с ошибкой, зависит от вас!

Сломай и изучи!

Ниже приведена программа для вычисления квадратного корня из числа. Но если передать функции `squareRoot` отрицательное число, она вернет ошибку. Внесите одно из указанных изменений и попробуйте откомпилировать программу; затем отмените изменение и переходите к следующему. Посмотрите, что из этого выйдет!

```
package main

import (
	"fmt"
	"math"
)

func squareRoot(number float64) (float64, error) {
	if number < 0 {
		return 0, fmt.Errorf("can't get square root of negative number")
	}
	return math.Sqrt(number), nil
}

func main() {
	root, err := squareRoot(-9.3)
	if err != nil {
		fmt.Println(err)
	} else {
		fmt.Printf("%0.3f", root)
	}
}
```

Если...	...программа не будет работать, потому что...
Удалить один из аргументов `return`: `return math.Sqrt(number)~~, nil~~`	Количество аргументов `return` всегда должно соответствовать количеству возвращаемых значений в объявлении функции
Удалить одну из переменных, которым присваиваются возвращаемые значения: `root~~, err~~ := squareRoot(-9.3)`	Если вы используете одно из возвращаемых значений функции, Go требует, чтобы вы использовали их все

Удалить код, использующий одно из возвращаемых значений:
`root, err := squareRoot(-9.3)`
`~~if err != nil {~~`
`~~fmt.Println(err)~~`
`~~} else {~~`
`fmt.Printf("%0.3f", root)`
`~~}~~`
	Go требует, чтобы каждая объявленная переменная использовалась в программе. И эта особенность весьма полезна для работы с возвращаемыми значениями, потому что она предотвращает случайное игнорирование ошибок

У бассейна

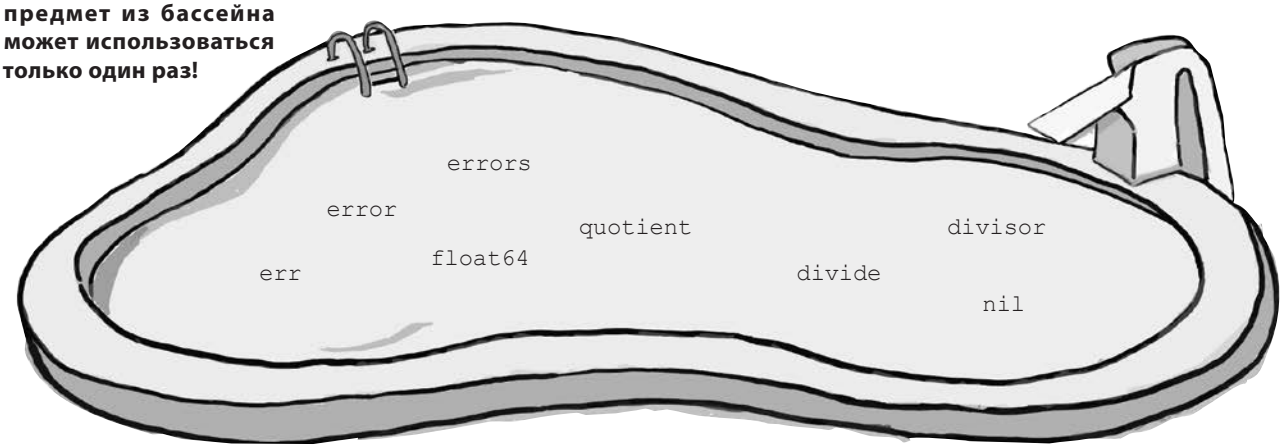
Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты необязательно. Ваша **задача**: построить код, который будет успешно выполняться и выведет показанный результат.

```
package main

import (
        "errors"
        "fmt"
)

func divide(dividend float64, divisor float64) (float64, ______) {
        if divisor == 0.0 {
                return 0, _________.New("can't divide by 0")
        }
        return dividend / divisor, _____
}

func main() {
        __________, _____ := divide(5.6, 0.0)
        if err != nil {
                fmt.Println(err)
        } else {
                fmt.Printf("%0.2f\n", quotient)
        }
}
```

Результат.

```
can't divide by 0
```

Примечание: каждый предмет из бассейна может использоваться только один раз!



Ответ на с. 146.

В параметрах функций хранятся копии аргументов

Как уже говорилось, при вызове функции с объявленными параметрами необходимо передать аргументы при вызове. Значение каждого аргумента *копируется* в соответствующую переменную-параметр. (Этот механизм в языках программирования иногда называют «передачей по значению».)

Во многих случаях этого достаточно. Но если вы хотите передать значение переменной функции, чтобы функция каким-то образом *изменила* его, возникает проблема. Функция может изменить только *копию* значения параметра, но не оригинал. Таким образом, любые изменения, внесенные внутри функции, не будут видны за ее пределами!

В Go используется механизм «передачи по значению»; в параметрах функций сохраняется копия аргументов, использованных при вызове функции.

Ниже приведена обновленная версия функции `double`, приведенной выше. Она получает число, умножает его на 2 и выводит результат. (Функция использует оператор `*=`, который работает аналогично `+=`, но умножает значение из переменной вместо того, чтобы выполнять сложение.)

```
package main

import "fmt"

func main() {
        amount := 6
        double(amount)
}

func double(number int) {
        number *= 2
        fmt.Println(number)
}
```

Аргумент передается функции.

В параметре сохраняется копия аргумента.

```
12
```

Выводит удвоенное значение.

Но предположим, вы хотите переместить команду вывода удвоенного значения из функции `double` в функцию, из которой она вызывается. Такое решение работать не будет, потому что `double` изменяет только *копию* значения. При попытке вывести значение в вызывающей функции будет выведено исходное значение, а не удвоенное!

```
func main() {
        amount := 6
        double(amount)
        fmt.Println(amount)
}

func double(number int) {
        number *= 2
}
```

Функции передается аргумент.

Выводится исходное значение!

Параметру присваивается копия аргумента.

```
6
```

Выводит исходное значение!

Изменяется скопированное значение, а не оригинал!

Как же добиться того, чтобы функция изменяла исходное значение, хранящееся в переменной, вместо его копии? Чтобы понять, как это делается, необходимо еще ненадолго отвлечься от основной темы и познакомиться с *указателями*.

Указатели

Оператор & (амперсанд) используется в Go для получения *адреса* переменной. Например, следующий код инициализирует переменную, сначала выводит ее значение, а затем адрес переменной...

```
amount := 6
fmt.Println(amount)
fmt.Println(&amount)
```

Выводит значение переменной.

Выводит адрес переменной.

```
6
0x1040a124
```

Значение переменной.

Адрес переменной.

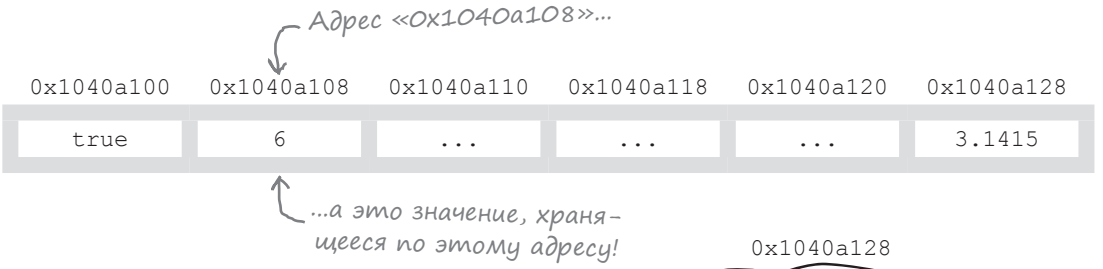
Адрес можно получить для переменной любого типа. Обратите внимание, все переменные имеют разные адреса.

```
var myInt int
fmt.Println(&myInt)
var myFloat float64
fmt.Println(&myFloat)
var myBool bool
fmt.Println(&myBool)
```

```
0x1040a128
0x1040a140
0x1040a148
```

И что же собой представляют эти «адреса»? Чтобы найти конкретный дом в плотно застроенном городе, вам нужно знать его адрес...

Память, выделяемая компьютером программе, так же переполнена, как и городские улицы. Она забита значениями переменных: логическими значениями, целыми числами, строками и т. д. Зная адрес переменной, вы сможете воспользоваться им для получения значения, хранящегося в переменной.



Значения, представляющие адреса переменных, называются **указателями**, потому что они *указывают* на область памяти, в которой хранится переменная.

Типы указателей

Тип указателя состоит из знака * и типа переменной, на которую ссылается указатель. Например, тип указателя на переменную int записывается в виде *int (читается «указатель на int»).

С помощью функции reflect.TypeOf можно вывести типы указателей из приведенной ранее программы:

```
package main

import (
       "fmt"
       "reflect"
)

func main() {
       var myInt int
       fmt.Println(reflect.TypeOf(&myInt))
       var myFloat float64
       fmt.Println(reflect.TypeOf(&myFloat))
       var myBool bool
       fmt.Println(reflect.TypeOf(&myBool))
}
```

Получает указатель на myInt и выводит тип указателя.

Получает указатель на myFloat и выводит тип указателя.

Получает указатель на myBool и выводит тип указателя.

Типы указателей.

```
*int
*float64
*bool
```

В программе можно объявлять переменные, содержащие указатели. В таких переменных могут храниться только указатели на один конкретный тип переменной, так что переменная может содержать только указатели *int, только указатели *float64 и т. д.

```
var myInt int
var myIntPointer *int
myIntPointer = &myInt
fmt.Println(myIntPointer)

var myFloat float64
var myFloatPointer *float64
myFloatPointer = &myFloat
fmt.Println(myFloatPointer)
```

Объявление переменной, содержащей указатель на int.

Указатель присваивается переменной.

Объявление переменной для хранения указателя на float64.

Указатель присваивается переменной.

```
0x1040a128
0x1040a140
```

Как и с другими типами, при немедленном присваивании исходного значения переменной-указателю можно воспользоваться коротким объявлением переменной.

```
var myBool bool
myBoolPointer := &myBool
fmt.Println(myBoolPointer)
```

Короткое объявление переменной-указателя.

```
0x1040a148
```

Чтение или изменение значения по указателю

Чтобы получить значение переменной, на которую ссылается указатель, поставьте оператор * прямо перед указателем в коде. Например, чтобы получить значение переменной по указателю `myIntPointer`, введите `*myIntPointer`. (Не существует официальной точки зрения на то, как читать конструкции с *; мы предпочитаем формулировку «значение по адресу», так что `*myIntPointer` читается как «значение по адресу `myIntPointer`».)

```
myInt := 4
myIntPointer := &myInt
fmt.Println(myIntPointer)
fmt.Println(*myIntPointer)

myFloat := 98.6
myFloatPointer := &myFloat
fmt.Println(myFloatPointer)
fmt.Println(*myFloatPointer)

myBool := true
myBoolPointer := &myBool
fmt.Println(myBoolPointer)
fmt.Println(*myBoolPointer)
```

Выводится сам указатель.

Выводится значение, на которое ссылается указатель.

Выводится сам указатель.

Выводится значение, на которое ссылается указатель.

Выводится сам указатель.

Выводится значение, на которое ссылается указатель.

```
0x1040a124
4
0x1040a140
98.6
0x1040a150
true
```

Оператор * может использоваться для обновления значения по указателю.

```
myInt := 4
fmt.Println(myInt)
myIntPointer := &myInt
*myIntPointer = 8
fmt.Println(*myIntPointer)
fmt.Println(myInt)
```

Новое значение присваивается переменной, на которую ссылается указатель (myInt).

Выводится значение переменной, на которую ссылается указатель.

Значение переменной выводится напрямую.

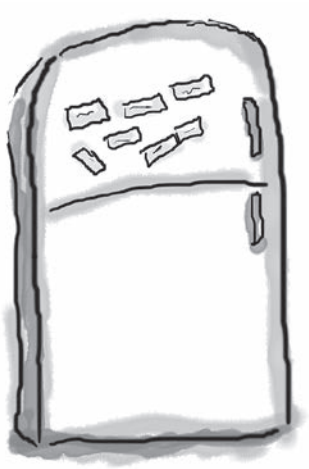
```
4
8
8
```

Исходное значение myInt.

*Результат обновления *myIntPointer.*

*Обновление значения myInt (то же, что *myIntPointer).*

В приведенном коде команда `*myIntPointer = 8` обращается к переменной, на которую ссылается указатель `myIntPointer` (то есть переменной `myInt`) и присваивает ей новое значение. Таким образом, обновляется не только значение `*myIntPointer`, но и `myInt`.



Развлечения с магнитами

На холодильнике была выложена программа Go, в которой используется переменная-указатель. Расставьте фрагменты кода, чтобы создать работоспособную программу, которая выводит указанный результат.

Программа должна объявлять целочисленную переменную `myInt` и переменную `myIntPointer` для хранения целочисленного указателя. Затем она должна присваивать значение `myInt` и указатель на `myInt` как значение `myIntPointer`. Наконец, программа должна выводить значение по указателю `myIntPointer`.

```
package main

import "fmt"

func main() {

    Добавьте свой код!

}
```

Результат.

```
42
```

Дополнительные магниты. Добавьте их в приведенную программу!

```
var   var   myInt   myInt   myInt   42
int   int   myIntPointer   myIntPointer   myIntPointer
=   =   &   *   *   fmt.Println(   )
```

Ответ на с. 146.

Использование указателей с функциями

Указатели также можно возвращать из функций; просто объявите возвращаемый тип функции как тип указателя.

```
func createPointer() *float64 {
       var myFloat = 98.5
       return &myFloat
}

func main() {
       var myFloatPointer *float64 = createPointer()
       fmt.Println(*myFloatPointer)
}
```

Объявляем, что функция возвращает указатель на float64.

Возвращается указатель заданного типа.

Назначает возвращенный указатель переменной.

Выводим значение, на которое ссылается указатель.

```
98.5
```

Кстати говоря, в отличие от некоторых других языков, в Go можно вернуть указатель на переменную, локальную по отношению к функции. И хотя эта переменная уже не находится в области видимости, пока у вас есть указатель, Go предоставит вам доступ к значению этой переменной.

Указатели также можно передавать функциям в аргументах. Для этого достаточно указать, что один или несколько параметров имеют тип указателя.

```
func printPointer(myBoolPointer *bool) {
       fmt.Println(*myBoolPointer)
}

func main() {
       var myBool bool = true
       printPointer(&myBool)
}
```

Для этого параметра используется тип указателя.

Выводим значение, на которое ссылается переданный указатель.

```
true
```

Передаем указатель функции.

Проследите за тем, чтобы в аргументах передавались именно указатели (если это соответствует объявлению). Если функция ожидает получить указатель, а вы попытаетесь передать ей значение, вы получите ошибку компиляции.

```
func main() {
       var myBool bool = true
       printPointer(myBool)
}
```

Ошибка.

```
cannot use myBool (type bool)
as type *bool in argument
to printPointer
```

Теперь вы освоили основные возможности использования указателей в Go. Пора вернуться к основной теме и исправить функцию `double`!

Исправление функции «double» с использованием указателей

У нас есть функция `double`, которая принимает значение типа `int` и умножает его на 2. Мы хотим передавать это значение и удваивать его. Но, как мы узнали, Go является языком передачи по значению, то есть параметры функции получают копию любых аргументов от вызывающей стороны. Наша функция удваивает копию значения и оставляет оригинал нетронутым!

```
func main() {
       amount := 6
       double(amount)
       fmt.Println(amount)
}

func double(number int) {
       number *= 2
}
```

Передает аргумент функции.

Выводит исходное значение!

Параметр настраивает копию аргумента.

Изменяет скопированное значение, а не исходное!

```
6
```

Выводится исходное значение!

Здесь-то нам и пригодится отступление от темы. Если передать указатель функции, а затем изменить значение, на которое он ссылается, изменения будут действовать и за пределами функции!

Для этого необходимо внести в программу несколько незначительных изменений. В функции `double` необходимо обновить тип параметра `number` — заменить `int` на `*int`. Затем нужно будет изменить код функции, чтобы она изменяла значение по указателю (вместо прямого обновления переменной). Наконец, в функции `main` необходимо обновить вызов `double`, чтобы в нем передавался указатель вместо значения.

```
func main() {
       amount := 6
       double(&amount)
       fmt.Println(amount)
}

func double(number *int) {
       *number *= 2
}
```

Вместо значения переменной передается указатель на нее.

Вместо значения int получает указатель.

Обновляем значение, на которое ссылается указатель.

```
12
```

Выводится удвоенное значение.

При выполнении обновленного кода функции `double` будет передаваться указатель на переменную `amount`. Функция `double` получает значение по указателю и удваивает его, что приводит к изменению значения в переменной `amount`. А когда управление снова передается в функцию `main` и последняя выводит значение `amount`, вы увидите удвоенное значение!

В этой главе вы узнали, как написать собственную функцию. Возможно, сейчас не до конца понятно, для чего нужны некоторые из представленных возможностей. Не беспокойтесь — вскоре наши программы станут сложнее, и мы начнем эффективно пользоваться всем, что узнали.

Упражнение

Ниже приведен код функции `negate`, которая *должна* обновлять значение переменной `truth` и заменять его противоположным (`false`), а также обновлять значение переменной `lies` и заменять его противоположным (`true`). Но если вызвать `negate` для переменных `truth` и `lies` и вывести их значения, становится ясно, что они не изменились!

```
package main

import "fmt"

func negate(myBoolean bool) bool {
       return !myBoolean
}

func main() {
       truth := true
       negate(truth)
       fmt.Println(truth)
       lies := false
       negate(lies)
       fmt.Println(lies)
}
```

Реальный результат.

```
true
false
```

Заполните пропуски в коде, чтобы функция `negate` получала указатель на логическое значение (вместо того, чтобы получать логическое значение напрямую), а затем обновляла значение, на которое ссылается этот указатель, обратным значением. Не забудьте изменить вызовы `negate`, чтобы вместо непосредственного значения передавался указатель!

```
package main

import "fmt"

func negate(myBoolean _______) {
       ___________ = !___________
}

func main() {
       truth := true
       negate(_______)
       fmt.Println(truth)
       lies := false
       negate(_______)
       fmt.Println(lies)
}
```

Нужный результат.

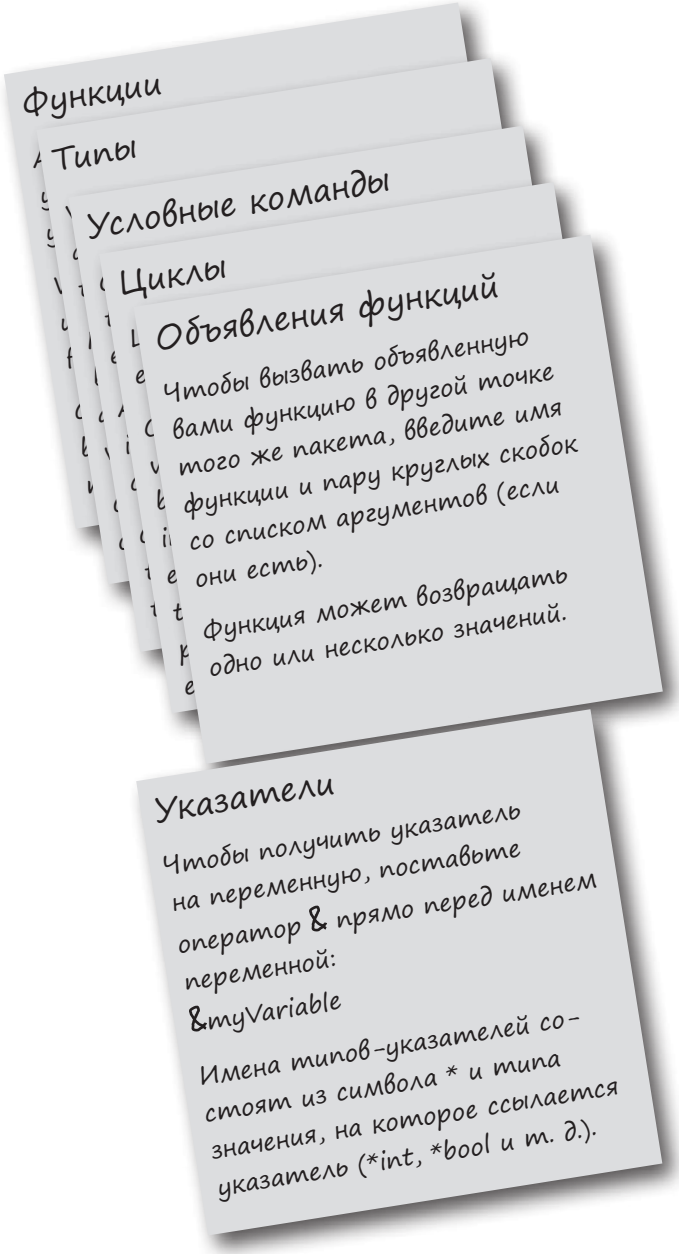
```
false
true
```

Ответ на с. 146.

Ваш инструментарий Go

Глава 3 осталась позади! В ней ваш инструментарий пополнился объявлениями функций и указателями.

ГЛАВА 3



КЛЮЧЕВЫЕ МОМЕНТЫ

- Функции `fmt.Printf` и `fmt.Sprintf` форматируют переданные им значения. В первом аргументе передается форматная строка с **глаголами** (`%d`, `%f`, `%s` и т. д.), на место которых подставляются отформатированные значения.
- Глагол форматирования может содержать **ширину**: минимальное количество символов, которые будет занимать отформатированное значение. Например, глагол `%12s` определяет строку из 12 символов (дополненную пробелами), `%2d` — целое число из двух цифр, а `%.3f` — число с плавающей точкой, округленное до трех цифр в дробной части.
- Если вы хотите, чтобы при вызовах вашей функции передавались аргументы, необходимо объявить один или несколько **параметров** (с указанием типа каждого параметра) в объявлении функции. Количество и тип аргументов всегда должны соответствовать количеству и типу параметров — в противном случае вы получите ошибку компиляции.
- Если вы хотите, чтобы ваша функция возвращала одно или несколько значений, объявите типы возвращаемых значений в объявлении функции.
- К переменным, объявленным внутри функции, невозможно обратиться вне этой функции. С другой стороны, внутри функции можно обращаться к переменным, объявленным за пределами функции (обычно на уровне пакета).
- Если функция возвращает несколько значений, последнее значение обычно имеет тип `error`. Значения ошибок содержат метод `Error()`, который возвращает строку с описанием ошибки.
- По умолчанию функции возвращают значение ошибки `nil`, указывающее на отсутствие ошибок.
- Чтобы обратиться к значению, на которое ссылается указатель, поставьте `*` перед именем: `*myPointer`.
- Если функция получает указатель в параметре и обновляет значение, на которое ссылается указатель, такое изменение будет видимо за пределами функции.

Упражнение
Решение

Внизу приведен код программы с объявлением нескольких функций и последующим вызовом этих функций из `main`. Напишите, какой результат должна выводить программа.

```
package main

import "fmt"

func functionA(a int, b int) {
       fmt.Println(a + b)
}
func functionB(a int, b int) {
       fmt.Println(a * b)
}
func functionC(a bool) {
       fmt.Println(!a)
}
func functionD(a string, b int) {
       for i := 0; i < b; i++ {
              fmt.Print(a)
       }
       fmt.Println()
}

func main() {
       functionA(2, 3)
       functionB(2, 3)
       functionC(true)
       functionD("$", 4)
       functionA(5, 6)
       functionB(5, 6)
       functionC(false)
       functionD("ha", 3)
}
```

Результат:

5

6

false

$$$$

11

30

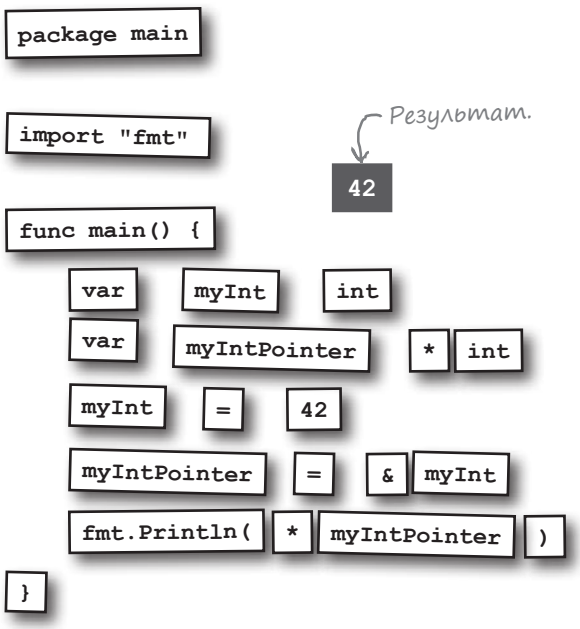
true

hahaha

У бассейна. Решение

```
package main

import (
	"errors"
	"fmt"
)

func divide(dividend float64, divisor float64) (float64, error) {
	if divisor == 0.0 {
		return 0, errors.New("can't divide by 0")
	}
	return dividend / divisor, nil
}

func main() {
	quotient, err := divide(5.6, 0.0)
	if err != nil {
		fmt.Println(err)
	} else {
		fmt.Printf("%0.2f\n", quotient)
	}
}
```

Развлечения с магнитами. Решение




```
package main

import "fmt"

func negate(myBoolean *bool) {
	*myBoolean = !*myBoolean
}

func main() {
	truth := true
	negate(&truth)
	fmt.Println(truth)
	lies := false
	negate(&lies)
	fmt.Println(lies)
}
```

Пакеты

Время навести порядок! До сих пор мы сваливали весь свой код в один файл. Но чем больше и сложнее будут становиться наши программы, тем скорее такой подход приведет к хаосу. В этой главе мы научим вас создавать **пакеты** для хранения взаимосвязанного кода в одном месте. Но пакеты нужны не только для организации кода. Пакеты предоставляют простые средства для *повторного использования кода в разных программах*. А еще это простой способ *распространения кода среди разработчиков*.

Разные программы, одна функция

Мы написали две программы, содержащие одинаковые копии функции. И это быстро создает проблемы с сопровождением... На этой странице приведена новая версия программы *pass_fail.go* из главы 2. Код ввода значения с клавиатуры был перемещен в новую функцию `getFloat`. Эта функция возвращает число с плавающей точкой, введенное пользователем, если только при вводе не произошла ошибка — в таком случае функция возвращает `0` и значение ошибки. Если функция вернула ошибку, программа сообщает о ней и завершается; в противном случае она, как и прежде, сообщает, сдал пользователь экзамен или нет.


pass_fail.go

```
// pass_fail сообщает, сдал ли пользователь экзамен.
package main

import (
        "bufio"
        "fmt"
        "log"
        "os"
        "strconv"
        "strings"
)

func getFloat() (float64, error) {
        reader := bufio.NewReader(os.Stdin)
        input, err := reader.ReadString('\n')
        if err != nil {
                return 0, err
        }

        input = strings.TrimSpace(input)
        number, err := strconv.ParseFloat(input, 64)
        if err != nil {
                return 0, err
        }
        return number, nil
}

func main() {
        fmt.Print("Enter a grade: ")
        grade, err := getFloat()
        if err != nil {
                log.Fatal(err)
        }

        var status string
        if grade >= 60 {
                status = "passing"
        } else {
                status = "failing"
        }
        fmt.Println("A grade of", grade, "is", status)
}
```

Не отличается от функции getFloat на следующей странице!

Почти не отличается от кода из главы 2, кроме одного...

...если при чтении ввода произошла ошибка, она будет возвращена функцией.

Также возвращаются ошибки, которые могут возникнуть при преобразовании строки в float64.

Вызываем getFloat для получения grade...

Если функция вернула ошибку, программа сообщает об этом и завершается.

Не отличается от кода из главы 2.

```
Enter a grade: 89.7
A grade of 89.7 is passing
```

Разные программы, одна функция (продолжение)

На этой странице приведена новая программа *tocelsius.go*, которая запрашивает у пользователя температуру в градусах по Фаренгейту и преобразует введенное значение в градусы по Цельсию.

Обратите внимание: функция `getFloat` в *tocelsius.go* идентична функции `getFloat` из *pass_fail.go*.


```
// tocelsius преобразует температуру в градусах по Фаренгейту
// в градусы по Цельсию.
package main

import (
        "bufio"
        "fmt"
        "log"
        "os"
        "strconv"
        "strings"
)

func getFloat() (float64, error) {
        reader := bufio.NewReader(os.Stdin)
        input, err := reader.ReadString('\n')
        if err != nil {
                return 0, err
        }

        input = strings.TrimSpace(input)
        number, err := strconv.ParseFloat(input, 64)
        if err != nil {
                return 0, err
        }
        return number, nil
}

func main() {
        fmt.Print("Enter a temperature in Fahrenheit: ")
        fahrenheit, err := getFloat()
        if err != nil {
                log.Fatal(err)
        }
        celsius := (fahrenheit - 32) * 5 / 9
        fmt.Printf("%0.2f degrees Celsius\n", celsius)
}
```

Не отличается от функции getFloat на предыдущей странице!

Вызываем getFloat для получения температуры.

Если функция вернула ошибку, программа сообщает об этом и завершается.

Температура преобразуется к шкале Цельсия...

...и выводится с двумя знаками после точки.

```
Enter a temperature in Fahrenheit: 98.6
37.00 degrees Celsius
```

Пакеты и повторное использование кода в программах

И снова повторяющийся код... Если когда-нибудь в функции `getFloat` обнаружится ошибка, ее придется исправлять в двух местах. С другой стороны, это две разные программы, и наверное, с этим ничего не сделать...

```
func getFloat() (float64, error) {
        reader := bufio.NewReader(os.Stdin)
        input, err := reader.ReadString('\n')
        if err != nil {
                return 0, err
        }

        input = strings.TrimSpace(input)
        number, err := strconv.ParseFloat(input, 64)
        if err != nil {
                return 0, err
        }
        return number, nil
}
```

На самом деле кое-что сделать можно, а именно переместить общую функцию в новый пакет!

Go позволяет нам определять собственные пакеты. Как упоминалось в главе 1, пакет представляет собой блок кода, который выполняет похожие операции. Пакет `fmt` форматирует вывод, пакет `math` работает с числами, пакет `strings` работает со строками и т. д. Мы уже использовали функции из всех этих пакетов в своих программах.

Возможность повторного использования кода в разных программах — одна из основных причин для существования пакетов. Если какие-то части вашего кода используются в разных программах, возможно, их стоит переместить в пакеты.

Если какие-то части вашего кода используются в разных программах, возможно, их стоит переместить в пакеты.

Хранение кода пакетов

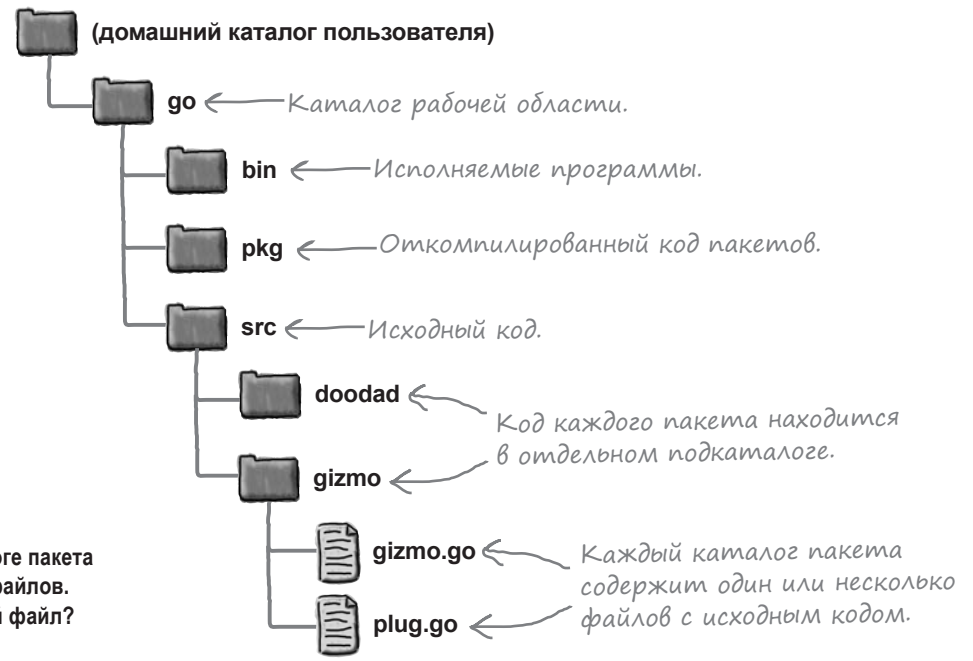
Инструменты Go ищут код пакетов в специальном каталоге (папке) на вашем компьютере, который называется **рабочей областью**. По умолчанию рабочей областью является каталог с именем *go* в домашнем каталоге текущего пользователя.

Каталог рабочей области содержит три подкаталога:

- *bin* для хранения откомпилированных двоичных исполняемых программ. (Каталог *bin* будет подробно описан позднее в этой главе.)
- *pkg* для хранения откомпилированных двоичных файлов пакетов. (Каталог *pkg* также будет описан в этой главе.)
- *src* для хранения исходного кода Go.

В каталоге *src* код каждого пакета размещается в отдельном подкаталоге. По соглашениям имя подкаталога должно совпадать с именем пакета (так что код пакета `gizmo` должен храниться в подкаталоге с именем *gizmo*).

Подкаталог каждого пакета должен содержать один или несколько файлов с исходным кодом. Имена файлов могут быть любыми, но они должны иметь расширение *.go*.



Часто Задаваемые Вопросы

В: Вы говорите, что в каталоге пакета могут находиться несколько файлов. Что должен содержать каждый файл?

О: Все что угодно! Весь код пакета можно хранить в одном файле, а можно разбить его на несколько файлов. В любом случае они станут частью одного пакета.

Создание нового пакета

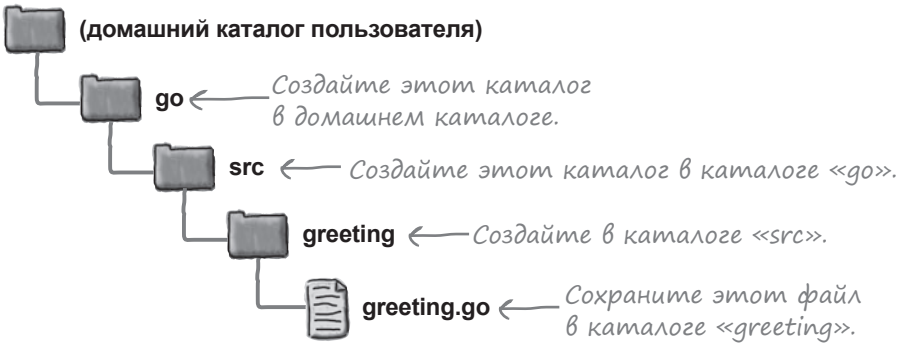
Попробуем создать пакет в рабочей области. Это будет простой пакет с именем `greeting`, который выводит приветствия на разных языках.

При установке Go каталог рабочей области не создается по умолчанию, так что вам придется создать его самостоятельно. Для начала перейдите в свой домашний каталог. (Путь имеет вид *C:\Users\<имя_пользователя>* в большинстве систем Windows, */Users/<имя_пользователя>* на Mac и */home/<имя_пользователя>* в большинстве систем Linux.) В домашнем каталоге создайте каталог с именем *go* — он станет новым каталогом рабочей области. Внутри каталога *go* создайте каталог с именем *src*.

Наконец, нам понадобится каталог для хранения кода пакета. По правилам имя каталога пакета должно совпадать с именем пакета. Так как наш пакет будет называться `greeting`, это имя должно быть присвоено и каталогу.

Да, мы знаем, кажется, что вложенных каталогов слишком много (а на самом деле их будет еще больше). Но поверьте, после того как вы построите коллекцию собственных пакетов, а также пакетов из других источников, такая структура поможет сохранить нормальную организацию кода.

И что еще важнее, эта структура помогает инструментам Go находить код. Поскольку код всегда находится в каталоге *src*, инструменты Go точно знают, где следует искать код импортируемых пакетов.



Затем создайте файл в каталоге *greeting* и присвойте ему имя *greeting.go*. Файл должен содержать код, приведенный ниже. Вскоре мы рассмотрим его более подробно, а пока следует обратить внимание на пару моментов...

Как и все файлы с исходным кодом, встречавшиеся вам до этого, файл начинается с директивы `package`. Но в отличие от других файлов, этот код не является частью пакета `main`; он принадлежит пакету с именем `greeting`.

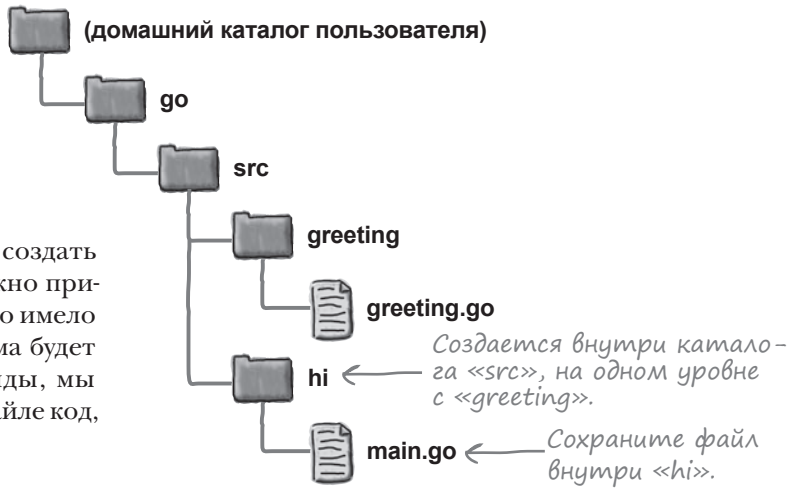
Также обратите внимание на определения двух функций. Они мало чем отличаются от других функций, встречавшихся вам ранее. Но так как мы хотим, чтобы к ним можно было обращаться за пределами пакета `greeting`, их имена начинаются с букв верхнего регистра, чтобы эти функции экспортировались.

```
package greeting          ← Принадлежит не пакету «main», а пакету «greeting»!

import "fmt"

func Hello() {            ← Первые буквы в верхнем регистре, чтобы функции экспортировались!
        fmt.Println("Hello!")
}

func Hi() {
        fmt.Println("Hi!")
}
```

greeting.go

Импорт пакета в программу

Теперь попробуем использовать новый пакет внутри программы. В каталоге рабочей области, внутри подкаталога *src*, создайте новый подкаталог с именем *hi*. (Хранить код исполняемых программ внутри рабочей области *необязательно*, но это хорошая мысль.)



Теперь в новом каталоге *hi* необходимо создать новый исходный файл. Этому файлу можно присвоить любое имя (важно лишь, чтобы оно имело расширение *.go*), но поскольку программа будет оформлена в виде исполняемой команды, мы присвоим ему имя *main.go*. Сохраните в файле код, приведенный ниже.

Как и любой файл с исходным кодом Go, этот код начинается с директивы `package`. Но поскольку предполагается, что это будет исполняемая команда, необходимо использовать имя пакета `main`. В общем случае имя пакета должно соответствовать имени каталога, в котором он хранится, но пакет `main` является исключением из этого правила.

```
package main

import "greeting"

func main() {
        greeting.Hello()
        greeting.Hi()
}
```

Чтобы использовать функции пакета, его необходимо сначала импортировать.

При вызове функций других пакетов необходимо указать имя пакета и точку.

main.go

Затем необходимо импортировать пакет `greeting`, чтобы использовать его функции. Инструменты Go ищут код пакета в подкаталоге каталога *src* рабочей области, имя которого соответствует имени в команде `import`. Чтобы приказать Go искать код в каталоге *src/greeting* из каталога рабочей области, мы используем директиву `import "greeting"`.

Наконец, поскольку пакет содержит исполняемый код, нам понадобится функция `main`, которая вызывается при запуске программы. В `main` вызываются обе функции, определенные в пакете `greeting`. В обоих вызовах указывается имя пакета и точка, чтобы компилятор Go знал, частью какого пакета являются функции.

Все готово! Попробуем запустить программу. В окне терминала или командной строки введите команду **`cd`**, чтобы перейти в каталог *src/hi* в каталоге рабочей области. (Путь зависит от местонахождения домашнего каталога.) Затем запустите программу командой **`go run main.go`**.

Вызов функций из пакета!

```
Shell  Edit  View  Window  Help
$ cd /Users/jay/go/src/hi
$ go run main.go
Hello!
Hi!
$
```

При достижении строки `import "greeting"` Go обращается к каталогу *greeting* в каталоге *src* вашего рабочего каталога и ищет в нем исходный код пакета. Этот код компилируется и импортируется, после чего мы получаем возможность вызывать функции пакета `greeting`!

Файлы пакетов имеют одинаковую структуру

Помните, в главе 1 говорилось о трех разделах, которые присутствуют практически во всех файлах с исходным кодом Go?

Вы быстро привыкнете к тому, что эти три части (именно в таком порядке) встречаются практически во всех файлах Go, с которыми вы будете работать:

1. Директива package.
2. Директива `import`.
3. Собственно код программы.

Директива package.
Директива import.
Собственно код программы.

```
package main
import "fmt"
func main() {
        fmt.Println("Hello, Go!")
}
```

Естественно, это правило выполняется и для пакета `main` в файле *main.go*. В нашем коде присутствует директива `package`, за которой следует директива `import` и собственно код нашего пакета.

Директива package.
Директива import.
Собственно код программы.

```
package main
import "greeting"
func main() {
        greeting.Hello()
        greeting.Hi()
}
```

Другие пакеты, кроме `main`, имеют тот же формат. Как видите, файл *greeting.go* тоже содержит директиву `package`, директиву `import` и завершается самим кодом пакета.

Директива package.
Директива import.
Собственно код программы.

```
package greeting
import "fmt"
func Hello() {
        fmt.Println("Hello!")
}
func Hi() {
        fmt.Println("Hi!")
}
```

Сломай и изучи!

Возьмите код пакета `greeting` и код программы, которая этот пакет импортирует. Внесите одно из указанных изменений и запустите программу; затем отмените изменение и переходите к следующему. Посмотрите, что из этого выйдет!


```
package greeting

import "fmt"

func Hello() {
       fmt.Println("Hello!")
}

func Hi() {
       fmt.Println("Hi!")
}
```


```
package main

import "greeting"

func main() {
       greeting.Hello()
       greeting.Hi()
}
```

Если...	*...программа не будет работать, потому что...*
Изменить имя каталога *greeting*  **~~greeting~~ salutation**	Инструменты Go используют имя в пути импорта как имя каталога, из которого должен загружаться исходный код пакета. Если имена не совпадают, то код не загрузится
Изменить имя в строке `package` файла *greeting.go* `package salutation`	Содержимое каталога *greeting* загрузится как пакет с именем `salutation`. Но поскольку в вызовах функций в *main.go* упоминается пакет `greeting`, вы получите сообщения об ошибке
Преобразовать имена функций в файлах *greeting.go* и *main.go* к нижнему регистру `func ~~H~~hello()` `func ~~H~~hi()` `greeting.~~H~~hello()` `greeting.~~H~~hi()`	Функции, имена которых начинаются с букв нижнего регистра, не экспортируются — это означает, что они могут использоваться только в своем пакете. Чтобы в программе можно было использовать функцию из другого пакета, эта функция должна экспортироваться, а для этого ее имя должно начинаться с буквы верхнего регистра

У бассейна

Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты необязательно. Ваша **задача**: создать в рабочей области Go пакет `calc`, чтобы функции могли использоваться в коде *main.go*.


```
package _____

func ____(first float64, second float64) float64 {
        return first + second
}

func _________(first float64, second float64) float64 {
        return first - second
}
```

calc.go

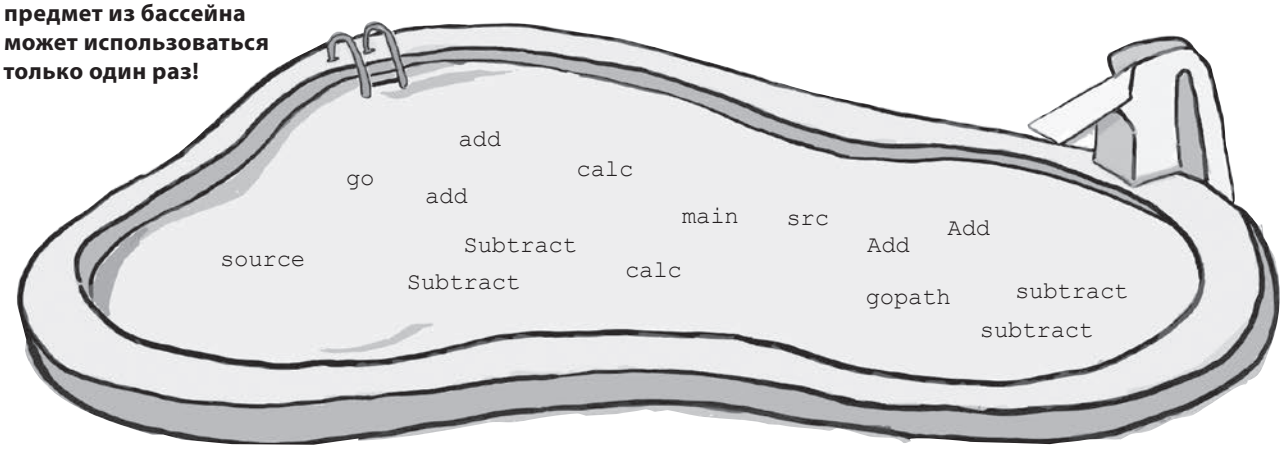
```
package main

import (
        "calc"
        "fmt"
)

func main() {
        fmt.Println(calc._____(1, 2))
        fmt.Println(calc._____(7, 3))
}
```

main.go

Результат.

3
4

Примечание: каждый предмет из бассейна может использоваться только один раз!



Ответ на с. 181.

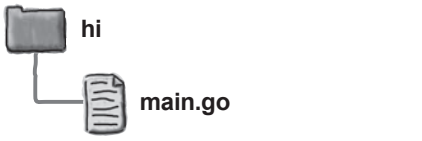
Соглашения по выбору имен пакетов

Разработчикам, использующим пакет, придется вводить его имя каждый раз, когда они вызывают функцию из этого пакета. (Вспомните `fmt.Printf`, `fmt.Println`, `fmt.Print` и т. д.) Чтобы не возникало проблем, при выборе имен пакетов следует соблюдать несколько правил:

- Имя пакета должно быть записано только символами нижнего регистра.
- Имя следует сокращать, если его смысл очевиден (например, `fmt`).
- По возможности имя должно состоять из одного слова. Если необходимы два слова, они *не* должны разделяться символами подчеркивания, а второе слово *не* должно начинаться с буквы верхнего регистра (пример — пакет `strconv`).
- Импортированные имена пакетов могут конфликтовать с именами локальных переменных, поэтому не используйте имя, которое с большой вероятностью может быть выбрано пользователями пакета. (Например, если бы пакет `fmt` назывался `format`, то импорт этого пакета создавал бы риск конфликта с локальной переменной `format`.)

Уточнение имен

При обращении к функции, переменной или чему-то еще, экспортируемому из другого пакета, необходимо уточнить имя функции или переменной, поставив перед именем функции или переменной имя пакета. Но при обращении к функции или переменной, определенной в *текущем* пакете, *не следует* уточнять имя пакета.




```
package main

import "greeting"

func main() {
    greeting.Hello()
    greeting.Hi()
}
```

Пакетные классификаторы.

Так как в нашем файле *main.go* код является частью пакета `main`, необходимо указать, что функции `Hello` и `Hi` принадлежат пакету `greeting` — для этого следует использовать имена **`greeting.Hello`** и **`greeting.Hi`**.


```
package greeting

import "fmt"

func Hello() {
        fmt.Println("Hello!")
}

func Hi() {
        fmt.Println("Hi!")
}

func AllGreetings() {
    Hello()
    Hi()
}
```

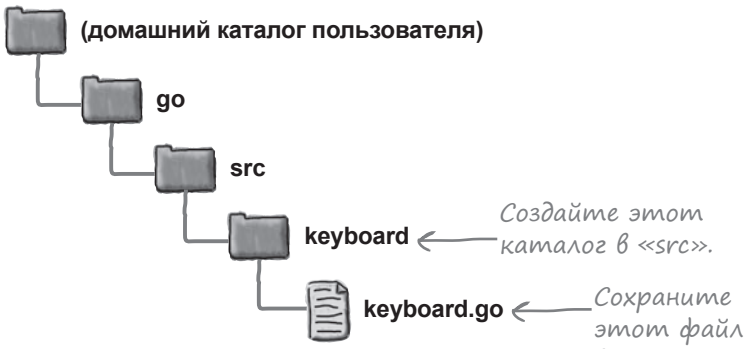
Классификаторы отсутствуют.

Но предположим, что функции `Hello` и `Hi` вызываются из другой функции пакета `greeting`. В этом случае достаточно указать имена `Hello` и `Hi` (без классификатора, то есть уточняющего имени пакета), потому что вызываются функции из того пакета, в котором были определены.

Перемещение общего кода в пакет

Теперь вы понимаете, как добавлять новые пакеты в рабочую область Go, и мы наконец готовы вынести функцию `getFloat` в пакет, который может использоваться программами *pass_fail.go* и *tocelsius.go*.

Назовем наш пакет `keyboard`, так как он предназначен для чтения ввода с клавиатуры. Начнем с создания нового каталога с именем *keyboard* в каталоге *src* рабочей области.



Затем создадим файл с исходным кодом в каталоге *keyboard*. Ему можно присвоить любое имя, но мы используем имя, соответствующее имени пакета: *keyboard.go*.

В начало файла необходимо добавить директиву `package` с именем пакета: `keyboard`.

Так как мы создаем отдельный файл, необходимо включить директиву `import` со всеми пакетами, используемыми в коде: `bufio`, `os`, `strconv` и `strings`. (Пакеты `fmt` и `log` не указываются, так как они используются только в файлах *pass_fail.go* и *tocelsius.go*.)

Наконец, код старой функции `getFloat` можно скопировать в неизменном виде. Но не забудьте переименовать функцию в `GetFloat`, чтобы функция экспортировалась, она должна начинаться с буквы верхнего регистра.

```
package keyboard        <- Добавляем директиву package.

import (
    "bufio"             <- Импортируются только пакеты, используемые в этом файле.
    "os"
    "strconv"
    "strings"
)
                        <- Имя функции начинается с буквы верхнего регистра, чтобы функция экспортировалась.
func GetFloat() (float64, error) {
    reader := bufio.NewReader(os.Stdin)
    input, err := reader.ReadString('\n')
    if err != nil {
        return 0, err
    }

    input = strings.TrimSpace(input)
    number, err := strconv.ParseFloat(input, 64)
    if err != nil {
        return 0, err
    }
    return number, nil
}
```

keyboard.go

Перемещение общего кода в пакет (продолжение)

Теперь можно обновить программу *pass_fail.go*, чтобы в ней использовался новый пакет `keyboard`.

```
// pass_fail сообщает, сдал ли пользователь экзамен.
package main

import (
        "fmt"
        "keyboard"
        "log"
)

func main() {
        fmt.Print("Enter a grade: ")
        grade, err := keyboard.GetFloat()
        if err != nil {
                log.Fatal(err)
        }

        var status string
        if grade >= 60 {
                status = "passing"
        } else {
                status = "failing"
        }
        fmt.Println("A grade of", grade, "is", status)
}
```

Импортируются только пакеты, используемые в этом файле.

Не забудьте импортировать новый пакет.

Функцию getFloat, которая здесь находилась, можно удалить.

Вызываем функцию из пакета «keyboard».

Так как мы удаляем старую функцию `getFloat`, также необходимо удалить неиспользуемые директивы `bufio`, `os`, `strconv` и `strings`. Вместо них будет импортироваться новый пакет `keyboard`.

В функции `main` старый вызов `getFloat` заменяется вызовом новой функции `keyboard.GetFloat`. Остальной код — без изменений.

При запуске обновленной программы будет выведен тот же результат.

```
Enter a grade: 89.7
A grade of 89.7 is passing
```

Те же изменения вносятся в программу *tocelsius.go*.

```
// tocelsius преобразует температуру...
package main

import (
        "fmt"
        "keyboard"
        "log"
)

func main() {
        fmt.Print("Enter a temperature in Fahrenheit: ")
        fahrenheit, err := keyboard.GetFloat()
        if err != nil {
                log.Fatal(err)
        }
        celsius := (fahrenheit - 32) * 5 / 9
        fmt.Printf("%0.2f degrees Celsius\n", celsius)
}
```

Импортируются только пакеты, используемые в этом файле.

Не забудьте импортировать новый пакет.

Функцию getFloat, которая здесь находилась, можно удалить.

Вызываем функцию из пакета «keyboard».

Мы обновляем директивы импорта, удаляем старую функцию `getFloat` и вызываем вместо нее `keyboard.GetFloat`.

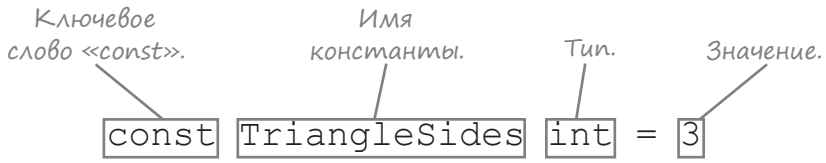
И снова при запуске обновленной программы будет получен тот же результат. Но на этот раз вместо дублирующегося кода будет использоваться функция из нашего нового пакета!

```
Enter a temperature in Fahrenheit: 98.6
37.00 degrees Celsius
```

Константы

Многие пакеты экспортируют **константы**: именованные значения, которые не изменяются за время работы программы. Объявление константы очень похоже на объявление переменной: в нем также указывается имя, необязательный тип и значение. Тем не менее правила несколько отличаются:

- Вместо ключевого слова `var` используется ключевое слово `const`.
- Значение константы должно быть задано в момент ее объявления. Вы не сможете присвоить его позднее, как с переменными.
- Для переменных доступен синтаксис короткого объявления `:=`, а у констант аналогичной конструкции не существует.



Как и при объявлении переменных, тип можно опустить, он будет автоматически определен по присваиваемому значению:

```
const SquareSides = 4
```

Присваивается целое число, поэтому для константы будет выбран тип «int».

Значение *переменной* может *изменяться*, но значение *константы* должно оставаться *постоянным*. Попытка присвоить новое значение константе приведет к ошибке компиляции. Эта особенность констант обеспечивает безопасность: константы должны использоваться для значений, которые *не должны* изменяться.

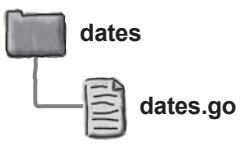
```
const PentagonSides = 5
PentagonSides = 6
```

Попытка присвоить новое значение константе!

Ошибка компиляции.

```
cannot assign to PentagonSides
```

Если ваша программа включает «фиксированные» значения литералов (особенно если эти значения используются в нескольких местах), подумайте о том, чтобы заменить их константами (даже если программа не разбита на пакеты). Ниже приведен пакет с двумя функциями, в обеих функциях целочисленный литерал 7 представляет количество дней в неделе:




```
package dates

func WeeksToDays(weeks int) int {
        return weeks * 7
}

func DaysToWeeks(days int) float64 {
        return float64(days) / float64(7)
}
```

Получаем количество недель.

Умножаем на количество дней в неделе, чтобы вычислить общее количество дней.

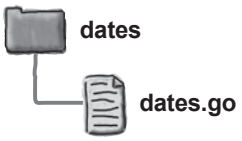
Получаем количество дней.

Делим на количество дней в неделе, чтобы получить количество недель.

Константы (продолжение)

Заменяя литералы константой `DaysInWeek`, мы тем самым документируем их смысл. (Другие разработчики увидят имя `DaysInWeek` и немедленно поймут, что число `7` было выбрано неслучайно.) Кроме того, если позднее в программу будут добавлены новые функции, для предотвращения возможных расхождений можно использовать константу `DaysInWeek` и в этих функциях.

Обратите внимание: константа объявляется за пределами любых функций, на уровне пакета. И хотя константу можно объявить внутри функции, это ограничит ее область видимости блоком этой функции. Вариант с объявлением констант на уровне пакета гораздо более типичен, так как эти константы будут доступны для всех функций этого пакета.



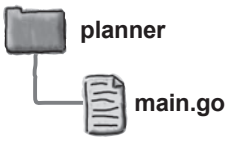

```
package dates

const DaysInWeek int = 7

func WeeksToDays(weeks int) int {
  return weeks * DaysInWeek
}
func DaysToWeeks(days int) float64 {
  return float64(days) / float64(DaysInWeek)
}
```

Объявляем константу.

Константа используется вместо целочисленного литерала.

Константа используется вместо целочисленного литерала.

Как и в случае с переменными и функциями, константа, имя которой начинается с буквы верхнего регистра, экспортируется, и к ней можно будет обращаться из других пакетов с уточнением имени. В следующем примере программа использует константу `DaysInWeek` из пакета `main`, для чего она импортирует пакет `dates` и уточняет имя константы в форме `dates.DaysInWeek`.




```
package main

import (
  "dates"
  "fmt"
)

func main() {
  days := 3
  fmt.Println("Your appointment is in", days, "days")
  fmt.Println("with a follow-up in", days + dates.DaysInWeek, "days")
}
```

Импортируем пакет, в котором объявлена константа.

Уточняется именем пакета.

Используется константа из пакета «dates».

```
Your appointment is in 3 days
with a follow-up in 10 days
```

Вложенные каталоги и пути импорта пакетов

При работе с пакетами, включенными в поставку Go (например, `fmt` и `strconv`), имя пакета обычно совпадает с путем импорта (строкой, которая используется в директиве импорта пакета). Но как было показано в главе 2, это не всегда так...

Однако путь импорта и имя пакета могут различаться. Многие пакеты Go классифицируются по категориям, например «сжатие» или «комплексные вычисления». По этой причине они часто группируются по префиксам пути импорта (например, `"archive/"` или `"math/"`). (Их можно рассматривать как аналоги путей каталогов на жестком диске.)

Путь импорта	Имя пакета
`"archive"`	`archive`
`"archive/tar"`	`tar`
`"archive/zip"`	`zip`
`"math"`	`math`
`"math/cmplx"`	`cmplx`
`"math/rand"`	`rand`

Некоторые наборы пакетов группируются по префиксам путей импорта, например `"archive/"` или `"math/"`. Мы предлагали рассматривать эти префиксы как аналоги путей каталогов на жестком диске... и это неслучайно. Эти префиксы путей импорта *формируются* на основе структуры каталогов!

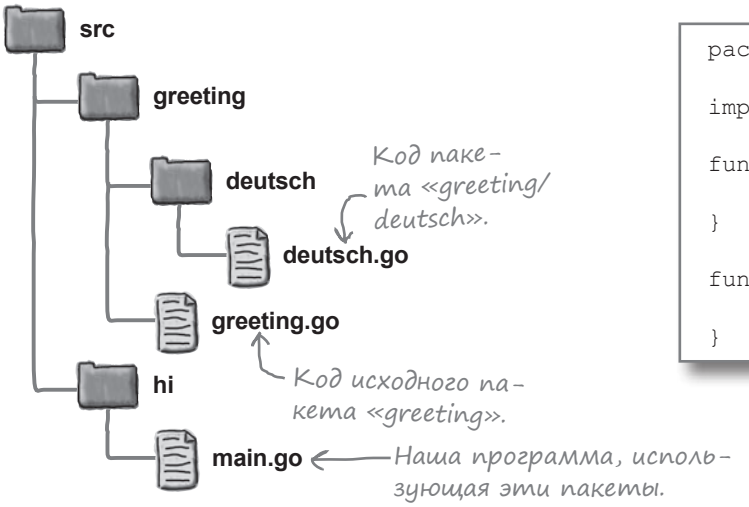
Группы взаимосвязанных пакетов можно разместить в каталоге в рабочей области Go. Этот каталог становится частью пути импорта для всех содержащихся в нем пакетов.

Допустим, вы хотите добавить пакеты для вывода приветствий на дополнительных языках. Если все эти пакеты будут размещаться прямо в пакете *src*, он быстро станет слишком громоздким и неудобным. Но если разместить все новые пакеты в каталоге *greeting*, они будут удобно организованы.

Размещение пакетов в каталоге *greeting* также влияет на их путь импорта. Например, если бы пакет `dansk` хранился непосредственно в каталоге *src*, то ему бы соответствовал путь импорта `"dansk"`. Но если разместить его в каталоге *greeting*, путь импорта преобразуется к виду `"greeting/dansk"`. Переместите пакет `deutsch` в каталог *greeting*, и ему будет назначен путь импорта `"greeting/deutsch"`. Исходный пакет `greeting` будет доступен по пути импорта `"greeting"` при условии, что файл с его исходным кодом хранится прямо в каталоге *greeting* (а не в одном из его подкаталогов).

Вложенные каталоги и пути импорта пакетов (продолжение)

Предположим, имеется пакет `deutsch`, вложенный в каталог пакета *greeting*, и его код выглядит примерно так:




```
package deutsch

import "fmt"

func Hallo() {
        fmt.Println("Hallo!")
}

func GutenTag() {
        fmt.Println("Guten Tag!")
}
```

deutsch.go

Давайте обновим код *hi/main.go*, чтобы в нем также использовался пакет `deutsch`. Поскольку каталог этого пакета вложен в каталог *greeting*, необходимо использовать путь импорта `"greeting/deutsch"`. Но после того, как пакет будет импортирован, к нему можно будет обращаться просто по имени пакета: `deutsch`.

```
package main

import (
        "greeting"
        "greeting/deutsch"
)

func main() {
        greeting.Hello()
        greeting.Hi()
        deutsch.Hallo()
        deutsch.GutenTag()
}
```

Импортируем пакет «greeting», как и прежде.

Также импортируем пакет «deutsch».

Добавляем вызовы функций нового пакета.

main.go

Как и прежде, чтобы запустить код, мы используем команду **cd** для перехода в каталог *src/hi* каталога рабочей области, после чего запускаем программу командой **`go run main.go`**. Результаты вызовов функций пакета `deutsch` видны в выходных данных.

```
Shell Edit View Window Help
$ cd /Users/jay/go/src/hi
$ go run main.go
Hello!
Hi!
Hallo!
Guten Tag!
```

Результат пакета «deutsch».

Установка исполняемых файлов командой «go install»

При использовании команды `go run` программа должна быть сначала откомпилирована, как и все пакеты, от которых она зависит. И весь откомпилированный код будет потерян после завершения программы.

В главе 1 была описана команда `go build`, которая компилирует и сохраняет исполняемый двоичный файл (файл, который может выполняться даже без установки Go). Но если вы будете использовать ее слишком часто, ваша рабочая область будет забита исполняемыми файлами в случайных и неподходящих местах.

Команда `go install` также сохраняет откомпилированные бинарные версии исполняемых программ, но в четко определенном и легкодоступном месте: в каталоге *bin* вашей рабочей области Go. Просто передайте `go install` имя каталога из *src*, содержащего код исполняемой программы (то есть файлов *.go*, начинающихся с `package main`). Программа будет откомпилирована, а исполняемый файл будет сохранен в стандартном каталоге.

Попробуем установить исполняемый файл для нашей программы *hi/main.go*. Как и прежде, введите в терминале команду **`go install`**, пробел и имя подкаталога в каталоге *src* (**`hi`**). Еще раз подчеркнем: неважно, из какого каталога вы это сделаете — компилятор будет искать каталог внутри каталога *src*.

```
Shell Edit View Window Help
$ go install hi
$
```

(Не забудьте, что команде «go install» должно передаваться имя каталога в «src», а не имя файла .go! По умолчанию команда «go install» не настроена для прямой работы с файлами .go.)

Когда компилятор Go видит, что файл в каталоге *hi* содержит объявление `package main`, он понимает, что это код исполняемой программы. Он компилирует исполняемый файл и сохраняет результат в каталоге с именем *bin* в рабочей области Go. (Каталог *bin* будет создан автоматически, если он до этого не существовал.)

В отличие от команды `go build`, которая присваивает исполняемому файлу имя файла *.go*, на основе которого он был создан, `go install` назначает имя исполняемого файла по имени каталога, содержащего код. Так как мы компилируем содержимое каталога *hi*, исполняемый файл будет назван `hi` (или `hi.exe` в Windows).

Теперь воспользуйтесь командой **`cd`** для перехода к каталогу *bin* в рабочей области Go. В каталоге *bin* исполняемый файл запускается командой **`./hi`** (или **`hi.exe`** в Windows).

```
Shell Edit View Window Help
$ cd /Users/jay/go/bin
$ ./hi
Hello!
Hi!
Hallo!
Guten Tag!
```

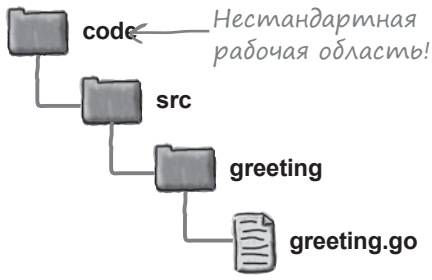
Вы также можете включить каталог «bin» своей рабочей области в системную переменную «PATH». Тогда вы сможете запускать исполняемые файлы из «bin» из любого каталога вашей системы! Современные программы установки Go для Mac и Windows автоматически вносят изменения в «PATH».

Переменная GOPATH и смена рабочих областей

На разных веб-сайтах можно увидеть, как разработчики говорят о «настройке GOPATH» при обсуждении рабочей области Go. GOPATH — переменная среды, к которой инструменты Go обращаются за информацией о местонахождении рабочей области. Большинство разработчиков хранит весь свой код Go в одной рабочей области и не меняет ее местонахождения по умолчанию. Но при желании можно использовать GOPATH для перемещения рабочей области в другой каталог.

Переменная среды предназначена для хранения и чтения значений (как и переменные Go), но управляет ею операционная система, а не Go. Некоторые программы настраиваются при помощи переменных среды; к их числу относится и компилятор Go.

Допустим, вместо домашнего каталога вы разместили свой пакет greeting в каталоге *code* в корневом каталоге своего жесткого диска. И теперь вы хотите запустить файл *main.go*, который зависит от greeting.




```
package main

import "greeting"

func main() {
       greeting.Hello()
       greeting.Hi()
}
```

main.go

Но вы получаете сообщение об ошибке. В нем говорится, что пакет greeting не найден, так как компилятор продолжает искать пакет в подкаталоге *go* вашего домашнего каталога:

```
Shell Edit View Window Help
$ go run main.go
command.go:3:8: cannot find package "greeting" in any of:
      /usr/local/go/libexec/src/greeting (from $GOROOT)
      /Users/jay/go/src/greeting (from $GOPATH)
```

Настройка GOPATH

Если ваш код хранится в другом каталоге (вместо каталога по умолчанию), вы должны настроить компилятор Go и передать информацию о местонахождении кода. Для этого можно воспользоваться переменной среды `GOPATH`, а конкретный способ зависит от операционной системы.

Системы Mac и Linux systems:

Для настройки переменной среды используется команда `export`. В приглашении терминала введите команду:

```
export GOPATH="/code"
```

Для каталога с именем *code* в корневом каталоге жесткого диска используется путь «`/code`». Если вы храните код в другом месте, укажите нужный путь.

Системы Windows:

Для настройки переменной среды используется команда `set`. В приглашении командной строки введите команду:

```
set GOPATH="C:\code"
```

Для каталога с именем *code* в корневом каталоге жесткого диска используется путь «`C:\code`». Если вы храните код в другом месте, укажите нужный путь.

После того как это будет сделано, при выполнении команды `go run` (как и других средств Go) будет использоваться рабочая область, определяемая заданным каталогом. Таким образом, библиотека `greeting` будет найдена, а программа успешно выполнится!

В Mac/Linux.

```
Shell Edit View Window Help
$ export GOPATH="/code"
$ go run main.go
Hello!
Hi!
```

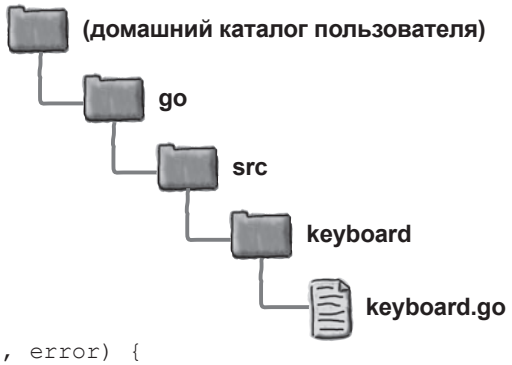
В Windows.

```
Shell Edit View Window Help
C:\Users\jay>set GOPATH="C:\code"
C:\Users\jay>go run main.go
Hello!
Hi!
```

Обратите внимание: описанные выше способы настраивают переменную среды `GOPATH` только для *текущего* окна терминала/командной строки. Вам придется настраивать ее заново для каждого нового окна. Впрочем, есть способы выполнить постоянную настройку переменной среды. Конкретный способ также зависит от операционной системы, и у нас нет возможности описывать их здесь. Введите строку «переменные среды» и название вашей ОС в поисковой системе — вы наверняка найдете полезные инструкции в результатах поиска.

Публикация пакетов

Наш замечательный пакет `keyboard` оказался настолько полезным, что другим разработчикам он бы тоже мог пригодиться.

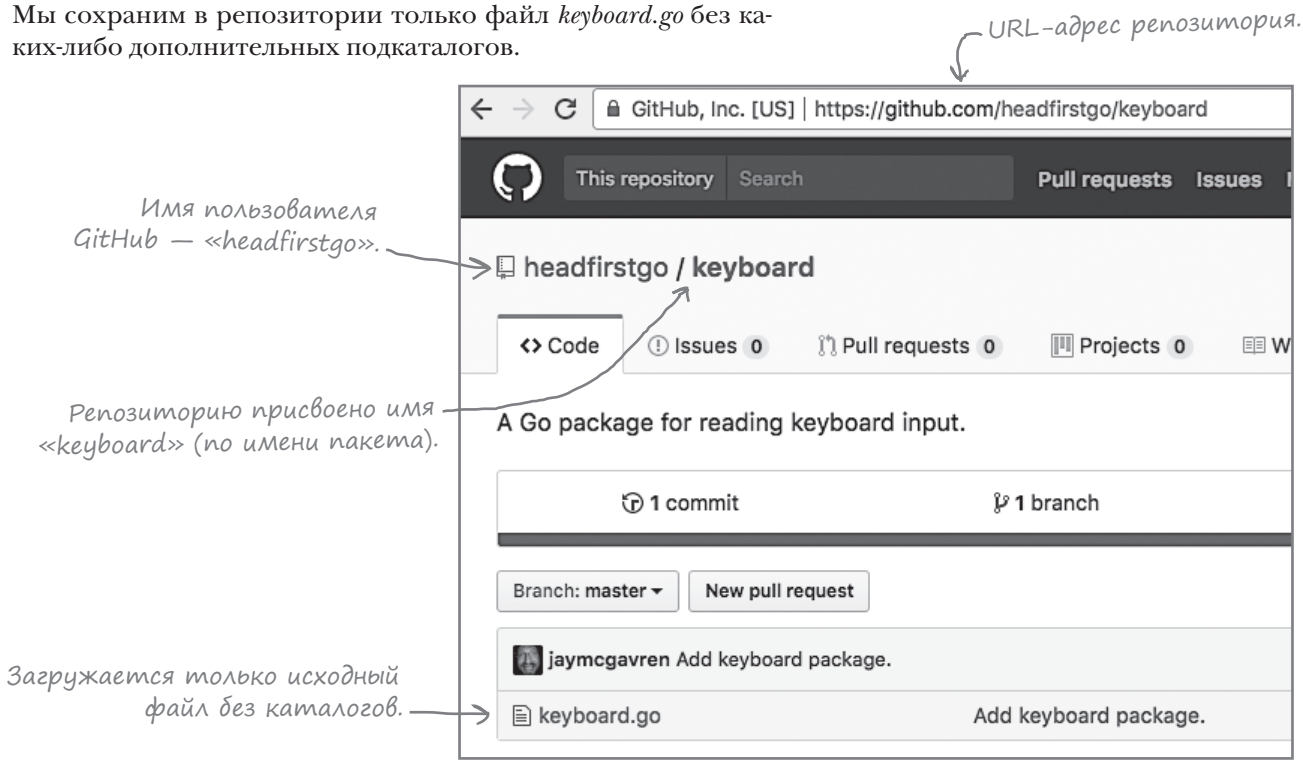



```
package keyboard

import (
       "bufio"
       "os"
       "strconv"
       "strings"
)

func GetFloat() (float64, error) {
       // Код GetFloat...
}
```

Создадим репозиторий для хранения нашего кода на GitHub — популярном веб-сайте, специализирующемся на публикации и распространении кода. Другие разработчики смогут загрузить пакет и использовать его в своих проектах! На GitHub мы используем имя пользователя `headfirstgo`, а репозиторий назовем *keyboard*, поэтому URL-адрес будет выглядеть так:

https://github.com/headfirstgo/keyboard

Мы сохраним в репозитории только файл *keyboard.go* без каких-либо дополнительных подкаталогов.

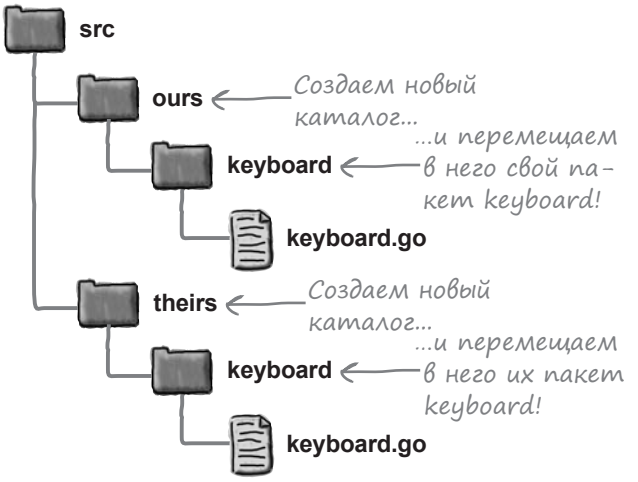


Публикация пакетов (продолжение)

Спасибо, но кажется, мы не сможем пользоваться вашим пакетом. В моем приложении для магазина музыкальных инструментов уже есть пакет `keyboard`, и если я установлю ваш пакет `keyboard`, возникнет конфликт!

Да, опасения вполне обоснованны. В каталоге *src* рабочего пространства Go может быть только один каталог *keyboard*. *Похоже*, может быть только один пакет с именем `keyboard`!

Погодите... А если снова воспользоваться вложением каталогов? В одном каталоге будет храниться **наш** пакет `keyboard`, а в другом — **их** пакет `keyboard`!

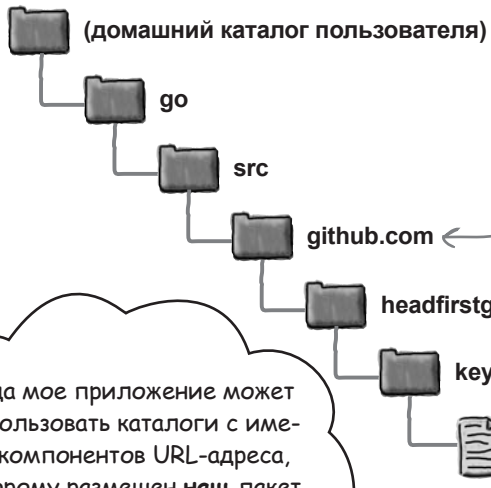


Но как назвать каталоги, в которых содержатся пакеты? Где **их** пакеты, где **наши**?

Публикация пакетов (продолжение)

Возможно, нам бы пригодился более универсальный идентификатор автора пакета. Наш пакет `keyboard` доступен только по адресу *http://github.com/headfirstgo/keyboard*. Почему бы не разбить этот URL-адрес на части и не использовать их как имена каталогов?

Тогда мое приложение может использовать каталоги с именами компонентов URL-адреса, по которому размещен **наш** пакет `keyboard`. Никаких конфликтов. Мне нравится!



Давайте попробуем переместить наш пакет в структуру каталогов, представляющих URL-адрес его размещения. Создайте в каталоге *src* другой каталог с именем *github.com*. Внутри этого каталога создайте каталог с именем следующего сегмента URL-адреса *headfirstgo*. А после этого переместите каталог *keyboard* из каталога *src* в каталог *headfirstgo*.

Хотя перемещение пакета в новый подкаталог изменяет его *путь импорта*, оно не изменит *имя* пакета. А раз в самом пакете для обращений к пакету используется только имя, вносить изменения в коде пакета не придется!

```
package keyboard

import (
        "bufio"
        "os"
        "strconv"
        "strings"
)

// Сюда перемещается код
keyboard.go...
```

Имя пакета не изменилось, поэтому изменять код пакета не нужно.

keyboard.go

Публикация пакетов (продолжение)

Тем не менее нам *придется* обновить программы, зависящие от этого пакета, потому что путь импорта программы изменился. Так как имена подкаталогов были выбраны по частям URL-адреса, по которому размещается пакет, новый путь импорта очень похож на URL:

```
"github.com/headfirstgo/keyboard"
```

Необходимо только обновить директиву `import` в каждой программе. Так как имя пакета осталось неизменным, все обращения к пакету в остальном коде не изменяются.

После внесения этих изменений все программы, зависящие от нашего пакета `keyboard`, должны работать нормально.

```
// pass_fail сообщает, сдал ли пользователь экзамен.
package main

import (
        "fmt"
        "github.com/headfirstgo/keyboard"
        "log"
)

func main() {
        fmt.Print("Enter a grade: ")
        grade, err := keyboard.GetFloat()
        if err != nil {
                log.Fatal(err)
        }
        // ...
}
```

Обновляем путь импорта.

Здесь ничего не изменилось: имя пакета осталось прежним.

```
Enter a grade: 89.7
A grade of 89.7 is passing
```

Конечно, нам хотелось бы быть авторами идеи об использовании доменных имен и путей для обеспечения уникальности путей импорта, но на самом деле это не наша заслуга. Сообщество Go с первых дней использовало эту схему именования пакетов как стандартную. Аналогичные схемы уже десятилетиями применяются в других языках (например, в Java).

```
// tocelsius converts a temperature...
package main

import (
        "fmt"
        "github.com/headfirstgo/keyboard"
        "log"
)

func main() {
        fmt.Print("Enter a temperature in Fahrenheit: ")
        fahrenheit, err := keyboard.GetFloat()
        if err != nil {
                log.Fatal(err)
        }
        // ...
}
```

Обновляем путь импорта.

Здесь ничего не изменилось: имя пакета осталось прежним.

```
Enter a temperature in Fahrenheit: 98.6
37.00 degrees Celsius
```

Загрузка и установка пакетов командой «go get»

У использования URL-адреса размещения пакета в качестве пути импорта есть еще одно преимущество. У команды `go` существует еще одна субкоманда `go get`, которая может автоматически загружать и устанавливать пакеты за вас.

Создадим репозиторий Git для описанного выше пакета `greeting` по следующему URL-адресу:

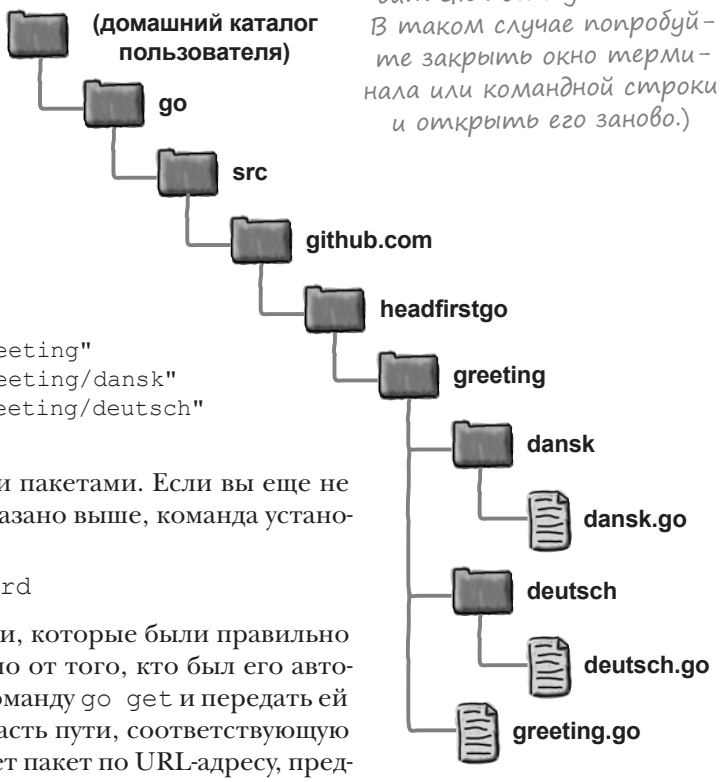
https://github.com/headfirstgo/greeting

Это означает, что на любом компьютере с установленным экземпляром Go необходимо ввести в терминале следующую команду:

```
go get github.com/headfirstgo/greeting
```

После команды `go get` следует URL-адрес репозитория с отсеченным сегментом «схемы» («https://»). Команда подключается к *github.com*, загружает репозиторий Git по пути */headfirstgo/greeting* и сохраняет его в каталоге *src* вашей рабочей области Go. (Примечание: если в вашей системе не установлена поддержка Git, вам будет предложено установить ее при выполнении команды `go get`. Просто следуйте инструкциям на экране. Команда `go get` также работает с репозиториями Subversion, Mercurial и Bazaar.)

Команда `go get` автоматически создает подкаталоги, необходимые для настройки подходящего пути импорта (каталог *github.com*, каталог *headfirstgo* и т. д.). Пакеты, сохраненные в каталоге *src*, выглядят так:



(Примечание: иногда команда «go get» не находит Git после установки. В таком случае попробуйте закрыть окно терминала или командной строки и открыть его заново.)

Пакеты, сохраненные в рабочей области Go, готовы к использованию в программах. Например, если вы хотите использовать пакеты `greeting`, `dansk` и `deutsch` в программе, включите в нее директиву импорта, которая выглядит примерно так:

```
import (
        "github.com/headfirstgo/greeting"
        "github.com/headfirstgo/greeting/dansk"
        "github.com/headfirstgo/greeting/deutsch"
)
```

Команда `go get` работает также и с другими пакетами. Если вы еще не настроили пакет `keyboard` так, как было показано выше, команда установит его:

```
go get github.com/headfirstgo/keyboard
```

Команда `go get` работает с любыми пакетами, которые были правильно настроены в сервисе размещения, независимо от того, кто был его автором. Все, что для этого нужно — выполнить команду `go get` и передать ей путь импорта пакета. Команда анализирует часть пути, соответствующую адресу хоста, подключается к хосту и загружает пакет по URL-адресу, представленному остальной частью пути импорта. Это значительно упрощает использование кода других разработчиков!

Упражнение

Мы создали рабочую область Go с простым пакетом `mypackage`. Завершите приведенную программу, чтобы она импортировала пакет `mypackage` и вызывала его функцию `MyFunction`.


```
package mypackage

func MyFunction() {
}
```

mypackage.go

Запишите свой код:

```
package main

import ____________________________________

func main() {

      ____________________

}
```

Ответ на с. 181.

Чтение документации пакетов командой «go doc»

Я установил ваш пакет `keyboard`. Но я понятия не имею, как им пользоваться! Где можно найти эту информацию?

Введите команду `go doc`, чтобы вывести документацию по любому пакету или функции.

Чтобы вывести документацию по пакету, передайте его путь импорта команде `go doc`. Например, информация о пакете `strconv` выводится командой `go doc strconv`.

Вывести документацию по пакету strconv.

Имя и путь импорта пакета.

Описание пакета.

Включенные функции.

(Часть вывода пропущена для экономии места.)

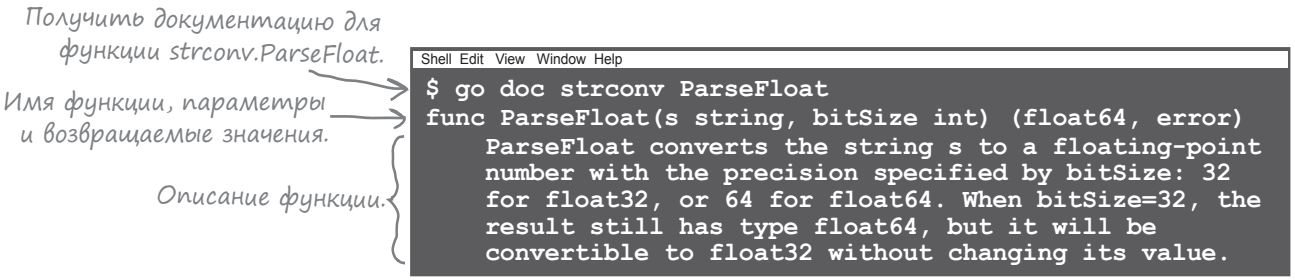
```
Shell Edit View Window Help
$ go doc strconv
package strconv // import "strconv"

Package strconv implements conversions to and from
string representations of basic data types.


Numeric Conversions

The most common numeric conversions are Atoi (string
to int) and Itoa (int to string).

    i, err := strconv.Atoi("-42")
    s := strconv.Itoa(-42)

[...Further description of the package here...]

[...Function names...]
func Itoa(i int) string
func ParseBool(str string) (bool, error)
func ParseFloat(s string, bitSize int) (float64, error)
[...More function names...]
```

В выходных данных приводится имя и путь импорта пакета (в данном случае это одно и то же), описание пакета в целом и список всех функций, экспортируемых пакетом.

Чтение документации пакета командой «go doc» (продолжение)

Команда `go doc` также может использоваться для получения подробной информации по конкретной функции. Для этого укажите имя функции после имени пакета. Предположим, вы увидели функцию `ParseFloat` в списке функций пакета `strconv` и хотите узнать о ней больше. Для этого можно воспользоваться командой `go doc strconv ParseFloat`.

Вы получите описание самой функции и того, что она делает:



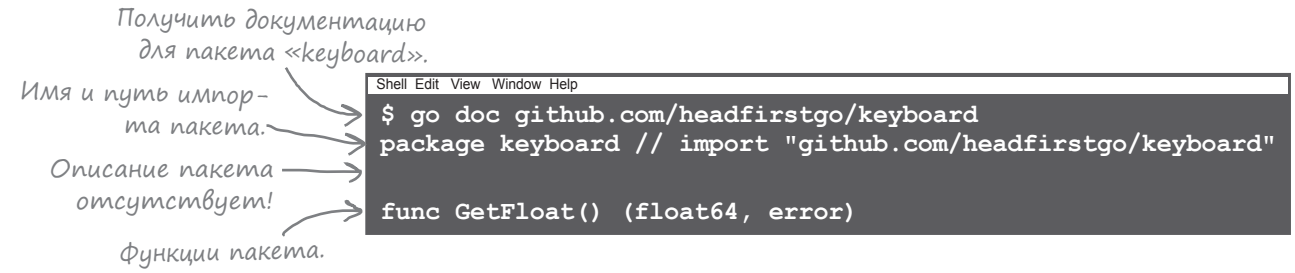
Первая строка выглядит как обычное объявление функции в коде. Она состоит из имени функции, за которым следуют круглые скобки с именами и типами параметров (если они есть). Если у функции есть возвращаемые значения, они указываются после параметров.

Далее идет подробное описание того, что делает функция, а также другая информация, необходимая для использования этой функции.

Таким же способом можно запросить информацию о пакете `keyboard`: передайте `go doc` его путь импорта. Давайте посмотрим, есть ли в ней что-то, что может пригодиться будущим пользователям. В терминале выполните команду:

```
go doc github.com/headfirstgo/keyboard
```

Команда `go doc` сможет извлечь основную информацию — имя пакета, путь импорта — из кода. Но пакет не содержит описания, поэтому польза от этих данных весьма ограниченна.



Запросив информацию о функции `GetFloat`, вы также не увидите описания:

Документирование пакетов

Команда `go doc` старается получить полезную информацию на основании анализа кода. Имена пакетов и пути импорта включаются автоматически, как и имена функций, параметры и возвращаемые типы. И все же команда `go doc` не умеет творить чудеса. Если вы хотите, чтобы пользователи смогли узнать из документации о предназначении пакета или функции, придется добавить эту информацию самостоятельно.

К счастью, это делается просто: нужно добавить в код **документирующие комментарии**. Обычные комментарии Go, размещенные непосредственно перед директивой `package` или объявлением функции, считаются документирующими комментариями и включаются в вывод `go doc`.

Давайте добавим doc-комментарии для пакета `keyboard`. В начале файла *keyboard.go*, непосредственно перед строкой `package`, мы добавим комментарий с описанием того, что делает пакет. А сразу же перед объявлением `GetFloat` будет добавлена пара строк комментариев с описанием этой функции.

Перед строкой «package» добавляются обычные комментарии.

Перед объявлением функции добавляются обычные комментарии.

```
// Package keyboard reads user input from the keyboard.
package keyboard

import (
        "bufio"
        "os"
        "strconv"
        "strings"
)

// GetFloat reads a floating-point number from the keyboard.
// It returns the number read and any error encountered.
func GetFloat() (float64, error) {
        // No changes to GetFloat code
}
```

При следующем выполнении для пакета команда `go doc` найдет комментарий перед строкой `package` и преобразует его в описание пакета. А при выполнении команды `go doc` для функции `GetFloat` вы увидите описание из комментария, добавленного перед объявлением `GetFloat`.

Описание пакета.

```
File  Edit  Window  Help
$ go doc github.com/headfirstgo/keyboard
package keyboard // import "github.com/headfirstgo/keyboard"

Package keyboard reads user input from the keyboard.

func GetFloat() (float64, error)
```

Описание функции.

```
File  Edit  Window  Help
$ go doc github.com/headfirstgo/keyboard GetFloat
func GetFloat() (float64, error)
    GetFloat reads a floating-point number from the
    keyboard. It returns the number read and any error
    encountered.
```

Документирование пакетов (продолжение)

Возможность получения документации командой `go doc` очень сильно поможет разработчику, установившему пакет.

Кроме того, документирующие комментарии упрощают жизнь разработчика, который пишет код пакета! Это самые обычные комментарии, поэтому добавляются они легко. И вы сможете легко просматривать их при внесении изменений в код.

Комментарий для пакета.

```
// Package keyboard reads user input from the keyboard.
package keyboard

import (
        "bufio"
        "os"
        "strconv"
        "strings"
)
```

Комментарий для функции.

```
// GetFloat reads a floating-point number from the keyboard.
// It returns the number read and any error encountered.
func GetFloat() (float64, error) {
        // Код GetFloat
}
```

При добавлении документирующих комментариев следует соблюдать ряд правил:

- Комментарии должны состоять из полноценных предложений.
- Комментарии для пакетов должны начинаться со слова «Package», за которым следует имя пакета:

```
// Package mypackage enables widget management.
```

- Комментарии для функций должны начинаться с имени функции, которую они описывают:

```
// MyFunction converts widgets to gizmos.
```

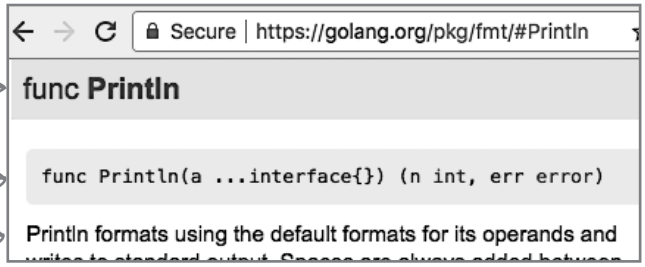
- В комментарии также можно включать примеры кода, которые должны снабжаться отступами.
- Не включайте дополнительные символы для выразительности или форматирования (кроме отступов в примерах кода). Документирующие комментарии будут выводиться в виде простого текста и должны форматироваться соответствующим образом.

Просмотр документации в браузере

Если вам удобнее работать в браузере, чем в терминале, существуют и другие способы просмотра документации пакетов. Самый простой способ — ввести в вашей любимой поисковой системе слово «golang», за которым следует имя пакета. («Golang» обычно используется в поисковых запросах, относящихся к языку Go, потому что слово «go» слишком часто встречается в других контекстах и не позволяет отфильтровать большинство нерелевантных результатов.) Например, если вам нужна документация для пакета `fmt`, проведите поиск по «golang fmt»:

Результаты должны включать сайты, которые предлагают документацию Go в формате HTML. Если вы ищете пакет в стандартной библиотеке Go (например, `fmt`), одним из лучших будет сайт *golang.org*, который ведет команда разработчиков Go. Документация будет похожа на выходные данные инструмента `go doc`, с именами пакетов, путями импорта и описаниями.

Одно из главных достоинств документации HTML заключается в том, что каждое имя в списке функций пакета представляет собой ссылку, которая открывает описание этой функции.



Впрочем, информация будет такой же, как при выполнении команды `go doc` в вашем терминале. Она тоже определяется все теми же простыми документирующими комментариями в коде.

Запуск сервера документации HTML командой «godoc»

Программа, которая обслуживает раздел документации сайта *golang.org*, на самом деле доступна и на *вашем* компьютере. Эта программа называется `godoc` (не путайте с командой `go doc`), и она автоматически устанавливается вместе с Go. Программа `godoc` генерирует документацию HTML на основании кода основной установки Go и вашей рабочей области. Она включает веб-сервер, который может передавать полученные веб-страницы браузеру. (Не беспокойтесь, с настройками по умолчанию `godoc` не будет принимать подключения с других компьютеров, кроме вашего.)

Чтобы запустить `godoc` в режиме веб-сервера, введите в терминале команду `godoc` (еще раз: не перепутайте с `go doc`) со специальным параметром `-http=:6060`.

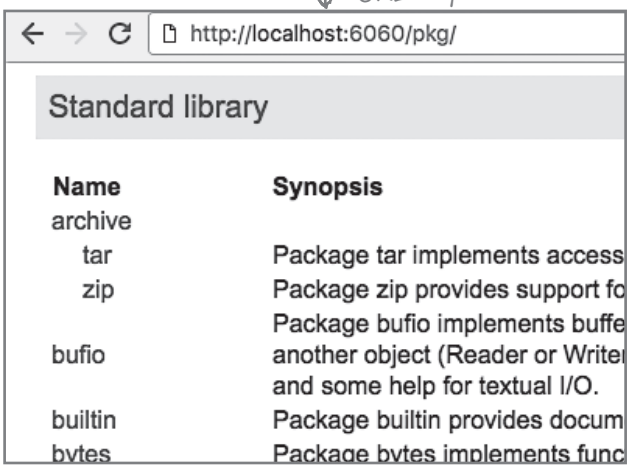
Запуск веб-сервера godoc.

После того как сервер `godoc` будет запущен, введите URL-адрес

```
http://localhost:6060/pkg
```

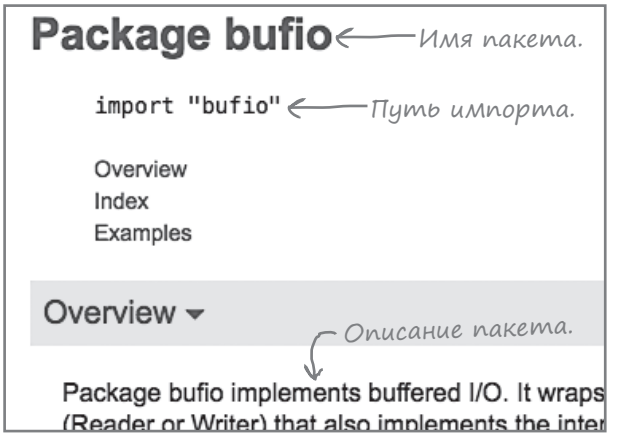
в адресной строке браузера и нажмите Enter. Браузер подключится к вашему собственному компьютеру, а сервер `godoc` ответит HTML-страницей. На экране появится список всех пакетов, установленных на вашей машине.

Введите этот URL-адрес.



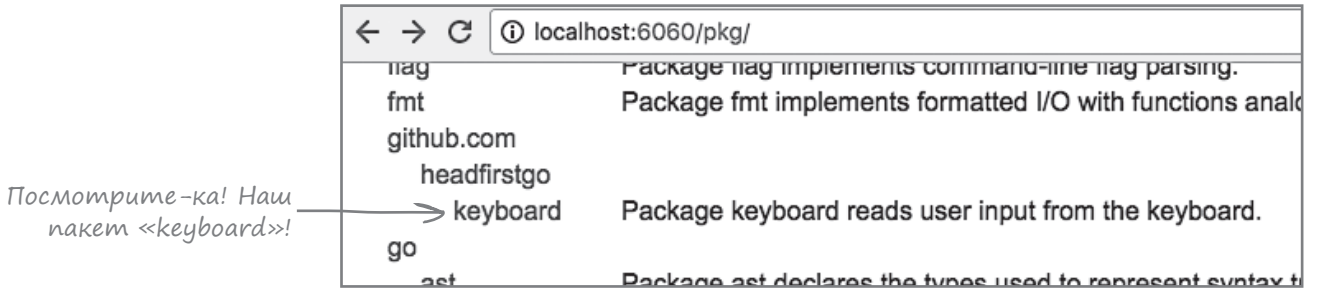
Ссылки на документацию пакетов.

Каждое имя пакета в списке представляет собой ссылку на документацию для этого пакета. Щелкните на ссылке, и вы увидите ту же документацию пакетов, которую обычно просматриваете на *golang.org*.



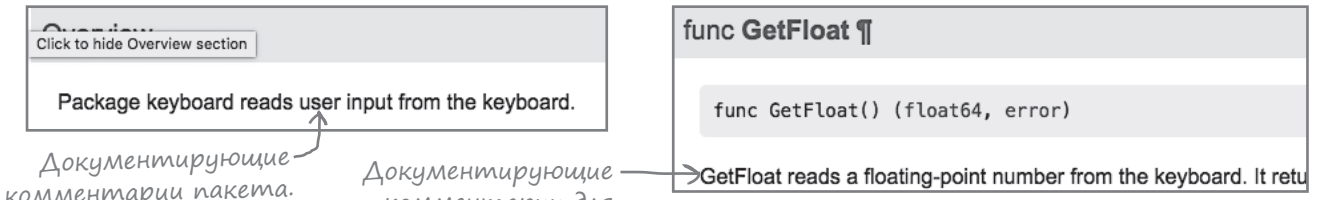
Сервер «godoc» включает ВАШИ пакеты!

Прокрутив список пакетов локального сервера `godoc`, вы обнаружите в нем нечто интересное: наш пакет `keyboard`!



В дополнение к пакетам из стандартной библиотеки Go, сервер `godoc` также строит документацию HTML для любых пакетов в вашей рабочей области Go. Это могут быть как сторонние пакеты, установленные вами, так и пакеты, которые вы написали сами.

Щелкните на ссылке *keyboard*, и в браузере откроется документация пакета. В нее будут включены все документирующие комментарии из вашего кода!



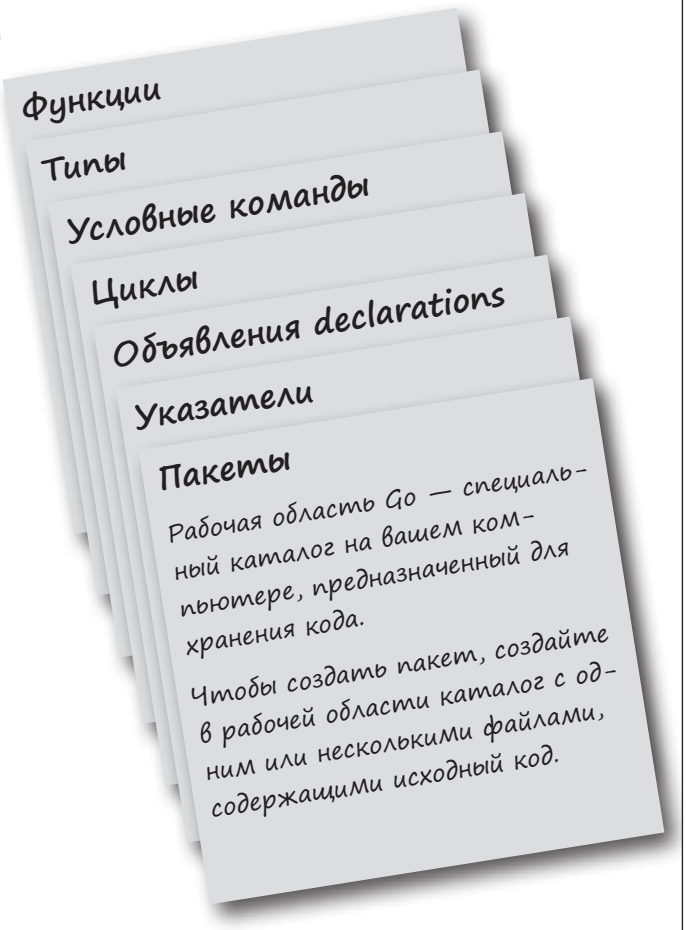
Когда вы захотите остановить сервер `godoc`, вернитесь к окну терминала и, удерживая нажатой клавишу Ctrl, нажмите C. На экране снова появляется системное приглашение.

В Go вы можете легко документировать свои пакеты, упрощая их распространение и использование другими разработчиками. И это всего лишь еще одна причина, по которой пакеты считаются столь удобным механизмом повторного использования кода!

Ваш инструментарий Go

Глава 4 подошла к концу! В ней ваш инструментарий пополнился пакетами.

ГЛАВА 4

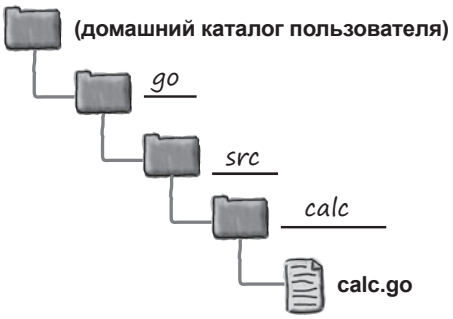


КЛЮЧЕВЫЕ МОМЕНТЫ

- По умолчанию рабочей областью является каталог с именем *go* в домашнем каталоге пользователя.
- Чтобы использовать в качестве рабочей области другой каталог, настройте переменную среды `GOPATH`.
- Go использует три подкаталога в рабочей области: в каталоге *bin* хранятся откомпилированные исполняемые программы, в каталоге *pkg* — откомпилированный код пакетов, а в каталоге *src* — исходный код Go.
- Имена подкаталогов каталога *src* формируют путь импорта пакета. Имена вложенных каталогов разделяются символами `/` в пути импорта.
- Имя пакета определяется директивами `package` в начале файла с исходным кодом в каталоге пакета. За исключением пакета `main`, имя пакета должно совпадать с именем каталога, в котором он находится.
- Имена пакетов должны записываться в нижнем регистре. В идеале они состоят из одного слова.
- Функции пакета могут вызываться за пределами пакета только в том случае, если они **экспортированы**. Функция экспортируется, если ее имя начинается с буквы верхнего регистра.
- **Константой** называется имя для обращения к значению, которое никогда не изменяется.
- Команда `go install` компилирует код пакета и сохраняет его в каталоге *pkg* для пакетов общего назначения или в каталоге *bin* для исполняемых программ.
- В качестве пути импорта пакета принято использовать URL-адрес размещения пакета. В этом случае команда `go get` может находить, загружать и устанавливать пакеты, зная только их путь импорта.
- Команда `go doc` выводит документацию пакетов. В выходные данные `go doc` включаются документирующие комментарии в коде.

У бассейна. Решение упражнения

Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты необязательно. Ваша **задача**: создать в рабочей области Go пакет `calc`, чтобы функции могли использоваться в коде *main.go*.




```
package calc
```

Убедитесь, что имя начинается с буквы верхнего регистра!

```
func Add(first float64, second float64) float64 {
        return first + second
}
```

Убедитесь, что имя начинается с буквы верхнего регистра!

```
func Subtract(first float64, second float64) float64 {
        return first - second
}
```

calc.go

```
package main

import (
        "calc"
        "fmt"
)

func main() {
        fmt.Println(calc.Add(1, 2))
        fmt.Println(calc.Subtract(7, 3))
}
```

main.go

Результат.

```
3
4
```

Упражнение
Решение

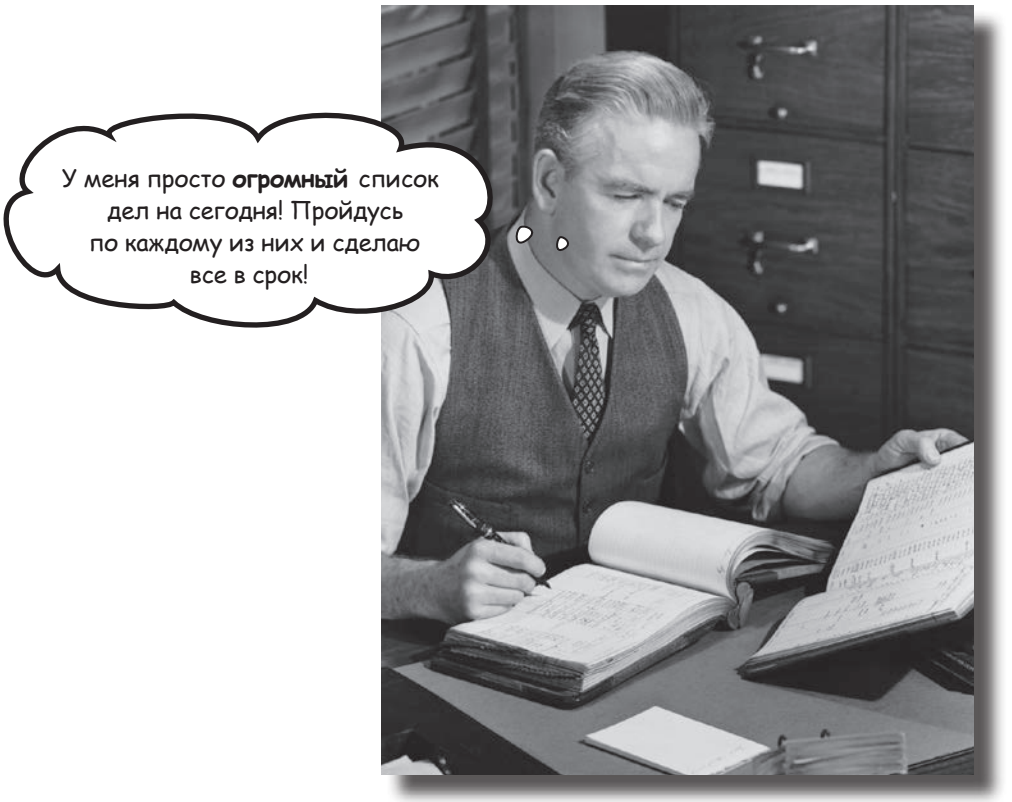
Мы создали рабочую область Go с простым пакетом `mypackage`. Завершите приведенную программу, чтобы она импортировала пакет `mypackage` и вызывала его функцию `MyFunction`.

```
package mypackage

func MyFunction() {
}
```

mypackage.go

```
package main

import «my.com/me/myproject/mypackage»

func main() {
        mypackage.MyFunction()
}
```

Массивы



Многие программы работают со списками. Списки адресов. Списки телефонных номеров. Списки товаров. В Go существуют *два* встроенных способа хранения списков. В этой главе мы разберем первый способ: **массивы**. Вы научитесь создавать массивы, заполнять их данными и извлекать сохраненные данные. Далее рассмотрим способы обработки всех элементов в массиве: сначала *сложный* вариант с циклами `for`, а затем *простой* — с циклами `for...range`.

В массивах хранятся наборы значений

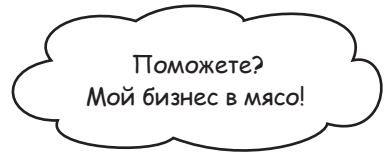
У владельца местного ресторана появилась проблема. Он должен знать, сколько говядины заказывать на следующую неделю. Если заказать слишком много, излишки испортятся. Если слишком мало — ему придется говорить клиентам, что он не может приготовить их любимые блюда.

Он хранит информацию о том, сколько мяса было использовано в предыдущие три недели. Ему нужна программа, которая даст некоторое представление об объеме заказа.

Неделя A:
71,8 фунта

Неделя B:
56,2 фунта

Неделя C:
89,5 фунта

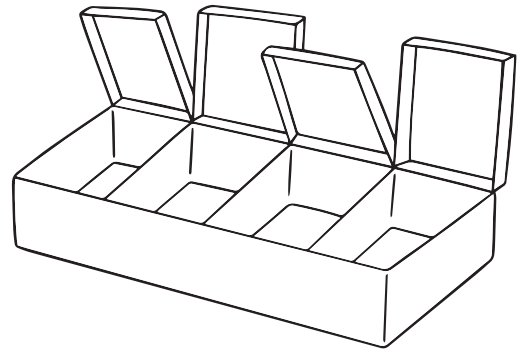


Задача не из сложных: нужно вычислить среднее значение, то есть взять три числа, сложить их и разделить на 3. Среднее значение дает неплохую оценку того, сколько мяса следует заказывать.

```
(неделя A + неделя B + неделя C) ÷ 3 = среднее
```

Первая проблема — хранение значений. Было бы неудобно объявлять три разные переменные, а если в будущем потребуется сложить больше значений, станет еще неудобнее. Но как и большинство языков программирования, Go предоставляет структуру данных, которая идеально подходит для подобных задач...

Массив представляет собой набор значений, относящихся к одному типу. Представьте себе таблетницу — вы можете класть и доставать таблетки в любое отделение, но при этом ее удобно переносить как единое целое.



Значения, хранящиеся в массиве, называются **элементами**. Вы можете создать массив строк, массив логических значений или массив любого другого типа Go (даже массив массивов). Весь массив можно сохранить в одной переменной, а затем обратиться к любому нужному элементу.

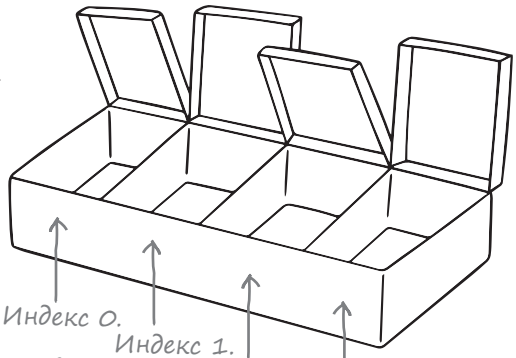
В массивах хранятся наборы значений (продолжение)

Массив содержит заранее заданное количество элементов, а его размер не может увеличиваться или уменьшаться. Чтобы объявить переменную для хранения массива, следует указать количество хранящихся в нем элементов в квадратных скобках (`[]`), а затем тип элементов в массиве.

Количество элементов в массиве.

Тип элементов в массиве.

```
var myArray [4]string
```



Чтобы присвоить значения элементам массива или прочитать их позднее, необходимо каким-то образом указать, какой элемент вам нужен. Элементы в массиве нумеруются, начиная с 0. Номер массива называется его **индексом**.

Например, если вы хотите создать массив с названиями нот, первой ноте будет присвоен индекс `0`, второй — индекс `1`, и т. д. Индекс задается в квадратных скобках.

Создается массив из семи строк.

```
var notes [7]string
notes[0] = "do"
notes[1] = "re"
notes[2] = "mi"
fmt.Println(notes[0])
fmt.Println(notes[1])
```

Присваивается значение первого элемента.

Присваивается значение второго элемента.

Присваивается значение третьего элемента.

Выводится первый элемент.

Выводится второй элемент.

```
do
re
```

Массив целых чисел:

Создается массив из пяти целых чисел.

```
var primes [5]int
primes[0] = 2
primes[1] = 3
fmt.Println(primes[0])
```

Присваивается значение первого элемента.

Присваивается значение второго элемента.

Выводится значение третьего элемента.

```
2
```

Массив значений `time.Time`:

Создается массив из трех значений Time.

```
var dates [3]time.Time
dates[0] = time.Unix(1257894000, 0)
dates[1] = time.Unix(1447920000, 0)
dates[2] = time.Unix(1508632200, 0)
fmt.Println(dates[1])
```

Присваивается значение первого элемента.

Присваивается значение второго элемента.

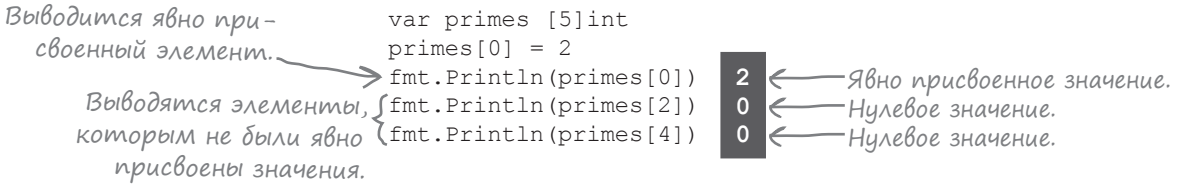
Присваивается значение третьего элемента.

Выводится второй элемент.

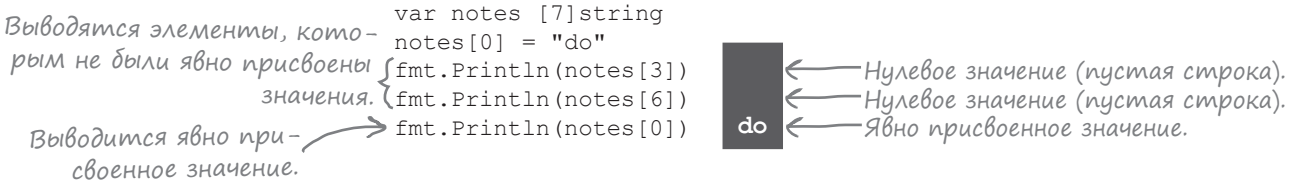
```
2015-11-19 08:00:00 +0000 UTC
```

Нулевые значения в массивах

Как и в случае с переменными, при создании массивов все содержащиеся в них значения инициализируются нулевым значением для типа, содержащегося в массиве. Так массив значений `int` по умолчанию заполняется нулями:



С другой стороны, нулевым значением для строк является пустая строка, так что массив строковых значений по умолчанию заполняется пустыми строками:



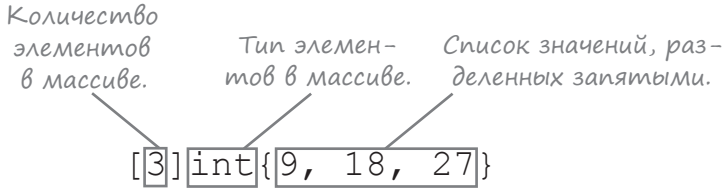
Нулевые значения позволяют безопасно выполнять операции с элементами массивов, даже если им не были присвоены значения. Например, в следующем массиве хранятся целочисленные счетчики. Любой элемент можно увеличить на 1 даже без предварительного присваивания значения, потому что мы знаем, что все значения счетчиков начинаются с 0.

```
var counters [3]int
counters[0]++   // Первый элемент увеличивается с 0 до 1.
counters[0]++   // Первый элемент увеличивается с 1 до 2.
counters[2]++   // Третий элемент увеличивается с 0 до 1.
fmt.Println(counters[0], counters[1], counters[2])
```

При создании массива все содержащиеся в нем элементы инициализируются нулевым значением для типа, хранящегося в массиве.

Литералы массивов

Если вам заранее известны значения, которые должны храниться в массиве, вы можете инициализировать массив этими значениями в форме **литерала массива**. Литерал массива начинается как тип массива — с количества элементов в квадратных скобках, за которым следует тип элементов. Далее в фигурных скобках идет список исходных значений элементов массива. Значения элементов должны разделяться запятыми.



Эти примеры почти не отличаются от предыдущих, если не считать того, что вместо последовательного присваивания значений элементам массива весь массив инициализируется с использованием литерала массива.

```
var notes [7]string = [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}
fmt.Println(notes[3], notes[6], notes[0])
var primes [5]int = [5]int{2, 3, 5, 7, 11}
fmt.Println(primes[0], primes[2], primes[4])
```

Присваивание значений в форме литерала массива.

Присваивание значений в форме литерала массива.

```
fa ti do
2 5 11
```

Литералы массивов также позволяют использовать короткие объявления переменных с помощью `:=`.

Короткое объявление переменной.

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}
primes := [5]int{2, 3, 5, 7, 11}
```

Короткое объявление переменной.

Литералы массивов могут распространяться на несколько строк, но перед каждым переносом строки в коде должна стоять запятая. Запятая даже должна стоять после последнего элемента в литерале массива, если за ним следует перенос строки. (На первый взгляд этот синтаксис выглядит неуклюже, но он упрощает последующее добавление новых элементов в коде.)

```
text := [3]string{
       "This is a series of long strings",
       "which would be awkward to place",
       "together on a single line",
}
```

Все это один массив.

Запятая в конце обязательна.

Упражнение

Следующая программа объявляет пару массивов и выводит все их элементы. Запишите результат, который будет выведен программой.

```
package main

import "fmt"

func main() {
       var numbers [3]int
       numbers[0] = 42
       numbers[2] = 108
       var letters = [3]string{"a", "b", "c"}

                                    Результат:

       fmt.Println(numbers[0])      .....

       fmt.Println(numbers[1])      .....

       fmt.Println(numbers[2])      .....

       fmt.Println(letters[2])      .....

       fmt.Println(letters[0])      .....

       fmt.Println(letters[1])      .....
}
```

Ответ на с. 207.

Функции пакета «fmt» умеют работать с массивами

Когда вы занимаетесь отладкой кода, вам не нужно передавать элементы массивов `Println` и другим функциям пакета `fmt` один за одним. Просто передайте весь массив. Пакет `fmt` содержит логику форматирования и вывода массивов. (Пакет `fmt` также умеет работать с сегментами, картами и другими структурами данных, которые будут описаны позднее.)

```
var notes [3]string = [3]string{"do", "re", "mi"}
var primes [5]int = [5]int{2, 3, 5, 7, 11}
fmt.Println(notes)
fmt.Println(primes)
```

Функции fmt.Println передаются целые массивы.

```
[do re mi]
[2 3 5 7 11]
```

Возможно, вы также помните глагол `"%#v"`, используемый функциями `Printf` и `Sprintf`, — он форматирует значения так, как они отображаются в коде Go. При форматировании с `"%#v"` массивы отображаются в форме литералов массивов Go.

```
fmt.Printf("%#v\n", notes)
fmt.Printf("%#v\n", primes)
```

Массивы форматируются так, как они записываются в коде Go.

```
[3]string{"do", "re", "mi"}
[5]int{2, 3, 5, 7, 11}
```

Обращение к элементам массива в цикле

Вы не обязаны явно записывать целочисленные индексы элементов массивов, к которым обращаетесь в своем коде. В качестве индекса также можно использовать значение целочисленной переменной.

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}
index := 1
fmt.Println(index, notes[index]) <-- Выводит элемент массива с индексом 1.
index = 3
fmt.Println(index, notes[index]) <-- Выводит элемент массива с индексом 3.
```

```
1 re
3 fa
```

Это означает, что элементы массивов можно перебирать в цикле `for`. Цикл перебирает индексы массива, а переменная цикла используется для обращения к элементу с текущим индексом.

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}
for i := 0; i <= 2; i++ {   <-- Перебирает индексы 0, 1 и 2.
        fmt.Println(i, notes[i])
}
```

Выводит элемент с текущим индексом.

```
0 do
1 re
2 mi
```

При обращении к элементам массивов через переменную необходимо действовать внимательно и следить за тем, какие значения индексов используются в программе. Как упоминалось ранее, массивы содержат конкретное число элементов. Попытка обратиться к индексу за пределами массива приводит к **панике** — ошибке, происходящей во время выполнения программы (а не на стадии компиляции).

Массив содержит семь элементов.

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}

for i := 0; i <= 7; i++ {   <-- Перебирает индексы до 7 (восьмой элемент, который не существует)!
        fmt.Println(i, notes[i])
}
```

Обычно в ситуации паники программа аварийно завершается с выводом сообщения об ошибке для пользователя. Не стоит и говорить, что таких ситуаций следует по возможности избегать.

Обращения к индексам от 0 до 6.

Обращение к индексу 7 приводит к панике!

```
0 do
1 re
2 mi
3 fa
4 so
5 la
6 ti
panic: runtime error: index out of range

goroutine 1 [running]:
main.main()
        /tmp/sandbox732328648/main.go:8 +0x140
```

Проверка длины массива функцией «len»

Написание циклов, которые ограничиваются только правильными индексами, сопряжено с определенным риском ошибок. К счастью, есть пара приемов, которые упрощают этот процесс.

Во-первых, вы можете проверить фактическое количество элементов в массиве перед обращением к элементу. Для этого можно воспользоваться встроенной функцией `len`, которая возвращает длину массива (количество содержащихся в нем элементов).

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}
fmt.Println(len(notes))
primes := [5]int{2, 3, 5, 7, 11}
fmt.Println(len(primes))
```

Выводит длину массива «notes».

Выводит длину массива «primes».

```
7
5
```

В цикле обработки всего массива можно воспользоваться функцией `len` для определения того, по каким индексам можно обращаться безопасно.

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}

for i := 0; i < len(notes); i++ {
        fmt.Println(i, notes[i])
}
```

Наибольшее значение, которого достигнет переменная «i», равно 6.

Возвращает длину массива 7.

```
0 do
1 re
2 mi
3 fa
4 so
5 la
6 ti
```

Впрочем, и здесь существует некоторый риск ошибок. Хотя `len(notes)` возвращает `7`, наибольший индекс, к которому вы можете обращаться, равен `6` (потому что индексирование массивов начинается с `0`, а не с `1`). При попытке обратиться по индексу `7` возникнет ситуация паники.

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}

for i := 0; i <= len(notes); i++ {
        fmt.Println(i, notes[i])
}
```

Наибольшее значение, которого достигнет переменная «i», равно 7!

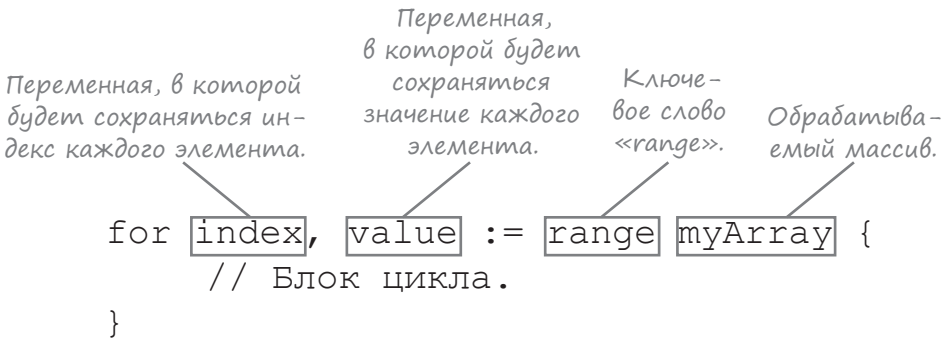
Возвращает длину массива 7.

```
0 do
1 re
2 mi
3 fa
4 so
5 la
6 ti
panic: runtime error: index out of range

goroutine 1 [running]:
main.main()
        /tmp/sandbox094804331/main.go:11 +0x140
```

Обращение к индексу 7 приводит к панике!

Безопасный перебор массивов в цикле «for...range»

В другом, еще более безопасном способе обработки всех элементов массива используется специальный цикл `for...range`. В форме с `range` указывается переменная для хранения целочисленного индекса каждого элемента, другая переменная для хранения значения самого элемента и перебираемый массив. Цикл выполняется по одному разу для каждого элемента в массиве; индекс элемента присваивается первой переменной, а значение элемента — второй переменной. В блок цикла включается код для обработки этих значений.



Эта форма цикла `for` не содержит запутанных выражений инициализации, условия и завершения. А поскольку значение элемента автоматически присваивается переменной, риск обращения к недействительному индексу массива исключен. Форма цикла `for` с `range` читается безопаснее и проще, поэтому именно она чаще всего встречается при работе с массивами и другими коллекциями.

Ниже приведен пример с выводом всех значений из массива `notes`, преобразованный для использования цикла `for...range`:

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}
```

Переменная для хранения индексов. Переменная для хранения строк. Обработка каждого значения в массиве.

```
for index, note := range notes {
        fmt.Println(index, note)
}
```

```
0 do
1 re
2 mi
3 fa
4 so
5 la
6 ti
```

Цикл выполняется семь раз, по одному разу для каждого элемента в массиве `notes`. Для каждого элемента переменной `index` присваивается индекс элемента, а переменной `note` присваивается значение элемента. После этого мы выводим индекс и значение.

Пустой идентификатор в циклах «for...range»

Как обычно, Go требует, чтобы каждая объявленная переменная использовалась в программе. Если отказаться от использования переменной `index` в цикле `for...range`, вы получите сообщение об ошибке:

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}

for index, note := range notes {
       fmt.Println(note)
}
```

Переменная «index» исключена из вывода.

Ошибка компиляции.

```
index declared and not used
```

То же самое произойдет и в том случае, если мы не будем использовать переменную со значением элемента:

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}

for index, note := range notes {
       fmt.Println(index)
}
```

Переменная «note» не используется.

Ошибка компиляции.

```
note declared and not used
```

Помните, как в главе 2 при вызове функции с несколькими возвращаемыми значениями мы хотели проигнорировать одно из них? Это значение присваивалось пустому идентификатору (_), чтобы компилятор Go просто отбросил это значение без выдачи сообщения об ошибке...

То же самое можно проделать со значениями из циклов `for...range`. Если вам не нужен индекс каждого элемента массива, присвойте его пустому идентификатору:

Пустой идентификатор используется как заменитель для индекса.

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}

for _, note := range notes {
       fmt.Println(note)
}
```

Используется только переменная «note».

```
do
re
mi
fa
so
la
ti
```

А если вам не нужна переменная для значения, замените ее пустым идентификатором:

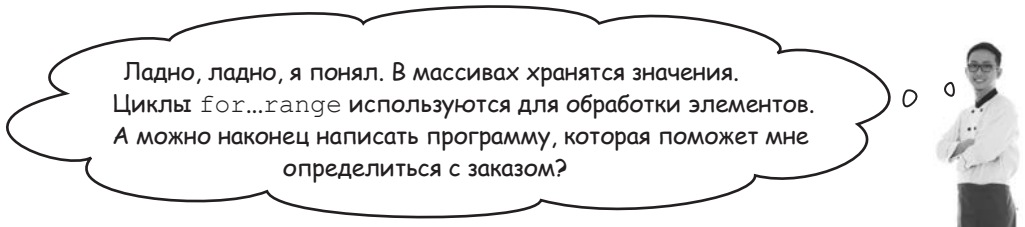
Пустой идентификатор используется как заменитель для значения элемента.

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}

for index, _ := range notes {
       fmt.Println(index)
}
```

Используется только переменная «index».

```
0
1
2
3
4
5
6
```

Суммирование чисел в массиве



Наконец-то мы знаем все, что нам нужно для создания массива значений `float64` и вычисления их среднего значения. А теперь возьмем объемы заказов за прошлые недели и встроим их в свою программу, которая называется `average`.

Все начинается с создания файла программы. В каталоге вашей рабочей области Go (каталог *go* в домашнем каталоге пользователя, если только вы не изменили переменную среды `GOPATH`) создайте следующую структуру вложенных каталогов (если они еще не существуют). В последнем каталоге *average* сохраните файл с именем *main.go*.

Код нашей программы будет содержаться в файле *main.go*. Так как это исполняемая программа, код будет частью пакета `main` и будет находиться в функции `main`.

Начнем с простого вычисления суммы трех значений, а потом вернемся к вычислению среднего. Мы воспользуемся литералом массива для создания массива трех значений `float64`, предварительно инициализированных данными за предыдущие недели. Объявим переменную `float64` с именем `sum` для хранения суммы, которая изначально равна `0`.

Затем все числа будут обработаны в цикле `for...range`. Индексы элементов не нужны, поэтому мы игнорируем их при помощи пустого идентификатора `_`. Каждое число прибавляется к значению `sum`. После того как все значения будут просуммированы, результат выводится перед завершением программы.

```
// average вычисляет среднее значение.
package main
```
Это исполняемая программа, поэтому используем пакет «main».

```
import "fmt"

func main() {
        numbers := [3]float64{71.8, 56.2, 89.5}
        var sum float64 = 0
        for _, number := range numbers {
                sum += number
        }
        fmt.Println(sum)
}
```

Литерал массива используется для создания массива с тремя числами, для которых вычисляется среднее.

Объявляем переменную float64 для хранения суммы трех чисел.

Перебор всех чисел в массиве.

Индекс элемента игнорируется.

Текущее число прибавляется к сумме.

Суммирование чисел в массиве (продолжение)

Попробуем откомпилировать и выполнить программу. Мы используем команду `go install` для построения исполняемого файла. Команде `go install` нужно передать путь импорта нашего исполняемого файла. Если используется следующая структура каталогов...

...это означает, что путь импорта для нашего пакета имеет вид `github.com/headfirstgo/average`. Итак, введите в терминале следующую команду:

```
go install github.com/headfirstgo/average
```

Это можно сделать из любого каталога. Команда ищет каталог *github.com/headfirstgo/average* внутри каталога *src* вашей рабочей области и компилирует все содержащиеся в нем файлы *.go*. Полученный исполняемый файл с именем `average` будет находиться в каталоге *bin* в вашей рабочей области Go.

После этого перейдите к каталогу *bin* своей рабочей области Go при помощи команды **`cd`**. В каталоге *bin* исполняемый файл запускается командой **`./average`** (или **`average.exe`** в Windows).

Компилирует содержимое каталога «average» и устанавливает полученный исполняемый файл.

Переход к каталогу «bin» вашей рабочей области.

Запуск исполняемого файла.

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average
$ cd /Users/jay/go/bin
$ ./average
217.5
$
```

Программа выводит сумму трех чисел из массива и завершает работу.

Вычисление среднего значения

Программа `average` выводит сумму значений из массива; теперь обновим ее, чтобы она выводила среднее значение. Для этого следует разделить сумму на длину массива.

Если передать массив функции `len`, функция вернет длину массива в формате `int`. Но поскольку сумма в переменной `sum` относится к типу `float64`, длину также необходимо преобразовать к типу `float64`, чтобы эти числа можно было использовать в одной математической операции. Результат сохраняется в переменной `sampleCount`. Когда это будет сделано, остается лишь разделить `sum` на `sampleCount` и вывести результат.

```
// average вычисляет среднее значение.
package main

import "fmt"

func main() {
        numbers := [3]float64{71.8, 56.2, 89.5}
        var sum float64 = 0
        for _, number := range numbers {
                sum += number
        }
        sampleCount := float64(len(numbers))
        fmt.Printf("Average: %0.2f\n", sum/sampleCount)
}
```

Получить длину массива в виде значения int и преобразовать ее к типу float64.

Чтобы вычислить среднее, делим сумму значений на длину массива.

После того как код будет обновлен, повторите предыдущие действия, чтобы просмотреть новый результат: выполните команду **`go install`**, чтобы заново откомпилировать программу, перейдите в каталог *bin* и запустите обновленный исполняемый файл `average`. Вместо суммы элементов массива будет выведено среднее значение.

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average
$ cd /Users/jay/go/bin
$ ./average
Average: 72.50
$
```

Среднее значение элементов массива.

У бассейна

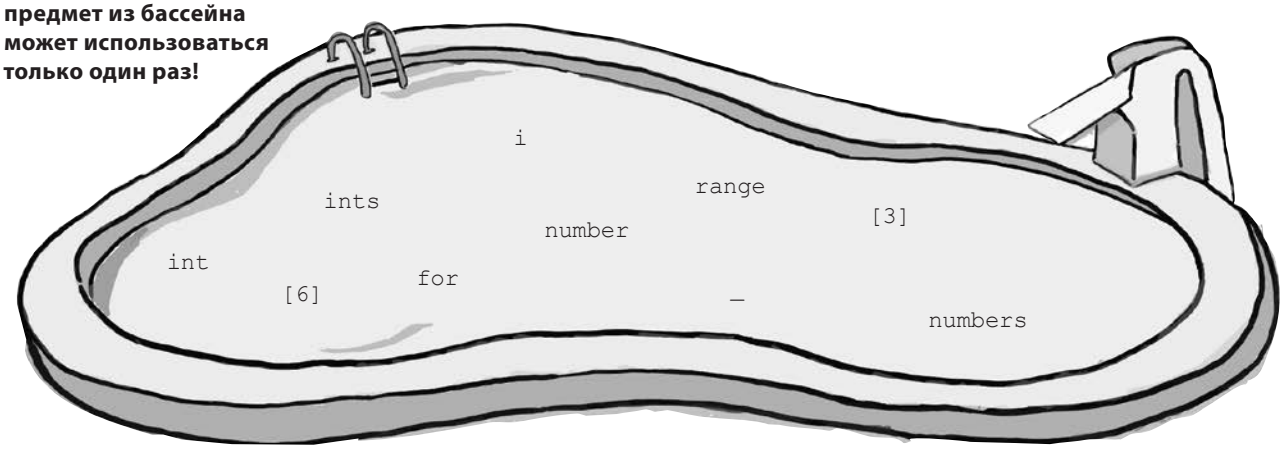
Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: создать программу, которая выводит индексы и значения всех элементов массива, лежащих в диапазоне от 10 до 20 (по приведенному образцу).

```
package main

import "fmt"

func main() {
        _________ := ____int{3, 16, -2, 10, 23, 12}
        for i, _________ := ________ numbers {
                if number >= 10 && number <= 20 {
                        fmt.Println(__, number)
                }
        }
}
```

Результат.

```
1 16
3 10
5 12
```

Примечание: каждый предмет из бассейна может использоваться только один раз!



Ответ на с. 207.

Чтение текстового файла

Превосходно, но ваша программа сообщает только объем заказа на **эту** неделю. А если у меня есть данные за несколько недель? Я не могу отредактировать код, чтобы изменить значения в массиве; у меня на компьютере даже не установлен компилятор Go!

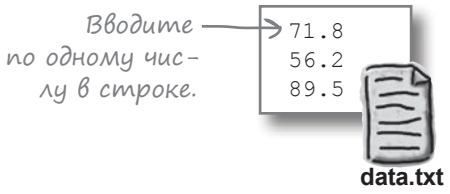
Действительно, программы, в которых пользователю приходится самостоятельно редактировать и компилировать исходный код, не очень-то удобны.

Ранее мы использовали пакеты `os` и `bufio` стандартной библиотеки для чтения данных по строкам с клавиатуры. Те же пакеты могут использоваться и для построчного чтения данных из текстовых файлов. Давайте ненадолго отвлечемся от основной темы и посмотрим, как это делается.

А потом мы вернемся обратно и обновим программу `average`, чтобы она читала числа из текстового файла.

В своем любимом текстовом редакторе создайте новый файл с именем *data.txt*. Сохраните его где-нибудь *за пределами своего каталога рабочей области Go*.

Запишите в файле три наших значения с плавающей точкой, по одному числу в строке.



Чтение текстового файла (продолжение)

Прежде чем браться за обновление программы для вычисления среднего значения чисел из текстового файла, необходимо прочитать содержимое файла. Для начала напишем программу, которая только читает текстовый файл, а затем применим новые знания в своей программе `average`.

Создайте в каталоге с файлом *data.txt* новую программу с именем *readfile.go*. Мы будем запускать *readfile.go* командой `go run` (вместо установки), поэтому программа может храниться за пределами каталога рабочей области Go. Сохраните приведенный ниже код в файле *readfile.go*. (На следующей странице мы более подробно разберемся в том, как работает этот код.)


```
package main

import (
        "bufio"
        "fmt"
        "log"
        "os"
)

func main() {
        file, err := os.Open("data.txt")
        if err != nil {
                log.Fatal(err)
        }
        scanner := bufio.NewScanner(file)
        for scanner.Scan() {
                fmt.Println(scanner.Text())
        }
        err = file.Close()
        if err != nil {
                log.Fatal(err)
        }
        if scanner.Err() != nil {
                log.Fatal(scanner.Err())
        }
}
```

Файл данных открывается для чтения.

Если при открытии файла произошла ошибка, сообщить о ней и завершить работу.

Для файла создается новое значение Scanner.

Цикл выполняется до того, как будет достигнут конец файла, а scanner.Scan вернет false.

Читает строку из файла.

Выводит строку.

Закрывает файл для освобождения ресурсов.

Если при закрытии файла произошла ошибка, сообщить о ней и завершить работу.

Если при сканировании файла произошла ошибка, сообщить о ней и завершить работу.

Затем в терминале перейдите в каталог, в котором хранятся эти два файла, и выполните команду **`go run readfile.go`**. Программа читает содержимое файла *data.txt* и выводит его.

Переходит в каталог, в котором были сохранены файлы data.txt b readfile.go.

Запускает readfile.go.

Содержимое data.txt выводится в терминале.

```
Shell Edit View Window Help
$ cd /Users/jay/code
$ go run readfile.go
71.8
56.2
89.5
```

Чтение текстового файла (продолжение)

Наша тестовая программа *readfile.go* успешно читает данные из файла *data.txt* и выводит их. А теперь разберемся, как же она работает.

Сначала строка с именем открываемого файла передается функции `os.Open`. Эта функция возвращает два значения: указатель `os.File`, представляющий открытый файл, и значение ошибки. Как и в случае с другими функциями, если значение ошибки равно `nil`, это означает, что файл был открыт успешно, но любое другое значение указывает на то, что произошла ошибка (например, если файл отсутствует или не читается). В таком случае программа выводит сообщение об ошибке и завершается.

Файл данных открывается для чтения.

```
file, err := os.Open("data.txt")
if err != nil {
        log.Fatal(err)
}
```

Если при открытии файла происходит ошибка, выводится сообщение, а программа завершается.

Затем значение `os.File` передается функции `bufio.NewScanner`. Функция возвращает значение `bufio.Scanner` для чтения данных из файла.

Создает новое значение Scanner для файла.

```
scanner := bufio.NewScanner(file)
```

Метод `Scan` значения `bufio.Scanner` создан для использования в циклах `for`. Он читает одну строку текста из файла и возвращает `true`, если данные прочитаны успешно, или `false` в случае ошибки. Если `Scan` используется как условие цикла `for`, цикл продолжит выполняться, пока остаются данные для чтения. При достижении конца файла (или возникновении ошибки) `Scan` вернет `false` и цикл завершится.

После вызова метода `Scan` для значения `bufio.Scanner` вызов метода `Text` возвращает строку с прочитанными данными. В этой программе мы просто вызываем `Println` в цикле, чтобы вывести каждую прочитанную строку.

Цикл выполняется до того, как будет достигнут конец файла, а scanner.Scan вернет false

```
for scanner.Scan() {
        fmt.Println(scanner.Text())
}
```

Читает строку из файла.

Выводит строку.

После выхода из цикла чтение файла завершено. Если файл останется открытым, он расходует ресурсы операционной системы, поэтому файлы всегда должны закрываться после завершения работы с ними. Для этого следует вызвать метод `Close` для значения `os.File`. Как и функция `Open`, метод `Close` возвращает значение ошибки, которое равно `nil` при отсутствии проблем. (В отличие от `Open`, `Close` возвращает *только одно* значение — в данном случае нет осмысленного значения, которое можно было бы вернуть, кроме значения ошибки.)

Закрывает файл для освобождения ресурсов.

```
err = file.Close()
if err != nil {
        log.Fatal(err)
}
```

Если при закрытии файла произошла ошибка, сообщить о ней и завершить программу.

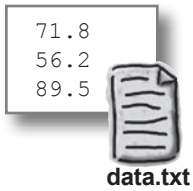
Также нельзя исключать, что метод `bufio.Scanner` вернет ошибку при сканировании файла. В таком случае вызов метода `Err` вернет эту ошибку, которая выводится перед завершением программы.

Если при сканировании файла произошла ошибка, сообщить о ней и завершить программу.

```
if scanner.Err() != nil {
        log.Fatal(scanner.Err())
}
```

Чтение текстового файла в массив

Программа *readfile.go* прекрасно работает — она читает данные из файла *data.txt* в формате строк и выводит их. Теперь необходимо преобразовать эти строки в числа и сохранить их в массиве. Создадим пакет с именем `datafile`, который сделает это за нас.



Создайте в своем каталоге рабочей области Go каталог *datafile* внутри каталога *headfirstgo*. В каталоге *datafile* сохраните файл с именем *floats.go*. (Этот файл будет содержать код для чтения данных с плавающей точкой из файлов.)

Сохраните в файле *floats.go* приведенный ниже код. Большая его часть основана на коде тестовой программы *readfile.go*; фрагменты с идентичным кодом выделены серым цветом. Новый код будет подробно описан на следующей странице.

```
// Пакет datafile предназначен для чтения данных из файлов.
package datafile

import (
        "bufio"
        "os"
        "strconv"
)

// GetFloats читает значение float64 из каждой строки файла.
func GetFloats(fileName string) ([3]float64, error) {
        var numbers [3]float64
        file, err := os.Open(fileName)
        if err != nil {
                return numbers, err
        }
        i := 0
        scanner := bufio.NewScanner(file)
        for scanner.Scan() {
                numbers[i], err = strconv.ParseFloat(scanner.Text(), 64)
                if err != nil {
                        return numbers, err
                }
                i++
        }
        err = file.Close()
        if err != nil {
                return numbers, err
        }
        if scanner.Err() != nil {
                return numbers, scanner.Err()
        }
        return numbers, nil
}
```

Имя файла с данными передается в аргументе.

Функция возвращает массив чисел и любую обнаруженную ошибку.

Объявление возвращаемого массива.

Открывает файл с переданным именем.

Если при открытии файла произошла ошибка, вернуть ее.

Переменная для хранения индекса, по которому должно выполняться присваивание.

Если при преобразовании значения в число произошла ошибка, вернуть ее.

Строка, прочитанная из файла, преобразуется в float64.

Переход к следующему индексу массива.

Если при закрытии файла произошла ошибка, вернуть ее.

Если при сканировании файла произошла ошибка, вернуть ее.

Если выполнение дошло до этой точки, значит, ошибок не было, поэтому программа возвращает массив чисел и значение ошибки «nil».

Чтение текстового файла в массив (продолжение)

Мы хотим, чтобы данные можно было читать из других файлов (не только из *data.txt*), поэтому имя открываемого файла передается в параметре. Согласно объявлению функция возвращает два значения: массив значений `float64` и значение ошибки. Как и у большинства функций, возвращающих значение ошибки, первое возвращаемое значение должно считаться достоверным только в том случае, если значение ошибки равно `nil`.

Имя файла с данными передается в аргументе.

Функция будет возвращать массив чисел и обнаруженную ошибку.

```
func GetFloats(fileName string) ([3]float64, error) {
```

Затем объявляется массив с тремя элементами `float64`, в которых будут храниться числа, прочитанные из файла.

```
var numbers [3]float64
```
Объявление возвращаемого массива.

Как и в случае с *readfile.go*, файл открывается для чтения. Новый вариант отличается тем, что вместо фиксированной строки `"data.txt"` используется имя файла, переданное функции. При обнаружении ошибки необходимо вернуть массив вместе со значением ошибки, поэтому мы просто возвращаем массив `numbers` (несмотря на то, что ему еще ничего не присвоено).

Если при открытии файла произошла ошибка, вернуть ее.

```
file, err := os.Open(fileName)
if err != nil {
        return numbers, err
}
```
Открывает файл с переданным именем.

Необходимо знать, какому элементу массива должна присваиваться очередная строка, поэтому мы создаем переменную для отслеживания текущего индекса.

```
i := 0
```
Переменная для хранения индекса, по которому должно выполняться присваивание.

Код создания `bufio.Scanner` и перебора строк файла идентичен коду из *readfile.go*. Тем не менее код в цикле отличается: мы вызываем `strconv.ParseFloat` для строки, прочитанной из файла, чтобы преобразовать ее в `float64`, после чего результат присваивается массиву. Если при выполнении `ParseFloat` происходит ошибка, ее необходимо вернуть. А если преобразование прошло успешно, необходимо увеличить `i`, чтобы следующее число присваивалось следующему элементу массива.

Если при преобразовании значения в число произошла ошибка, вернуть ее.

```
numbers[i], err = strconv.ParseFloat(scanner.Text(), 64)
if err != nil {
        return numbers, err
}
i++
```
Строка, прочитанная из файла, преобразуется в float64.

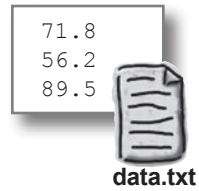
Переход к следующему индексу в массиве.

Код закрытия файла и вывода сообщений об ошибках идентичен коду *readfile.go*, если не считать того, что код возвращает ошибки вместо того, чтобы сразу завершать выполнение программы. Если ошибок не было, конец функции `GetFloats` будет успешно достигнут, а функция вернет массив значений `float64` вместе с ошибкой `nil`.

Если при сканировании файла произошла ошибка, вернуть ее.

```
if scanner.Err() != nil {
        return numbers, scanner.Err()
}
return numbers, nil
```
Если выполнение дошло до этой точки, значит, ошибок не было, поэтому программа возвращает массив чисел и значение ошибки «nil».

Чтение текстового файла в программе «average»



Все готово для замены жестко запрограммированного массива в программе `average` массивом, который читается из файла *data.txt*!

Самое сложное — написание пакета `datafile` — уже сделано. В программе `main` осталось сделать еще три вещи:

- Обновить директиву `import` для включения пакетов `datafile` и `log`.
- Заменить массив жестко запрограммированных чисел вызовом `datafile.GetFloats("data.txt")`.
- Проверить, была ли возвращена ошибка при вызове `GetFloats`, вывести сообщение и завершить работу программы.

Весь остальной код останется точно таким же.


```
// average вычисляет среднее значение.
package main

import (
        "fmt"
        "github.com/headfirstgo/datafile"
        "log"
)

func main() {
        numbers, err := datafile.GetFloats("data.txt")
        if err != nil {
                log.Fatal(err)
        }
        var sum float64 = 0
        for _, number := range numbers {
                sum += number
        }
        sampleCount := float64(len(numbers))
        fmt.Printf("Average: %0.2f\n", sum/sampleCount)
}
```

Импортируем пакет.

Импортируем пакет «log».

Загружает файл data.txt, разбирает содержащиеся в нем числа и сохраняет массив.

Если произошла ошибка, программа сообщает о ней и завершает работу.

Чтение текстового файла в программе «average» (продолжение)

Программа компилируется той же командой терминала:

```
go install github.com/headfirstgo/average
```

Так как наша программа импортирует пакет `datafile`, этот пакет тоже будет автоматически откомпилирован.

Компилирует как программу «average», так и пакет «datafile», от которого она зависит.

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average
```

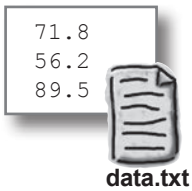
Файл *data.txt* необходимо переместить в подкаталог *bin* рабочей области Go. Дело в том, что исполняемый файл `average` будет запускаться из этого каталога и будет искать файл *data.txt* в этом же каталоге. После перемещения файла *data.txt* перейдите в подкаталог *bin*.

Переместите файл data.txt в подкаталог «bin» рабочей области. (Используйте соответствующую команду своей операционной системы или сохраните файл заново в текстовом редакторе.)

```
Shell Edit View Window Help
$ mv data.txt /Users/jay/go/bin
$ cd /Users/jay/go/bin
```

Переход в подкаталог «bin».

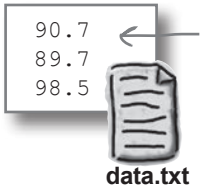
Когда вы запустите исполняемый файл `average`, он загрузит значения из файла *data.txt* в массив и использует их для вычисления среднего.



Среднее значение для данных из data.txt.

```
Shell Edit View Window Help
$ ./average
Average: 72.50
```

Если изменить значения в файле *data.txt*, среднее значение также изменится.



Изменяем данные.

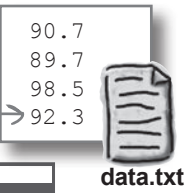
Среднее значение изменилось.

```
Shell Edit View Window Help
$ ./average
Average: 92.97
```

Наша программа может обрабатывать только три значения!

Но тут возникает проблема — программа `average` работает только в том случае, если файл *data.txt* состоит из трех и менее строк. Если их четыре и более, в программе `average` происходит ошибка и она завершает работу!

Если добавить четвертую строку...



В программе произойдет ошибка, и она завершится!

```
Shell Edit View Window Help
$ ./average
panic: runtime error: index out of range

goroutine 1 [running]:
github.com/headfirstgo/datafile.GetFloats(0x10cd018, ...)
        /Users/jay/go/src/github.com/headfirstgo/
        datafile/floats.go:20 +0x39d
```

Компилятор сообщает об ошибке в строке 20 файла floats.go...

Когда в программе Go возникает ситуация паники, она выводит отчет с информацией о строке кода, в которой возникла ошибка. В нашем случае проблема возникла в строке 20 файла *floats.go*.

Взгляните на строку 20 файла *floats.go* — это часть функции `GetFloats`, в которой числа из файла добавляются в массив!

```
// ...предшествующий код пропущен...
func GetFloats(fileName string) ([3]float64, error) {
        var numbers [3]float64
        file, err := os.Open(fileName)
        if err != nil {
                return numbers, err
        }
        i := 0
        scanner := bufio.NewScanner(file)
        for scanner.Scan() {
                numbers[i], err = strconv.ParseFloat(scanner.Text(), 64)
                if err != nil {
                        return numbers, err
                }
                i++
        }
        // ...
}
```

Вот строка 20, в которой число присваивается элементу массива!

Наша программа может обрабатывать только три значения! (продолжение)

Помните, как ошибка в предыдущем примере привела к попытке обращения к восьмому элементу массива из семи элементов? В той программе тоже возникла ситуация паники с аварийным завершением.

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}

for i := 0; i <= 7; i++ {
       fmt.Println(i, notes[i])
}
```

Массив содержит только семь элементов.

Перебор до индекса 7 (восьмой элемент, который не существует)!

```
0 do
1 re
2 mi
3 fa
4 so
5 la
6 ti
panic: runtime error: index out of range
```

Обращения к индексам от 0 до 6.

Обращение к индексу 7 порождает ситуацию паники!

Та же проблема возникает в функции `GetFloats`. Поскольку мы объявили, что массив `numbers` содержит три элемента, *других* элементов в нем быть не может. При достижении четвертой строки файла *data.txt* программа пытается присвоить значение *четвертому* элементу `numbers`, а это приводит к ситуации паники.

```
func GetFloats(fileName string) ([3]float64, error) {
       var numbers [3]float64
       file, err := os.Open(fileName)
       if err != nil {
              return numbers, err
       }
       i := 0
       scanner := bufio.NewScanner(file)
       for scanner.Scan() {
              numbers[i], err = strconv.ParseFloat(scanner.Text(), 64)
              if err != nil {
                     return numbers, err
              }
              i++
       }
       // ...
}
```

Допустимы только индексы с numbers[0] до numbers [2]...

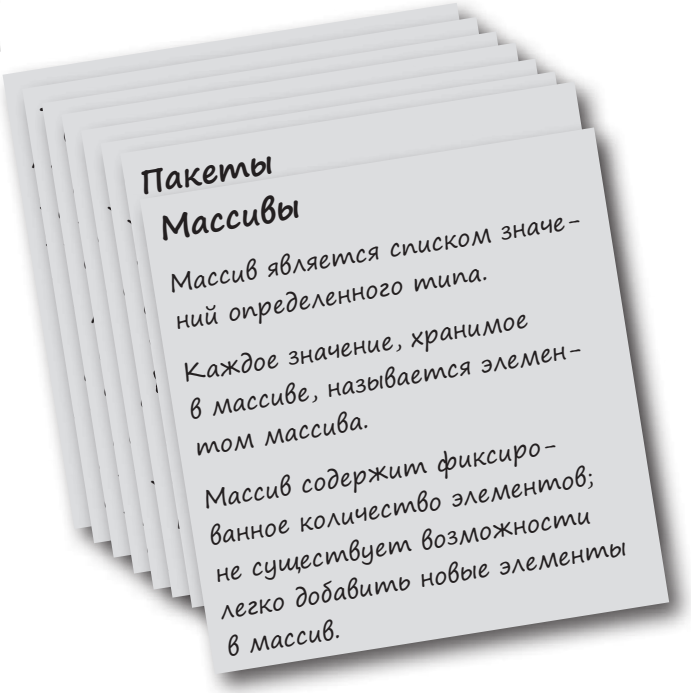
Попытка присваивания numbers[3] приводит к ситуации паники!

Массивы в Go имеют фиксированный размер; они не могут увеличиваться или уменьшаться. Но файл *data.txt* может содержать столько строк, сколько в него добавит пользователь. В следующей главе мы расскажем, как разрешается эта проблема!

Ваш инструментарий Go

Глава 5 подошла к концу! В ней ваш инструментарий пополнился массивами.

ГЛАВА 5



КЛЮЧЕВЫЕ МОМЕНТЫ

- Чтобы объявить переменную-массив, укажите длину массива в квадратных скобках и тип хранящихся в нем элементов: `var myArray [3]int`
- Чтобы прочитать или присвоить значение элемента массива, укажите его индекс в квадратных скобках. Индексы начинаются с 0, поэтому первый элемент `myArray` обозначается `myArray[0]`.
- Как и переменные, по умолчанию все элементы массива инициализируются нулевым значением для типа элемента.
- Элементы массива можно инициализировать в момент создания; для этого используется **литерал массива**: `[3]int{4, 9, 6}`
- Если сохранить недопустимый индекс массива в переменной, а потом попытаться обратиться к элементу с использованием этой переменной в качестве индекса, возникнет ситуация паники — ошибка времени выполнения.
- Для получения количества элементов в массиве используется встроенная функция `len`.
- Все элементы массива можно удобно обработать в специальном синтаксисе цикла `for... range`. Этот цикл перебирает все элементы и присваивает индекс и значение каждого элемента указанным вами переменным.
- При использовании цикла `for...range` можно игнорировать индекс или значение каждого элемента при помощи пустого идентификатора `_`.
- Функция `os.Open` открывает файл. Она возвращает указатель на значение `os.File`, представляющее открытый файл.
- При передаче значения `os.File` функции `bufio.NewScanner` возвращается значение `bufio.Scanner`. Его методы `Scan` и `Text` используются для последовательного чтения файла по строкам.

Упражнение
Решение

Следующая программа объявляет пару массивов и выводит все их элементы. Запишите результат, который будет выведен программой.

```
package main

import "fmt"

func main() {
       var numbers [3]int
       numbers[0] = 42
       numbers[2] = 108
       var letters = [3]string{"a", "b", "c"}

                                          Результат:

       fmt.Println(numbers[0])            42
       fmt.Println(numbers[1])            0
       fmt.Println(numbers[2])            108
       fmt.Println(letters[2])            c
       fmt.Println(letters[0])            a
       fmt.Println(letters[1])            b
}
```

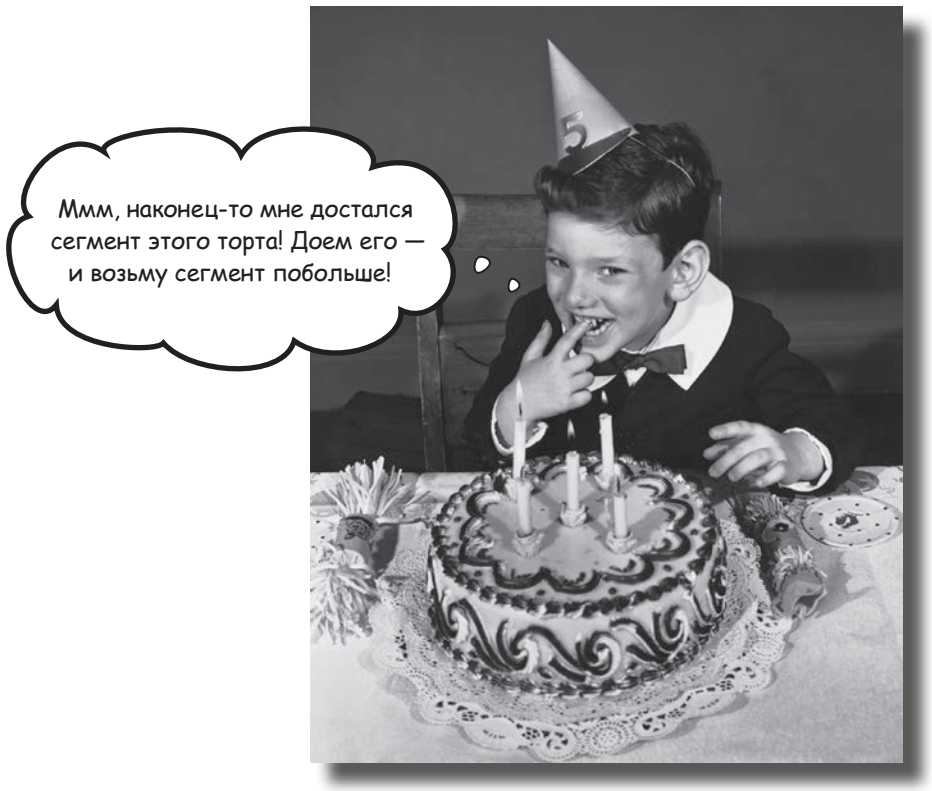
У бассейна. Решение

```
package main

import "fmt"

func main() {
       numbers := [6]int{3, 16, -2, 10, 23, 12}
       for i, number := range numbers {
              if number >= 10 && number <= 20 {
                     fmt.Println(i, number)
              }
       }
}
```

Результат.

```
1 16
3 10
5 12
```

Сегменты

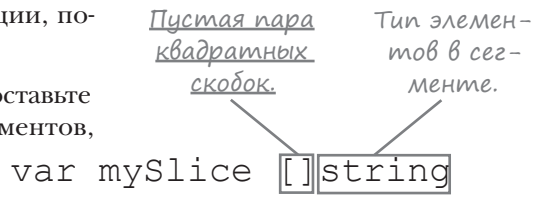


Вы уже знаете, что в массив нельзя добавить новые элементы. В нашей программе это создает настоящие проблемы, потому что количество значений данных в файле неизвестно заранее. На помощь приходят **сегменты** Go. Сегменты — разновидность коллекций, которые могут расширяться для хранения дополнительных элементов; а это как раз то, что нужно! Вы также увидите, как использовать сегменты для простой передачи данных *любым* программам и как с их помощью пишутся функции, которые удобно вызывать.

Сегменты

Оказывается, в Go *существует* структура данных, в которую можно добавлять новые значения, — она называется **сегментом**. Как и массив, сегмент состоит из нескольких элементов, относящихся к одному типу. *В отличие* от массивов, существуют функции, позволяющие добавлять новые элементы в конец сегмента.

Чтобы объявить тип переменной для хранения сегмента, поставьте пустую пару квадратных скобок — за ней следует тип элементов, которые будут храниться в сегменте.



Фактически это уже знакомый синтаксис объявления массива, только без указания размера.

```
var myArray [5]int
var mySlice []int
```

Массив — обратите внимание на размер.

Сегмент — размер не задан.

В отличие от переменных для массивов, объявление переменной для сегмента не приводит к автоматическому созданию сегмента. Для этого следует вызвать встроенную функцию `make`. Функции передается тип создаваемого сегмента (он должен соответствовать типу переменной, которой вы собираетесь присвоить сегмент) и длина сегмента при создании.

```
var notes []string
notes = make([]string, 7)
```

Объявление переменной для сегмента.

Создание сегмента из семи строк.

Для присваивания и чтения элементов созданного сегмента используется тот же синтаксис, который использовался бы для массива.

```
notes[0] = "do"
notes[1] = "re"
notes[2] = "mi"
fmt.Println(notes[0])
fmt.Println(notes[1])
```

Присваивает значение первому элементу.

Присваивает значение второму элементу.

Присваивает значение третьему элементу.

Выводит первый элемент.

Выводит второй элемент.

```
do
re
```

Объявлять переменную и создавать сегмент по отдельности необязательно; если вы воспользуетесь коротким объявлением переменной, то тип переменной будет определен автоматически.

```
primes := make([]int, 5)
primes[0] = 2
primes[1] = 3
fmt.Println(primes[0])
```

Создает сегмент из пяти целых чисел и присваивает его переменной.

```
2
```

Сегменты (продолжение)

Встроенная функция `len` для сегментов работает так же, как и для массивов. Передайте `len` сегмент, и функция вернет его длину в виде целого числа.

```
notes := make([]string, 7)
primes := make([]int, 5)
fmt.Println(len(notes))
fmt.Println(len(primes))
```

```
7
5
```

Циклы `for` и `for...range` работают с сегментами точно так же, как и с массивами:

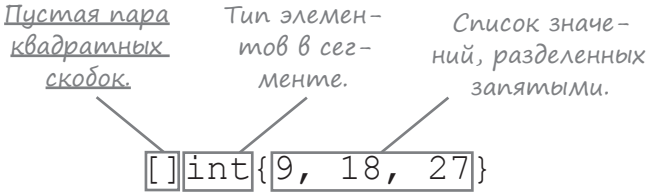
```
letters := []string{"a", "b", "c"}
for i := 0; i < len(letters); i++ {
      fmt.Println(letters[i])
}
for _, letter := range letters {
      fmt.Println(letter)
}
```

```
a
b
c
a
b
c
```

Литералы сегментов

Как и с массивами, если вы заранее знаете, какими значениями должен быть заполнен сегмент в исходном состоянии, то можете инициализировать сегмент этими значениями при помощи **литерала сегмента**. Литерал сегмента очень похож на литерал массива, но если литерал массива содержит длину массива в квадратных скобках, у литерала сегмента квадратные скобки пусты. За пустыми скобками следует тип элементов, которые будут храниться в сегменте, и список исходных значений всех элементов, заключенный в фигурные скобки.

Вызывать функцию `make` необязательно; при использовании литерала сегмента ваш код создаст сегмент *и* заполнит его.



Эти примеры похожи на те, что приводились выше, если не считать того, что вместо последовательного присваивания значений элементам весь сегмент инициализируется с использованием литерала сегмента.

```
notes := []string{"do", "re", "mi", "fa", "so", "la", "ti"}
fmt.Println(notes[3], notes[6], notes[0])
primes := []int{
      2,
      3,
      5,
}
fmt.Println(primes[0], primes[1], primes[2])
```

Значения присваиваются с помощью литерала сегмента.

Многострочный литерал сегмента.

```
fa ti do
2 3 5
```

У бассейна

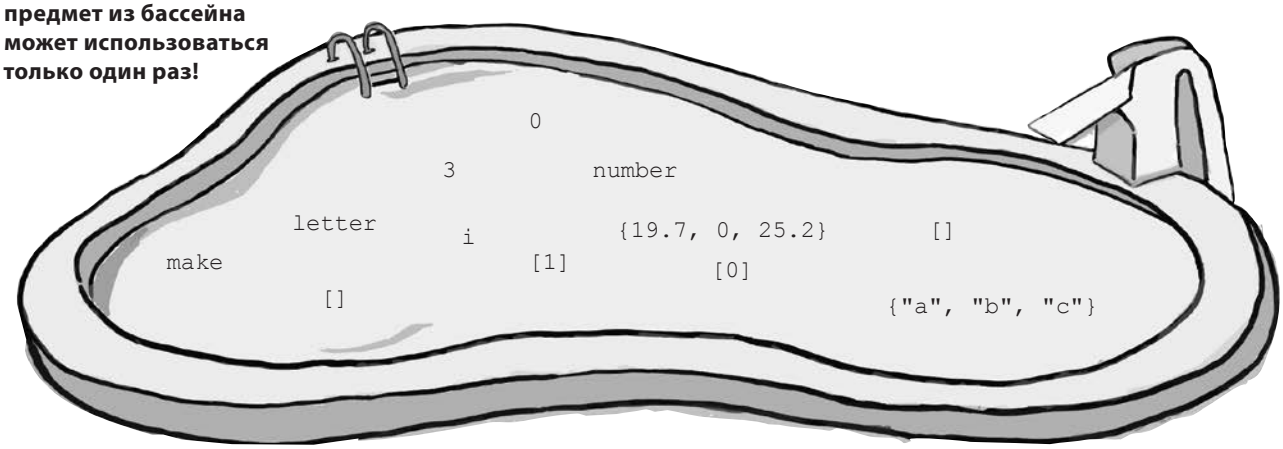
Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты необязательно. Ваша **задача**: создать программу, которая работает и выводит показанный результат.

```
package main

import "fmt"

func main() {
        numbers := ______(__float64, __)
        numbers____ = 19.7
        numbers[2] = 25.2
        for __, _______ := range numbers {
                fmt.Println(i, number)
        }
        var letters = __string_______________
        for i, letter := range letters {
                fmt.Println(i, ________)
        }
}
```

Результат.

```
0 19.7
1 0
2 25.2
0 a
1 b
2 c
```

Примечание: каждый предмет из бассейна может использоваться только один раз!



Ответ на с. 237.

Погодите! Похоже, что сегменты могут делать все, что делают массивы, **и** в них можно добавлять элементы. Тогда почему бы не ограничиться сегментами и не забыть про эту ерунду с массивами?

Потому что сегменты построены на основе массивов. И вы не сможете понять, как работают сегменты, не понимая массивы. Сейчас мы покажем почему.

Оператор сегмента

Каждый массив существует на основе **базового массива**. Данные сегмента на самом деле хранятся в базовом массиве, а сегмент всего лишь предоставляет «окно» для работы с некоторыми (или всеми) элементами массива.

Когда вы используете функцию `make` или литерал сегмента для создания сегмента, базовый массив при этом создается автоматически (и вы не можете обратиться к нему иначе как через сегмент). Но вы также можете создать массив самостоятельно, а затем создать сегмент на основе этого массива при помощи **оператора сегмента**.

Индекс массива, с которого должен начинаться сегмент.

Индекс, перед которым завершается сегмент.

```
mySlice := myArray[1:3]
```

Оператор сегмента напоминает синтаксис обращения к отдельному элементу или сегменту массива, не считая того, что он имеет два индекса: индекс, с которого должен начинаться сегмент, и индекс, перед которым сегмент завершается.

Индекс 0 – массив начинается с этой позиции.

Индекс 3 – сегмент завершается перед этой позицией.

```
underlyingArray := [5]string{"a", "b", "c", "d", "e"}
slice1 := underlyingArray[0:3]
fmt.Println(slice1)
```

```
[a b c]
```

Элементы underlyingArray с 0 по 2.

Обратите внимание: мы неоднократно подчеркиваем, что второй индекс определяет позицию, перед которой завершается сегмент. Другими словами, сегмент должен включать все элементы до второго индекса, *не включая* его. Если использовать оператор сегмента в конструкции `underlyingArray[i:j]`, полученный сегмент будет содержать элементы от `underlyingArray[i]` до `underlyingArray[j-1]`.

Индекс 1 – массив начинается с этой позиции.

Индекс 4 – массив завершается перед этой позицией.

```
underlyingArray := [5]string{"a", "b", "c", "d", "e"}
i, j := 1, 4
slice2 := underlyingArray[i:j]
fmt.Println(slice2)
```

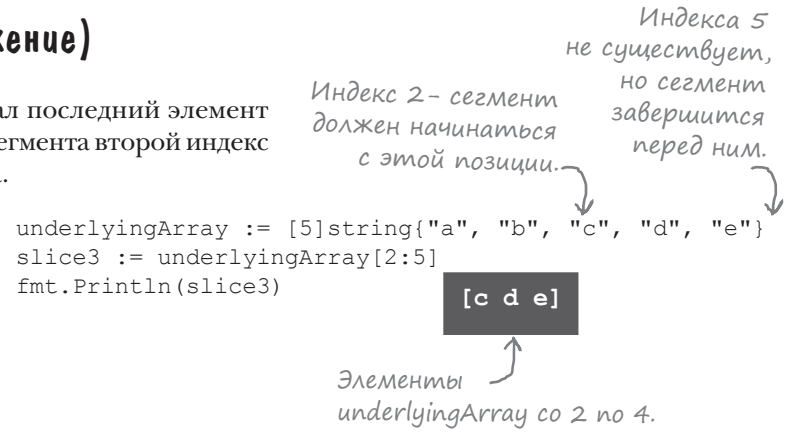
```
[b c d]
```

Элементы underlyingArray с 1 по 3.

(Знаем, это выглядит как-то нелогично. Но аналогичная запись используется в языке программирования Python более 20 лет, и никто не жалуется.)

Оператор сегмента (продолжение)

Если вы хотите, чтобы сегмент включал последний элемент базового массива, укажите в операторе сегмента второй индекс на 1 *больше* последнего индекса массива.

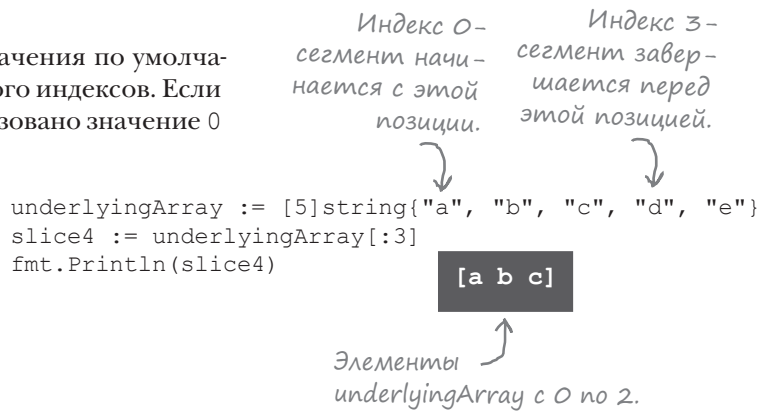


Но будьте внимательны и не превышайте это значение, иначе произойдет ошибка:

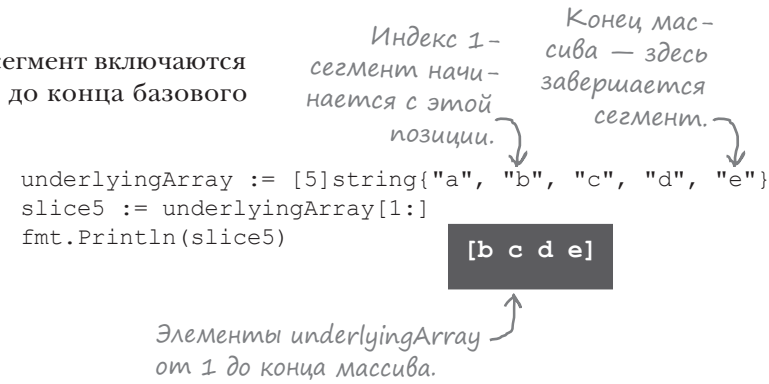
```
underlyingArray := [5]string{"a", "b", "c", "d", "e"}
slice3 := underlyingArray[2:6]
```

```
invalid slice index 6 (out of bounds for 5-element array)
```

У оператора сегмента предусмотрены значения по умолчанию как для начального, так и для конечного индексов. Если начальный индекс не указан, будет использовано значение 0 (первый элемент массива).



А если не указан конечный индекс, то в сегмент включаются все элементы от начального индекса и до конца базового массива.



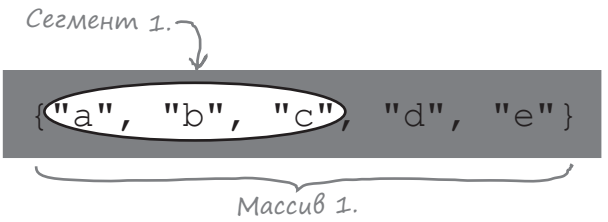
Базовые массивы

Как упоминалось ранее, сам сегмент не содержит данных, это всего лишь «окно» для просмотра элементов базового массива. Сегмент можно представить себе как микроскоп, направленный на определенную часть предметного стекла (базовый массив).

Когда вы берете сегмент базового массива, то «видите» только ту часть элементов массива, которая видна через этот сегмент.

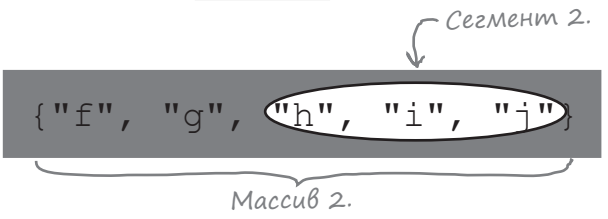
```
array1 := [5]string{"a", "b", "c", "d", "e"}
slice1 := array1[0:3]
fmt.Println(slice1)
```

```
[a b c]
```




```
array2 := [5]string{"f", "g", "h", "i", "j"}
slice2 := array2[2:5]
fmt.Println(slice2)
```

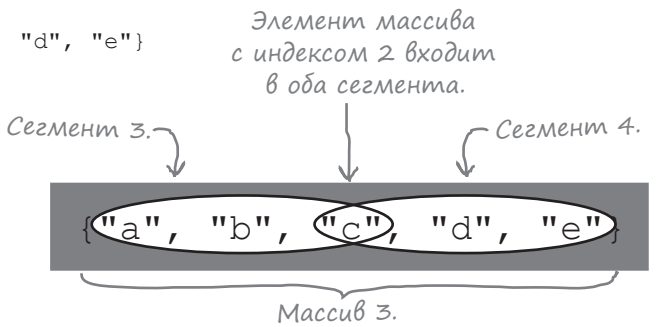
```
[h i j]
```



Несколько сегментов могут существовать на основе одного базового массива. В этом случае каждый сегмент становится «окном» для отдельного подмножества элементов массива. Сегменты даже могут перекрываться!

```
array3 := [5]string{"a", "b", "c", "d", "e"}
slice3 := array3[0:3]
slice4 := array3[2:5]
fmt.Println(slice3, slice4)
```

```
[a b c] [c d e]
```



При изменении базового массива изменяется сегмент

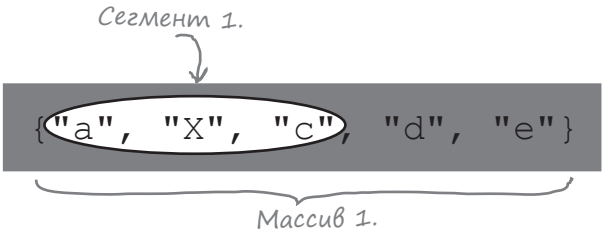
А теперь один нюанс: поскольку сегмент является всего лишь «окном» для работы с содержимым массива, в случае изменения базового массива эти изменения *также* отразятся в сегменте!

```
array1 := [5]string{"a", "b", "c", "d", "e"}
slice1 := array1[0:3]
array1[1] = "X"
fmt.Println(array1)
fmt.Println(slice1)
```

Изменяем элемент базового массива...

```
[a X c d e]
[a X c]
```

...и изменения отражаются в сегменте!



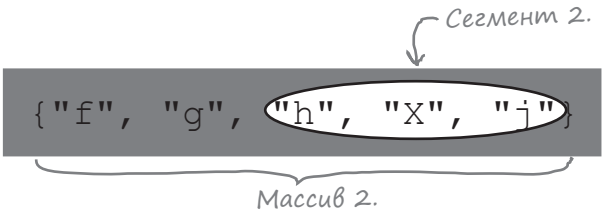
Присваивание нового значения элементу сегмента приводит к изменению соответствующего элемента в базовом массиве.

```
array2 := [5]string{"f", "g", "h", "i", "j"}
slice2 := array2[2:5]
slice2[1] = "X"
fmt.Println(array2)
fmt.Println(slice2)
```

Изменяем элемент сегмента...

```
[f g h X j]
[h X j]
```

...и изменяется базовый массив!



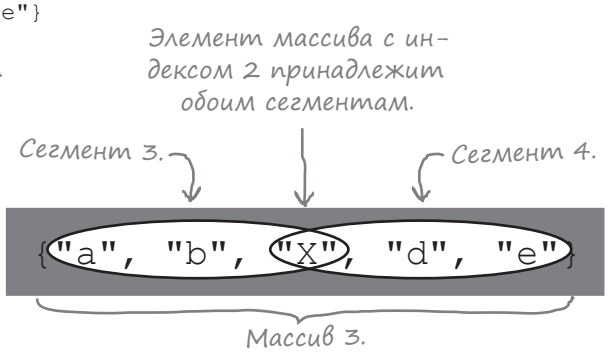
Если на один и тот же базовый массив указывают несколько сегментов, то и изменения элементов массива будут видны во *всех* сегментах.

```
array3 := [5]string{"a", "b", "c", "d", "e"}
slice3 := array3[0:3]
slice4 := array3[2:5]
array3[2] = "X"
fmt.Println(array3)
fmt.Println(slice3, slice4)
```

Изменяем элемент базового массива...

```
[a b X d e]
[a b X] [X d e]
```

...и изменения отражаются в обоих сегментах!



Из-за этих потенциальных проблем обычно рекомендуется создавать сегменты с использованием `make` или литерала сегмента (вместо того, чтобы создать массив и применять к нему оператор сегмента). С `make` и литералами сегментов вам никогда не приходится иметь дела с базовым массивом.

Расширение сегментов функцией «append»

Хорошо, все эти сегменты — отличная штука. Честно. Но у меня по-прежнему остается программа, которая может читать только три строки из текстового файла, потому что использует массив. Вы сказали, что в сегмент можно добавлять новые значения — я хочу побольше узнать об этом!

В Go существует встроенная функция `append`, которая получает сегмент и одно или несколько значений, которые присоединяются в конец сегмента. Функция возвращает новый, расширенный сегмент со всеми элементами исходного сегмента и новыми элементами, добавленными в его конец.

Возвращаемое значение «append» присваивается той же переменной slice.

Возвращаемое значение «append» присваивается той же переменной slice.

```
slice := []string{"a", "b"}
fmt.Println(slice, len(slice))
slice = append(slice, "c")
fmt.Println(slice, len(slice))
slice = append(slice, "d", "e")
fmt.Println(slice, len(slice))
```

Создание сегмента.

Элемент присоединяется в конец сегмента.

Два элемента присоединяются в конец сегмента.

Содержит еще один элемент, а длина сегмента увеличивается на 1.

Содержит еще два элемента, а длина увеличивается на 2.

```
[a b] 2
[a b c] 3
[a b c d e] 5
```

Вам не нужно следить за тем, по какому индексу присваиваются новые значения, или за чем-нибудь еще! Просто вызовите функцию `append` и передайте ей сегмент со значениями, которые добавляются в конец сегмента, и вы получите новый расширенный сегмент. Да, так просто!

Хотя нужно учитывать один момент...

Расширение сегментов функцией «append» (продолжение)

Обратите внимание: возвращаемое значение `append` во всех случаях присваивается *той же* переменной сегмента, которая передается `append`. Таким образом предотвращается возможность непоследовательного поведения сегментов, возвращаемых `append`.

Базовый массив сегмента не может увеличиваться в размерах. Если в массиве не остается места для добавления элементов, все элементы копируются в новый, бóльший массив, а сегмент обновляется, чтобы он базировался на новом массиве. Но поскольку все это происходит где-то за кулисами внутри функции `append`, невозможно простым способом определить, имеет ли возвращенный сегмент *тот же* базовый массив, как и переданный сегмент, или *другой*. Если в программе будут оставаться оба сегмента, это может привести к непредсказуемому поведению.

Например, ниже приведены четыре сегмента; последние три создаются вызовами `append`. Ниже мы *не* следуем соглашению о присваивании возвращаемого значения той же переменной. Когда значение присваивается элементу сегмента `s4`, мы видим, что изменения отражаются в `s3`, потому что `s4` и `s3` используют один и тот же базовый массив. Однако изменения *не* отражаются в `s2` или `s1`, потому что они используют *другой* базовый массив.

Сегменты, возвращаемые «append», присваиваются новым переменным!

```
s1 := []string{"s1", "s1"}
s2 := append(s1, "s2", "s2")
s3 := append(s2, "s3", "s3")
s4 := append(s3, "s4", "s4")
fmt.Println(s1, s2, s3, s4)
s4[0] = "XX"
fmt.Println(s1, s2, s3, s4)
```

Вывод сегментов.

Присваивание элементу четвертого сегмента.

Видим, что изменилось.

Сегменты «s1» и «s2» используют разные базовые массивы, поэтому в данном случае изменения не отражаются!

Сегмент «s3» использует один базовый массив с «s4», поэтому изменения в «s4» отражаются здесь!

Происходит в результате присваивания «s4[0]».

По этой причине при вызове `append` возвращаемое значение обычно присваивается той же переменной сегмента, которая была передана `append`. Если в программе хранится только один сегмент, то вам не придется беспокоиться о том, используют ли два сегмента один базовый массив!

Сегменты, возвращаемые «append», присваиваются той же переменной.

```
s1 := []string{"s1", "s1"}
s1 = append(s1, "s2", "s2")
s1 = append(s1, "s3", "s3")
s1 = append(s1, "s4", "s4")
fmt.Println(s1)
```

```
[s1 s1 s2 s2 s3 s3 s4 s4]
```

И никаких неприятных сюрпризов!

Сегменты и нулевые значения

Как и в случае с массивами, при обращении к элементу сегмента, которому не было присвоено значение, вы получите нулевое значение для этого типа:

Создаем сегменты без присваивания значений их элементам.

```
floatSlice := make([]float64, 10)
boolSlice := make([]bool, 10)
fmt.Println(floatSlice[9], boolSlice[5])
```

```
0 false
```

В отличие от массивов, переменная сегмента сама по себе *тоже* имеет нулевое значение: `nil`. Другими словами, переменная сегмента, которой не был присвоен сегмент, будет содержать значение `nil`.

Напоминаем: «%#v» форматирует значение в том виде, в котором оно будет записываться в коде Go.

Объявление переменных для сегментов без создания сегментов.

```
var intSlice []int
var stringSlice []string
fmt.Printf("intSlice: %#v, stringSlice: %#v\n", intSlice, stringSlice)
```

Значения обеих переменных — nil.

```
intSlice: []int(nil), stringSlice: []string(nil)
```

В других языках перед попыткой использования переменной вам, возможно, придется проверить, действительно ли она содержит сегмент. Но в Go функции намеренно написаны так, чтобы значение сегмента `nil` интерпретировалось как пустой сегмент. Например, функция `len`, если передать ей вместо сегмента `nil`, вернет `0`:

Функции «len» передается сегмент nil.

Вернет 0, как если бы вы передали пустой сегмент!

```
fmt.Println(len(intSlice))
```

```
0
```

Функция `append` также интерпретирует значения `nil` как пустые сегменты. Если передать функции `append` пустой сегмент, она добавит заданный элемент в сегмент и вернет сегмент с одним элементом. Если же передать `append` значение `nil`, вы *тоже* получите сегмент с одним элементом, хотя формально не было сегмента, к которому бы присоединялся элемент. Функция `append` создает сегмент незаметно для вас.

Функции «append» передается сегмент nil.

Вернет сегмент из одного элемента, поскольку элемент присоединяется к пустому сегменту!

```
intSlice = append(intSlice, 27)
fmt.Printf("intSlice: %#v\n", intSlice)
```

```
stringSlice: []string{27}
```

А это означает, что в общем случае вам не нужно беспокоиться о том, что используется в программе: пустой сегмент или `nil`-сегмент. В обоих случаях можно действовать одинаково, а ваш код будет «просто работать»!

Переменная будет содержать nil.

Функция «len» возвращает 0.

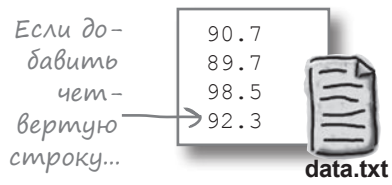
Функция «append» возвращает сегмент из одного элемента, так как ей был передан пустой сегмент.

```
var slice []string
if len(slice) == 0 {
      slice = append(slice, "first item")
}
fmt.Printf("%#v\n", slice)
```

```
[]string{"first item"}
```

Чтение строк из файла с использованием сегментов и «append»

Теперь, когда вы знаете о сегментах и функции append, мы наконец-то можем исправить программу average! Напомним, что сбой в функции average происходил сразу же после добавления четвертой строки в файл *data.txt*, из которого читаются данные:



...в программе возникает ситуация паники и все завершается!

```
Shell Edit View Window Help
$ ./average
panic: runtime error: index out of range
...
```

Как мы выяснили, проблема возникала из-за пакета datafile. Пакет сохранял строки файла в массиве, размер которого не мог вырасти за пределы трех элементов:


```
// Пакет datafile предназначен для чтения данных из файлов.
package datafile

import (
	"bufio"
	"os"
	"strconv"
)

// GetFloats читает значение float64 из каждой строки файла.
func GetFloats(fileName string) ([3]float64, error) {
	var numbers [3]float64
	file, err := os.Open(fileName)
	if err != nil {
		return numbers, err
	}
	i := 0
	scanner := bufio.NewScanner(file)
	for scanner.Scan() {
		numbers[i], err = strconv.ParseFloat(scanner.Text(), 64)
		if err != nil {
			return numbers, err
		}
		i++
	}
	err = file.Close()
	if err != nil {
		return numbers, err
	}
	if scanner.Err() != nil {
		return numbers, scanner.Err()
	}
	return numbers, nil
}
```

Функция возвращает массив значений float64.

Допустимыми являются только индексы с numbers[0] до numbers [2]...

При попытке присваивания numbers[3] возникает паника!

Чтение строк из файла с использованием сегментов и «append» (продолжение)

До этого мы разбирали принципы работы с сегментами. Теперь, когда мы понимаем, как они устроены, проблем с обновлением функции `GetFloats`, чтобы в ней использовался сегмент вместо массива, уже не будет.

Сначала нужно обновить объявление функции, чтобы она возвращала сегмент значений `float64` вместо массива. Ранее массив хранился в переменной с именем `numbers`; мы воспользуемся той же переменной для хранения сегмента. Переменная `numbers` не инициализируется, поэтому в исходном состоянии она содержит `nil`.

Вместо того чтобы присваивать значения, прочитанные из файла, элементу массива с конкретным индексом, мы просто вызовем `append`, чтобы расширить сегмент (или создать сегмент, если переменная содержит `nil`) и добавить в него новое значение. Это позволит обойтись без создания и обновления переменной `i`, в которой отслеживается текущий индекс. Значение `float64`, возвращенное функцией `ParseFloat`, сохраняется в новой временной переменной — просто для того, чтобы данные где-то хранились во время проверки ошибок разбора. Затем сегмент `numbers` и новое значение из файла передается функции `append`, а возвращаемое значение функции снова присваивается той же переменной `numbers`.

В остальном код `GetFloats` остается неизменным — просто вместо массива в функцию подставляется сегмент.

Переходим на возвращение сегмента.

```
// ...
func GetFloats(fileName string) ([]float64, error) {
    var numbers []float64
    file, err := os.Open(fileName)
    if err != nil {
        return numbers, err
    }
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        number, err := strconv.ParseFloat(scanner.Text(), 64)
        if err != nil {
            return numbers, err
        }
        numbers = append(numbers, number)
    }
    err = file.Close()
    if err != nil {
        return numbers, err
    }
    if scanner.Err() != nil {
        return numbers, scanner.Err()
    }
    return numbers, nil
}
```

Переменная по умолчанию содержит nil. (Помните, «append» интерпретирует nil как пустой сегмент.)

В обработке ошибок ничего менять не нужно; с сегментом все работает так же, как с массивом.

Строка преобразуется в float64 и присваивается временной переменной.

Новое число присоединяется к сегменту.

Здесь тоже ничего не меняется.

Тестирование измененной программы

Сегмент, возвращаемый функцией `GetFloats`, также заменяет массив в основной программе `average`. В основной программе *ничего* изменять не придется!

Так как мы использовали короткое объявление переменной `:=` для присваивания возвращаемого значения `GetFloats` переменной, переменная `numbers` автоматически переключается с типа `[3]float64` (массив) на тип `[]float64` (сегмент). А поскольку цикл `for...range` и функция `len` работают с сегментами точно так же, как с массивами, в этом коде тоже не придется ничего изменять!


```
// average вычисляет среднее значение.
package main

import (
        "fmt"
        "github.com/headfirstgo/datafile"
        "log"
)

func main() {
        numbers, err := datafile.GetFloats("data.txt")
        if err != nil {
                log.Fatal(err)
        }
        var sum float64 = 0
        for _, number := range numbers {
                sum += number
        }
        sampleCount := float64(len(numbers))
        fmt.Printf("Average: %0.2f\n", sum/sampleCount)
}
```

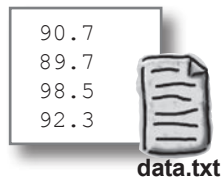
Ничего изменять не нужно!

Автоматически получает тип []float64 вместо [3]float64.

Работает с сегментом так же, как с массивом.

Тоже работает с сегментом так же, как с массивом.

Мы можем опробовать новую версию программы! Убедитесь в том, что файл *data.txt* хранится в подкаталоге *bin* рабочей области Go, затем откомпилируйте и запустите программу уже известными вам командами. Программа читает все числа из файла *data.txt* и выводит их среднее значение. Попробуйте изменить *data.txt* и добавить (или удалить) строки данных; программа все равно работает!



Компилирует обновленный пакет «datafile», потому что «average» зависит от этого пакета.

Переход в подкаталог «bin».

Запускаем программу.

Среднее значение чисел из всех четырех строк файла!

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average
$ cd /Users/jay/go/bin
$ ./average
Average: 92.80
```

Возвращение сегмента nil в случае ошибки

Внесем еще одно небольшое улучшение в функцию `GetFloats`. В настоящее время функция возвращает сегмент `numbers` даже при возникновении ошибки. А это означает, что она может возвращать сегмент с недействительными данными:

```
number, err := strconv.ParseFloat(scanner.Text(), 64)
if err != nil {
       return numbers, err
}
```

Возвращаются недействительные данные, которые не должны использоваться в программе!

Код, который вызывает `GetFloats`, *должен* проверять возвращаемое значение ошибки, убеждаться в том, что оно отлично от `nil`, и при обнаружении ошибки игнорировать содержимое возвращаемого сегмента. Но зачем вообще возвращать сегмент, если содержащиеся в нем данные недействительны? Обновим функцию `GetFloats`, чтобы она возвращала `nil` вместо сегмента в случае ошибки.


```
// ...
func GetFloats(fileName string) ([]float64, error) {
       var numbers []float64
       file, err := os.Open(fileName)
       if err != nil {
              return nil, err
       }
       scanner := bufio.NewScanner(file)
       for scanner.Scan() {
              number, err := strconv.ParseFloat(scanner.Text(), 64)
              if err != nil {
                     return nil, err
              }
              numbers = append(numbers, number)
       }
       err = file.Close()
       if err != nil {
              return nil, err
       }
       if scanner.Err() != nil {
              return nil, scanner.Err()
       }
       return numbers, nil
}
```

Возвращает nil вместо сегмента. (Переменная в этот момент в любом случае равна nil, но так этот факт становится еще более очевидным.)

Возвращает nil вместо сегмента.

Возвращает nil вместо сегмента.

Возвращает nil вместо сегмента.

Перекомпилируйте программу (вместе с обновленным пакетом `datafile`) и запустите ее. Программа должна работать так же, как и прежде, но код обработки ошибки стал немного чище.

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average
$ cd /Users/jay/go/bin
$ ./average
Average: 92.80
```

Упражнение

Ниже приведена программа, которая берет сегмент массива и присоединяет элементы к сегменту. Напишите, какой результат должна выводить программа.

```
package main

import "fmt"

func main() {
       array := [5]string{"a", "b", "c", "d", "e"}
       slice := array[1:3]
       slice = append(slice, "x")
       slice = append(slice, "y", "z")
       for _, letter := range slice {
              fmt.Println(letter)
       }
}
```

Результат:

.....

.....

.....

.....

.....

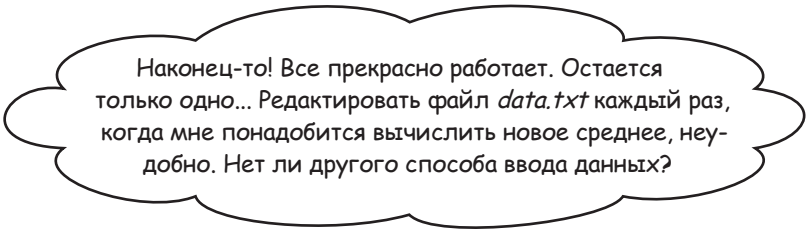
.....

.....

Пропусков здесь больше, чем нужно. Насколько больше? Определите это сами!

Ответ на с. 237.

Аргументы командной строки



Есть и другой вариант — пользователи могут передавать значения программе в аргументах командной строки.

Как вы уже знаете, поведением многих функций Go можно управлять, передавая им аргументы. Точно так же можно передавать аргументы программам, запускаемым в терминале или командной строке. Такой механизм передачи данных называется *интерфейсом командной строки* программы.

Примеры использования аргументов командной строки уже встречались вам в этой книге. Так, команде `cd` (изменение каталога) в аргументе передается имя каталога, в который вы хотите перейти. При выполнении команды `go` тоже часто передаются аргументы: используемая подкоманда (`run`, `install` и т. д.) и имя файла или пакета, с которым должна работать подкоманда.

Получение аргументов командной строки из сегмента os.Args

Создадим новую версию программы `average` с именем `average2`, которая получает усредняемые значения в аргументах командной строки.

В пакете `os` объявляется пакетная переменная `os.Args`, которая содержит сегмент строк, представляющих аргументы командной строки, переданные текущей программе при запуске. Для начала просто выведем элементы `os.Args` и посмотрим, что же содержит этот сегмент.

Создайте новый каталог *average2* рядом с каталогом *average* в своей рабочей области. Сохраните в нем файл *main.go*.

Затем сохраните приведенный ниже код в *main.go*. Этот код просто импортирует пакеты `fmt` и `os` и передает сегмент `os.Args` функции `fmt.Println`.

```
// average2 вычисляет среднее значение.
package main

import (
       "fmt"
       "os"
)

func main() {
       fmt.Println(os.Args)
}
```

Вывод сегмента os.Args.

Посмотрим, что же получилось. В терминале или командной строке введите следующую команду, чтобы откомпилировать и установить программу:

```
go install github.com/headfirstgo/average2
```

Команда устанавливает исполняемый файл с именем *average2* (или *average2.exe* в Windows) в подкаталоге *bin* вашей рабочей области. Перейдите в каталог *bin* командой `cd` и введите **`average2`**, но пока не нажимайте клавишу Enter. После имени программы введите пробел, а затем один или несколько аргументов, разделенных пробелами. *После этого* нажмите Enter. Программа запускается и выводит значение `os.Args`.

Запустите `average2` с другими аргументами, и вы увидите другой результат.

Компилирует и устанавливает исполняемый файл.

Переход в подкаталог «bin».

Запуск исполняемого файла с несколькими аргументами.

Выводит значение os.Args.

average2 запускается с другими аргументами, чтобы увидеть другой результат.

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average2
$ cd /Users/jay/go/bin
$ ./average2 71.8 56.2 89.5
[./average2 71.8 56.2 89.5]
$ ./average2 do re mi fa so
[./average2 do re mi fa so]
```

Оператор сегмента может использоваться с другими сегментами

Этот механизм неплохо работает, но есть одна проблема: в первом элементе `os.Args` содержится имя исполняемого файла.

```
$ ./average2 71.8 56.2 89.5
[./average2 71.8 56.2 89.5]
```

Первый элемент содержит имя программы.

Впрочем, проблема довольно легко решается. Помните, как мы использовали оператор сегмента для получения сегмента, включавшего все элементы массива, кроме первого?

Индекс 1 — сегмент начнется с этой позиции.

Конец массива — здесь завершается сегмент.

```
underlyingArray := [5]string{"a", "b", "c", "d", "e"}
slice5 := underlyingArray[1:]
fmt.Println(slice5)
```

```
[b c d e]
```

От элемента 1 до конца underlyingArray.

Оператор сегмента может применяться не только с массивами, но и с сегментами. Если применить оператор сегмента `[1:]` к `os.Args`, вы получите новый сегмент, из которого исключен первый элемент (с индексом `0`) — от второго элемента (индекс `1`) до конца сегмента.

```
// average2 вычисляет среднее значение.
package main

import (
       "fmt"
       "os"
)

func main() {
       fmt.Println(os.Args[1:])
}
```

Получает новый сегмент от второго элемента (индекс 1) до конца os.Args.

Если заново откомпилировать и запустить `average2`, вы увидите, что на этот раз в вывод включаются только фактически введенные аргументы командной строки.

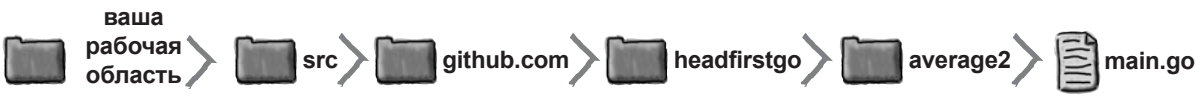
```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average2
$ ./average2 71.8 56.2 89.5
[71.8 56.2 89.5]
$ ./average2 do re mi fa so
[do re mi fa so]
```

Без имени исполняемого файла.

Без имени исполняемого файла.

Использование аргументов командной строки в программе

Теперь, когда мы можем получить аргументы командной строки в виде сегмента строк, обновим программу `average2` для преобразования аргументов в числа и вычисления их среднего значения. В основном мы сможем повторно использовать концепции, которые были описаны для исходной программы `average` и пакета `datafile`. Мы применим оператор сегмента к `os.Args` для удаления имени программы, а затем присвоим полученный сегмент переменной `arguments`. В программе создается переменная `sum` для хранения суммы всех переданных чисел. После этого для обработки всех элементов сегмента `arguments` будет использован цикл `for...range` (для игнорирования индекса элемента используется пустой идентификатор `_`). Функция `strconv.ParseFloat` преобразует строку аргумента в `float64`. Если будет обнаружена ошибка, программа выводит сообщение об ошибке и завершает работу, но в остальных случаях программа прибавляет текущее число к `sum`. При переборе аргументов функция `len(arguments)` определяет, сколько значений данных мы будем усреднять. Среднее значение вычисляется делением суммы на это количество.




```
// average2 вычисляет среднее значение.
package main

import (
        "fmt"
        "log"
        "os"
        "strconv"
)

func main() {
        arguments := os.Args[1:]
        var sum float64 = 0
        for _, argument := range arguments {
                number, err := strconv.ParseFloat(argument, 64)
                if err != nil {
                        log.Fatal(err)
                }
                sum += number
        }
        sampleCount := float64(len(arguments))
        fmt.Printf("Average: %0.2f\n", sum/sampleCount)
}
```

Импортируем пакеты «log» и «strconv».

Получение сегмента строк со всеми элементами os.Args, кроме первого.

Объявление переменной для хранения суммы чисел.

Обрабатываем каждый аргумент командной строки.

Строка преобразуется в float64.

Если при преобразовании строки произошла ошибка, программа выводит сообщение и завершается.

Число прибавляется к сумме.

Длина сегмента arguments используется как количество значений данных.

Вычисление среднего значения и его вывод.

Сохраните эти изменения, перекомпилируйте и запустите программу. Программа получает числа, переданные в аргументах, и вычисляет их среднее значение. Количество аргументов может быть сколь угодно большим или малым — программа все равно работает!

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average2
$ cd /Users/jay/go/bin
$ ./average2 71.8 56.2 89.5
Average: 72.50
$ ./average2 90.7 89.7 98.5 92.3
Average: 92.80
```

Запуск программы с несколькими аргументами.

Передайте любое количество аргументов на свое усмотрение.

Функции с переменным количеством аргументов

После знакомства с сегментами мы можем рассмотреть одну возможность Go, которая до сих пор явно не упоминалась. Вы заметили, что при некоторых вызовах функций может передаваться разное количество аргументов? Например, взгляните на функцию `fmt.Println` или `append`:

```
fmt.Println(1)
fmt.Println(1, 2, 3, 4, 5)
letters := []string{"a"}
letters = append(letters, "b")
letters = append(letters, "c", "d", "e", "f", "g")
```

Функция «Println» может получать один аргумент...
...или целых пять!
Функция «append» может получать два аргумента...
...или шесть!

Но это возможно далеко не для всех функций! У всех функций, которые определялись нами до сих пор, количество параметров в определении функции должно *точно* соответствовать количеству параметров в определении функции и количеству аргументов в вызове функции. Любые расхождения приведут к ошибке компиляции.

```
func twoInts(first int, second int) {
      fmt.Println(first, second)
}

func main() {
      twoInts(1)
      twoInts(1, 2, 3)
}
```

Если функция ожидает получить два параметра...
...то нельзя передать ей только один...
...и нельзя передать три аргумента.

```
tmp/sandbox815038307/main.go:10:9: not enough arguments in call to twoInts
      have (number)
      want (int, int)
tmp/sandbox815038307/main.go:11:9: too many arguments in call to twoInts
      have (number, number, number)
      want (int, int)
```

Как же это делают функции `Println` и `append`? Они объявляются как **функции с переменным количеством параметров**. Таким функциям при вызове может передаваться *разное* количество аргументов. Чтобы функция могла получать переменное количество аргументов, поставьте многоточие (`...`) перед типом последнего (или единственного) параметра функции в ее объявлении.

Многоточие. *Тип.*

```
func myFunc(param1 int, param2 ...string) {
    // код функции
}
```

Функции с переменным количеством аргументов (продолжение)

В последнем параметре функции с переменным количеством аргументов передается сегмент, который может обрабатываться функцией как любой другой сегмент. Ниже приведена версия функции `twoInts` с переменным количеством аргументов, которая нормально работает при любом количестве аргументов:

Переменная «numbers» содержит сегмент с аргументами.

```
func severalInts(numbers ...int) {
        fmt.Println(numbers)
}

func main() {
        severalInts(1)
        severalInts(1, 2, 3)
}
```

```
[1]
[1 2 3]
```

Ниже приведена аналогичная функция, работающая со строками. Обратите внимание: если переменная часть аргументов не передается, ошибки не будет; функция просто получает пустой сегмент.

Переменная «strings» содержит сегмент с аргументами.

```
func severalStrings(strings ...string) {
        fmt.Println(strings)
}

func main() {
        severalStrings("a", "b")
        severalStrings("a", "b", "c", "d", "e")
        severalStrings()
}
```

При отсутствии аргументов получает пустой сегмент.

```
[a b]
[a b c d e]
[]
```

Функция также может получать один или несколько фиксированных аргументов. Если при вызове переменную часть аргументов можно опускать (что приведет к созданию пустого сегмента), фиксированные аргументы всегда обязательны; если опустить их, произойдет ошибка компиляции. Переменным может быть только *последний* параметр в определении функции; он не может предшествовать обязательным параметрам.

Аргумент int должен стоять на первом месте.

Логический аргумент должен стоять на втором месте.

Все остальные аргументы должны быть строками и храниться в сегменте.

```
func mix(num int, flag bool, strings ...string) {
        fmt.Println(num, flag, strings)
}

func main() {
        mix(1, true, "a", "b")
        mix(2, false, "a", "b", "c", "d")
}
```

```
1 true [a b]
2 false [a b c d]
```

Использование функций с переменным количеством аргументов

Функция `maximum` получает любое количество аргументов `float64` и возвращает наибольшее значение среди всех переданных. Аргументы функции `maximum` хранятся в сегменте в параметре `numbers`. Сначала текущему максимуму присваивается — `Inf` — специальное значение, представляющее отрицательную бесконечность, для получения которого используется вызов `math.Inf`. (Также можно начать с текущего максимума `0`, но в нашем варианте `maximum` будет работать с отрицательными числами.) Затем цикл `for...range` обрабатывает каждый аргумент в сегменте `numbers`, сравнивает его с текущим максимумом и присваивает как новый максимум, если он больше текущего. Максимум, остающийся после обработки всех аргументов, возвращается функцией.

```
package main

import (
        "fmt"
        "math"
)

func maximum(numbers ...float64) float64 {
        max := math.Inf(-1)
        for _, number := range numbers {
                if number > max {
                        max = number
                }
        }
        return max
}

func main() {
        fmt.Println(maximum(71.8, 56.2, 89.5))
        fmt.Println(maximum(90.7, 89.7, 98.5, 92.3))
}
```

Получает любое количество аргументов float64.

Начинает с очень низкого значения.

Обработка всех аргументов в переменной части.

Находит наибольшее значение среди аргументов.

```
89.5
98.5
```

Ниже приведена функция `inRange`, которая получает минимальное значение, максимальное значение и любое количество дополнительных аргументов `float64`. Функция игнорирует аргументы, меньшие заданного минимума или большие максимума, и возвращает сегмент с аргументами, принадлежащими указанному диапазону.

```
package main

import "fmt"

func inRange(min float64, max float64, numbers ...float64) []float64 {
        var result []float64
        for _, number := range numbers {
                if number >= min && number <= max {
                        result = append(result, number)
                }
        }
        return result
}

func main() {
        fmt.Println(inRange(1, 100, -12.5, 3.2, 0, 50, 103.5))
        fmt.Println(inRange(-10, 10, 4.1, 12, -12, -5.2))
}
```

Минимальное значение в диапазоне.

Максимальное значение в диапазоне.

Любое количество дополнительных аргументов float64.

Сегмент для хранения аргументов, входящих в диапазон.

Обработка всех аргументов в переменной части.

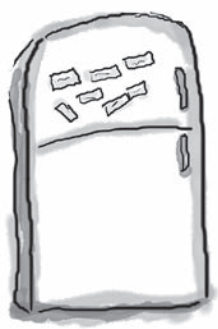
Если этот аргумент не ниже минимума и не выше максимума...

...он добавляется в возвращаемый сегмент.

Поиск аргументов >= 1 и <= 100.

Поиск аргументов >= -10 и <= 10.

```
[3.2 50]
[4.1 -5.2]
```



Развлечения с магнитами

На холодильнике была выложена программа Go, которая определяет и использует функцию с переменным количеством параметров. Расставьте фрагменты кода так, чтобы у вас получилась работоспособная программа, которая выводит указанный результат.

```
for _, number
:=
))
int
...
(
)
}
sum(
,
{
}
{
}
sum(
4
import "fmt"
func main() {
7
1
,
var sum int = 0
func sum
fmt.Println(
))
package main
return sum
int
9
2
sum += number
numbers
fmt.Println(
,
numbers
range
```

Результат.

```
16
7
```

Ответ на с. 238.

Использование функции с переменным количеством параметров для вычисления среднего значения

Создайте функцию `average` с переменным количеством аргументов, которая получает любое количество аргументов `float64` и возвращает их среднее значение. Большая часть логики очень похожа на программу `average2`. Мы создаем переменную `sum` для хранения суммы аргументов, после чего в цикле перебираем аргументы и прибавляем каждый к накапливаемой сумме. Наконец, сумма делится на количество аргументов (преобразованное к типу `float64`) для получения среднего значения. В итоге мы получаем функцию вычисления среднего, которой можно передать столько чисел, сколько вы посчитаете нужным.

```
package main

import "fmt"

func average(numbers ...float64) float64 {
    var sum float64 = 0
    for _, number := range numbers {
        sum += number
    }
    return sum / float64(len(numbers))
}

func main() {
    fmt.Println(average(100, 50))
    fmt.Println(average(90.7, 89.7, 98.5, 92.3))
}
```

Получает любое количество аргументов float64.

Обработать каждый аргумент из переменной части.

Создание переменной для хранения суммы аргументов.

Значение аргумента прибавляется к сумме.

Среднее значение вычисляется делением суммы на количество аргументов.

```
75
92.8
```

Передача сегментов функциям с переменным количеством аргументов

Новая функция `average` с переменным количеством аргументов работает просто отлично; попробуем обновить программу `average2`, чтобы в ней использовалась эта функция. Функцию `average` можно вставить в код `average2` без изменений.

В функции `main` каждый аргумент командной строки по-прежнему должен быть преобразован из строки в значение `float64`. Мы создадим сегмент для хранения полученных значений и сохраним его в переменной с именем `numbers`. После преобразования каждого аргумента командной строки, вместо того чтобы немедленно использоваться для вычисления среднего, он просто присоединяется к сегменту `numbers`.

А после этого мы *попробуем* передать сегмент `numbers` функции `average`. Но во время компиляции программы выдается сообщение об ошибке...


```
// average2 вычисляет среднее значение.
package main

import (
        "fmt"
        "log"
        "os"
        "strconv"
)

func average(numbers ...float64) float64 {
        var sum float64 = 0
        for _, number := range numbers {
                sum += number
        }
        return sum / float64(len(numbers))
}

func main() {
        arguments := os.Args[1:]
        var numbers []float64
        for _, argument := range arguments {
                number, err := strconv.ParseFloat(argument, 64)
                if err != nil {
                        log.Fatal(err)
                }
                numbers = append(numbers, number)
        }
        fmt.Printf("Average: %0.2f\n", average(numbers))
}
```

Функция «average» вставляется в исходном виде.

В этом сегменте будут храниться числа, для которых вычисляется среднее.

Преобразованное число присоединяется к сегменту.

При попытке передать числа функции с переменным количеством аргументов...

Ошибка

```
cannot use numbers (type []float64)
as type float64 in argument to average
```

Функция `average` рассчитывает получить один или несколько аргументов `float64`, а не *сегмент* значений `float64`...

Передача сегментов функциям с переменным количеством аргументов (продолжение)

И что теперь? Придется выбирать между объявлением функции с переменным количеством аргументов и возможностью передачи им сегментов?

К счастью, в Go предусмотрен специальный синтаксис для подобных ситуаций. При вызове функции с переменным количеством аргументов просто добавьте многоточие (`...`) после сегмента, который должен использоваться вместо переменного количества аргументов.

```
func severalInts(numbers ...int) {
       fmt.Println(numbers)
}

func mix(num int, flag bool, strings ...string) {
       fmt.Println(num, flag, strings)
}

func main() {
       intSlice := []int{1, 2, 3}
       severalInts(intSlice...)
       stringSlice := []string{"a", "b", "c", "d"}
       mix(1, true, stringSlice...)
}
```

Сегмент int используется вместо переменного количества аргументов.

Сегмент строк используется вместо переменного количества аргументов.

```
[1 2 3]
1 true [a b c d]
```

Итак, нужно совсем немного — добавить многоточие после сегмента `numbers` при вызове `average`.

```
func main() {
       arguments := os.Args[1:]
       var numbers []float64
       for _, argument := range arguments {
              number, err := strconv.ParseFloat(argument, 64)
              if err != nil {
                     log.Fatal(err)
              }
              numbers = append(numbers, number)
       }
       fmt.Printf("Average: %0.2f\n", average(numbers...))
}
```

Сегмент передается функции с переменным количеством аргументов.

Внесите это изменение, и вы снова сможете откомпилировать и запустить программу. Аргументы командной строки преобразуются в сегмент значений `float64`, а затем этот сегмент передается функции `average` с переменным количеством аргументов.

Работает!

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average2
$ cd /Users/jay/go/bin
$ ./average2 71.8 56.2 89.5
Average: 72.50
$ ./average2 90.7 89.7 98.5 92.3
Average: 92.80
```

Сегменты спасли положение!

Великолепно! Я просто ввожу объемы заказов за предыдущие недели и сразу же вижу среднее значение. Все так удобно — я могу планировать заказы по всем ингредиентам! Пожалуй, мне все же стоит установить Go!

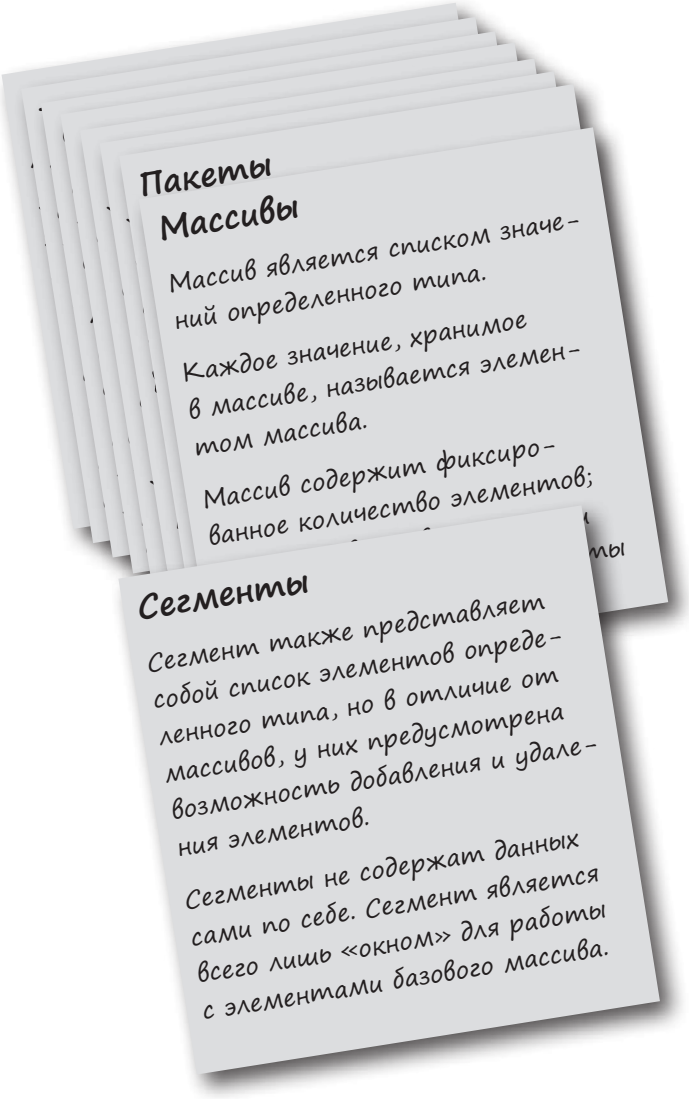
```
Shell Edit View Window Help
$ go install github.com/headfirstgo/average2
$ cd /Users/jay/go/bin
$ ./average2 71.8 56.2 89.5
Average: 72.50
$ ./average2 90.7 89.7 98.5 92.3
Average: 92.80
```

Работа со списками значений играет важную роль в любом языке программирования. С массивами и сегментами вы можете хранить свои данные в коллекциях любого нужного размера. А с такими средствами, как циклы `for...range`, Go еще и упрощает работу с данными в таких коллекциях!

Ваш инструментарий Go

Глава 6 осталась позади! В ней ваш инструментарий пополнился сегментами.

ГЛАВА 6



КЛЮЧЕВЫЕ МОМЕНТЫ

- Тип переменной-сегмента объявляется так же, как и тип переменной-массива, но без указания длины:
  `var mySlice []int`
- В основном код работы с сегментами идентичен коду работы с массивами. В частности, это относится к обращению к элементам, использованию нулевых значений, передаче сегментов функции `len` и циклам `for... range`.
- **Литерал сегмента** выглядит точно так же, как литерал массива, но без указания длины:
  `[]int{1, 7, 10}`
- Для получения сегмента, содержащего элементы с `i` по `j - 1` массива или сегмента, можно воспользоваться **оператором сегмента**: `s[i:j]`
- Пакетная переменная `os.Args` содержит сегмент строк с аргументами командной строки, с которыми была запущена текущая программа.
- Чтобы объявить функцию с переменным количеством аргументов, поставьте многоточие (`...`) перед типом последнего параметра в объявлении функции. Этому параметру будут присвоены все аргументы переменного набора в виде сегмента.
- При вызове функции с переменным количеством аргументов можно использовать сегмент вместо переменного набора аргументов; для этого поставьте многоточие после сегмента:
  `inRange(1, 10, mySlice...)`

У бассейна. Решение

```
package main

import "fmt"

func main() {
        numbers := make([]float64, 3)
        numbers[0] = 19.7
        numbers[2] = 25.2
        for i, number := range numbers {
                fmt.Println(i, number)
        }
        var letters = []string{"a", "b", "c"}
        for i, letter := range letters {
                fmt.Println(i, letter)
        }
}
```

Упражнение
Решение

Ниже приведена программа, которая берет сегмент массива и присоединяет элементы к сегменту. Напишите, какой результат должна выводить программа.

```
package main

import "fmt"

func main() {
        array := [5]string{"a", "b", "c", "d", "e"}
        slice := array[1:3]
        slice = append(slice, "x")
        slice = append(slice, "y", "z")
        for _, letter := range slice {
                fmt.Println(letter)
        }
}
```

Результат:

b

c

x

y

z

Развлечения с магнитами. Решение

```
package main

import "fmt"

func sum ( numbers ... int ) int {
    var sum int = 0
    for _, number := range numbers {
        sum += number
    }
    return sum
}

func main() {
    fmt.Println( sum( 7 , 9 ))
    fmt.Println( sum( 1 , 2 , 4 ))
}
```

Результат.

```
16
7
```

Карты

Сваливать все в одну кучу удобно до тех пор, пока не потребуется что-нибудь найти. Вы уже видели, как создавать списки значений в *массивах* и *сегментах*. Вы знаете, как применить одну операцию *к каждому значению* в массиве или сегменте. Но что, если требуется поработать с *конкретным* значением? Чтобы найти его, придется начать с начала массива или сегмента и *просмотреть. Каждое. Существующее. Значение.* А если бы существовала разновидность коллекций, в которой каждое значение снабжается специальной меткой? Тогда нужное значение можно было бы быстро найти по метке! Эта глава посвящена **картам**, которые предназначены именно для этого.

Подсчет голосов

Этот пример впервые встречается в книге Head First Ruby при описании хешей. Хеши Ruby имеют много общего с картами Go, поэтому этот пример отлично подойдет!

В школьном совете округа открылась вакансия, и опросы показывают, что выборы очень близки. Кандидаты с волнением следят за ходом голосования.

Уверена, что избиратели выберут кандидата, который обеспечит будущее наших детей!

Имя: Эмбер Грэм
Профессия: менеджер

Пора вернуть финансовую ответственность и подотчетность в нашу школьную систему!

Имя: Брайан Мартин
Профессия: бухгалтер

На вакансию претендуют два кандидата, Эмбер Грэм и Брайан Мартин. У избирателя также есть возможность «вписать» своего кандидата (то есть указать имя, отсутствующее в бюллетене для голосования). Таких кандидатов будет меньше, чем основных, но без них наверняка не обойдется.

В этом году используются электронные машины для голосования, которые записывают данные голосования в текстовые файлы, по одному результату на строку. (Бюджет ограничен, поэтому городской совет выбрал дешевую модель.)

Вот как выглядит файл с результатами голосования для участка А:

В каждой строке представлен один голос.

```
Amber Graham
Brian Martin
Amber Graham
Brian Martin
Amber Graham
```

votes.txt

Наша задача — обработать каждую строку файла и подсчитать общее количество вхождений каждого имени. Кандидат с наибольшим количеством голосов станет победителем!

Чтение имен из файла

Прежде всего необходимо прочитать содержимое файла *votes.txt*. Пакет `datafile` из предыдущих глав уже содержит функцию `GetFloats`, которая читает каждую строку файла в сегмент, но `GetFloats` умеет читать только значения `float64`. Нам понадобится отдельная функция, которая возвращает содержимое файла в виде сегмента строковых значений.

Начнем с создания файла *strings.go* наряду с файлом *floats.go* в каталоге пакета *datafile*. В этот файл следует добавить функцию `GetStrings`. Код `GetStrings` имеет много общего с кодом `GetFloats` (совпадающий код внизу выделен серым цветом). Но вместо того чтобы преобразовывать каждую строку в значение `float64`, `GetStrings` добавляет строку непосредственно в возвращаемый сегмент в виде строкового значения.


```
// Пакет datafile предназначен для чтения данных из файлов.
package datafile

import (
        "bufio"
        "os"
)

// GetStrings читает строку из каждой строки данных файла.
func GetStrings(fileName string) ([]string, error) {
        var lines []string
        file, err := os.Open(fileName)
        if err != nil {
                return nil, err
        }
        scanner := bufio.NewScanner(file)
        for scanner.Scan() {
                line := scanner.Text()
                lines = append(lines, line)
        }
        err = file.Close()
        if err != nil {
                return nil, err
        }
        if scanner.Err() != nil {
                return nil, scanner.Err()
        }
        return lines, nil
}
```

Принадлежит тому же пакету, что и GetFloats.

Пакет «strconv» не импортируется; в этом файле он не нужен.

В переменной хранится сегмент строк.

Возвращает сегмент строк вместо сегмента значений float64.

Вместо преобразования строки файла к типу float64 мы просто добавляем ее к сегменту.

Возвращает сегмент строк.

Чтение имен из файла (продолжение)

Перейдем к созданию программы, в которой происходит непосредственный подсчет голосов. Программа будет называться `count`. В рабочей области Go перейдите в каталог *src/github.com/headfirstgo* и создайте новый каталог с именем *count*. Создайте в нем файл с именем *main.go*.

Прежде чем писать полную программу, убедимся в том, что функция `GetStrings` работает. В начале функции `main` мы вызываем функцию `datafile.GetStrings` и передаем ей строку `"votes.txt"` — имя файла, из которого читаются данные. Возвращаемый сегмент строк сохраняется в новой переменной с именем `lines`, а ошибки — в переменной `err`. Как обычно, если значение `err` отлично от `nil`, программа выводит сообщение об ошибке и завершается. В противном случае мы просто вызываем функцию `fmt.Println` для вывода содержимого сегмента `lines`.


```
// count подсчитывает количество вхождений
// каждой строки в файле.
package main

import (
        "fmt"
        "github.com/headfirstgo/datafile"
        "log"
)

func main() {
        lines, err := datafile.GetStrings("votes.txt")
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(lines)
}
```

Импортируется пакет «datafile», который сейчас включает функцию GetStrings.

Читает файл «votes.txt» и возвращает сегмент, содержащий все строки из файла.

Если произошла ошибка, программа выводит сообщение и завершается.

Вывод сегмента строк.

Как и предыдущие программы, эта программа (вместе с пакетами, от которых она зависит, — `datafile` в данном случае) компилируется командой `go install` с передачей пути импорта пакета. Для приведенной выше структуры каталогов путь импорта имеет вид `github.com/headfirstgo/count`.

Компилирует содержимое каталога «count» и устанавливает полученный исполняемый файл.

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/count
```

Исполняемый файл с именем *count* (или *count.exe* в Windows) сохраняется в подкаталоге *bin* вашей рабочей области Go.

Чтение имен из файла (продолжение)

Как и в случае с файлом *data.txt* из предыдущих глав, файл *votes.txt* должен храниться в текущем каталоге при запуске программы. В подкаталоге *bin* рабочей области Go сохраните файл с содержимым, приведенным справа. В терминале перейдите в этот подкаталог командой **cd**.

Запустите исполняемый файл командой **./count** (или **count.exe** в Windows). Программа читает все строки файла *votes.txt* в сегмент строк, после чего выводит содержимое сегмента.

```
Amber Graham
Brian Martin
Amber Graham
Brian Martin
Amber Graham
```

votes.txt

Переход в каталог «bin» рабочей области.

Запуск исполняемого файла.

```
Shell Edit View Window Help
$ cd /Users/jay/go/bin
$ ./count
[Amber Graham Brian Martin Amber Graham Brian Martin
Amber Graham]
$
```

Подсчет имен: сложный способ с сегментами

Чтобы загрузить сегмент с именами из файла, ничего нового изучать не пришлось. Но тут мы сталкиваемся с проблемой: как подсчитать количество вхождений каждого имени? Мы продемонстрируем два способа: с сегментами и с новой структурой данных, которая называется *картой*.

Для первого решения мы создадим два сегмента с одинаковым количеством элементов, следующих в определенном порядке. В первом сегменте хранятся имена, найденные в файле; каждое имя встречается ровно один раз. Назовем этот сегмент `names`. Во втором сегменте `counts` будет храниться количество вхождений каждого имени в файле. Элемент `counts[0]` содержит счетчик для `names[0]`, `counts[1]` — счетчик для `names[1]` и т. д.

Индекс	names	Индекс	counts	
0	"Amber Graham"	0	3	← «Amber Graham» — три голоса.
1	"Brian Martin"	1	2	← «Brian Martin» — два голоса.
2	"Carlos Diaz"	2	1	← «Carlos Diaz» — один голос.
3	...	3	...	

Подсчет имен: сложный способ с сегментами (продолжение)

Обновим программу `count`, чтобы она подсчитывала количество вхождений каждого имени в файле. Опробуем план с использованием сегмента `names` с уникальными именами кандидатов и парным сегментом `counts` с количеством вхождений каждого имени.


```
//...
func main() {
        lines, err := datafile.GetStrings("votes.txt")
        if err != nil {
                log.Fatal(err)
        }
        var names []string
        var counts []int
        for _, line := range lines {
                matched := false
                for i, name := range names {
                        if name == line {
                                counts[i]++
                                matched = true
                        }
                }
                if matched == false {
                        names = append(names, line)
                        counts = append(counts, 1)
                }
        }
        for i, name := range names {
                fmt.Printf("%s: %d\n", name, counts[i])
        }
}
```

Переменная для хранения сегмента с именами кандидатов.

Сегмент с количеством вхождений каждого имени.

Обработка всех строк из файла.

Перебор всех значений из сегмента names.

Если эта строка совпадает с текущим именем...

Увеличивает соответствующий счетчик.

Устанавливаем признак обнаруженного совпадения.

Если совпадение не найдено...

...добавить его как новое имя в сегмент...

И добавить новый счетчик (текущая строка станет первым вхождением).

Все готово, остается вывести результаты.

Вывести каждый элемент из сегмента names...

...и соответствующий элемент из сегмента counts.

Как обычно, программа перекомпилируется командой `go install`. Если запустить полученный исполняемый файл, программа прочитает файл *votes.txt* и выведет все найденные в нем имена вместе со счетчиками вхождений!

Компилирует программу.

Переход в подкаталог «bin».

Запуск обновленной программы.

Выводятся счетчики вхождений для каждого имени.

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/count
$ cd /Users/jay/go/bin
$ ./count
Amber Graham: 3
Brian Martin: 2
```

Давайте разберемся, как работает программа...

Подсчет имен: сложный способ с сегментами (продолжение)

В программе `count` для подсчета имен используется цикл, вложенный *внутрь* другого цикла. Внешний цикл последовательно присваивает строки файла переменной `line`.

Обработка каждой строки файла.

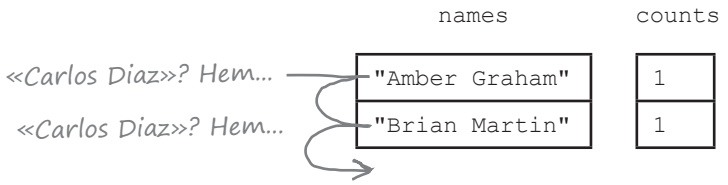
```
for _, line := range lines {
    // ...
}
```

Внутренний цикл просматривает каждый элемент сегмента в поисках имени, равного текущей строке из файла.

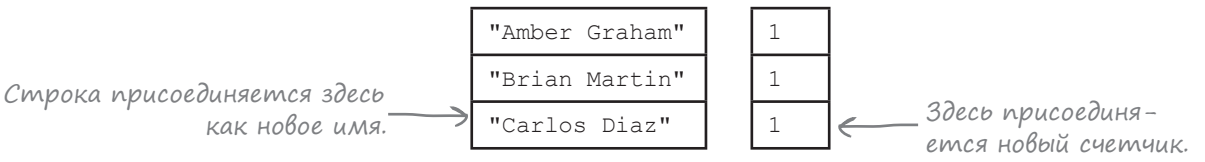
В сегменте «names» ищется элемент, совпадающий с текущей строкой файла.

```
for i, name := range names {
    if name == line {
        counts[i] += 1
        matched = true
    }
}
```

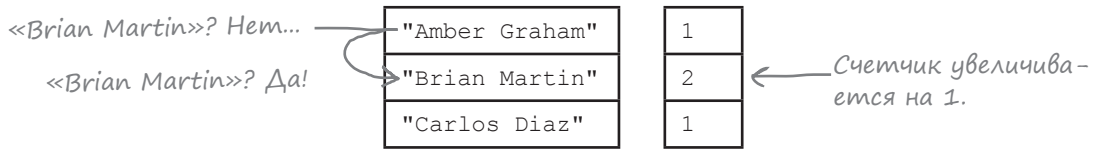
Допустим, кто-то вписал в бюллетень своего кандидата и в текстовом файле появилась строка `"Carlos Diaz"`. Программа последовательно перебирает элементы `names` и проверяет, равен ли каждый из них строке `"Carlos Diaz"`.



Если совпадение не находится, программа присоединяет строку `"Carlos Diaz"` к сегменту `names` и сохраняет соответствующий счетчик `1` в сегменте `counts` (потому что эта строка представляет первый голос, поданный за `"Carlos Diaz"`).

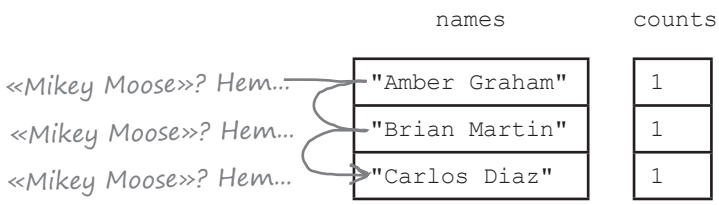


Но предположим, что следующая строка файла содержит имя `"Brian Martin"`. Так как эта строка уже существует в сегменте `names`, программа находит его и увеличивает соответствующее значение из `counts` на 1.



Карты

Но хранение имен в сегментах создает одну проблему: для каждой строки файла необходимо просмотреть многие (а то и все) значения в сегменте `names` для сравнения. Для небольших участков такое решение подойдет, но на большом участке с множеством избирателей такое решение может оказаться слишком медленным!



Хранение данных в сегменте можно сравнить с большой стопкой документов; вы сможете достать из стопки конкретный документ, но в худшем случае вам придется перебрать всю стопку *до последнего документа*.

В Go также существует другой способ хранения коллекций данных: *карты*. Карта представляет собой коллекцию, в которой вы обращаетесь к значениям по *ключу*. Ключи обеспечивают простой механизм извлечения данных из карты. Такую коллекцию можно сравнить с архивом, в котором документы разложены по аккуратно подписанным папкам.

Начать с верха, перебрать всю стопку.

Сегмент

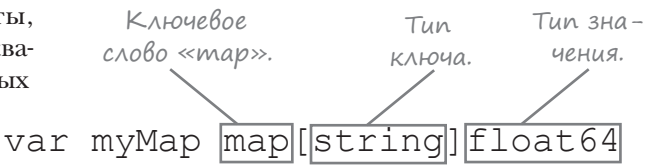
Ключи ускоряют поиск данных!

Карта

Если массивы и сегменты могут использовать в качестве индексов только *числа*, то в карте ключом может быть *любой* тип (при условии, что значения этого типа можно сравнивать оператором ==). К этой категории относятся числа, строки и т. д. Все значения должны относиться к одному типу, и все ключи тоже должны иметь одинаковый тип, но типы ключей и значений вполне могут быть разными.

Карты (продолжение)

Чтобы объявить переменную для хранения карты, введите ключевое слово `map`, за которым следуют квадратные скобки (`[]`) с типом ключа. После квадратных скобок указывается тип значения.



Как и в случае с сегментами, объявление переменной-карты не приводит к автоматическому созданию карты, для этого необходимо вызвать функцию `make` (ту же, которая используется для создания сегментов). Вместо типа сегмента функции `make` можно передать тип создаваемой карты (он будет совпадать с типом переменной, которой карта будет присвоена).

```
var ranks map[string]int
ranks = make(map[string]int)
```

Объявление переменной для карты.

Непосредственное создание карты.

А может, вам будет проще воспользоваться коротким объявлением переменной:

```
ranks := make(map[string]int)
```

Создание карты и объявление переменной для ее хранения.

Синтаксис присваивания значений в карте и их последующей выборки имеет много общего с синтаксисом присваивания и чтения значений в массивах и сегментах. Но если массивы и сегменты позволяют использовать в качестве индексов элементов только целые числа, то для ключей карт можно выбрать практически любой тип. Например, карта `ranks` использует строковые ключи:

```
ranks["gold"] = 1
ranks["silver"] = 2
ranks["bronze"] = 3
fmt.Println(ranks["bronze"])
fmt.Println(ranks["gold"])
```

```
3
1
```

У массивов и сегментов индексами могут быть только целые числа. Для ключей карты можно выбрать практически любой тип.

Другая карта, в которой и ключами, и значениями являются строки:

```
elements := make(map[string]string)
elements["H"] = "Hydrogen"
elements["Li"] = "Lithium"
fmt.Println(elements["Li"])
fmt.Println(elements["H"])
```

```
Lithium
Hydrogen
```

А это карта с целочисленными ключами и логическими значениями:

```
isPrime := make(map[int]bool)
isPrime[4] = false
isPrime[7] = true
fmt.Println(isPrime[4])
fmt.Println(isPrime[7])
```

```
false
true
```

Литералы карт

Если набор ключей и значений, которыми должна инициализироваться карта, известен заранее, то как и в случае с массивами и сегментами, вы можете воспользоваться **литералом карты** для ее создания. Литерал карты начинается с типа карты (в форме `map[`*ТипКлюча*`]`*ТипЗначения*). За ним следуют заключенные в фигурные скобки пары «ключ/значение», которыми должна инициализироваться карта. Каждая пара «ключ/значение» состоит из ключа, двоеточия и значения. Пары «ключ/значение» разделяются запятыми.

Пара предыдущих примеров, записанных в форме литералов карт:

```
ranks := map[string]int{"bronze": 3, "silver": 2, "gold": 1}
fmt.Println(ranks["gold"])
fmt.Println(ranks["bronze"])
elements := map[string]string{
      "H": "Hydrogen",
      "Li": "Lithium",
}
fmt.Println(elements["H"])
fmt.Println(elements["Li"])
```

Литерал карты.

Многострочный литерал карты.

```
1
3
Hydrogen
Lithium
```

Как и в случае с литералами сегментов, с пустыми фигурными скобками будет создана карта, пустая в исходном состоянии.

```
emptyMap := map[string]float64{}
```

Создание пустой карты.

Упражнение

Заполните пустые места в программе, чтобы она выдавала указанный результат.

```
jewelry := _____(map[string]float64)
jewelry["necklace"] = 89.99
jewelry[_____] = 79.99
clothing := ____[string]float64{_____: 59.99, "shirt": 39.99}
fmt.Println("Earrings:", jewelry["earrings"])
fmt.Println("Necklace:", jewelry[_____])
fmt.Println("Shirt:", clothing[_____])
fmt.Println("Pants:", clothing["pants"])
```

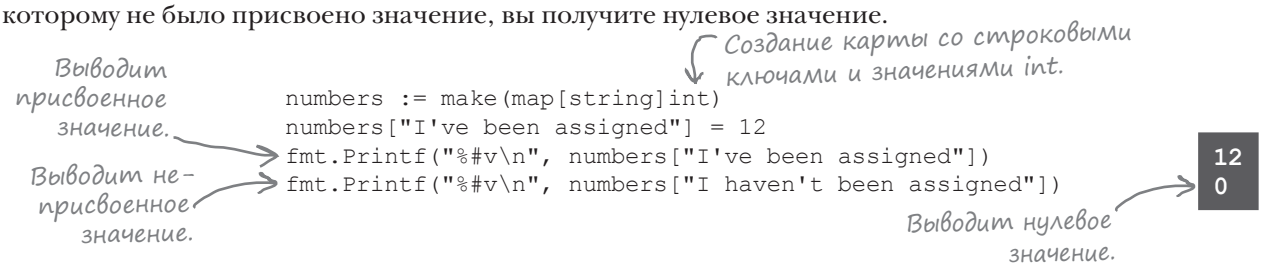
Результат.

```
Earrings: 79.99
Necklace: 89.99
Shirt: 39.99
Pants: 59.99
```

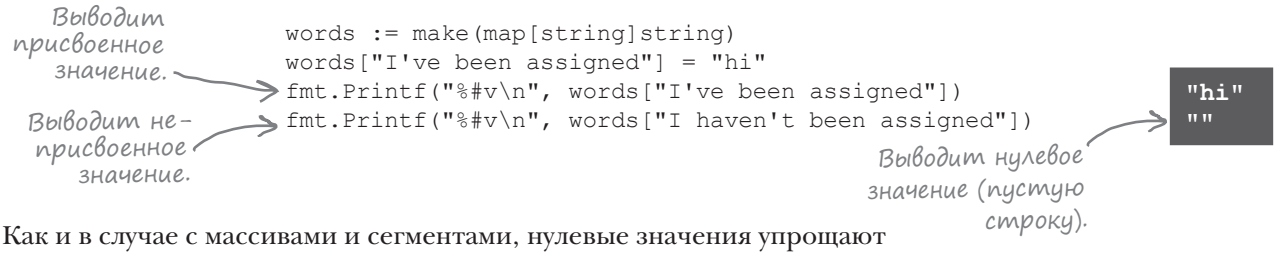
Ответ на с. 248.

Нулевые значения с картами

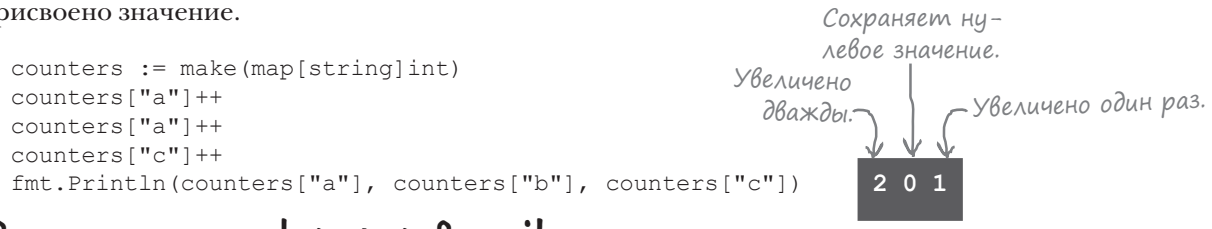
Как и в случае с массивами и сегментами, при обращении к ключу карты, которому не было присвоено значение, вы получите нулевое значение.



В зависимости от типа нулевое значение может быть отлично от 0. Например, для карт со строковым типом значения нулевым значением будет пустая строка.

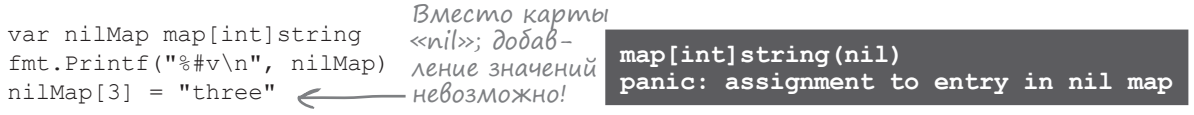


Как и в случае с массивами и сегментами, нулевые значения упрощают работу со значениями в картах даже в том случае, если им еще не было явно присвоено значение.

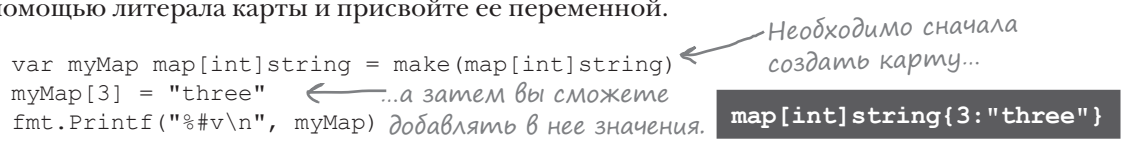


Нулевое значение для карты равно nil

Нулевым значением самой переменной, предназначенной для хранения карты, является `nil`. Если объявить переменную для карты, но не присвоить ей значение, она будет содержать `nil`. Это означает, что не существует карты, в которую можно было бы добавить новые ключи и значения. Попытавшись выполнить такую операцию, вы получите ситуацию паники:



Прежде чем пытаться добавлять ключи и значения, создайте карту функцией `make` или с помощью литерала карты и присвойте ее переменной.



Как отличить нулевые значения от присвоенных

Нулевые значения удобны, но иногда из-за них бывает трудно определить, было ли присвоено нулевое значение для заданного ключа.

Перед вами пример программы, в которой это может создать проблемы. Программа ошибочно сообщает, что студент "Carl" провалился на экзамене, тогда как на самом деле для него просто не было сохранено ни одной оценки:

```
func status(name string) {
        grades := map[string]float64{"Alma": 0, "Rohit": 86.5}
        grade := grades[name]
        if grade < 60 {
                fmt.Printf("%s is failing!\n", name)
        }
}

func main() {
        status("Alma")
        status("Carl")
}
```

Ключ карты с присвоенным значением 0.
Ключу карты не было присвоено значение.

```
Alma is failing!
Carl is failing!
```

Для подобных ситуаций обращение к ключу карты может возвращать второе (логическое) значение. Это значение равно `true`, если возвращаемое значение было реально присвоено в карте, или `false`, если возвращаемое значение просто представляет нулевое значение по умолчанию. Большинство разработчиков Go присваивает это логическое значение переменной с именем `ok` (потому что это удобное и короткое имя).

У разработчиков Go эта идиома получила название «запятая-OK». Мы снова увидим ее при рассмотрении отладочных условий в главе 11.

```
counters := map[string]int{"a": 3, "b": 0}
var value int
var ok bool
value, ok = counters["a"]
fmt.Println(value, ok)
value, ok = counters["b"]
fmt.Println(value, ok)
value, ok = counters["c"]
fmt.Println(value, ok)
```

Обращение к присвоенному значению.
«ok» содержит true.
Обращение к присвоенному значению.
«ok» содержит true.
Обращение к неприсвоенному значению.
«ok» содержит false.

```
3 true
0 true
0 false
```

Если вы хотите просто проверить, присутствует значение в карте или нет, то вы можете проигнорировать само значение, присвоив его пустому идентификатору _.

Проверяет присутствие значения, но игнорирует его.
Проверяет присутствие значения, но игнорирует его.

```
counters := map[string]int{"a": 3, "b": 0}
var ok bool
_, ok = counters["b"]
fmt.Println(ok)
_, ok = counters["c"]
fmt.Println(ok)
```

```
true
false
```

Как отличить нулевые значения от присвоенных (продолжение)

По второму возвращаемому значению можно решить, следует ли интерпретировать значение, полученное из карты, как присвоенное значение, которое просто случайно совпало с нулевым значением этого типа, или же это значение не присваивалось.

Ниже приведена обновленная версия кода, которая перед выводом сообщения о провале проверяет, было ли присвоено значение по заданному ключу:

Получаем значение и логический признак, который сообщает, является ли это значение присвоенным.

Если значение для заданного ключа не было присвоено...

...сообщить, что для студента информации об оценках нет.

В противном случае выполняется логика вывода сообщения о провале.

```
func status(name string) {
	grades := map[string]float64{"Alma": 0, "Rohit": 86.5}
	grade, ok := grades[name]
	if !ok {
		fmt.Printf("No grade recorded for %s.\n", name)
	} else if grade < 60 {
		fmt.Printf("%s is failing!\n", name)
	}
}

func main() {
	status("Alma")
	status("Carl")
}
```

```
Alma is failing!
No grade recorded for Carl.
```

Упражнение

Напишите, какой результат выведет этот фрагмент кода.

```
data := []string{"a", "c", "e", "a", "e"}
counts := make(map[string]int)
for _, item := range data {
	counts[item]++
}
letters := []string{"a", "b", "c", "d", "e"}
for _, letter := range letters {
	count, ok := counts[letter]
	if !ok {
		fmt.Printf("%s: not found\n", letter)
	} else {
		fmt.Printf("%s: %d\n", letter, count)
	}
}
```

Результат:

.....

.....

.....

.....

.....

Ответ на с. 262.

Удаление пар «ключ/значение» функцией «delete»

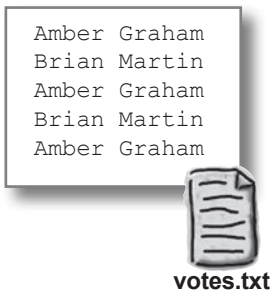
Возможно, после присваивания значения для ключа в какой-то момент вы захотите удалить его из карты. Go предоставляет для этой цели встроенную функцию `delete`. Передайте функции `delete` два аргумента: карту, из которой удаляется ключ, и удаляемый ключ. Ключ вместе с соответствующим значением удаляется из карты.

В следующем коде мы присваиваем значения для ключей в двух разных картах, а затем снова удаляем их. После этого при попытке обращения по этим ключам будет получено нулевое значение (`0` для карты `ranks`, `false` для карты `isPrime`). Вторичное логическое значение равно `false` в обоих случаях, что указывает на отсутствие ключа.

```
var ok bool
ranks := make(map[string]int)
var rank int
ranks["bronze"] = 3          // Присваиваем значение для ключа «bronze».
rank, ok = ranks["bronze"]   // «ok» содержит true, потому что значение присутствует.
fmt.Printf("rank: %d, ok: %v\n", rank, ok)
delete(ranks, "bronze")      // Удаляем ключ «bronze» с соответствующим значением.
rank, ok = ranks["bronze"]   // «ok» содержит false, потому что значение было удалено.
fmt.Printf("rank: %d, ok: %v\n", rank, ok)

isPrime := make(map[int]bool)
var prime bool
isPrime[5] = true            // Присваивает значение для ключа 5.
prime, ok = isPrime[5]       // «ok» содержит true, потому что значение присутствует.
fmt.Printf("prime: %v, ok: %v\n", prime, ok)
delete(isPrime, 5)           // Удаляет ключ 5 с соответствующим значением.
prime, ok = isPrime[5]       // «ok» содержит false, потому что значение было удалено.
fmt.Printf("prime: %v, ok: %v\n", prime, ok)
```

```
rank: 3, ok: true
rank: 0, ok: false
prime: true, ok: true
prime: false, ok: false
```

Версия программы с использованием карты



Итак, теперь вы немного лучше понимаете, как работают карты. Посмотрим, удастся ли нам применить новые знания для упрощения программы подсчета голосов.

В предыдущей версии использовалась пара сегментов: в сегменте с именем `names` хранились имена кандидатов, а в сегменте с именем `vote` — счетчики для всех имен. Для каждого имени, прочитанного из файла, нам приходилось перебирать сегмент с именами в поисках совпадения. После этого счетчик для найденного имени увеличивался в соответствующем элементе сегмента `counts`.

```
// ...
var names []string
var counts []int
for _, line := range lines {
       matched := false
       for i, name := range names {
              if name == line {
                     counts[i] += 1
// ...
```

Переменная для хранения сегмента с именами кандидатов.

Переменная для хранения сегмента с количеством вхождений каждого имени.

В цикле перебираем все элементы сегмента slice.

Если строка совпадает с текущим именем...

...увеличить соответствующий счетчик.

Решение с картой получается намного проще. Два сегмента заменяются одной картой (которую мы тоже назовем `counts`). В ней ключами станут имена кандидатов, а значениями — целые числа (счетчики голосов для имени). После того как карта будет создана, остается лишь использовать каждое имя кандидата, прочитанное из файла, в качестве ключа и увеличить значение, соответствующее ключу.

Ниже приведен упрощенный код, который создает карту и увеличивает значения для некоторых кандидатов напрямую:

```
counts := make(map[string]int)
counts["Amber Graham"]++
counts["Brian Martin"]++
counts["Amber Graham"]++
fmt.Println(counts)
```

```
map[Amber Graham:2 Brian Martin:1]
```

В предыдущей версии требовалась отдельная логика добавления новых элементов в оба сегмента в том случае, если имя не было найдено...

```
if matched == false {
       names = append(names, line)
       counts = append(counts, 1)
}
```

Если совпадение не найдено...

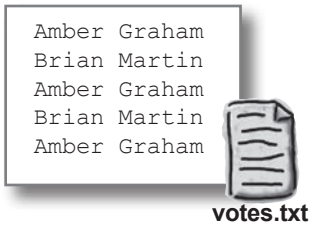
...добавить новое имя...

...и добавить новый счетчик (текущая строка станет первым вхождением).

Но с картой все это становится лишним. Если ключ, к которому мы обращаемся, не существует, карта вернет нулевое значение (в данном случае `0`, потому что значениями являются целые числа). После этого значение увеличивается, а полученный результат `1` присваивается в карте. При повторном обнаружении этого имени мы получим присвоенное значение, которое затем увеличивается как обычно.

Версия программы с использованием карты (продолжение)

А теперь попробуем внедрить карту `counts` в программу, чтобы она могла читать данные голосования из файла.



Будем откровенны: после того, как вы основательно потрудились, изучая карты, итоговый код не производит особого впечатления. Два объявления сегментов заменены одним объявлением карты. Далее идет код цикла, который обрабатывает строки из файла. Исходные 11 строк заменились всего одной строкой, которая увеличивает счетчик в карте для текущего имени кандидата. А цикл вывода результатов в самом конце заменен одной строкой, которая выводит всю карту `counts`.


```
package main

import (
        "fmt"
        "github.com/headfirstgo/datafile"
        "log"
)

func main() {
        lines, err := datafile.GetStrings("votes.txt")
        if err != nil {
                log.Fatal(err)
        }
        counts := make(map[string]int)
        for _, line := range lines {
                counts[line]++
        }
        fmt.Println(counts)
}
```

Объявляем карту, у которой ключами являются имена кандидатов, а значениями — счетчики голосов.

Увеличивает счетчик голосов для текущего кандидата.

Выводит заполненную карту.

Но поверьте, он только *кажется* простым. Здесь выполняются достаточно сложные операции. Однако карта берет все трудности на себя, а это означает, что вам придется писать меньше кода!

Как и прежде, программа перекомпилируется командой `go install`. При запуске исполняемая программа загружает и обрабатывает файл *votes.txt*. Она выводит карту `counts` со счетчиками, которые показывают, сколько раз каждое имя встречается в файле.

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/count
$ cd /Users/jay/go/bin
$ ./count
map[Amber Graham:3 Brian Martin:2]
```

Циклы for...range с картами

Программа очень полезная. Но мы же не можем показывать результаты прессе в таком виде... Нельзя ли вывести их в более осмысленном формате?

Наш формат.

```
map[Amber Graham:3 Brian Martin:2]
```

Имя: Кевин Вагнер
Профессия: волонтер на выборах

Верно. Формат с одним именем и одним счетчиком на строку будет безусловно удобнее:

Нужный формат.

```
Amber Graham: 3
Brian Martin: 2
```

Чтобы отформатировать каждый ключ и значение из карты в отдельной строке, необходимо перебрать все элементы карты.

Цикл `for...range`, который ранее использовался для обработки элементов массивов и сегментов, работает и для карт. Но вместо целочисленного индекса первой указанной переменной будет присвоен текущий ключ карты.

Переменная для каждого ключа карты.

Переменная для каждого соответствующего значения.

Ключевое слово «range».

Обрабатываемая карта.

```
for key, value := range myMap {
    // Блок цикла.
}
```

Циклы for...range с картами (продолжение)

Цикл `for...range` предоставляет простой способ перебора ключей и значений карты. Предоставьте переменную для хранения каждого ключа, другую переменную для хранения соответствующего значения — и цикл автоматически переберет все элементы карты.

```
package main

import "fmt"

func main() {
      grades := map[string]float64{"Alma": 74.2, "Rohit": 86.5, "Carl": 59.7}
      for name, grade := range grades {
            fmt.Printf("%s has a grade of %0.1f%%\n", name, grade)
      }
}
```

Перебор всех пар «ключ/значение».

Выводит все ключи с соответствующими значениями.

```
Carl has a grade of 59.7%
Alma has a grade of 74.2%
Rohit has a grade of 86.5%
```

Если вы хотите перебрать только ключи, опустите переменную для хранения значений:

Обрабатываются только ключи.

```
fmt.Println("Class roster:")
for name := range grades {
      fmt.Println(name)
}
```

```
Class roster:
Alma
Rohit
Carl
```

А если нужны только значения, укажите для ключей пустой идентификатор _:

Обрабатываются только значения.

```
fmt.Println("Grades:")
for _, grade := range grades {
      fmt.Println(grade)
}
```

```
Grades:
59.7
74.2
86.5
```

Но с этим примером связана одна потенциальная проблема. Если сохранить приведенный пример в файле и запустить его командой `go run`, выясняется, что ключи и значения карты выводятся в случайном порядке. При многократном запуске программы вы будете каждый раз получать новый порядок.

(Примечание: это не относится к коду, запускаемому на сайте Go Playground. Там порядок будет оставаться случайным, но при каждом запуске будет выводиться один и тот же результат.)

Каждый раз цикл выполняется в другом порядке!

```
Shell Edit View Window Help
$ go run temp.go
Alma has a grade of 74.2%
Rohit has a grade of 86.5%
Carl has a grade of 59.7%
$ go run temp.go
Carl has a grade of 59.7%
Alma has a grade of 74.2%
Rohit has a grade of 86.5%
```

Цикл for...range обрабатывает карты в случайном порядке!

Цикл `for...range` обрабатывает ключи и значения карты в случайном порядке, так как карта является *неупорядоченной* коллекцией ключей и значений. Используя цикл `for...range` с картой, вы никогда не знаете, в каком порядке получите доступ к ее содержимому! Иногда это нормально, но если вам требуется более последовательное упорядочение, соответствующий код придется написать самостоятельно.

Ниже приведена обновленная версия программы, которая всегда выводит имена в алфавитном порядке. Для этого она использует два разных цикла `for`. Первый цикл перебирает все ключи в карте, игнорируя значения, и добавляет их в сегмент строк. Затем сегмент передается функции `Strings` пакета `sort`, которая сортирует их на месте в алфавитном порядке.

Второй цикл `for` не перебирает содержимое карты — он перебирает отсортированный список имен (который благодаря предшествующему фрагменту содержит все ключи из карты в алфавитном порядке). Он выводит имя, а затем получает из карты значение, соответствующее этому имени. При этом будут обработаны все ключи и значения из карты, но ключи берутся из отсортированного сегмента, а не из самой карты.

```
package main

import (
        "fmt"
        "sort"
)

func main() {
        grades := map[string]float64{"Alma": 74.2, "Rohit": 86.5, "Carl": 59.7}
        var names []string
        for name := range grades {
                names = append(names, name)
        }
        sort.Strings(names)
        for _, name := range names {
                fmt.Printf("%s has a grade of %0.1f%%\n", name, grades[name])
        }
}
```

Построение сегмента со всеми ключами карты.

Сегмент сортируется в алфавитном порядке. Имена обрабатываются в алфавитном порядке.

Текущее имя используется для выборки оценки из карты.

Если сохранить этот код и запустить его, то имена студентов будут выводиться в алфавитном порядке. Этот порядок сохранится, сколько бы раз вы ни запустили программу.

Если порядок обработки данных карты для вас неважен, вероятно, непосредственное использование цикла `for...range` с картой приведет вас к нужному результату. Но если порядок для вас существенен, подумайте над тем, чтобы самостоятельно реализовать нужный порядок обработки.

```
Shell Edit View Window Help
$ go run temp.go
Alma has a grade of 74.2%
Carl has a grade of 59.7%
Rohit has a grade of 86.5%
$ go run temp.go
Alma has a grade of 74.2%
Carl has a grade of 59.7%
Rohit has a grade of 86.5%
```

Каждый раз имена обрабатываются в алфавитном порядке.

Обновление программы подсчета голосов циклом for...range

Кандидатов на выборах в школьный совет не так много, поэтому сортировка результатов по имени не имеет смысла. Мы просто воспользуемся циклом `for...range`, чтобы обработать ключи и значения прямо из карты.

```
Amber Graham
Brian Martin
Amber Graham
Brian Martin
Amber Graham
```

votes.txt

Изменение вносится достаточно просто: строка вывода всей карты заменяется циклом `for...range`. Каждый ключ присваивается переменной `name`, а каждое значение — переменной `count`. Затем имя текущего кандидата и количество голосов выводятся вызовом функции `Printf`.


```
package main

import (
        "fmt"
        "github.com/headfirstgo/datafile"
        "log"
)

func main() {
        lines, err := datafile.GetStrings("votes.txt")
        if err != nil {
                log.Fatal(err)
        }
        counts := make(map[string]int)
        for _, line := range lines {
                counts[line]++
        }
        for name, count := range counts {
                fmt.Printf("Votes for %s: %d\n", name, count)
        }
}
```

Обработка каждого ключа и значения в карте.

Вывод ключа (имя кандидата).

Вывод значения (количество голосов).

Снова компиляция командой `go install`, снова запуск исполняемого файла — и наш вывод отображается в новом формате. Программа выводит имя каждого кандидата и соответствующий ему счетчик голосов в отдельной строке.

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/count
$ cd /Users/jay/go/bin
$ ./count
Votes for Amber Graham: 3
Votes for Brian Martin: 2
```

Программа подсчета голосов готова!

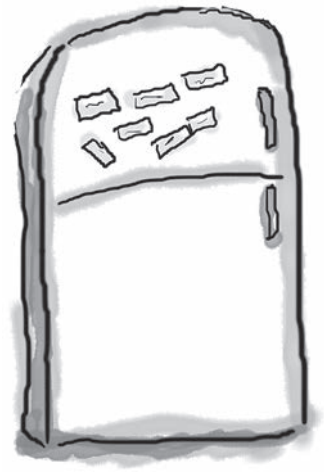
Я знала, что избиратели сделают правильный выбор! Хочу поздравить своего соперника с непростой избирательной кампанией...

```
Shell Edit View Window Help
$ go install github.com/headfirstgo/count
$ cd /Users/jay/go/bin
$ ./count
Votes for Amber Graham: 3
Votes for Brian Martin: 2
```

Программа подсчета голосов готова!

Когда из всех коллекций данных в нашем распоряжении были только массивы и сегменты, для выборки значений нам требовалось много лишнего кода и времени. С картами эта задача решается элементарно! Каждый раз, когда вам снова потребуется организовать поиск значений в коллекции, подумайте об использовании карты.

Развлечения с магнитами



На холодильнике была выложена программа Go, в которой цикл `for...range` используется для вывода содержимого карты. Расставьте фрагменты кода так, чтобы у вас получилась работоспособная программа, которая выводит указанный результат. (Порядок вывода может изменяться при разных запусках программы.)

```
package main
import "fmt"
func main() {
ranks
{
}
}
}
{
"bronze": 3
"silver": 2
"gold": 1
:=
:=
,
,
,
,
fmt.Printf(
)
"The %s medal's rank is %d\n"
```

```
map
range
ranks
for
[string]
int
medal
rank
medal
rank
```

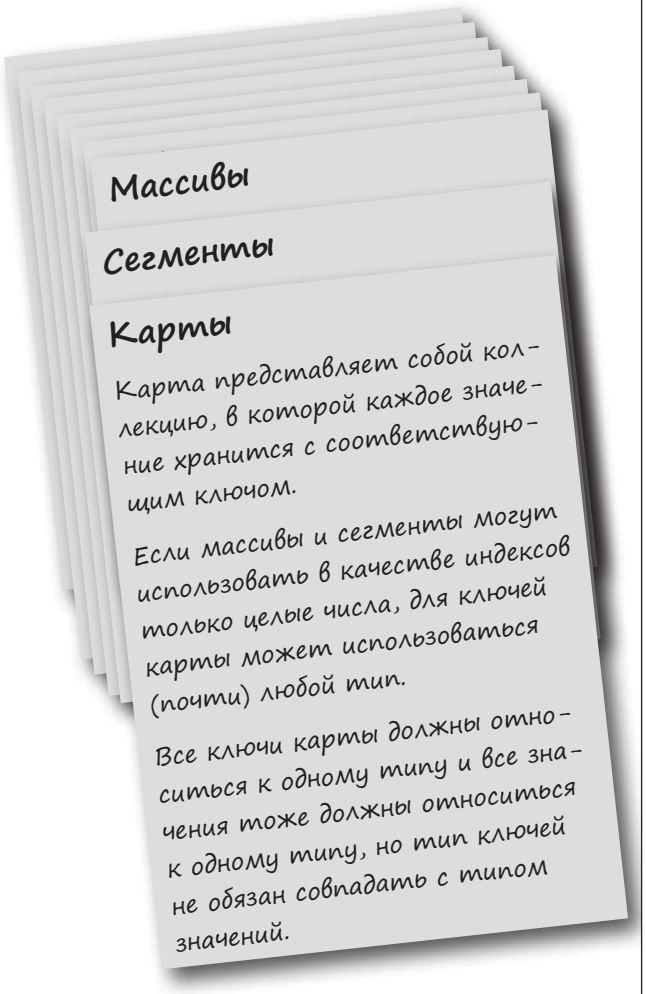
Результат.

```
The gold medal's rank is 1
The bronze medal's rank is 3
The silver medal's rank is 2
```

Ответ на с. 263.

Ваш инструментарий Go

Глава 7 осталась позади! В ней ваш инструментарий пополнился картами.



КЛЮЧЕВЫЕ МОМЕНТЫ

- При объявлении переменной для карты необходимо указать типы ключей и значений:
  `var myMap map[string]int`
- Чтобы создать новую карту, вызовите функцию `make` с типом карты:
  `myMap = make(map[string]int)`
- Чтобы присвоить значение в карте, укажите ключ, по которому присваивается значение, в квадратных скобках:
  `myMap["my key"] = 12`
- Чтобы прочитать значение из карты, также следует указать ключ:
  `fmt.Println(myMap["my key"])`
- Инициализацию карты можно совместить с ее созданием при помощи **литерала карты**:
  `map[string]int{"a": 2, "b": 3}`
- Как и в случае с массивами и сегментами, при обращении к ключу карты, для которого не было присвоено значение, вы получите нулевое значение.
- При получении значения от карты может возвращаться второе необязательное логическое значение, которое указывает, было ли это значение присвоено или же представляет нулевое значение по умолчанию:
  `value, ok := myMap["c"]`
- Если вы хотите только проверить, связано ли с ключом значение, проигнорируйте значение с использованием пустого идентификатора `_`:
  `_, ok := myMap["c"]`
- Чтобы удалить ключи и соответствующие значения из карты, используйте встроенную функцию `delete`:
  `delete(myMap, "b")`
- Циклы `for...range` могут использоваться с картами по аналогии с тем, как они используются с массивами или сегментами. Вы предоставляете только одну переменную, которой будет последовательно присваиваться каждый ключ, и вторую переменную, которой будет последовательно присваиваться каждое значение.

```
for key, value := range myMap {
    fmt.Println(key, value)
}
```

Упражнение
Решение

Заполните пустые места в программе, чтобы она выдавала указанный результат.

```
jewelry := make(map[string]float64)
jewelry["necklace"] = 89.99
jewelry["earrings"] = 79.99
clothing := map[string]float64{"pants": 59.99, "shirt": 39.99}
fmt.Println("Earrings:", jewelry["earrings"])
fmt.Println("Necklace:", jewelry["necklace"])
fmt.Println("Shirt:", clothing["shirt"])
fmt.Println("Pants:", clothing["pants"])
```

Создаем новую, пустую карту.

Значения присваиваются по ключам.

Литерал карты используется для создания новой инициализированной карты.

Вывод разных значений из карты.

Результат.

```
Earrings: 79.99
Necklace: 89.99
Shirt: 39.99
Pants: 59.99
```

Упражнение
Решение

Напишите, какой результат выведет этот фрагмент кода.

```
data := []string{"a", "c", "e", "a", "e"}
counts := make(map[string]int)
for _, item := range data {
    counts[item]++
}
letters := []string{"a", "b", "c", "d", "e"}
for _, letter := range letters {
    count, ok := counts[letter]
    if !ok {
        fmt.Printf("%s: not found\n", letter)
    } else {
        fmt.Printf("%s: %d\n", letter, count)
    }
}
```

Подсчет количества вхождений каждой буквы в сегменте.

Карта для хранения счетчиков.

Обработка каждой буквы.

Увеличение счетчика для текущей буквы.

Проверяем, существует ли каждая из этих букв в карте.

Получение счетчика для текущей буквы, а также признака ее обнаружения.

Если буква не найдена...

...сообщить об этом.

В противном случае буква была найдена...

...тогда выводится буква и счетчик.

Результат:

a: 2

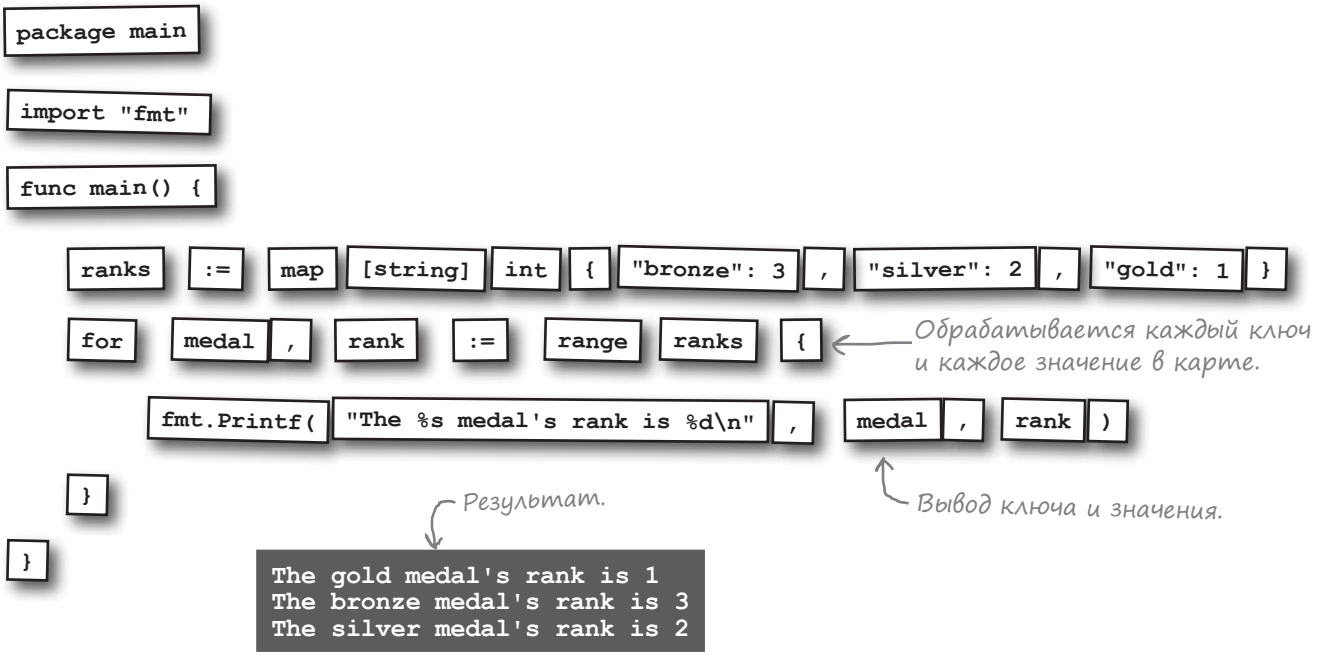
b: not found

c: 1

d: not found

e: 2

Развлечения с магнитами. Решение



Структуры

Иногда требуется хранить вместе несколько типов данных. Сначала вы познакомились с сегментами, предназначенными для хранения списков. Затем были рассмотрены карты, связывающие список ключей со списком значений. Но обе структуры данных позволяют хранить значения только *одного* типа, а в некоторых ситуациях требуется сгруппировать значения *нескольких* типов. Например, в почтовых адресах названия улиц (строки) группируются с почтовыми индексами (целые числа). Или в информации о студентах имена (строки) объединяются со средними оценками (вещественные числа). Сегменты и карты не позволяют смешивать разные типы. Тем не менее это *возможно* при использовании другого типа данных, называемого **структурой**. В этой главе мы подробно изучим структуры.

В сегментах и картах хранятся значения ОДНОГО типа

Gopher Fancy — новый журнал, посвященный симпатичным грызунам. В настоящее время редакция работает над системой хранения информации о своих подписчиках.

Для начала нам нужно сохранить имя подписчика, ежемесячную плату и признак активности подписки. Но имя — строка, плата имеет тип `float64`, а признак активности — тип `bool`. Сохранить все эти типы в одном сегменте не удастся!

Сегмент можно настроить для хранения только одного типа.

```
subscriber := []string{}
subscriber = append(subscriber, "Aman Singh")
subscriber = append(subscriber, 4.99)
subscriber = append(subscriber, true)
```

Мы не можем добавить тип float64!

Мы не можем добавить тип bool!

```
cannot use 4.99 (type float64) as type string in append
cannot use true (type bool) as type string in append
```

Потом мы попытались использовать карты. Это было бы удобно, потому что ключи можно использовать для пометки того, что представляет каждое значение. Но как и сегменты, карты могут хранить значения только одного типа!

В картах могут храниться значения только одного типа.

```
subscriber := map[string]float64{}
subscriber["name"] = "Aman Singh"
subscriber["rate"] = 4.99
subscriber["active"] = true
```

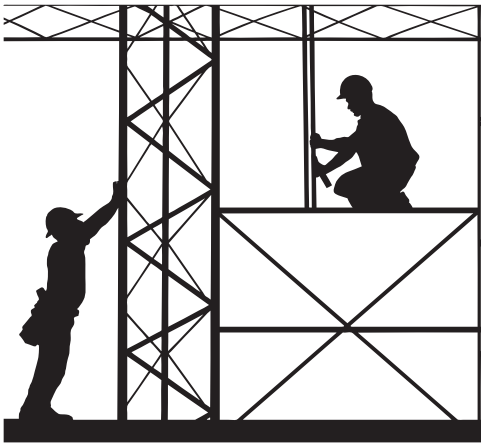
Сохранить строку не удастся!

И логическое значение тоже!

```
cannot use "Aman Singh" (type string)
as type float64 in assignment
cannot use true (type bool)
as type float64 in assignment
```

Да, это правда: если вам нужно хранить смешанные значения разных типов, то массивы, сегменты и карты не подойдут. Их можно настроить только для хранения значений одного типа. Однако в Go существует способ решения этой проблемы...

Структуры формируются из значений МНОГИХ типов



Структура представляет собой значение, которое строится из других значений разных типов. Если в сегменте могут храниться только строковые значения, а в карте — только целочисленные значения, вы можете создать структуру для хранения строковых значений, значений `int`, `float64`, `bool` и т. д. — и все это в одной удобной группе.

Тип структуры объявляется ключевым словом `struct`, за которым следуют фигурные скобки. В фигурных скобках определяются одно или несколько **полей**: значений, группируемых в структуре. Определение каждого поля размещается в отдельной строке и состоит из имени поля, за ним следует тип значения, которое будет храниться в этом поле.

Ключевое слово «struct».

```
struct {
    field1 string
    field2 int
}
```

Имя поля. — Тип поля.

Тип структуры может использоваться в качестве типа объявляемой переменной. В этом коде объявляется переменная с именем `myStruct` для хранения структуры с полем `number` типа `float64`, полем `word` типа `string` и полем `toggle` типа `bool`:

(Чаще для объявления структурных переменных используется определение типа, но мы рассмотрим эту тему через несколько страниц, поэтому пока будем использовать такую форму записи.)

Объявление переменной с именем «myStruct».

```
var myStruct struct {
    number float64
    word   string
    toggle bool
}
fmt.Printf("%#v\n", myStruct)
```

Переменная «myStruct» может хранить структуры, состоящие из поля float64 с именем «number», поля string с именем «word» и поля bool с именем «toggle».

Значение структуры выводится в том виде, в котором оно записывается в коде Go.

Поля структуры, каждому из которых присваивается нулевое значение данного типа.

```
struct { number float64; word string; toggle bool }
{number:0, word:"", toggle:false}
```

Если вызвать функцию `Printf` с глаголом `%#v`, она выведет значение из `myStruct` в виде литерала структуры. Литералы структур будут рассмотрены позднее в этой главе, а пока мы видим, что полю `number` структуры присвоено значение `0`, полю `word` — пустая строка, а полю `toggle` — значение `false`. Каждое поле заполняется нулевым значением своего типа.

Не беспокойтесь о количестве пробелов между именами и типами полей структур.

Когда вы записываете поля структуры, поставьте один пробел между именем поля и его типом. Когда вы выполните команду `go fmt` для своего файла (а это следует делать всегда), команда вставит дополнительные пробелы, чтобы все типы были выровнены по вертикали. Выравнивание всего лишь упрощает чтение кода: его смысл при этом совершенно не изменяется!

```
var aStruct struct {
        shortName   int
        longerName  float64
        longestName string
}
```

Дополнительные пробелы выравнивают типы.

Обращение к полям структуры

Теперь вы знаете, как определяются структуры. Но чтобы использовать структуру в программе, необходимо каким-то образом сохранять новые значения в полях структуры и после этого читать их.

До настоящего момента мы использовали оператор «точка» для обозначения функций, «принадлежащих» другому пакету, или методов, «принадлежащих» некоторому значению:

```
fmt.Println("hi")
```

Вызов функции, принадлежащей пакету «fmt».

```
var myTime time.Time
myTime.Year()
```

Вызов метода, принадлежащего значению «Time».

Аналогичным образом оператор «точка» может использоваться для обозначения полей, «принадлежащих» структуре. Этот синтаксис подходит как для присваивания значений, так и для их чтения.

Значение-структура. *Имя поля.*

```
myStruct.number = 3.14
fmt.Println(myStruct.number)
```

Значение-структура. *Имя поля.*

Теперь мы можем воспользоваться оператором «точка» для присваивания значений всем полям `myStruct` и их последующего вывода:

```
var myStruct struct {
        number float64
        word   string
        toggle bool
}
myStruct.number = 3.14
myStruct.word = "pie"
myStruct.toggle = true
fmt.Println(myStruct.number)
fmt.Println(myStruct.word)
fmt.Println(myStruct.toggle)
```

Присваивание значений полям структуры.

Чтение значений из полей структуры.

```
3.14
pie
true
```

Хранение данных подписчиков в структуре

Итак, вы знаете, как объявить переменную для хранения структуры и как присваивать значения ее полям. Теперь мы можем создать структуру для хранения данных подписчиков журнала.

Начнем с определения переменной с именем `subscriber`. Эта переменная будет иметь структурный тип с полями `name` (`string`), `rate` (`float64`) и `active` (`bool`).

После объявления переменной и ее типа к полям структуры можно обращаться при помощи оператора «точка». Каждому полю присваивается значение соответствующего типа, после чего программа выводит значения.

Объявляем переменную «subscriber»...

...для хранения структуры.

```
var subscriber struct {
        name   string
        rate   float64
        active bool
}
```

Структура состоит из поля «name» для хранения строки...

...поля «rate» для хранения float64...

...и поля «active» для хранения bool.

Присваиваем значения полям структуры.

```
subscriber.name = "Aman Singh"
subscriber.rate = 4.99
subscriber.active = true
```

А теперь читаем значения из полей структуры.

```
fmt.Println("Name:", subscriber.name)
fmt.Println("Monthly rate:", subscriber.rate)
fmt.Println("Active?", subscriber.active)
```

```
Name: Aman Singh
Monthly rate: 4.99
Active? true
```

Хотя данные подписчиков состоят из значений нескольких типов, структуры позволяют объединить их в одном удобном пакете!

Упражнение

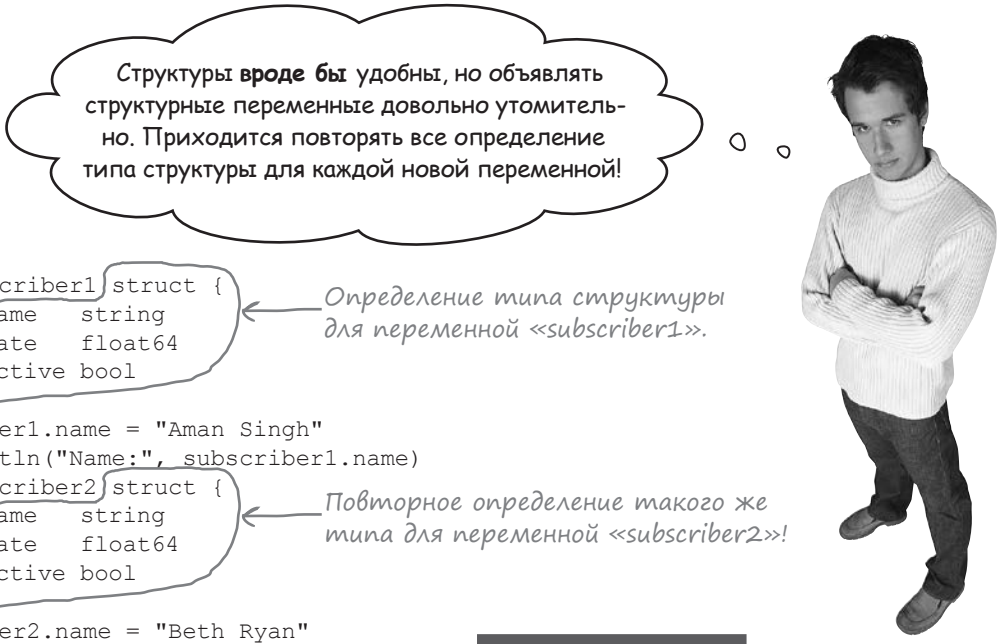
Справа приведена программа, в которой создается структурная переменная для хранения имени домашнего питомца (`string`) и его возраста (`int`). Заполните пустые места в коде, чтобы он выводил показанный результат.

```
package main

import "fmt"

func main() {
        var pet _______ {
                name  _______
                ____  int
        }
        pet._____ = "Max"
        pet.age = 5
        fmt.Println("Name:", ____.name)
        fmt.Println("Age:", pet.____)
}
```

```
Name: Max
Age: 5
```

Ответ на с. 296.

Определения типов и структуры

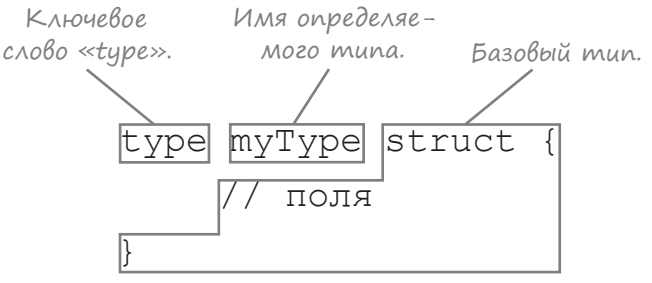


В этой книге использовались разные типы: `int`, `string`, `bool`, сегменты, карты, теперь структуры. Но до сих пор вы еще не создавали полностью *новые* типы.

Определения типов позволяют создавать собственные типы. Они создают новый **определяемый тип** на основе базового типа.

И хотя в качестве базового может использоваться любой тип (например, `float64`, `string` или даже сегменты или карты), в этой главе мы сосредоточимся на использовании структурных типов как базовых. Другие базовые типы будут использоваться после того, как мы поближе познакомимся с определениями типов в следующей главе.

Чтобы написать определение типа, используйте ключевое слово `type`, за которым следует имя нового определяемого типа и имя базового типа, на основе которого он должен создаваться. Если в качестве базового используется тип структуры, укажите ключевое слово `struct` со списком определений полей, заключенным в фигурные скобки — по аналогии с тем, как это делалось при объявлении переменных для структур.

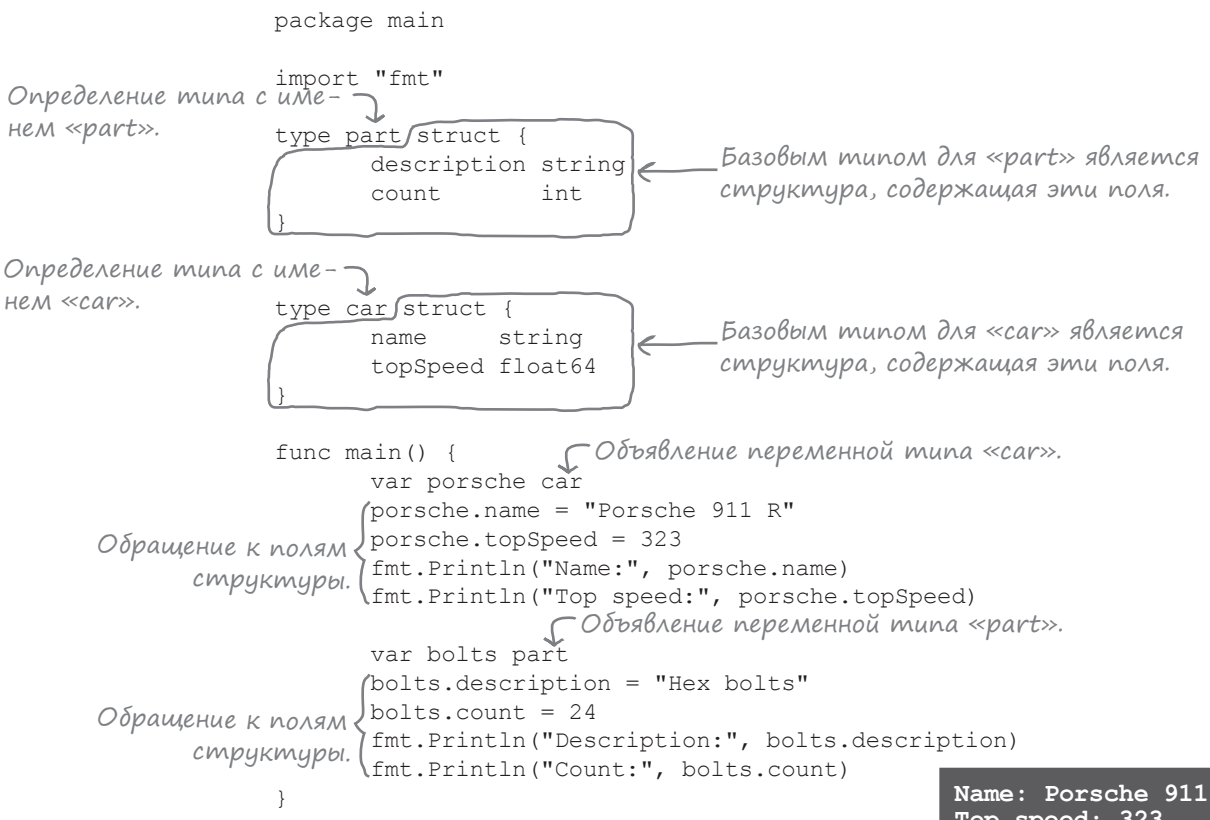


Определения типов и структуры (продолжение)

Определения типов, как и переменные, *могут* записываться внутри функций. Но в этом случае область видимости определения ограничивается блоком этой функции, что означает, что его нельзя будет использовать за пределами функции. По этой причине типы обычно определяются вне любых функций, на уровне пакета.

Простой пример: в приведенном ниже коде определяются два типа — `part` и `car`. Каждый определяемый тип использует структуру в качестве базового типа.

Затем в функции `main` объявляется переменная `porsche` типа `car` и переменная `bolts` типа `part`. Переписывать длинные определения структур при объявлении переменных не нужно; мы просто используем имена определяемых типов.




```
package main

import "fmt"

type part struct {
        description string
        count       int
}

type car struct {
        name     string
        topSpeed float64
}

func main() {
        var porsche car
        porsche.name = "Porsche 911 R"
        porsche.topSpeed = 323
        fmt.Println("Name:", porsche.name)
        fmt.Println("Top speed:", porsche.topSpeed)
        var bolts part
        bolts.description = "Hex bolts"
        bolts.count = 24
        fmt.Println("Description:", bolts.description)
        fmt.Println("Count:", bolts.count)
}
```

```
Name: Porsche 911 R
Top speed: 323
Description: Hex bolts
Count: 24
```

После того как переменные будут объявлены, мы можем присваивать значения полям их структур и читать их, как это делалось в предыдущих программах.

Использование определяемого типа для информации о подписчиках

Ранее для того, чтобы создать несколько переменных для хранения данных подписчиков в структурах, нам приходилось полностью записывать тип структуры (включая все ее поля) для каждой переменной.

```
var subscriber1 struct {
        name   string
        rate   float64
        active bool
}
// ...
var subscriber2 struct {
        name   string
        rate   float64
        active bool
}
// ...
```

Определение типа структуры.

Определение идентичного типа.

Но теперь мы можем определить тип `subscriber` на уровне пакета. Тип структуры записывается только один раз, как базовый тип для определяемого типа. Когда дойдет до объявления переменных, тип структуры не нужно будет записывать снова, достаточно указать `subscriber` в качестве типа. Повторять все определение структуры уже не нужно!

```
package main

import "fmt"
type subscriber struct {
        name   string
        rate   float64
        active bool
}

func main() {
        var subscriber1 subscriber
        subscriber1.name = "Aman Singh"
        fmt.Println("Name:", subscriber1.name)
        var subscriber2 subscriber
        subscriber2.name = "Beth Ryan"
        fmt.Println("Name:", subscriber2.name)
}
```

Определение типа с именем «subscriber».

Тип структуры используется для переменных как базовый для определения типа.

Объявление переменной типа «subscriber».

Тип «subscriber» также используется для второй переменной.

```
Name: Aman Singh
Name: Beth Ryan
```

Использование определяемых типов с функциями

Возможности определяемых типов не ограничиваются типами переменных. Определяемые типы также могут использоваться для параметров функций и возвращаемых значений.

Ниже снова приведен тип `part` с функцией `showInfo`, которая выводит поля `part`. Функция получает один параметр с типом `part`. Внутри `showInfo` мы обращаемся к полям переменной-параметра точно так же, как к полям любой другой структурной переменной.

```
package main

import "fmt"

type part struct {
        description string
        count       int
}

func showInfo(p part) {
        fmt.Println("Description:", p.description)
        fmt.Println("Count:", p.count)
}

func main() {
        var bolts part
        bolts.description = "Hex bolts"
        bolts.count = 24
        showInfo(bolts)
}
```

Объявление одного параметра с типом «part».

Обращение к полям параметра.

Создается значение «part».

Тип «part» передается функции.

```
Description: Hex bolts
Count: 24
```

А здесь функция `minimumOrder` создает значение `part` с заданным описанием `description` и заранее определенным значением поля `count`. Тип `part` также объявляется возвращаемым типом `minimumOrder`, чтобы функция могла вернуть созданную структуру.

```
// Директивы package и imports, определения
типов пропущены

func minimumOrder(description string) part {
        var p part
        p.description = description
        p.count = 100
        return p
}

func main() {
        p := minimumOrder("Hex bolts")
        fmt.Println(p.description, p.count)
}
```

Объявляется одно возвращаемое значение типа «part».

Создание нового значения «part».

Функция возвращает тип «part».

Вызывает minimumOrder. Для сохранения возвращаемого значения «part» используется короткое определение переменной.

```
Hex bolts 100
```

Использование определяемых типов с функциями (продолжение)

Рассмотрим пару функций, которые работают с типом `subscriber`.

Функция `printInfo` получает значение `subscriber` в параметре и выводит значения полей структуры.

Также имеется функция `defaultSubscriber`, которая создает новую структуру `subscriber` и заполняет ее значениями по умолчанию. Функция получает строковый параметр с именем `name` и использует его для инициализации поля `name` нового значения `subscriber`. Затем она заполняет поля `rate` и `active` значениями по умолчанию. Наконец, функция возвращает заполненную структуру `subscriber` на сторону вызова.

```
package main

import "fmt"

type subscriber struct {
        name   string
        rate   float64
        active bool
}
func printInfo(s subscriber) {
        fmt.Println("Name:", s.name)
        fmt.Println("Monthly rate:", s.rate)
        fmt.Println("Active?", s.active)
}

func defaultSubscriber(name string) subscriber {
        var s subscriber
        s.name = name
        s.rate = 5.99
        s.active = true
        return s
}

func main() {
        subscriber1 := defaultSubscriber("Aman Singh")
        subscriber1.rate = 4.99
        printInfo(subscriber1)

        subscriber2 := defaultSubscriber("Beth Ryan")
        printInfo(subscriber2)
}
```

Объявляется один параметр... *...с типом «subscriber».*

Возвращает значение «subscriber».

Создается новое значение «subscriber».

Заполняются поля структуры.

Возвращает «subscriber».

Создает subscriber с заданным значением name.

Использует заданное значение rate.

Вывод значений полей.

Создает subscriber с заданным значением name.

Вывод значений полей.

```
Name: Aman Singh
Monthly rate: 4.99
Active? true
Name: Beth Ryan
Monthly rate: 5.99
Active? true
```

В функции `main` имя подписчика передается `defaultSubscriber` для получения новой структуры `subscriber`. Один подписчик пользуется льготной ставкой оплаты, поэтому это поле структуры заполняется напрямую. Заполненные структуры `subscriber` передаются `printInfo` для вывода их содержимого.

Будьте осторожны!

Не используйте существующее имя типа как имя переменной!

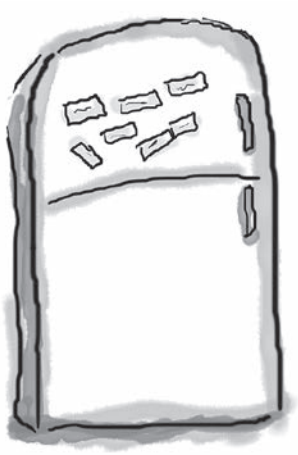
Если вы определили тип с именем `car` *в текущем пакете и объявили переменную, которой также присвоено имя* `car`*, то имя переменной заместит имя типа и сделает его недоступным.*

Обозначает тип.

```
var car car
var car2 car
```

Обозначает переменную, что приводит к ошибке!

На практике эта проблема встречается нечасто, так как определяемые типы часто экспортируются из своих пакетов (поэтому их имена записываются с буквы верхнего регистра), а переменные экспортируются реже (поэтому их имена начинаются с буквы нижнего регистра). Другие примеры экспортирования определяемых типов приводятся позднее в этой главе. Тем не менее замещение имен может стать весьма коварной проблемой, и следует учитывать такую возможность.



Развлечения с магнитами

На холодильнике была выложена программа Go. К сожалению, некоторые магниты упали на пол. Удастся ли вам расставить фрагменты кода по местам и создать работоспособную программу, которая будет выводить нужный результат? Программа должна определять тип структуры с именем `student` и содержать функцию `printInfo`, которая получает значение `student` в параметре.

```
package main
{
}
}
import "fmt"
) {
}
type
struct
s student
student
var s
fmt.Println("Name:", s.name)
printInfo(s)
fmt.Printf("Grade: %0.1f\n", s.grade)
name string
func printInfo(
func main() {
grade float64
s.name = "Alonzo Cole"
s.grade = 92.3
student
```

Результат.

```
Name: Alonzo Cole
Grade: 92.3
```

Ответ на с. 296.

Изменение структуры в функции

Мы предоставляем льготную подписку $4.99 многим читателям, поэтому я решил написать функцию applyDiscount, которая бы заполняла поле rate за нас. Но функция не работает!

```
func applyDiscount(s subscriber) {
        s.rate = 4.99
}

func main() {
        var s subscriber
        applyDiscount(s)
        fmt.Println(s.rate)
}
```

Получает параметр «subscriber».

Заполняет поле «rate».

Пытается присвоить полю «rate» структуры «subscriber» значение 4.99.

0

Поле остается равным 0!

Наши друзья в *Gopher Fancy* попробовали написать функцию, которая получает структуру в параметре и обновляет одно из полей этой структуры.

Помните, как в главе 3 мы пытались написать функцию double, которая получает число и удваивает его? После возвращения из функции число снова возвращалось к исходному значению!

Тогда мы упомянули о том, что в Go используется «передача по значению»; это означает, что параметры функций получают *копию* аргументов, с которыми вызывалась функция. Если функция изменяет значение параметра, то изменится *копия*, а не *оригинал*.

```
func main() {
        amount := 6
        double(amount)
        fmt.Println(amount)
}

func double(number int) {
        number *= 2
}
```

Функции передается аргумент.

Выводит исходное значение!

Параметру присваивается копия аргумента.

Изменяет скопированное, а не исходное значение!

6

В результате ничего не меняется!

Это относится и к структурам. Когда мы передаем applyDiscount структуру subscriber, функция получает *копию* структуры. Таким образом, при присваивании значения поля rate структуры изменяется *скопированная* структура, а не оригинал.

```
func applyDiscount(s subscriber) {
        s.rate = 4.99
}
```

Получает копию структуры!

Изменяется копия, а не оригинал!

Изменение структуры в функции (продолжение)

В главе 3 проблема решалась изменением параметра функции: в нем передавался *указатель* на значение вместо самого значения. При вызове функции благодаря оператору & передавался указатель на то значение, которое мы бы хотели обновить. Затем внутри функции оператор * использовался для обновления значения по этому указателю.

В результате обновленное значение сохранится и после возвращения из функции.

```
func main() {
        amount := 6
        double(&amount)
        fmt.Println(amount)
}

func double(number *int) {
        *number *= 2
}
```

Вместо значения переменной передается указатель.

Получает указатель вместо значения int.

Обновляется значение по указателю.

```
12
```

Выводит удвоенное значение.

Используя указатели, вы также можете добиться того, чтобы функция обновляла передаваемую структуру.

Ниже приведена обновленная версия функции `applyDiscount`, которая должна работать правильно. Мы обновляем параметр `s`, чтобы он содержал указатель на структуру `subscriber` вместо самой структуры. После этого обновляется значение в поле `rate` структуры.

В `main` функция `applyDiscount` вызывается с передачей указателя на структуру `subscriber`. А при выводе значения в поле `rate` структуры становится видно, что оно было успешно обновлено!

```
package main

import "fmt"

type subscriber struct {
        name   string
        rate   float64
        active bool
}

func applyDiscount(s *subscriber) {
        s.rate = 4.99
}

func main() {
        var s subscriber
        applyDiscount(&s)
        fmt.Println(s.rate)
}
```

Получает указатель на структуру, а не саму структуру.

Обновляет поле структуры.

Передается указатель, а не структура.

```
4.99
```

Погодите, как это работает? В функции `double` мы использовали оператор * для получения значения, на которое ссылается указатель. Разве не нужно ставить * при заполнении поля `rate` в `applyDiscount`?

Оказывается, нет! Точечная запись при обращении к полям работает как с указателями на структуры, так и с самими структурами.

Обращение к полям структур по указателю

При попытке вывести переменную, содержащую указатель, вы увидите адрес памяти, на который эта переменная указывает. Как правило, это совсем не то, что вам нужно.

```
func main() {
       var value int = 2
       var pointer *int = &value
       fmt.Println(pointer)
}
```

Создает значение.

Получает указатель на это значение.

Сюрприз! Выводится указатель, а не значение!

```
0xc420014100
```

Для получения значения, на которое ссылается указатель, следует воспользоваться оператором `*`.

```
func main() {
       var value int = 2
       var pointer *int = &value
       fmt.Println(*pointer)
}
```

Выводит значение, на которое ссылается указатель.

```
2
```

Казалось бы, оператор `*` также следует использовать и с указателями на структуры. Но если вы просто поставите `*` перед указателем на структуру, такое решение работать не будет:

```
type myStruct struct {
       myField int
}

func main() {
       var value myStruct
       value.myField = 3
       var pointer *myStruct = &value
       fmt.Println(*pointer.myField)
}
```

Создание значения структуры.

Получает указатель на значение структуры.

Ошибка!

Пытаемся получить значение структуры, на которое ссылается указатель.

```
invalid indirect of
pointer.myField (type int)
```

Но если вы используете запись `*pointer.myField`, Go посчитает, что поле `myField` должно содержать указатель. В действительности это не так, поэтому происходит ошибка. Чтобы это решение заработало, необходимо заключить `*pointer` в круглые скобки. Тогда вы сначала получите значение `myStruct`, после чего сможете обратиться к полю структуры.

```
func main() {
       var value myStruct
       value.myField = 3
       var pointer *myStruct = &value
       fmt.Println((*pointer).myField)
}
```

Получаем значения структуры по указателю, а затем обращаемся к полю структуры.

```
3
```

Обращение к полям структур по указателю (продолжение)

Но каждый раз вводить конструкцию `(*pointer).myField` быстро надоест. По этой причине оператор «точка» позволяет обращаться к полям по *указателям* на структуры точно так же, как вы обращаетесь к полям напрямую по значениям структур. Круглые скобки и оператор * не обязательны.

```
func main() {
       var value myStruct
       value.myField = 3
       var pointer *myStruct = &value
       fmt.Println(pointer.myField)
}
```

Обращение к полю структуры по указателю.

```
3
```

Этот способ также работает для присваивания значений полям структур по указателю:

```
func main() {
       var value myStruct
       var pointer *myStruct = &value
       pointer.myField = 9
       fmt.Println(pointer.myField)
}
```

Присваивание полю структуры по указателю.

```
9
```

Вот так функция `applyDiscount` может обновлять поле структуры без помощи оператора `*`. Происходит присваивание полю структуры по указателю.

```
func applyDiscount(s *subscriber) {
       s.rate = 4.99
}

func main() {
       var s subscriber
       applyDiscount(&s)
       fmt.Println(s.rate)
}
```

Присваивание полю структуры по указателю.

```
4.99
```

Часто Задаваемые Вопросы

В: Ранее вы уже приводили функцию `defaultSubscriber`, которая присваивает значения полям структуры, и указатели для этого не понадобились! Почему?

О: Функция `defaultSubscriber` ***возвращала*** значение структуры. Если вызывающая сторона сохраняет возвращенное значение, то значения в его полях останутся. Только функции, которые ***изменяют существующие*** структуры, не возвращая их, должны использовать указатели, чтобы эти изменения не были потеряны при выходе из функции.

Однако при желании функция `defaultSubscriber` ***могла бы*** возвращать указатель на структуру. Кстати, это изменение будет внесено в следующем разделе.

Передача больших структур с помощью указателей

Итак, параметры получают копию аргументов при вызове функции даже для структур... Но если вы передаете большую структуру с большим количеством полей, разве это не приведет к большим затратам памяти?

Да, приведет. В памяти будет храниться как исходная структура, так и копия.

Функции получают копию аргументов, с которыми они вызываются, даже если занимают много места (например, структуры).

Следовательно, если ваша структура не ограничивается парой небольших полей, часто бывает лучше передать функции *указатель* на структуру вместо нее самой (даже в том случае, если функции не нужно изменять структуру). При передаче указателя на структуру в памяти существует всего один экземпляр исходной структуры. Функция просто получает адрес памяти этого единственного экземпляра, а затем может читать данные структуры, изменять их или делать что-то еще, и все это без создания дополнительной копии.

Ниже приведена наша функция `defaultSubscriber`, измененная для возвращения указателя, и функция `printInfo`, измененная для получения указателя. Ни одна из этих функций не изменяет существующую структуру, как это делает `applyDiscount`. Однако указатели гарантируют, что в памяти будет храниться только одна копия каждой структуры, а программа будет нормально работать.

```
// ...
type subscriber struct {
        name   string
        rate   float64
        active bool
}

func printInfo(s *subscriber) {
        fmt.Println("Name:", s.name)
        fmt.Println("Monthly rate:", s.rate)
        fmt.Println("Active?", s.active)
}

func defaultSubscriber(name string) *subscriber {
        var s subscriber
        s.name = name
        s.rate = 5.99
        s.active = true
        return &s
}

func applyDiscount(s *subscriber) {
        s.rate = 4.99
}

func main() {
        subscriber1 := defaultSubscriber("Aman Singh")
        applyDiscount(subscriber1)
        printInfo(subscriber1)
        subscriber2 := defaultSubscriber("Beth Ryan")
        printInfo(subscriber2)
}
```

Изменяется для передачи указателя.

Изменяется для возвращения указателя.

Возвращает указатель на структуру вместо самой структуры.

Это уже не структура, а указатель на структуру...

Так как это структура, оператор & не нужен.

```
Name: Aman Singh
Monthly rate: 4.99
Active? true
Name: Beth Ryan
Monthly rate: 5.99
Active? true
```

Упражнение

Следующие две программы работают неправильно. Функция `nitroBoost` в программе слева должна увеличивать скорость машины на 50 км/ч, но не делает этого. А функция `doublePack` в программе справа должна удваивать поле `count` значения `part`, но и этого не происходит.

Удастся ли вам исправить программы? Для этого достаточно минимальных изменений; мы оставили немного свободного места в коде, чтобы вы могли внести необходимые обновления.

```
package main

import "fmt"

type car struct {
      name      string
      topSpeed  float64
}

func nitroBoost( c  car ) {
      c.topSpeed += 50
}

func main() {
      var mustang car
      mustang.name = "Mustang Cobra"
      mustang.topSpeed = 225
      nitroBoost( mustang )
      fmt.Println( mustang.name )
      fmt.Println( mustang.topSpeed )
}
```

Должно быть на 50 км/ч выше! →

```
Mustang Cobra
225
```

```
package main

import "fmt"

type part struct {
      description  string
      count        int
}

func doublePack( p  part ) {
      p.count *= 2
}

func main() {
      var fuses part
      fuses.description = "Fuses"
      fuses.count = 5
      doublePack( fuses )
      fmt.Println( fuses.description )
      fmt.Println( fuses.count )
}
```

Должно было удвоиться! →

```
Fuses
5
```

Ответ на с. 297.

Перемещение типа структуры в другой пакет

Да, мы начинаем понимать, насколько удобен тип структуры `subscriber`. Но код в пакете `main` становится слишком длинным. Можно ли вынести определение `subscriber` в другой пакет?

Сделать это несложно. Найдите каталог *headfirstgo* в своей рабочей области Go и создайте в нем новый каталог для хранения пакета с именем `magazine`. В каталоге *magazine* создайте файл с именем *magazine.go*.

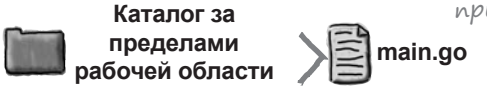
Не забудьте добавить объявление пакета `magazine` в начало файла *magazine.go*. Затем скопируйте определение структуры `subscriber` из существующего кода и вставьте его в *magazine.go*.

Попробуем вставить сюда определение типа без внесения каких-либо изменений.

```
package magazine

type subscriber struct {
        name   string
        rate   float64
        active bool
}
```

Теперь создадим программу, чтобы опробовать новый пакет. Так как сейчас мы просто экспериментируем, не обязательно создавать для этого кода отдельный каталог пакета — мы просто запустим его командой `go run`. Создайте файл с именем *main.go*. Его можно сохранить в любом каталоге, но проследите за тем, чтобы он был сохранен *за пределами* рабочей области Go, чтобы избежать возможных конфликтов с другими пакетами.

(Позднее этот код можно переместить в рабочую область Go — при условии, что для него будет создан отдельный каталог пакета.)



Этот код должен быть сохранен в файле *main.go*. Он просто создает новую структуру `subscriber` и обращается к одному из ее полей.

Мы видим два отличия от предыдущих примеров. Во-первых, пакет `magazine` должен импортироваться в начале файла. Во-вторых, имя типа должно записываться в форме `magazine.subscriber`, потому что тип теперь принадлежит другому пакету.

```
package main

import (
        "fmt"
        "github.com/headfirstgo/magazine"
)

func main() {
        var s magazine.subscriber
        s.rate = 4.99
        fmt.Println(s.rate)
}
```

Импортируем необходимые пакеты...

...включая новый пакет «magazine».

Имя типа теперь должно иметь префикс с именем пакета.

Экспорт определяемых типов

Посмотрим, сможет ли наш экспериментальный код обратиться к типу структуры `subscriber` из нового пакета. В терминале перейдите в каталог, в котором был сохранен файл *main.go*, после чего введите команду **`go run main.go`**.

```
Shell Edit View Window Help
$ cd temp
$ go run main.go
./main.go:9:18: cannot refer to unexported name magazine.subscriber
./main.go:9:18: undefined: magazine.subscriber
```

Компилятор выдает пару сообщений об ошибках, но важно только сообщение о невозможности обращения к неэкспортированному имени `magazine.subscriber`.

Имена типов Go записываются по тем же правилам, что и имена переменных и функций: если имя переменной, функции или типа начинается с буквы верхнего регистра, такое имя считается *экспортированным*, и к нему можно обращаться за пределами пакета, в котором оно было объявлено. Но имя нашего типа `subscriber` начинается с буквы нижнего регистра. Это означает, что оно может использоваться только в пределах пакета `magazine`.

Вроде бы проблема легко решается. Просто откройте файл *magazine.go* и замените первую букву в имени определяемого типа. Откройте файл *main.go* и измените регистр во всех упоминаниях этого типа (пока всего в одном месте).

Чтобы к типу можно было обращаться за пределами пакета, в котором он был определен, этот тип должен экспортироваться: для этого его имя должно начинаться с буквы верхнего регистра.

magazine.go

```
package magazine

type Subscriber struct {
        name   string
        rate   float64
        active bool
}
```

Имя типа начинается с буквы верхнего регистра.

main.go

```
package main

import (
        "fmt"
        "github.com/headfirstgo/magazine"
)

func main() {
        var s magazine.Subscriber
        s.rate = 4.99
        fmt.Println(s.rate)
}
```

Имя типа начинается с буквы верхнего регистра.

Если вы попытаетесь запустить обновленный код командой `go run main.go`, ошибка об отсутствии экспортированного типа `magazine.subscriber` исчезает. Казалось бы, проблема решена. Но внезапно вместо старой ошибки появляется пара новых...

```
Shell Edit View Window Help
$ go run main.go
./main.go:10:13: s.rate undefined
(cannot refer to unexported field or method rate)
./main.go:11:25: s.rate undefined
(cannot refer to unexported field or method rate)
```

Экспорт полей структур

После того как имя типа `Subscriber` будет записано с буквы верхнего регистра, к нему можно будет обращаться из пакета `main`. Но теперь мы получаем ошибку с сообщением о том, что программе не удается обратиться к *полю* `rate`, потому что *это поле* не экспортируется.

```
Shell Edit View Window Help
$ go run main.go
./main.go:10:13: s.rate undefined
(cannot refer to unexported field or method rate)
./main.go:11:25: s.rate undefined
(cannot refer to unexported field or method rate)
```

Даже если тип структуры экспортируется из пакета, его поля *не экспортируются*, если их имена не начинаются с буквы верхнего регистра. Попробуем перейти на имя `Rate` (в *magazine.go* и *main.go*)...

Чтобы имена полей структур экспортировались из своих пакетов, они тоже должны записываться с буквы верхнего регистра.

magazine.go

```
package magazine

type Subscriber struct {
        name   string
Rate    float64
        active bool
}
```

Верхний регистр.

main.go

```
package main

import (
        "fmt"
        "github.com/headfirstgo/magazine"
)

func main() {
        var s magazine.Subscriber
        s.Rate = 4.99
        fmt.Println(s.Rate)
}
```

Верхний регистр.

Верхний регистр.

Снова запустите *main.go* — вы увидите, что теперь все работает нормально. С экспортированием имен мы можем обращаться к типу `Subscriber` *и* его полю `Rate` из пакета `main`.

```
Shell Edit View Window Help
$ go run main.go
4.99
```

Обратите внимание: код работает даже при том, что поля `name` и `active` не экспортируются. Вы можете смешивать экспортируемые и неэкспортируемые поля в одном типе структуры.

Впрочем, в случае с типом `Subscriber` так поступать, пожалуй, не стоит. Странно разрешать другим пакетам доступ к стоимости подписки, но не к имени или адресу. Вернемся к файлу *magazine.go* и экспортируем другие поля. Просто замените первую букву в их именах: `Name` и `Active`.

magazine.go

```
package magazine

type Subscriber struct {
Name    string
Rate    float64
Active  bool
}
```

Верхний регистр.

Верхний регистр.

Литералы структур

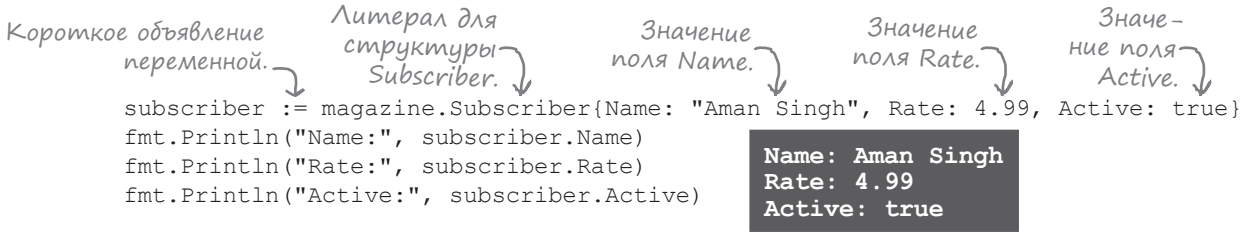
Код определения структуры и последующего присваивания значений ее полям быстро надоедает:

```
var subscriber magazine.Subscriber
subscriber.Name = "Aman Singh"
subscriber.Rate = 4.99
subscriber.Active = true
```

Как и в случае с сегментами и картами, Go предоставляет **литералы структур** для создания структур одновременно с инициализацией их полей.

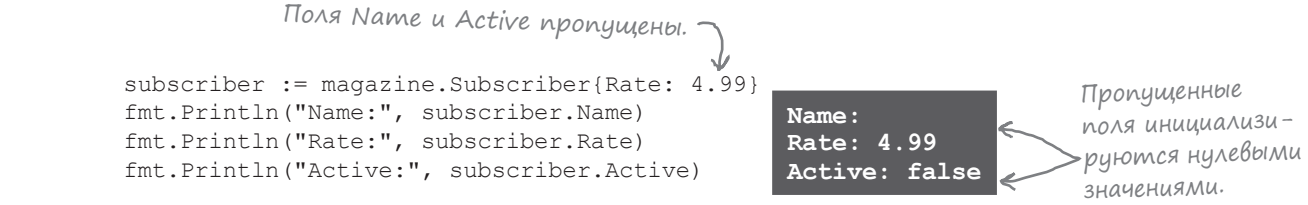
Синтаксис имеет много общего с литералами карт. Сначала указывается тип, за ним идут фигурные скобки. В фигурных скобках можно задать значения полей структуры (всех или некоторых); для каждого поля указывается имя, двоеточие и значение. Если вы указываете несколько полей, разделите их запятыми.

Выше был приведен код, который создает структуру `Subscriber` и заполняет ее отдельные поля. Этот код делает то же самое, что и следующий литерал структуры, состоящий всего из одной строки:



Возможно, вы заметили, что в этой главе нам в основном приходилось использовать длинные объявления для переменных структур (если только структура не возвращалась функцией). Литералы структур позволяют использовать короткие объявления переменных для только что созданных структур.

Некоторые (и даже все) поля не нужно указывать в фигурных скобках. Пропущенные поля инициализируются нулевыми значениями для своих типов.



У бассейна

Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты необязательно. Ваша **задача**: создать программу, которая работает и выводит показанный результат.

```
package main

import (
        "fmt"
        "geo"
)

func main() {
        location := geo._____{_____: 37.42, _____: -122.08}
        fmt.Println("Latitude:", location.Latitude)
        fmt.Println("Longitude:", location.Longitude)
}
```

main.go

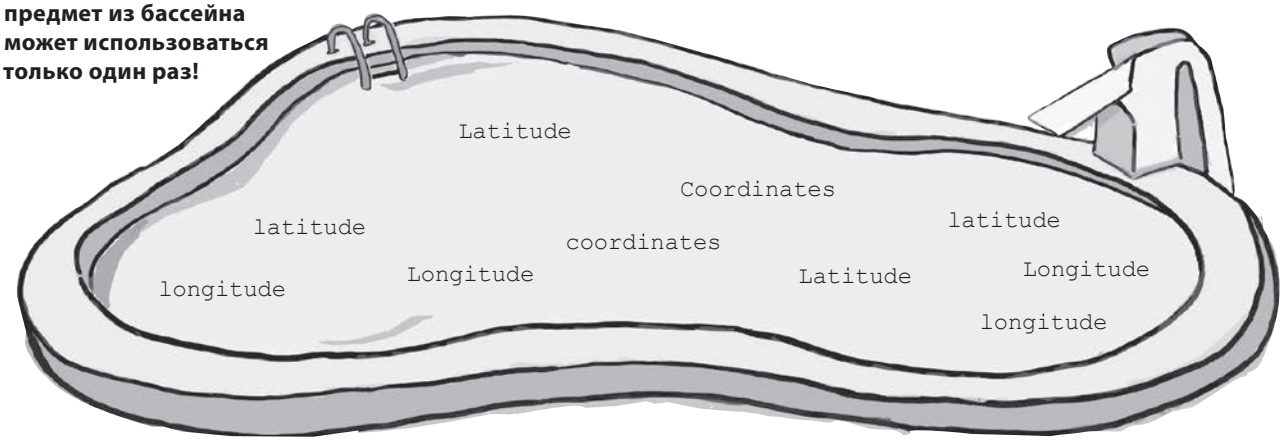
```
package geo

type Coordinates struct {
        _____ float64
        _____ float64
}
```

geo.go

Результат.

```
Latitude: 37.42
Longitude: -122.08
```

Примечание: каждый предмет из бассейна может использоваться только один раз!



Ответ на с. 298.

Создание типа структуры Employee

Новый пакет `magazine` работает просто замечательно! Еще пара мелких исправлений перед публикацией... Нам понадобится тип структуры `Employee` для отслеживания имен и зарплат наших работников. А еще нужно хранить адреса — как работников, так и подписчиков.

Создать тип структуры `Employee` будет несложно. Просто добавим его в пакет `magazine` вместе с типом `Subscriber`. В *magazine.go* определите новый тип `Employee` с базовым структурным типом. Этот тип должен содержать поле `Name` типа `string` и поле `Salary` типа `float64`. Помните, что имя типа *и* имена всех полей должны начинаться с буквы верхнего регистра, чтобы они экспортировались из пакета `magazine`.

Мы можем обновить функцию `main` в *main.go*, чтобы опробовать новый тип. Для начала объявите переменную с типом `magazine.Employee`, а затем присвойте значения соответствующих типов всем полям. Наконец, выведите значения полей.

magazine.go

```
package magazine

type Subscriber struct {
        Name   string
        Rate   float64
        Active bool
}

type Employee struct {
        Name   string
        Salary float64
}
```

Имя начинается с буквы верхнего регистра, чтобы оно экспортировалось.

Имена полей тоже должны экспортироваться.

main.go

```
package main

import (
        "fmt"
        "github.com/headfirstgo/magazine"
)

func main() {
        var employee magazine.Employee
        employee.Name = "Joy Carr"
        employee.Salary = 60000
        fmt.Println(employee.Name)
        fmt.Println(employee.Salary)
}
```

Пытаемся создать значение Employee.

Выполните команду `go run main.go` в терминале. Программа запускается, создает новую структуру `magazine.Employee`, инициализирует ее поля, а затем выводит их значения.

```
Joy Carr
60000
```

Создание типа структуры Address

Следующая задача — хранение информации адресов для типов `Subscriber` и `Employee`. Полный адрес должен содержать поля для улицы и номера дома, города, штата и почтового индекса.

Мы *можем* добавить отдельные поля для типов `Subscriber` и `Employee`:

```
type Subscriber struct {
      Name   string
      Rate   float64
      Active bool
      Street     string
      City       string
      State      string
      PostalCode string
}
```

Если мы добавим здесь поля...

```
type Employee struct {
      Name   string
      Salary float64
      Street     string
      City       string
      State      string
      PostalCode string
}
```

...их придется повторять здесь...

Но почтовые адреса должны иметь постоянный формат независимо от того, к какому типу принадлежат. Было бы утомительно повторять все эти поля для разных типов.

Поля структур могут содержать значения любого типа, *включая другие структуры*. Попробуем поступить иначе — построить тип структуры `Address`, а затем добавить поле `Address` в типы `Subscriber` и `Employee`. Такое решение сэкономит сейчас немного времени и усилий, а также поможет согласовать изменения в типах, если в будущем потребуется изменить формат адреса.

Начнем с создания типа `Address`, чтобы убедиться в правильности его работы. Включите код в пакет `magazine` вместе с типами `Subscriber` и `Employee`. Затем замените код в *main.go* несколькими строками, в которых создается значение `Address` и проверяется доступность его полей.

magazine.go

```
package magazine

// Код Subscriber
// и Employee пропущен...

type Address struct {
      Street     string
      City       string
      State      string
      PostalCode string
}
```

Здесь добавляется новый тип.

main.go

```
package main

import (
      "fmt"
      "github.com/headfirstgo/magazine"
)

func main() {
      var address magazine.Address
      address.Street = "123 Oak St"
      address.City = "Omaha"
      address.State = "NE"
      address.PostalCode = "68111"
      fmt.Println(address)
}
```

Пытаемся создать значение Address.

Введите команду **`go run main.go`** в терминале. Команда должна создать структуру `Address`, заполнить ее поля, а затем вывести всю структуру.

```
{123 Oak St Omaha NE 68111}
```

Добавление структуры как поля в другой тип

Итак, мы знаем, что тип структуры `Address` работает сам по себе. Добавим поля `HomeAddress` в типы `Subscriber` и `Employee`.

Добавить в структуру новое поле, которое само по себе является структурой, не сложнее, чем добавить поле любого другого типа. Укажите имя поля, за которым идет тип поля (которым в данном случае будет тип структуры).

Добавьте поле с именем `HomeAddress` в структуру `Subscriber`. Проследите за тем, чтобы имя поля начиналось с буквы верхнего регистра — так оно было доступно за пределами пакета `magazine`. Затем укажите тип поля, то есть `Address`.

Также добавьте поле `HomeAddress` в тип `Employee`.

magazine.go

```
package magazine

type Subscriber struct {
        Name        string
        Rate        float64
        Active      bool
        HomeAddress Address
}
type Employee struct {
        Name        string
        Salary      float64
        HomeAddress Address
}
type Address struct {
        // ...
}
```

Имя поля начинается с символа верхнего регистра. → `HomeAddress Address`

Тип поля. → `Address`

Имя поля начинается с символа верхнего регистра. → `HomeAddress Address`

Тип поля. → `Address`

Создание вложенной структуры

Посмотрим, как заполнить поля структуры `Address` *внутри* структуры `Subscriber`. Это можно сделать двумя способами.

Во-первых, можно создать отдельную структуру `Address`, а затем использовать ее для заполнения всего поля `Address` структуры `Subscriber`. Ниже приведена обновленная версия *main.go*, в которой реализован этот способ.

main.go

```
package main

import (
        "fmt"
        "github.com/headfirstgo/magazine"
)

func main() {
        address := magazine.Address{Street: "123 Oak St",
                City: "Omaha", State: "NE", PostalCode: "68111"}
        subscriber := magazine.Subscriber{Name: "Aman Singh"}
        subscriber.HomeAddress = address
        fmt.Println(subscriber.HomeAddress)
}
```

Создание значения Address с заполнением полей. → `address := magazine.Address{...}`

Создание структуры Subscriber, которой будет принадлежать Address. → `subscriber := magazine.Subscriber{...}`

Инициализирует поле HomeAddress. → `subscriber.HomeAddress = address`

Выводит поле HomeAddress. → `fmt.Println(subscriber.HomeAddress)`

Введите команду **`go run main.go`** в терминале. Вы увидите, что поле `HomeAddress` в данных подписчика заполнено созданной вами структурой.

```
{123 Oak St Omaha NE 68111}
```

Создание вложенной структуры (продолжение)

Другое решение основано на присваивании значений полей внутренней структуры *через* внешнюю структуру. При создании структуры `Subscriber` ее поле `HomeAddress` уже задано: оно содержит структуру `Address`, все поля которой заполнены нулевыми значениями. Если снова вывести `HomeAddress`, передав `fmt.Printf` глагол `"%#v"`, функция выведет структуру в том виде, в котором она записывается в коде Go — то есть в виде литерала структуры. Мы видим, что каждое из полей `Address` инициализируется пустой строкой (нулевое значение для строкового типа).

```
subscriber := magazine.Subscriber{}
fmt.Printf("%#v\n", subscriber.HomeAddress)
```

Каждое из полей Address инициализируется пустой строкой (нулевое значение для строкового типа).

Поле настроено как новая структура Address.

```
magazine.Address{Street:"", City:"", State:"", PostalCode:""}
```

Если `subscriber` — переменная, содержащая структуру `Subscriber`, то конструкция `subscriber.HomeAddress` дает вам структуру `Address`, хотя вы и не задали `HomeAddress` явно.

Этот факт позволяет использовать «сцепленные» операторы «точка» для обращения к полям структуры `Address`. Введите выражение **`subscriber.HomeAddress`**, чтобы обратиться к структуре `Address`, а затем *другой* оператор «точка» и имя поля структуры `Address`, к которому хотите обратиться.

```
subscriber.HomeAddress.City
```

Эта часть дает структуру Address.

Эта часть обращается к полю City структуры Address.

Этот синтаксис работает как для присваивания значений полям внутренней структуры...

```
subscriber.HomeAddress.PostalCode = "68111"
```

...так и для последующего чтения этих значений.

```
fmt.Println("Postal Code:", subscriber.HomeAddress.PostalCode)
```

Создание вложенной структуры (продолжение)

Ниже приведена обновленная версия *main.go*, где используются сцепленные операторы «точка». Сначала структура `Subscriber` сохраняется в переменной `subscriber`. При этом в поле `HomeAddress` переменной `subscriber` автоматически создается структура `Address`. Мы задаем значения полей `subscriber.HomeAddress.Street`, `subscriber.HomeAddress.City` и т. д., а затем снова выводим эти значения.

Структура `Employee` сохраняется в переменной `employee`, после чего то же самое происходит с ее структурой `HomeAddress`.

main.go

```
package main

import (
        "fmt"
        "github.com/headfirstgo/magazine"
)

func main() {
        subscriber := magazine.Subscriber{Name: "Aman Singh"}
        subscriber.HomeAddress.Street = "123 Oak St"
        subscriber.HomeAddress.City = "Omaha"
        subscriber.HomeAddress.State = "NE"
        subscriber.HomeAddress.PostalCode = "68111"
        fmt.Println("Subscriber Name:", subscriber.Name)
        fmt.Println("Street:", subscriber.HomeAddress.Street)
        fmt.Println("City:", subscriber.HomeAddress.City)
        fmt.Println("State:", subscriber.HomeAddress.State)
        fmt.Println("Postal Code:", subscriber.HomeAddress.PostalCode)

        employee := magazine.Employee{Name: "Joy Carr"}
        employee.HomeAddress.Street = "456 Elm St"
        employee.HomeAddress.City = "Portland"
        employee.HomeAddress.State = "OR"
        employee.HomeAddress.PostalCode = "97222"
        fmt.Println("Employee Name:", employee.Name)
        fmt.Println("Street:", employee.HomeAddress.Street)
        fmt.Println("City:", employee.HomeAddress.City)
        fmt.Println("State:", employee.HomeAddress.State)
        fmt.Println("Postal Code:", employee.HomeAddress.PostalCode)
}
```

Задаем значения полей subscriber.HomeAddress.

Получаем значения полей из subscriber.HomeAddress.

Задаем поля employee.HomeAddress.

Получаем значения полей из employee.HomeAddress.

Введите команду **`go run main.go`** в терминале. Программа выводит заполненные поля `subscriber.HomeAddress` и `employee.HomeAddress`.

```
Subscriber Name: Aman Singh
Street: 123 Oak St
City: Omaha
State: NE
Postal Code: 68111
Employee Name: Joy Carr
Street: 456 Elm St
City: Portland
State: OR
Postal Code: 97222
```

Анонимные поля структур

Код обращения к полям внутренней структуры через внешнюю структуру получается довольно однообразным. Имя поля внутренней структуры (`HomeAddress`) нужно указывать каждый раз, когда вы хотите обратиться к содержащимся в ней полям.

```
subscriber := magazine.Subscriber{Name: "Aman Singh"}
subscriber.HomeAddress.Street = "123 Oak St"
subscriber.HomeAddress.City = "Omaha"
subscriber.HomeAddress.State = "NE"
subscriber.HomeAddress.PostalCode = "68111"
```

Сначала указывается имя поля с внутренней структурой...

...и только после этого вы сможете обращаться к ее полям.

Go позволяет определять анонимные поля: поля структур, которые не имеют собственного имени, а обладают только типом. Анонимные поля упрощают доступ к внутренней структуре.

Перед вами обновленные версии типов `Subscriber` и `Employee`, в которых поля `HomeAddress` преобразованы в анонимные поля. Для этого достаточно удалить имя поля, оставив только тип.

magazine.go

```
package magazine

type Subscriber struct {
        Name   string
        Rate   float64
        Active bool
        Address
}

type Employee struct {
        Name   string
        Salary float64
        Address
}

type Address struct {
        // Поля пропущены
}
```

Удаляем имя поля («HomeAddress»), оставляя только тип.

Удаляем имя поля («HomeAddress»), оставляя только тип.

При объявлении анонимного поля вы можете использовать имя типа поля так, как если бы оно было именем поля. Таким образом, выражения `subscriber.Address` и `employee.Address` в приведенном ниже коде обозначают структуру `Address`:

```
subscriber := magazine.Subscriber{Name: "Aman Singh"}
subscriber.Address.Street = "123 Oak St"
subscriber.Address.City = "Omaha"
fmt.Println("Street:", subscriber.Address.Street)
fmt.Println("City:", subscriber.Address.City)
employee := magazine.Employee{Name: "Joy Carr"}
employee.Address.State = "OR"
employee.Address.PostalCode = "97222"
fmt.Println("State:", employee.Address.State)
fmt.Println("Postal Code:", employee.Address.PostalCode)
```

Обращаемся к полю внутренней структуры по его новому «имени», то есть «Address».

```
Street: 123 Oak St
City: Omaha
State: OR
Postal Code: 97222
```

Встроенные структуры

Польза анонимных полей не ограничивается экономией на вводе имени поля в определении структуры. Внутренняя структура, хранящаяся во внешней структуре с обращением через анонимное поле, называется **встроенной** во внешнюю структуру. Поля встроенной структуры **повышаются** до уровня внешней структуры; то есть к ним можно обращаться так, будто они принадлежат внешней структуре.

Итак, когда тип структуры `Address` встроен в типы структур `Subscriber` и `Employee`, вам не придется использовать запись `subscriber.Address.City` для обращения к полю `City`, достаточно выражения `subscriber.City`. Не нужно использовать запись `employee.Address.State`; хватит и `employee.State`.

Ниже приведена последняя версия *main.go*, в которой структура `Address` реализована в виде встроенного типа. Код можно писать так, словно никакого типа `Address` не существует; все выглядит так, будто поля `Address` принадлежат типу структуры, в которую они встроены.

main.go

```
package main

import (
        "fmt"
        "github.com/headfirstgo/magazine"
)

func main() {
        subscriber := magazine.Subscriber{Name: "Aman Singh"}
        subscriber.Street = "123 Oak St"
        subscriber.City = "Omaha"
        subscriber.State = "NE"
        subscriber.PostalCode = "68111"
        fmt.Println("Street:", subscriber.Street)
        fmt.Println("City:", subscriber.City)
        fmt.Println("State:", subscriber.State)
        fmt.Println("Postal Code:", subscriber.PostalCode)

        employee := magazine.Employee{Name: "Joy Carr"}
        employee.Street = "456 Elm St"
        employee.City = "Portland"
        employee.State = "OR"
        employee.PostalCode = "97222"
        fmt.Println("Street:", employee.Street)
        fmt.Println("City:", employee.City)
        fmt.Println("State:", employee.State)
        fmt.Println("Postal Code:", employee.PostalCode)
}
```

Поля Address задаются так, как если бы они были определены в Subscriber.

Получение значений полей Address через Subscriber.

Поля Address задаются так, как если бы они были определены в Employee.

Получение значений полей Address через Employee.

```
Street: 123 Oak St
City: Omaha
State: NE
Postal Code: 68111
Street: 456 Elm St
City: Portland
State: OR
Postal Code: 97222
```

Помните, что вы не обязаны использовать встроенные внутренние структуры. Вы вообще не обязаны использовать внутренние структуры. Иногда добавление новых полей на уровне внешней структуры делает код более элегантным. Проанализируйте текущую ситуацию и выберите то решение, которое лучше подойдет вам и вашим пользователям.

Наши определяемые типы готовы!

Типы структур, которые вы написали для нас, просто великолепны! Теперь не нужно передавать кучу переменных, представляющих одного подписчика. Все необходимое хранится в одном удобном наборе. Спасибо, остается запустить печатный станок и напечатать первый номер!

Отличная работа! Мы определили структурные типы `Subscriber` и `Employee`, а также встроили в каждый из них структуру `Address`. Нам удалось найти способ представления всех данных, необходимых для журнала!

Впрочем, в определяемых типах все еще отсутствует один важный аспект. В предыдущих главах использовались такие типы, как `time.Time` или `strings.Replacer`, — эти типы обладали *методами*: функциями, которые можно вызывать *для* их значений. Однако вы еще пока не знаете, как определять методы для ваших собственных типов. Не беспокойтесь, эта тема будет рассмотрена в следующей главе!

Упражнение

Ниже приведен исходный файл пакета `geo` из предыдущего упражнения. Ваша задача — сделать так, чтобы код *main.go* работал правильно. Но тут есть небольшая загвоздка: задачу необходимо решить добавлением всего двух полей в тип структуры `Landmark` внутри *geo.go*.

geo.go

```
package geo

type Coordinates struct {
        Latitude  float64
        Longitude float64
}

type Landmark struct {
        _____
        _____
}
```

Здесь добавляются два поля!

main.go

```
package main

import (
        "fmt"
        "geo"
)

func main() {
        location := geo.Landmark{}
        location.Name = "The Googleplex"
        location.Latitude = 37.42
        location.Longitude = -122.08
        fmt.Println(location)
}
```

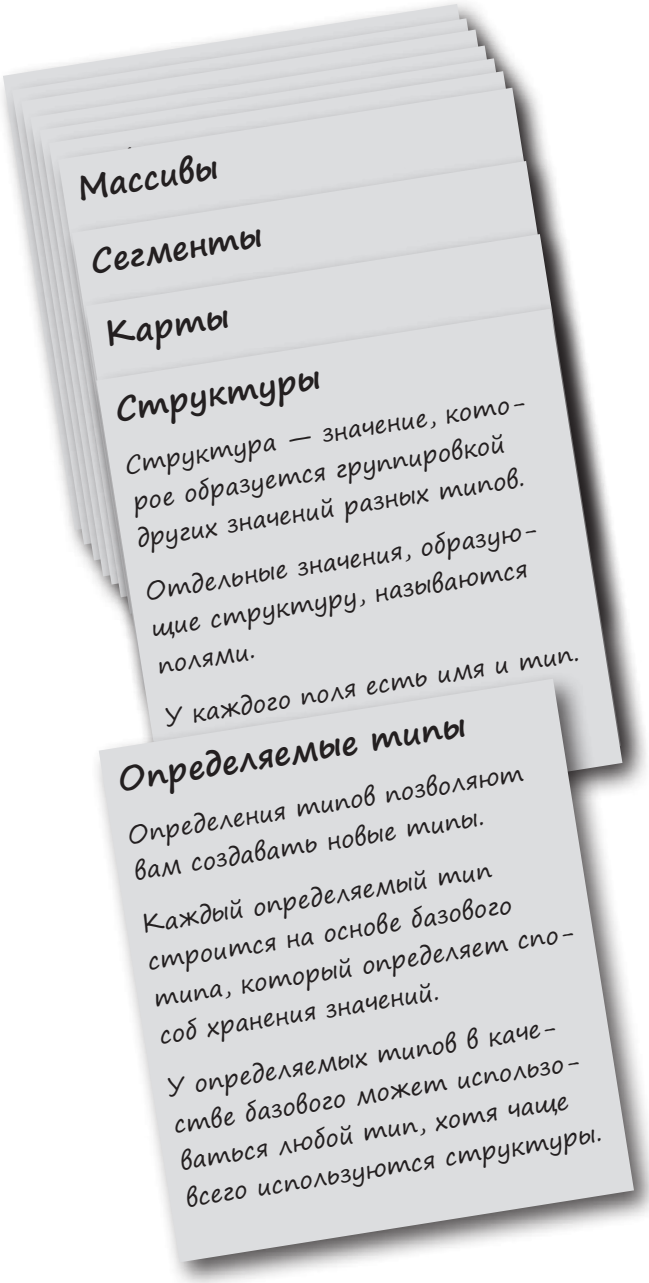
Результат.

```
{The Googleplex {37.42 -122.08}}
```

Ответ на с. 298.

Ваш инструментарий Go

Глава 8 осталась позади! В ней ваш инструментарий пополнился структурами и определяемыми типами.



КЛЮЧЕВЫЕ МОМЕНТЫ

- Переменную можно объявить с типом структуры. Чтобы задать тип структуры, используйте ключевое слово `struct` со списком имен полей и типов, заключенным в фигурные скобки.

```
var myStruct struct {
    field1 string
    field2 int
}
```

- Многократно записывать типы структур неудобно, поэтому обычно бывает удобнее определить тип с базовым типом структуры. После этого определяемый тип может использоваться для переменных, параметров функций, возвращаемых значений и т. д.

```
type myType struct {
    field1 string
}
var myVar myType
```

- Для обращения к полям структур используется оператор «точка».

```
myVar.field1 = "value"
fmt.Println(myVar.field1)
```

- Если структура должна изменяться внутри функции или занимает много памяти, следует передавать ее функции как указатель.
- Чтобы типы экспортировались из пакета, в котором они определяются, их имена должны начинаться с буквы верхнего регистра.
- Так же и поля структур доступны за пределами своего пакета только в том случае, если их имена начинаются с букв верхнего регистра.
- Литералы структур позволяют создать структуру одновременно с инициализацией ее полей.

```
myVar := myType{field1: "value"}
```

- При добавлении в структуру поля, у которого нет имени (а есть только тип), определяется анонимное поле.
- Внутренняя структура, добавляемая в составе внешней структуры в виде анонимного поля, называется **встроенной**.
- К полям встроенной структуры можно обращаться так, словно они принадлежат внешней структуре.

Упражнение
Решение

Справа приведена программа, в которой создается структурная переменная для хранения имени домашнего питомца (`string`) и его возраста (`int`). Заполните пустые места в коде, чтобы он выводил показанный результат.

```
package main

import "fmt"

func main() {
        var pet  struct  {
                name   string
                age    int
        }
        pet. name  = "Max"
        pet.age = 5
        fmt.Println("Name:", pet .name)
        fmt.Println("Age:", pet. age )
}
```

```
Name: Max
Age: 5
```

Развлечения с магнитами. Решение

```
package main

import "fmt"

type student struct {
        name string
        grade float64
}

func printInfo( s student ) {
        fmt.Println("Name:", s.name)
        fmt.Printf("Grade: %0.1f\n", s.grade)
}

func main() {
        var s student
        s.name = "Alonzo Cole"
        s.grade = 92.3
        printInfo(s)
}
```

Определение типа структуры «student».

Определение функции, которая получает структуру «student» в параметре.

Функции передается структура.

Результат.

```
Name: Alonzo Cole
Grade: 92.3
```

Упражнение
Решение

Следующие две программы работают неправильно. Функция `nitroBoost` в программе слева должна увеличивать скорость машины на 50 км/ч, но не делает этого. А функция `doublePack` в программе справа должна удваивать поле `count` значения `part`, но и этого не происходит. В обеих программах было достаточно изменить функции, чтобы они получали указатели, и обновить вызовы функций для получения указателей. Код внутри функций, обновляющий поля структур, изменять не нужно; код обращения к полю по указателю на структуру идентичен коду с прямым обращением к полю структуры.

```
package main

import "fmt"

type car struct {
       name      string
       topSpeed  float64
}

func nitroBoost( c *car ) {
       c.topSpeed += 50
}

func main() {
       var mustang car
       mustang.name = "Mustang Cobra"
       mustang.topSpeed = 225
       nitroBoost(&mustang )
       fmt.Println( mustang.name )
       fmt.Println( mustang.topSpeed )
}
```

Получает указатель на структуру вместо структуры.

Ничего изменять не придется; работает как с указателем, так и с самой структурой.

Передается указатель.

Исправлено; увеличивается на 50 км/ч.

```
Mustang Cobra
275
```

```
package main

import "fmt"

type part struct {
       description  string
       count        int
}

func doublePack( p *part ) {
       p.count *= 2
}

func main() {
       var fuses part
       fuses.description = "Fuses"
       fuses.count = 5
       doublePack(&fuses )
       fmt.Println( fuses.description )
       fmt.Println( fuses.count )
}
```

Получает указатель на структуру вместо структуры.

Ничего изменять не придется; работает как с указателем, так и с самой структурой.

Передается указатель.

Исправлено; исходное значение удвоилось.

```
Fuses
10
```

У бассейна. Решение

```
package main

import (
        "fmt"
        "geo"
)

func main() {
        location := geo.Coordinates{Latitude: 37.42, Longitude: -122.08}
        fmt.Println("Latitude:", location.Latitude)
        fmt.Println("Longitude:", location.Longitude)
}
```

Типы имен должны начинаться с буквы верхнего регистра.

Имена полей тоже должны начинаться с буквы верхнего регистра.

main.go

```
package geo

type Coordinates struct {
        Latitude  float64
        Longitude float64
}
```

geo.go

Результат.

```
Latitude: 37.42
Longitude: -122.08
```

Упражнение
Решение

Ниже приведен исходный файл пакета `geo` из предыдущего упражнения. Ваша задача — сделать так, чтобы код *main.go* работал правильно. Но тут есть небольшая загвоздка: задачу необходимо решить добавлением всего двух полей в тип структуры `Landmark` внутри *geo.go*.

```
package geo

type Coordinates struct {
        Latitude  float64
        Longitude float64
}

type Landmark struct {
        Name string
        Coordinates
}
```

geo.go

Структура Coordinates встраивается как анонимное поле, что позволяет вам обращаться к ее полям Latitude и Longitude так, как если бы они были определены в Landmark.

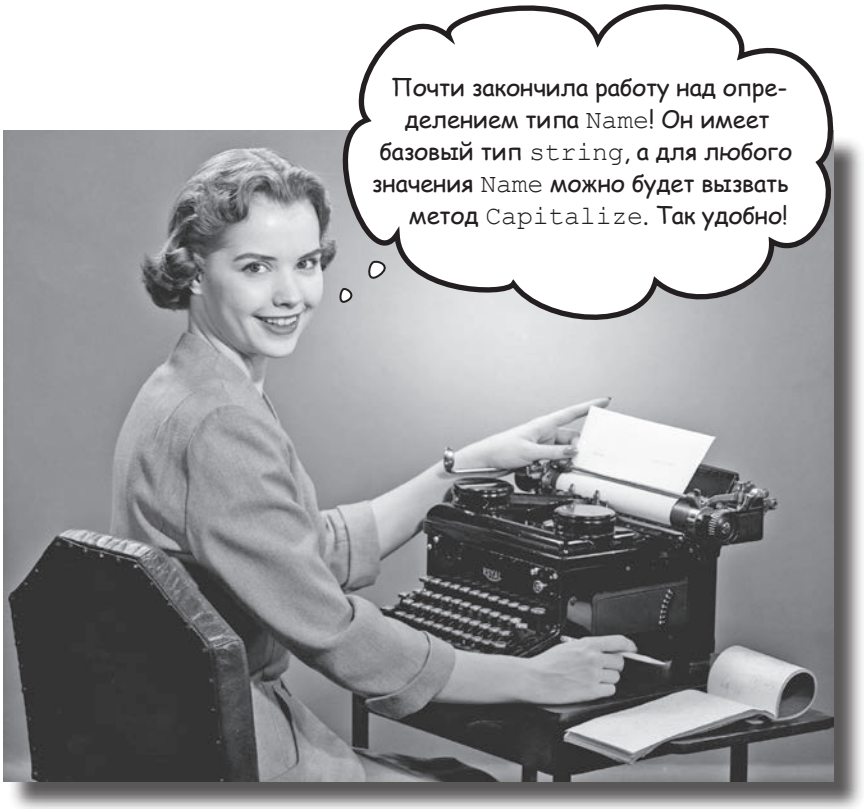
```
package main

import (
        "fmt"
        "geo"
)

func main() {
        location := geo.Landmark{}
        location.Name = "The Googleplex"
        location.Latitude = 37.42
        location.Longitude = -122.08
        fmt.Println(location)
}
```

main.go

Результат.

```
{The Googleplex {37.42 -122.08}}
```

Определяемые типы



Мы еще не все рассказали об определяемых типах. В предыдущей главе вы узнали, как определяются типы, у которых базовым типом является тип структуры. Но при этом мы *не сказали*, что в качестве базового может использоваться *любой* тип.

Помните о методах — особой разновидности функций, связываемой со значениями определенного типа? В книге неоднократно встречались примеры вызова методов для разных значений, но *собственные* методы вы еще не умеете определять. В этой главе мы исправим это. А теперь за дело!

Ошибки типов в реальной жизни

Жители США привыкли к экзотической системе мер, используемой в этой стране. Например, на заправках бензин продается в галлонах — эта единица объема почти вчетверо больше литра, принятого в большинстве стран мира.

Американец Стив взял напрокат машину в другой стране и заехал на заправку. Он собирается купить 10 галлонов — по его расчетам этого хватит, чтобы доехать до отеля в другом городе.

Стив проехал только четверть пути до места назначения, когда у него кончился бензин.

Если бы Стив повнимательнее присмотрелся к надписям на колонке, то понял бы, что объем измеряется в литрах, а не в галлонах, и для получения объема, эквивалентного 10 галлонам, следует купить 37,85 литров.

Даже если вам известно число единиц, стоит лишний раз проверить, что это число измеряет: литры или галлоны, килограммы или фунты, доллары или иены?

Определяемые типы с базовыми основными типами

Допустим, имеется следующая переменная:

```
var fuel float64 = 10
```

...Что она представляет — 10 галлонов или 10 литров? Это известно человеку, который написал это объявление, но никто другой этого не знает (по крайней мере наверняка).

Определяемые типы Go помогают четко обозначить, для чего должно использоваться значение. Хотя определяемые типы чаще всего используют структуры в качестве базовых типов, они также могут использовать `int`, `float64`, `string`, `bool` или любой другой тип.

Определяемые типы Go чаще всего используют структуры в качестве базовых типов, но вместо них также могут использоваться int, строки, логические значения или любой другой тип.

Следующая программа определяет два новых типа, `Liters` и `Gallons`, в обоих случаях в качестве базового используется тип `float64`. Так как эти типы определяются на уровне пакета, они доступны в любой функции текущего пакета.

Внутри функции `main` объявляется переменная с типом `Gallons` и другая переменная с типом `Liters`. Программа присваивает значения каждой переменной, а потом выводит их.

Определяются два новых типа, оба используют базовый тип float64.

```
package main

import "fmt"

type Liters float64
type Gallons float64

func main() {
        var carFuel Gallons
        var busFuel Liters
        carFuel = Gallons(10.0)
        busFuel = Liters(240.0)
        fmt.Println(carFuel, busFuel)
}
```

Определяется переменная с типом Gallons.

Определяется переменная с типом Liters.

Преобразует float64 в Gallons.

Преобразует float64 в Liters.

```
10 240
```

После того как тип будет определен, вы можете выполнить преобразование к этому типу от любого значения базового типа. Как и при любых других преобразованиях, сначала записывается тип, к которому приводится значение, а затем преобразуемое значение в круглых скобках.

При желании в коде можно было бы использовать короткие объявления переменных с преобразованиями типов:

Короткие объявления переменных используются с преобразованиями типов.

```
carFuel := Gallons(10.0)
busFuel := Liters(240.0)
```

Определяемые типы с базовыми основными типами (продолжение)

Если у вас имеется переменная, использующая определяемый тип, вы *не сможете* присвоить ей значение другого определяемого типа, даже если другой тип имеет такой же базовый тип. Это сделано для того, чтобы разработчики не путали два типа.

```
carFuel = Liters(240.0)
busFuel = Gallons(10.0)
```

Ошибки. →
```
cannot use Liters(240) (type Liters) as type Gallons in assignment
cannot use Gallons(10) (type Gallons) as type Liters in assignment
```

Тем не менее вы можете *выполнять преобразование* между типами, имеющими один базовый тип. Таким образом, значение `Liters` можно преобразовать в `Gallons` и наоборот, потому что оба типа имеют базовый тип `float64`. Но в Go при преобразовании рассматривается только значение базового типа; между `Gallons(Liters(240.0))` и `Gallons(240.0)` нет никаких различий. Прямое преобразование исходных значений от одного типа к другому нарушает защиту от ошибок преобразования, которая должна обеспечиваться типами.

```
carFuel = Gallons(Liters(40.0))   ← 40 литров НЕ РАВНЫ 40 галлонам!
busFuel = Liters(Gallons(63.0))   ← 63 галлона НЕ РАВНЫ 63 литрам!
fmt.Printf("Gallons: %0.1f Liters: %0.1f\n", carFuel, busFuel)
```

Работает, но неверно! →
```
Gallons: 40.0 Liters: 63.0
```

Вместо этого следует выполнить операции преобразования значения основополагающего типа в значение, соответствующее тому типу, к которому выполняется преобразование.

Быстрый поиск в интернете показывает, что один литр равен приблизительно 0,264 галлона, а один галлон равен приблизительно 3,785 литра. Умножая на эти коэффициенты, мы можем преобразовать `Gallons` в `Liters` и наоборот.

```
carFuel = Gallons(Liters(40.0) * 0.264)   ← Преобразование Liters в Gallons.
busFuel = Liters(Gallons(63.0) * 3.785)   ← Преобразование Gallons в Liters.
fmt.Printf("Gallons: %0.1f Liters: %0.1f\n", carFuel, busFuel)
```

Правильные значения. →
```
Gallons: 10.6 Liters: 238.5
```

Определяемые типы и операторы

Определяемый тип поддерживает все те же операции, что и базовый тип. Например, типы, основанные на `float64`, поддерживают арифметические операторы (такие, как +, -, * и /), а также операторы преобразования (==, >, < и т. д.).

```
fmt.Println(Liters(1.2) + Liters(3.4))
fmt.Println(Gallons(5.5) - Gallons(2.2))
fmt.Println(Liters(2.2) / Liters(1.1))
fmt.Println(Gallons(1.2) == Gallons(1.2))
fmt.Println(Liters(1.2) < Liters(3.4))
fmt.Println(Liters(1.2) > Liters(3.4))
```

```
4.6
3.3
2
true
true
false
```

С другой стороны, тип, основывающийся на базовом типе `string`, поддерживает операторы +, ==, > и <, но не -, потому что оператор — не поддерживается для строк.

```
// Директивы package и import пропущены
type Title string
```

Определение типа с базовым типом «string».

```
func main() {
    fmt.Println(Title("Alien") == Title("Alien"))
    fmt.Println(Title("Alien") < Title("Zodiac"))
    fmt.Println(Title("Alien") > Title("Zodiac"))
    fmt.Println(Title("Alien") + "s")
    fmt.Println(Title("Jaws 2") - " 2")
}
```

Эти операции работают...

А эта нет!

Ошибка.

```
invalid operation:
Title("Jaws 2") - " 2"
(operator - not defined
on string)
```

Определяемый тип может использоваться в операциях вместе с литералами:

```
fmt.Println(Liters(1.2) + 3.4)
fmt.Println(Gallons(5.5) - 2.2)
fmt.Println(Gallons(1.2) == 1.2)
fmt.Println(Liters(1.2) < 3.4)
```

```
4.6
3.3
true
true
```

Но определяемые типы *не могут* использоваться в операциях вместе со значениями другого типа, даже если другой тип имеет такой же базовый тип. Это также сделано для того, чтобы разработчики случайно не перепутали два типа.

```
fmt.Println(Liters(1.2) + Gallons(3.4))
fmt.Println(Gallons(1.2) == Liters(1.2))
```

Ошибки.

```
invalid operation: Liters(1.2) + Gallons(3.4)
(mismatched types Liters and Gallons)
invalid operation: Gallons(1.2) == Liters(1.2)
(mismatched types Gallons and Liters)
```

Если вы хотите сложить значение типа `Liters` со значением типа `Gallons`, сначала необходимо преобразовать один тип и привести его в соответствие с другим.

У бассейна

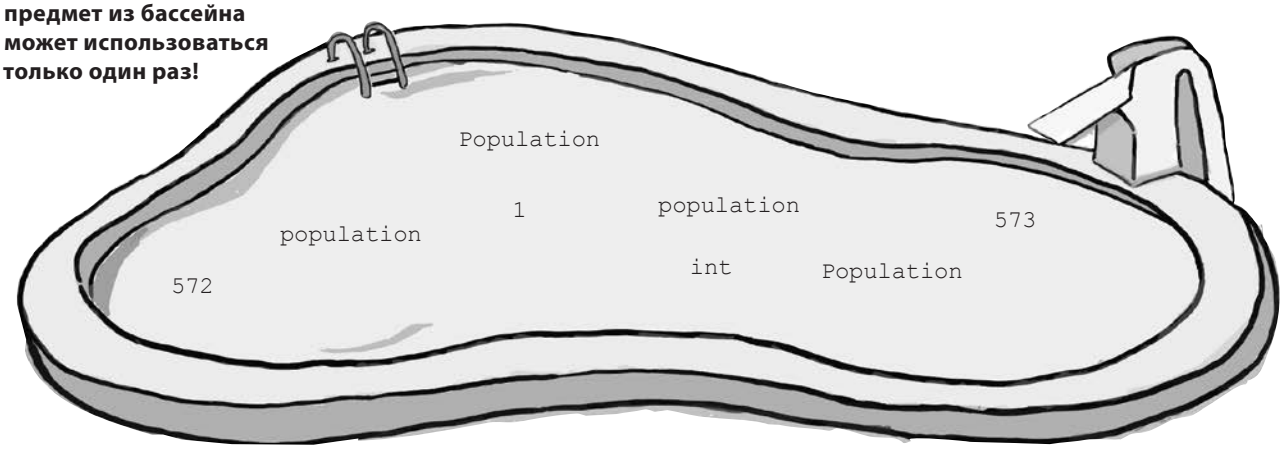
Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты необязательно. Ваша **задача**: создать программу, которая работает и выводит показанный результат.

```
package main

import "fmt"

type _____________ int

func main() {
       var _____________ Population
       population = _____________(_____)
       fmt.Println("Sleepy Creek County population:", population)
       fmt.Println("Congratulations, Kevin and Anna! It's a girl!")
       population += ___
       fmt.Println("Sleepy Creek County population:", population)
}
```

Результат.

```
Sleepy Creek County population: 572
Congratulations, Kevin and Anna! It's a girl!
Sleepy Creek County population: 573
```

Примечание: каждый предмет из бассейна может использоваться только один раз!



Ответ на с. 320.

Преобразования типов с помощью функций

Допустим, у вас есть машина, у которой уровень топлива измеряется в галлонах, и вы хотите заправить ее на колонке, измеряющей объем топлива в литрах. Или вам нужно заправить автобус, у которого уровень топлива измеряется в литрах, на колонке, измеряющей объем топлива в галлонах. Чтобы защититься от ошибок в вычислениях, Go выдает ошибку компиляции при попытке использовать в одной операции разнотипные значения:

```
package main

import "fmt"

type Liters float64
type Gallons float64

func main() {
      carFuel := Gallons(1.2)
      busFuel := Liters(2.5)
      carFuel += Liters(8.0)
      busFuel += Gallons(30.0)
}
```

Значение Liters нельзя сложить со значением Gallons!

Значение Gallons нельзя сложить со значением Liters!

Ошибки.

```
invalid operation: carFuel += Liters(8)
(mismatched types Gallons and Liters)
invalid operation: busFuel += Gallons(20)
(mismatched types Liters and Gallons)
```

Чтобы выполнить операции со значениями разных типов, сначала необходимо привести их к одному типу. Ранее мы показали, как значение `Liters` умножается на 0,264, а результат преобразуется в `Gallons`. Также значение типа `Gallons` умножается на 3,785, а результат преобразуется в `Liters`.

```
carFuel = Gallons(Liters(40.0) * 0.264)
busFuel = Liters(Gallons(63.0) * 3.785)
```

Преобразование Liters в Gallons.

Преобразование Gallons в Liters.

Мы можем создать функции `ToGallons` и `ToLiters`, которые делают то же самое, а затем вызвать их, чтобы они выполнили преобразование за нас:

```
// Imports, type declarations omitted
func ToGallons(l Liters) Gallons {
      return Gallons(l * 0.264)
}

func ToLiters(g Gallons) Liters {
      return Liters(g * 3.785)
}

func main() {
      carFuel := Gallons(1.2)
      busFuel := Liters(4.5)
      carFuel += ToGallons(Liters(40.0))
      busFuel += ToLiters(Gallons(30.0))
      fmt.Printf("Car fuel: %0.1f gallons\n", carFuel)
      fmt.Printf("Bus fuel: %0.1f liters\n", busFuel)
}
```

Значение типа Gallons чуть больше 1/4 от значения типа Liters.

Значение типа Liters почти вчетверо больше значения типа Gallons.

Liters преобразуется в Gallons перед сложением.

Gallons преобразуется в Liters перед сложением.

```
Car fuel: 11.8 gallons
Bus fuel: 118.1 liters
```

Преобразования типов с помощью функций (продолжение)

Бензин — не единственная жидкость, объем которой нам приходится измерять. Можно вспомнить еще растительное масло, газировку, соки и т. д. Также есть много других единиц измерения объема, кроме литров и галлонов. В США это чашки, кварты, чайные ложки и т. д. В метрической системе тоже существуют другие единицы измерения, но чаще всего используются миллилитры (1/1000 литра).

Добавим новый тип `Milliliters`. Как и другие типы, он использует `float64` в качестве базового типа.

Добавляется новый тип.

```
type Liters float64
type Milliliters float64
type Gallons float64
```

Также нам потребуется способ преобразования `Milliliters` в другие типы. Но если попытаться добавить функцию для преобразования `Milliliters` в `Gallons`, возникает проблема: в одном пакете не может быть двух функций с именем `ToGallons`!

```
func ToGallons(l Liters) Gallons {
      return Gallons(l * 0.264)
}
func ToGallons(m Milliliters) Gallons {
      return Gallons(m * 0.000264)
}
```

Мы не сможем добавить другую функцию для преобразования Milliliters в Gallons, если она имеет такое же имя!

Ошибка.

```
12:31: ToGallons redeclared in this block
      previous declaration at prog.go:9:26
```

Мы можем переименовать две функции `ToGallons` и включить в имена исходный тип: `LitersToGallons` и `MillilitersToGallons` соответственно. Но записывать такие имена будет неудобно, а с добавлением новых функций для преобразования между типами становится ясно, что этот путь ведет в тупик.

```
func LitersToGallons(l Liters) Gallons {
      return Gallons(l * 0.264)
}
func MillilitersToGallons(m Milliliters) Gallons {
      return Gallons(m * 0.000264)
}
func GallonsToLiters(g Gallons) Liters {
      return Liters(g * 3.785)
}
func GallonsToMilliliters(g Gallons) Milliliters {
      return Milliliters(g * 3785.41)
}
```

Устраняет конфликт, но имя получается очень длинным!

Устраняет конфликт, но имя получается очень длинным!

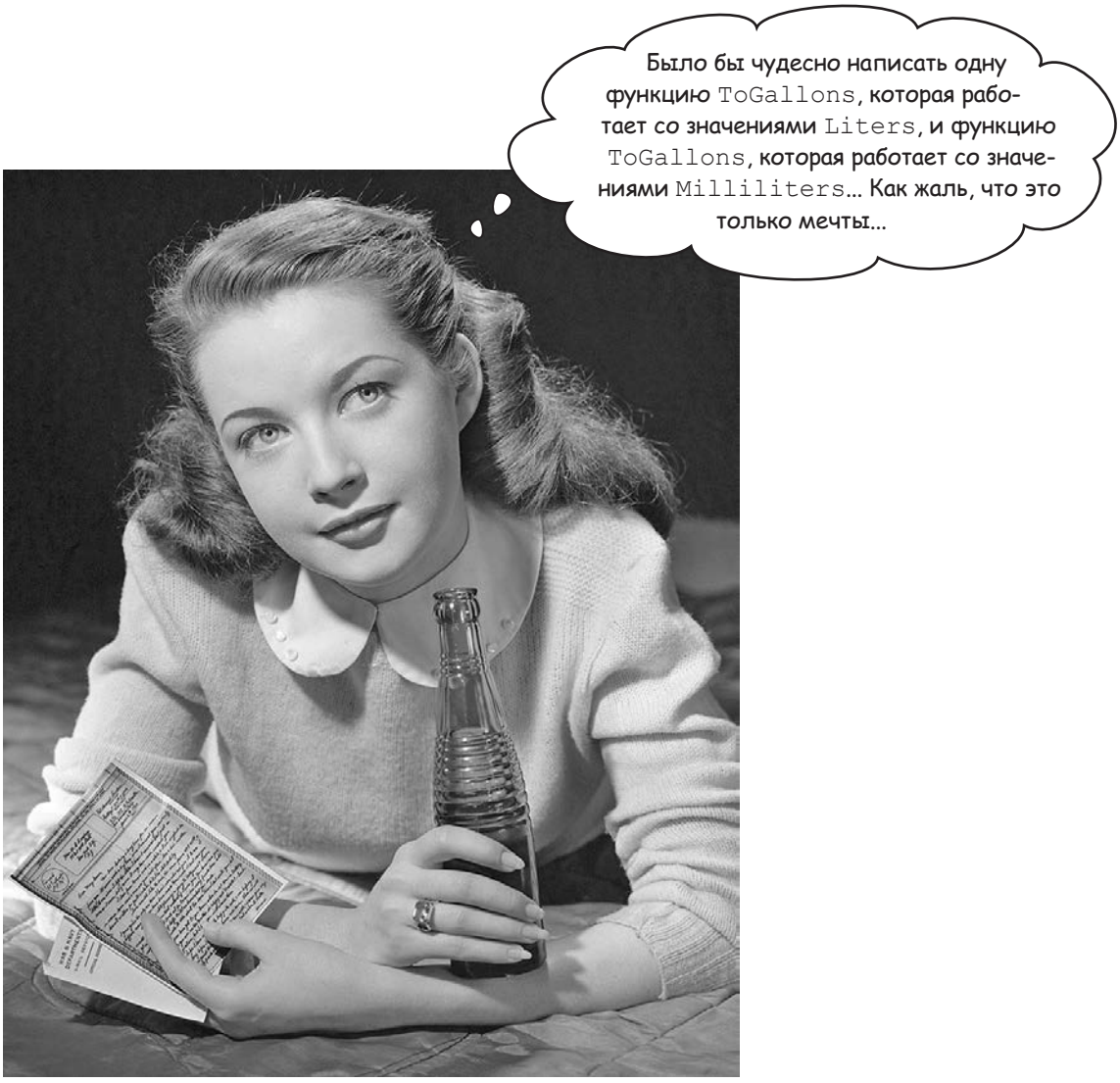
Предотвращает конфликт, но имя получается очень длинным!

Предотвращает конфликт, но имя получается очень длинным!

Часто Задаваемые Вопросы

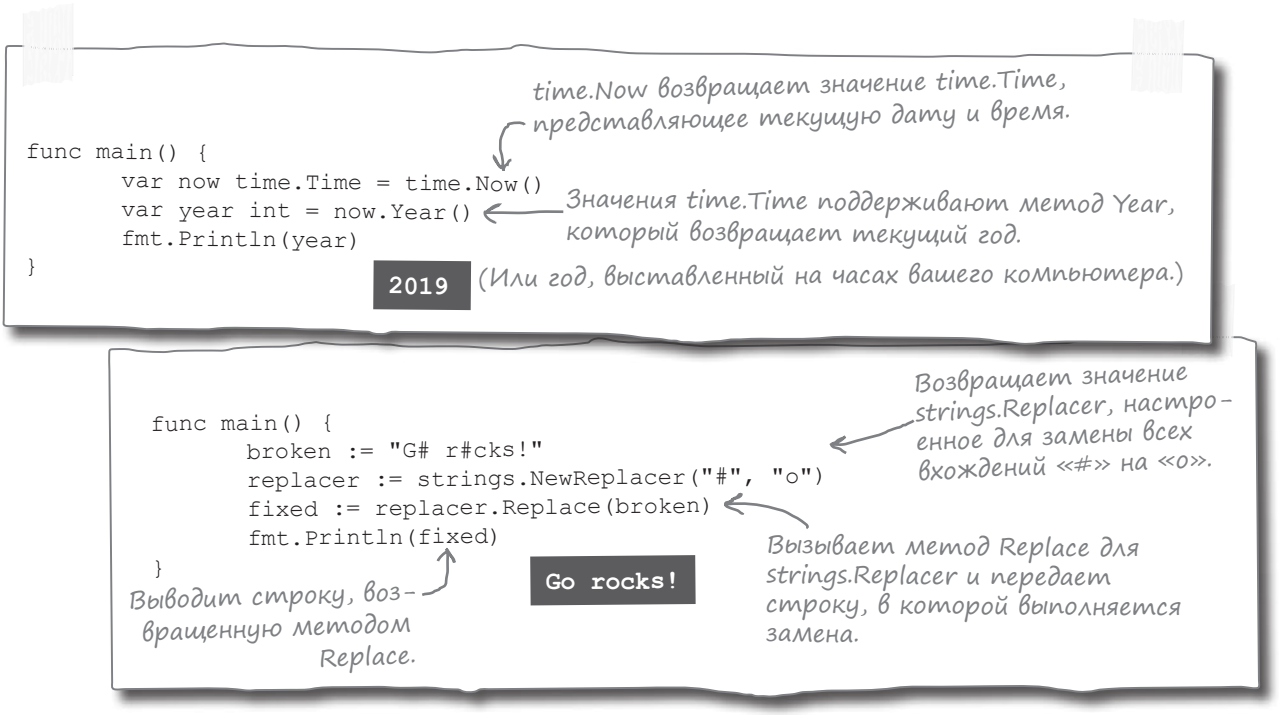
В: Я видел другие языки, поддерживающие *перегрузку* функций: в таких языках можно создавать несколько одноименных функций при условии, что их типы параметров различны. В Go перегрузка не поддерживается?

О: Специалисты по сопровождению Go тоже часто слышат этот вопрос, поэтому ответили на него по адресу *https://golang.org/doc/faq#overloading*: «Опыт работы с другими языками говорит нам, что возможность создания нескольких одноименных методов с разными сигнатурами иногда бывает полезной, но она также приводит к созданию запутанного и ненадежного кода». Отказ от перегрузки упрощает язык, поэтому в Go она не поддерживается. Как вы увидите далее в книге, команда Go принимала аналогичные решения в других областях языка: когда приходилось выбирать между простотой и расширением возможностей, обычно предпочтение отдавалось простоте. И это правильно! Вскоре вы увидите, что к тому же результату можно прийти другими средствами...


Было бы чудесно написать одну функцию ToGallons, которая работает со значениями Liters, и функцию ToGallons, которая работает со значениями Milliliters... Как жаль, что это только мечты...

Разрешение конфликтов имен с использованием методов

Помните, как в главе 2 мы рассказали вам о *методах* — функциях, связанных со значениями конкретного типа? Среди прочего мы создали значение `time.Time`, вызвали его метод `Year`, а также создали значение `strings.Replacer` и вызвали его метод `Replace`.



Мы можем определить собственные методы, которые помогут в решении задачи преобразования типов.

Создать несколько функций с именем `ToGallons` не удастся, поэтому нам приходилось писать длинные, неудобные имена функций, включающие исходный тип:

```
LitersToGallons(Liters(2))
MillilitersToGallons(Milliliters(500))
```

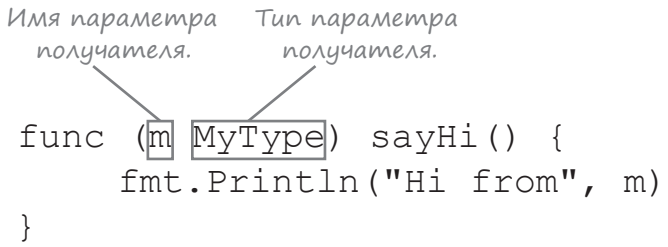
Но вы *можете* определить несколько *методов* с именем `ToGallons` при условии, что они определяются для разных типов. Если вам не придется беспокоиться о возможных конфликтах имен, имена методов станут намного короче.

```
Liters(2).ToGallons()
Milliliters(500).ToGallons()
```

Но не будем забегать вперед — для начала нужно научиться определять методы...

Определение методов

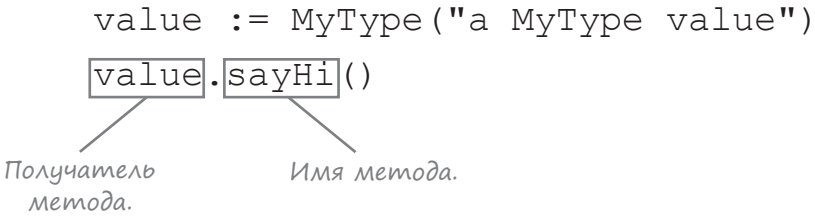
Определение метода очень похоже на определение функции. На самом деле между ними существует только одно отличие: у метода *перед* именем функции добавляется один дополнительный параметр, называемый **параметром получателя**.



Как и у любых параметров функций, необходимо передать имя параметра получателя, за которым следует тип.

Вызов метода, который вы определили, состоит из значения, для которого вызывается метод, точки, имени вызываемого метода и пары круглых скобок. Значение, для которого вызывается метод, называется **получателем** метода.

Сходство между вызовами методов и определениями методов поможет вам запомнить синтаксис: получатель стоит на первом месте при *вызове* метода, а параметр получателя стоит на первом месте при *определении* метода.



Имя параметра получателя в определении метода выбирается произвольно, но важен тип; метод, который вы определяете, связывается со всеми значениями этого типа.

Ниже мы определяем тип с именем `MyType`, имеющий базовый тип `string`. Затем определяется метод с именем `sayHi`. Так как `sayHi` имеет параметр получателя с типом `MyType`, метод `sayHi` можно будет вызывать для любого значения `MyType`. (Многие разработчики скажут, что сам метод `sayHi` определяется для типа `MyType`.)

```
package main

import "fmt"

type MyType string

func (m MyType) sayHi() {
        fmt.Println("Hi")
}

func main() {
        value := MyType("a MyType value")
        value.sayHi()
        anotherValue := MyType("another value")
        anotherValue.sayHi()
}
```

Определяется новый тип.

Определяется параметр получателя.

Для MyType определяется метод.

Создается значение MyType.

Вызывается sayHi для этого значения.

Вызывается sayHi для нового значения.

Создается другое значение MyType.

```
Hi
Hi
```

После того как метод будет определен для типа, его можно будет вызывать для любого значения этого типа.

В следующем примере мы создаем два разных значения `MyType` и вызываем `sayHi` для каждого из них.

Параметр получателя (почти) не отличается от других параметров

Типом параметра получателя является тип, с которым связывается метод. Но в остальном параметр получателя не имеет никаких особых прав в Go. К его содержимому можно обратиться из блока метода точно так же, как и к любому другому параметру функции.

Приведенный ниже пример кода почти не отличается от предыдущего, не считая того, что в нем выводится значение параметра получателя. Мы видим эти значения в выводимых результатах.

```
package main

import "fmt"

type MyType string

func (m MyType) sayHi() {
       fmt.Println("Hi from", m)
}

func main() {
       value := MyType("a MyType value")
       value.sayHi()
       anotherValue := MyType("another value")
       anotherValue.sayHi()
}
```

Вывод значения параметра получателя.

Значение, для которого вызывается метод.

Получатели, передаваемые в параметрах получателей.

Значение, для которого вызывается метод.

```
Hi from a MyType value
Hi from another value
```

См. значения получателей в результатах.

Go позволяет присвоить параметру получателя любое имя, но код будет лучше читаться, если все методы, определяемые для типа, имеют параметры получателей с одинаковыми именами.

По общепринятым соглашениям разработчики Go обычно используют имя, состоящее из одной буквы — первой буквы имени типа получателя в нижнем регистре. (Именно поэтому мы использовали `m` как имя параметра получателя `MyType`.)

В Go параметры получателей занимают место значений «self» или «this» из других языков программирования.

Часто Задаваемые Вопросы

В: Могу ли я определять новые методы для *любого* типа?

О: Только для типов, определяемых в том же пакете, в котором определяется метод. Это означает, что вы не сможете определять методы для типов из стороннего пакета `security`, из своего пакета `hacking` или определять новые методы для таких фундаментальных типов, как `int` или `string`.

В: Но мне очень нужно использовать свои методы с чужими типами!

О: Для начала подумайте, не подойдет ли для этого функция, ведь функции могут получать любые типы в параметрах. Но если вам *действительно* необходимо значение, которое имеет собственные методы, а также некоторые методы типа из другого пакета, создайте тип структуры, в котором тип из другого пакета встроен как анонимное поле. В следующей главе вы увидите, как это делается.

В: Я видел другие языки, в которых получатель метода был доступен в блоке метода в виде специальной переменной с именем `self` или `this`. А как дело обстоит в Go?

О: Go использует параметры получателей вместо `self` и `this`. Принципиальное отличие заключается в том, что `self` и `this` задаются *неявно*, тогда как параметр получателя объявляется *явно*. В остальном параметры получателей работают точно так же, а языку Go не нужно резервировать `self` или `this` как ключевые слова! (Вы даже сможете присвоить параметру получателя имя `this`, но делать так не стоит: по общепринятым соглашениям вместо этого используется первая буква имени типа.)

Метод (почти) не отличается от функции

Помимо того что методы вызываются для получателя, в остальном они очень похожи на любые другие функции.

Как и в случае с другими функциями, вы можете определять дополнительные параметры в круглых скобках после имени метода. К этим переменным/параметрам можно обращаться в блоке метода наряду с параметром получателя. При вызове метода необходимо предоставить аргумент для каждого параметра.

```
func (m MyType) MethodWithParameters(number int, flag bool) {
        fmt.Println(m)
        fmt.Println(number)
        fmt.Println(flag)
}

func main() {
        value := MyType("MyType value")
        value.MethodWithParameters(4, true)
}
```

Параметр получателя. Параметр. Параметр. Получатель. Аргумент. Аргумент.

```
MyType value
4
true
```

Как и в случае с другими функциями, для метода можно объявить одно или несколько возвращаемых значений, которые будут возвращаться при вызове метода:

```
func (m MyType) WithReturn() int {
        return len(m)
}

func main() {
        value := MyType("MyType value")
        fmt.Println(value.WithReturn())
}
```

Возвращаемое значение. Возвращает длину основополагающего строкового значения получателя. Выводит возвращаемое значение метода.

```
12
```

Как и в случае с любой другой функцией, метод экспортируется из текущего пакета, если его имя начинается с буквы верхнего регистра, и не экспортируется, если имя начинается с буквы нижнего регистра. Если вы хотите использовать свой метод за пределами текущего пакета, проследите за тем, чтобы его имя начиналось с буквы верхнего регистра.

```
func (m MyType) ExportedMethod() {
}

func (m MyType) unexportedMethod() {
}
```

Экспортируется — имя начинается с буквы верхнего регистра. Не экспортируется — имя начинается с буквы нижнего регистра.

Упражнение

Заполните пропуски в определении типа `Number` с методами `Add` и `Subtract`, которые должны выводить приведенный результат.

```
type Number int

func (__ _________) ____(_____________ int) {
       fmt.Println(n, "plus", otherNumber, "is", int(n)+otherNumber)
}

func (__ _________) _________(_____________ int) {
       fmt.Println(n, "minus", otherNumber, "is", int(n)-otherNumber)
}

func main() {
       ten := Number(10)
       ten.Add(4)
       ten.Subtract(5)
       four := Number(4)
       four.Add(3)
       four.Subtract(2)
}
```

```
10 plus 4 is 14
10 minus 5 is 5
4 plus 3 is 7
4 minus 2 is 2
```

Ответ на с. 320.

Указатели и параметры получателей

А теперь проблема, которая покажется вам знакомой. Мы определяем новый тип `Number` с базовым типом `int`. Для `Number` определяется метод `double`, который должен умножать базовое значение получателя на 2 и обновлять получателя. Но из результатов видно, что получатель метода не обновляется!

```
package main

import "fmt"

type Number int

func (n Number) Double() {
      n *= 2
}

func main() {
      number := Number(4)
      fmt.Println("Original value of number:", number)
      number.Double()
      fmt.Println("number after calling Double:", number)
}
```

Определяем тип с базовым типом «int».

Определяем метод для типа Number.

Умножаем получателя на 2 и пытаемся обновить.

Создается значение Number.

Удваиваем Number.

```
Original value of number: 4
number after calling Double: 4
```

Number не меняется!

В главе 3 у *функции* `double` возникала аналогичная проблема. Тогда вы узнали, что параметры функций получают копии значений, с которыми вызывалась функция, а не исходные значения, а все изменения в копии теряются при выходе из функции. Чтобы функция `double` работала, нужно было передать *указатель* на обновляемое значение, а затем обновить значение по указателю внутри функции.

```
func main() {
      amount := 6
      double(&amount)
      fmt.Println(amount)
}

func double(number *int) {
      *number *= 2
}
```

Вместо значения переменной передается указатель.

Получает указатель вместо значения int.

Обновляется значение, на которое ссылается указатель.

```
12
```

Выводит удвоенное значение.

Указатели и параметры получателей (продолжение)

Мы сказали, что параметры указателей принципиально ничем не отличаются от обычных параметров. Как и любые параметры, параметр указателя получает *копию* значения получателя. Если вы внесете изменения в получатель внутри метода, то изменится копия, а не оригинал.

Как и в случае с функцией `double` из главы 3, проблема решается обновлением метода `Double` и использованием указателя в качестве параметра получателя. Это делается так же, как и с обычными параметрами: перед типом получателя ставится знак `*` — признак типа-указателя. Также необходимо изменить блок метода, чтобы в нем изменялось значение по указателю. Когда это будет сделано, при вызове `Double` для значения `Number` последнее должно обновиться.

```
// Директивы package, imports, типы пропущены
func (n *Number) Double() {
        *n *= 2
}

func main() {
        number := Number(4)
        fmt.Println("Original value of number:", number)
        number.Double()
        fmt.Println("number after calling Double:", number)
}
```

Параметр получателя преобразуется в тип указателя.

Обновляется значение по указателю.

Изменять вызов метода НЕ НУЖНО!

```
Original value of number: 4
number after calling Double: 8
```

Значение, на которое ссылается указатель, изменилось.

Обратите внимание: нам вообще *не пришлось* изменять вызов метода. При вызове метода, которому требуется получатель-указатель на переменную, не обладающую типом указателя, Go автоматически преобразует получатель в указатель. То же относится к переменным с типами указателей; при вызове метода, требующего получателя-значения, Go автоматически получит значение по указателю и передаст его методу.

Пример справа показывает, как это происходит. У метода с именем `method` получателем является значение, но его можно вызывать как для непосредственных значений, так и для указателей, потому что Go при необходимости выполняет преобразование автоматически. А у метода с именем `pointerMethod` получателем является указатель, но его тоже можно вызывать как для непосредственных значений, так и для указателей, потому что Go при необходимости выполнит преобразование автоматически.

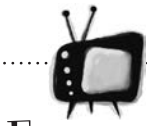
```
// Директивы package, imports пропущены
type MyType string

func (m MyType) method() {
        fmt.Println("Method with value receiver")
}
func (m *MyType) pointerMethod() {
        fmt.Println("Method with pointer receiver")
}

func main() {
        value := MyType("a value")
        pointer := &value
        value.method()
        value.pointerMethod()
        pointer.method()
        pointer.pointerMethod()
}
```

Значение автоматически преобразуется в указатель.

Автоматически читается значение, на которое ссылается указатель.

```
Method with value receiver
Method with pointer receiver
Method with value receiver
Method with pointer receiver
```

Кстати, код справа нарушает общепринятое соглашение: для единства стиля у всех методов типа получателем должно быть либо значение, либо указатель; смешивать их не рекомендуется. Мы же делаем это только для демонстрации.



Будьте осторожны!

Чтобы вызвать метод, которому требуется получатель-указатель, необходимо иметь возможность получить указатель на значение!

Вы можете получать указатели только на значения, хранящиеся в переменных. При попытке получить адрес значения, не хранящегося в переменной, вы получите сообщение об ошибке:

```
&MyType("a value")
```

Ошибка. →

```
cannot take the address
of MyType("a value")
```

То же ограничение действует при вызове методов с получателями-указателями. Go может автоматически преобразовать значения в указатели, но только если значение указателя хранится в переменной. При попытке вызвать метод для самого значения Go не сможет получить указатель, и вы получите похожую ошибку:

```
MyType("a value").pointerMethod()
```

Ошибки. →

```
cannot call pointer method
on MyType("a value")
cannot take the address
of MyType("a value")
```

Вместо этого нужно сохранить значение в переменной; это позволит Go получить указатель на нее:

```
value := MyType("a value")
value.pointerMethod()
```

Go преобразует значение в указатель.

Сломай и изучи!

Перед вами уже знакомый тип `Number` с определениями пары методов. Внесите одно из указанных изменений и запустите программу; затем отмените изменение и переходите к следующему. Посмотрите, что из этого выйдет!

```
package main

import "fmt"

type Number int

func (n *Number) Display() {
       fmt.Println(*n)
}
func (n *Number) Double() {
       *n *= 2
}

func main() {
       number := Number(4)
       number.Double()
       number.Display()
}
```

Если...	*...программа не будет работать, потому что...*

Заменить тип параметра получателя типом, не определенным в текущем пакете:
`func (n *~~Number~~int) Double() {`
`  *n *= 2`
`}`
	Новые методы могут определяться только для типов, объявленных в текущем пакете. Определение метода для глобально определяемого типа (такого, как `int`) приведет к ошибке компиляции

Заменить тип параметра получателя `Double` типом, который не является указателем:
`func (n *Number) Double() {`
`  *n *= 2`
`}`
	Параметры получателей получают копию значения, для которого был вызван метод. Если функция `Double` изменит только копию, то исходное значение останется неизменным при выходе из `Double`

Вызвать метод, которому необходим получатель-указатель, для значения, которое не хранится в переменной: `Number(4).Double()`	При вызове метода с получателем, который является указателем, Go может автоматически преобразовать значение в указатель на получателя, если он хранится в переменной. В противном случае произойдет ошибка

Заменить тип параметра получателя `Display` типом, который не является указателем:
`func (n *Number) Display() {`
`  fmt.Println(*n)`
`}`
	На самом деле после внесения этого изменения код *будет* работать, но нарушит общепринятые соглашения! Параметры получателей в методах типа могут быть либо указателями, либо значениями, и смешивать их не рекомендуется

Преобразование литров и миллилитров в галлоны с помощью методов

При добавлении типа `Milliliters` в определяемые типы для измерения объема оказалось, что мы не можем определить одноименные функции `ToGallons` как для `Liters`, так и для `Milliliters`. Для решения этой проблемы пришлось создавать функции с длинными именами:

```
func LitersToGallons(l Liters) Gallons {
      return Gallons(l * 0.264)
}
func MillilitersToGallons(m Milliliters) Gallons {
      return Gallons(m * 0.000264)
}
```

Но в отличие от функций, имена методов не обязаны быть уникальными, если определяются для разных типов.

Попробуем реализовать методы `ToGallons` для типа `Liters`. Код будет почти идентичен коду функции `LitersToGallons`, но значение `Liters` станет параметром получателя вместо обычного параметра. Затем то же самое нужно проделать для типа `Milliliters`: функция `MillilitersToGallons` преобразуется в метод `ToGallons`.

Обратите внимание: для параметров получателей не используются типы-указатели. Получатели не изменяются и не занимают много памяти, поэтому ничто не мешает передать копию значения в параметре.

```
package main

import "fmt"

type Liters float64
type Milliliters float64
type Gallons float64

func (l Liters) ToGallons() Gallons {
      return Gallons(l * 0.264)
}
func (m Milliliters) ToGallons() Gallons {
      return Gallons(m * 0.000264)
}

func main() {
      soda := Liters(2)
      fmt.Printf("%0.3f liters equals %0.3f gallons\n", soda, soda.ToGallons())
      water := Milliliters(500)
      fmt.Printf("%0.3f milliliters equals %0.3f gallons\n", water, water.ToGallons())
}
```

Имена могут быть одинаковыми, если они определяются для разных типов.

Метод для Liters.

Блок метода не отличается от блока функции.

Имена могут быть одинаковыми, если они определяются для разных типов.

Метод для Milliliters.

Блок метода не отличается от блока функции.

Создание значения Liters.

Преобразование Liters в Gallons.

Создание значения Milliliters.

Преобразование Milliliters в Gallons.

```
2.000 liters equals 0.528 gallons
500.000 milliliters equals 0.132 gallons
```

В функции `main` создается значение `Liters`, для которого затем вызывается метод `ToGallons`. Так как получатель имеет тип `Liters`, будет вызван метод `ToGallons` для типа `Liters`. Аналогичным образом при вызове `ToGallons` для значения `Milliliters` будет вызван метод `ToGallons` для типа `Milliliters`.

Преобразование Gallons в Liters и Milliliters с помощью методов

Примерно то же самое происходит и тогда, если функции `GallonsToLiters` и `GallonsToMilliliters` будут преобразованы в методы. Просто параметр `Gallons` в каждом случае преобразуется в параметр получателя.

```
func (g Gallons) ToLiters() Liters {
       return Liters(g * 3.785)
}
func (g Gallons) ToMilliliters() Milliliters {
       return Milliliters(g * 3785.41)
}

func main() {
       milk := Gallons(2)
       fmt.Printf("%0.3f gallons equals %0.3f liters\n", milk, milk.ToLiters())
       fmt.Printf("%0.3f gallons equals %0.3f milliliters\n", milk, milk.ToMilliliters())
}
```

Определяем метод ToLiters для типа Gallons.

Определяем метод ToMilliliters для типа Gallons.

Создаем значение Gallons.

Преобразуем в Liters.

Преобразуем в Milliliters.

```
2.000 gallons equals 7.570 liters
2.000 gallons equals 7570.820 milliliters
```

Упражнение

Приведенный ниже код должен добавлять метод `ToMilliliters` для типа `Liters` и метод `ToLiters` для типа `Milliliters`. Код функции `main` должен выдавать приведенный ниже результат. Заполните пропуски и завершите код.

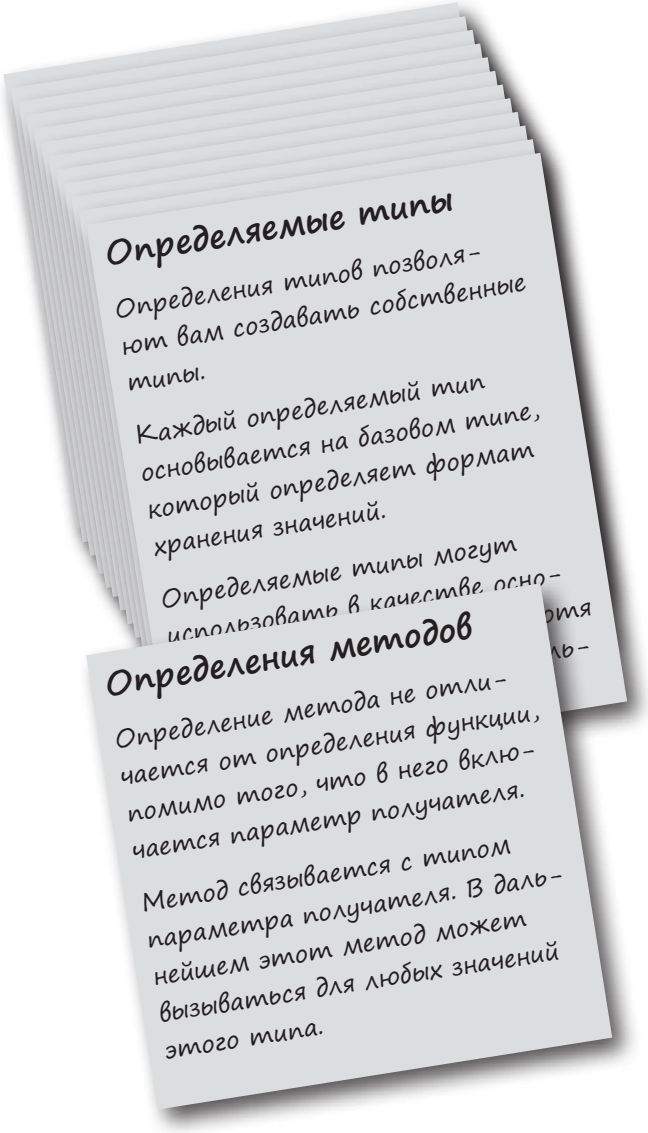
```
type Liters float64
type Milliliters float64
type Gallons float64

func _____ ToMilliliters() _____ {
       return Milliliters(l * 1000)
}
func _____ ToLiters() _____ {
       return Liters(m / 1000)
}

func main() {
       l := _____(3)
       fmt.Printf("%0.1f liters is %0.1f milliliters\n", l, l._____())
       ml := _____(500)
       fmt.Printf("%0.1f milliliters is %0.1f liters\n", ml, ml._____())
}
```

```
3.0 liters is 3000.0 milliliters
500.0 milliliters is 0.5 liters
```

Ответ на с. 321.

Ваш инструментарий Go

Глава 9 осталась позади! Ваш инструментарий пополнился определениями методов.



КЛЮЧЕВЫЕ МОМЕНТЫ

- После того как тип будет определен, вы можете выполнить преобразование к нему любого значения того же базового типа:
  `Gallons(10.0)`
- После определения типа переменной значения других типов не могут присваиваться этой переменной, даже если они имеют тот же базовый тип.
- Определяемый тип поддерживает все те же операторы, что и базовый тип. Например, тип, основывающийся на базовом типе `int`, будет поддерживать операторы `+`, `-`, `*`, `/`, `==`, `>` и `<`.
- Определяемый тип может использоваться в операциях совместно со значениями-литералами:
  `Gallons(10.0) + 2.3`
- Чтобы определить метод, укажите параметр получателя в круглых скобках перед именем метода:
```
func (m MyType) MyMethod() {
}
```
- Параметр получателя может использоваться в блоке метода, как любой другой параметр:
```
func (m MyType) MyMethod() {
    fmt.Println("called on", m)
}
```
- Для методов, как и для любых других функций, можно определять дополнительные параметры или возвращаемые значения.
- Определение нескольких одноименных функций в одном пакете запрещено, даже если они имеют параметры разных типов. С другой стороны, вы *можете* определить несколько *методов* с одинаковыми именами при условии, что они определяются для разных типов.
- Методы могут определяться только для типов, определенных в том же пакете.
- Как и для других параметров, в параметрах получателей передается копия исходного значения. Если ваш метод должен изменять получатель, используйте тип указателя для параметра получателя и измените значение по этому указателю.

У бассейна. Решение

```
package main

import "fmt"

type Population int

func main() {
    var population Population
    population = Population(572)
    fmt.Println("Sleepy Creek County population:", population)
    fmt.Println("Congratulations, Kevin and Anna! It's a girl!")
    population += 1
    fmt.Println("Sleepy Creek County population:", population)
}
```

Объявляем тип Population с базовым типом «int».

Целое число преобразуется в значение Population.

Базовый тип поддерживает оператор +=, поэтому тип Population его тоже поддерживает.

Результат.

```
Sleepy Creek County population: 572
Congratulations, Kevin and Anna! It's a girl!
Sleepy Creek County population: 573
```

Упражнение Решение

Заполните пропуски в определении типа `Number` с методами `Add` и `Subtract`, которые должны выводить приведенный результат.

```
type Number int

func (n Number) Add(otherNumber int) {
    fmt.Println(n, "plus", otherNumber, "is", int(n)+otherNumber)
}

func (n Number) Subtract(otherNumber int) {
    fmt.Println(n, "minus", otherNumber, "is", int(n)-otherNumber)
}

func main() {
    ten := Number(10)
    ten.Add(4)
    ten.Subtract(5)
    four := Number(4)
    four.Add(3)
    four.Subtract(2)
}
```

Параметр получателя.

Метод определяет тип Number.

Number невозможно сложить с int; необходимо выполнить преобразование.

Выводим получателя.

Выводим обычный параметр.

Параметр получателя.

Метод определяет тип Number.

Необходимо выполнить преобразование.

Выводим получателя.

Выводим обычный параметр.

Целое число преобразуется в Number.

Вызовы методов Number.

Целое число преобразуется в Number.

Вызовы методов Number.

```
10 plus 4 is 14
10 minus 5 is 5
4 plus 3 is 7
4 minus 2 is 2
```

Упражнение
Решение

Приведенный ниже код должен добавлять метод `ToMilliliters` для типа `Liters` и метод `ToLiters` для типа `Milliliters`. Код функции `main` должен выдавать приведенный ниже результат. Заполните пропуски и завершите код.

```
type Liters float64
type Milliliters float64
type Gallons float64

func (l Liters) ToMilliliters() Milliliters {
    return Milliliters(l * 1000)
}
func (m Milliliters) ToLiters() Liters {
    return Liters(m / 1000)
}

func main() {
    l := Liters(3)
    fmt.Printf("%0.1f liters is %0.1f milliliters\n", l, l.ToMilliliters())
    ml := Milliliters(500)
    fmt.Printf("%0.1f milliliters is %0.1f liters\n", ml, ml.ToLiters())
}
```

Значение получателя умножается на 1000, а результат преобразуется к типу Milliliters.

Значение получателя делится на 1000, а тип результата преобразуется в Liters.

```
3.0 liters is 3000.0 milliliters
500.0 milliliters is 0.5 liters
```

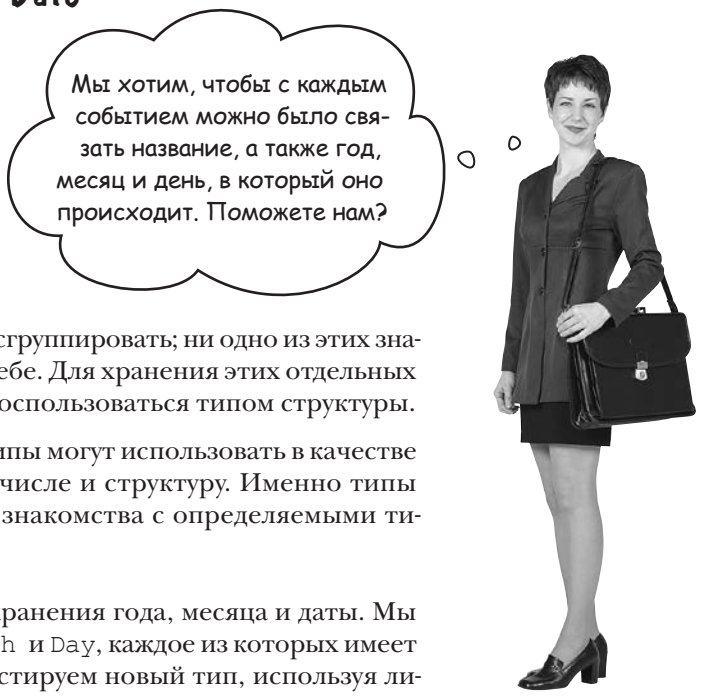
Инкапсуляция и встраивание

Я слышала, что ее тип `Paragraph` хранит свои данные в простом поле `string`! А этот экзотический метод `Replace`? Да он просто достался от встроенного `strings`. `Replacer`! Но когда работаешь с `Paragraph`, ни за что не догадаешься!

Ошибки случаются. Иногда программа получает недействительные данные от пользователя, из файла или другого источника. В этой главе рассматривается **инкапсуляция**: механизм защиты полей структурного типа от недействительных данных. И будьте уверены — теперь с данными полей можно безопасно работать! Мы также покажем, как **встраивать** другие типы в структуры. Если вашему типу структуры потребуются методы, которые уже существуют у другого типа, вам не придется копировать и вставлять код метода. Вы можете встроить другой тип в свой тип структуры, а затем воспользоваться методами встроенного типа так, если бы они были определены для вашего собственного типа!

Создание типа структуры Date

Местный стартап Remind Me разрабатывает календарное приложение, которое будет своевременно напоминать пользователям о днях рождения, юбилеях и т. д.



Похоже, что год, месяц и день стоит сгруппировать; ни одно из этих значений не принесет пользы само по себе. Для хранения этих отдельных значений в одном наборе разумно воспользоваться типом структуры.

Как вы уже видели, определяемые типы могут использовать в качестве базового любой другой тип, в том числе и структуру. Именно типы структур использовались нами для знакомства с определяемыми типами в главе 8.

Создадим тип структуры `Date` для хранения года, месяца и даты. Мы добавим в структуру поля `Year`, `Month` и `Day`, каждое из которых имеет тип `int`. В функции `main` мы протестируем новый тип, используя литерал структуры для создания значения `Date` с заполненными полями. После этого содержимое `Date` выводится функцией `Println`.

```
package main

import "fmt"

type Date struct {
        Year  int
        Month int
        Day   int
}

func main() {
        date := Date{Year: 2019, Month: 5, Day: 27}
        fmt.Println(date)
}
```

Определение нового типа структуры.

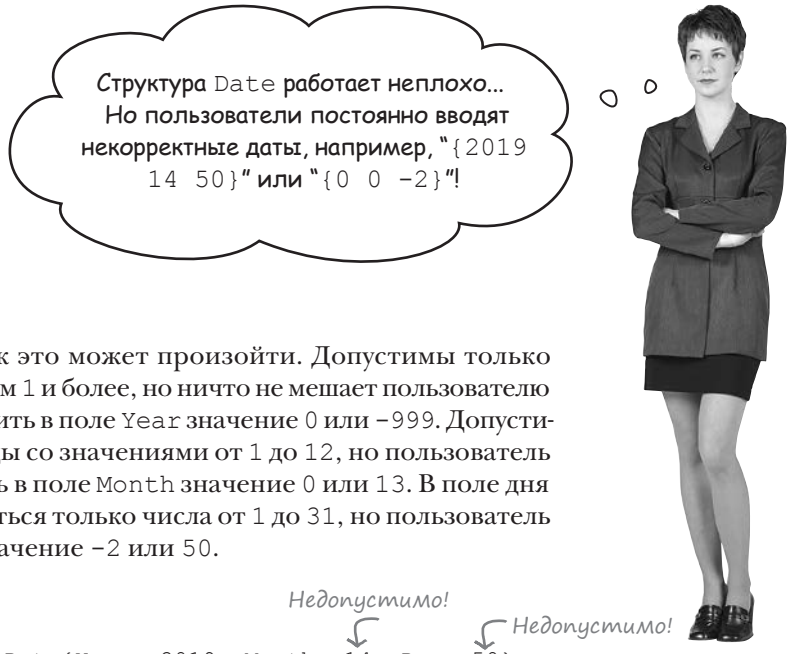
Определяем поля структуры.

Для создания значения Date используется литерал структуры.

```
{2019 5 27}
```

Запустив программу, вы увидите поля `Year`, `Month` и `Day` нашей структуры `Date`. Кажется, все работает!

Пользователи заполняют поле структуры Date недопустимыми значениями!



Да, понятно, как это может произойти. Допустимы только годы со значением 1 и более, но ничто не мешает пользователю случайно сохранить в поле `Year` значение `0` или `-999`. Допустимы только месяцы со значениями от 1 до 12, но пользователь может сохранить в поле `Month` значение `0` или `13`. В поле дня `Day` могут храниться только числа от 1 до 31, но пользователь может ввести значение `-2` или `50`.

```
date := Date{Year: 2019, Month: 14, Day: 50}
fmt.Println(date)
date = Date{Year: 0, Month: 0, Day: -2}
fmt.Println(date)
date = Date{Year: -999, Month: -1, Day: 0}
fmt.Println(date)
```

Недопустимо! *Недопустимо!* *Недопустимо!* *Недопустимо!* *Недопустимо!* *Недопустимо!* *Недопустимо!*

```
{2019 14 50}
{0 0 -2}
{-999 -1 0}
```

Необходимо каким-то образом проверить корректность данных пользователя, прежде чем сохранять их в структуре. Мы должны убедиться, что полю года `Year` присвоено значение 1 и более, полю месяца `Month` — значение от 1 до 12, а полю дня `Day` — значение от 1 до 31.

(Да, не в каждом месяце 31 день, но чтобы не усложнять код примера, мы ограничимся тем, что значение лежит в интервале от 1 до 31.)

Set-методы

Тип структуры — всего лишь разновидность определяемого типа; это означает, что для него, как и для любых других определяемых типов, могут определяться методы. Попробуем создать для типа `Date` методы `SetYear`, `SetMonth` и `SetDay`, которые получают значение, проверяют его, и если значение допустимо — присваивают его соответствующему полю структуры.

Такие методы часто называются **set-методами**, или сеттерами. По общепринятым соглашениям set-методам в Go присваиваются имена в форме `Set`*X*, где *X* — присваиваемое поле.

Ниже приведена первая версия метода `SetYear`. Параметр получателя представляет структуру `Date`, для которой вызывается метод. `SetYear` получает присваиваемый год в параметре и присваивает его полю `Year` получателя — структуры `Date`. Пока он не проверяет новое значение, но проверку мы скоро добавим.

В методе `main` мы создаем значение `Date` и вызываем для него `SetYear`. После этого выводится содержимое поля `Year` структуры.

Set-методы используются для присваивания значений полей или других данных в базовом значении определяемого типа.

```
package main

import "fmt"

type Date struct {
       Year  int
       Month int
       Day   int
}

func (d Date) SetYear(year int) {
       d.Year = year
}

func main() {
       date := Date{}
       date.SetYear(2019)
       fmt.Println(date.Year)
}
```

Получает значение, которое должно быть присвоено полю.

Присваивает значение полю структуры.

Создание Date.

Задает значение поля Year при помощи метода.

Выводит поле Year.

```
0
```

Year по-прежнему содержит нулевое значение!

При запуске программы становится очевидно, что не все пошло по плану. Хотя мы создаем значение `Date` и вызываем `SetYear` с новым значением, поле `Year` по-прежнему сохраняет нулевое значение!

Set-методам необходимы указатели на получателей

Помните метод `Double` приведенного ранее метода `Number`? Сначала мы написали его с получателем, который был простым значением `Number`. Но вы узнали, что параметры получателей, как и любые другие параметры, получают *копию* исходного значения. Метод `Double` обновлял копию, а изменения терялись при выходе из функции.

```
func (n *Number) Double() {
        *n *= 2
}
```

Параметр указателя преобразуется к типу указателя.

Обновление значения по указателю.

Проблема решалась обновлением `Double` с передачей типа указателя на получателя `*Number`. При обновлении значения по указателю изменения сохранялись после выхода из `Double`.

То же относится к `SetYear`. Получателю `Date` достается *копия* исходной структуры. Любые изменения полей копии будут потеряны при выходе из `SetYear`!

```
func (d Date) SetYear(year int) {
        d.Year = year
}
```

Получает копию структуры Date.

Обновляет копию, а не оригинал!

Проблема с `SetYear` решается заменой получателя указателем на получателя: `(d *Date)`. Это единственное необходимое изменение. Обновлять блок метода `SetYear` не нужно, так как `d.Year` автоматически получает значение по указателю (как если бы было введено выражение `(*d).Year`). Вызов `date.SetYear` в `main` изменять тоже не нужно, потому что значение `Date` автоматически преобразуется в `*Date` при передаче методу.

```
type Date struct {
        Year  int
        Month int
        Day   int
}

func (d *Date) SetYear(year int) {
        d.Year = year
}

func main() {
        date := Date{}
        date.SetYear(2019)
        fmt.Println(date.Year)
}
```

Здесь нужен указатель на получателя, чтобы обновлялась не копия, а исходное значение.

Теперь обновляется исходное значение, а не копия.

Автоматически получает значение по указателю.

Автоматически преобразуется в указатель.

```
2019
```

Поле Year было изменено.

Теперь, когда `SetYear` получает указатель на получателя, повторный запуск программы показывает, что поле `Year` было успешно изменено.

Добавление остальных set-методов

Теперь вы без особого труда по той же схеме определите методы `SetMonth` и `SetDay` для типа `Date`. Нужно только проследить за тем, чтобы в определении метода использовался указатель на получателя. Go преобразует получателя в указатель при вызове каждого метода и преобразует указатель обратно в значение структуры при обновлении полей.

```
package main

import "fmt"

type Date struct {
       Year  int
       Month int
       Day   int
}

func (d *Date) SetYear(year int) {
       d.Year = year
}
func (d *Date) SetMonth(month int) {
       d.Month = month
}
func (d *Date) SetDay(day int) {
       d.Day = day
}
```

Обязательно используйте указатели на получателей!

В `main` создается значение структуры `Date`; задайте его поля `Year`, `Month` и `Day` при помощи новых методов и выведите всю структуру, чтобы увидеть результат.

```
func main() {
       date := Date{}
       date.SetYear(2019)
       date.SetMonth(5)
       date.SetDay(27)
       fmt.Println(date)
}
```

Задает месяц.

Задает день месяца.

Выводит все поля.

```
{2019 5 27}
```

Теперь у нас имеются set-методы для всех полей типа `Date`. Но несмотря на методы, пользователи все равно могут случайно присвоить полям недопустимые значения. Давайте посмотрим, как этого избежать.

```
date := Date{}
date.SetYear(0)
date.SetMonth(14)
date.SetDay(50)
fmt.Println(date)
```

Недопустимо!

Недопустимо!

Недопустимо!

```
{0 14 50}
```

Упражнение

В упражнениях главы 8 был приведен код типа структуры `Coordinates`. Это определение типа было перемещено в файл *coordinates.go* в каталоге пакета *geo*.

Необходимо добавить в тип `Coordinates` set-методы для каждого из его полей. Заполните пропуски в файле *coordinates.go*, чтобы код *main.go* выполнялся и выводил показанный результат.

```
package geo

type Coordinates struct {
        Latitude  float64
        Longitude float64
}

func (c _______________) SetLatitude(__________ float64) {
        _____________ = latitude
}

func (c _______________) SetLongitude(__________ float64) {
        _____________ = longitude
}
```

coordinates.go

```
package main

import (
        "fmt"
        "geo"
)

func main() {
        coordinates := geo.Coordinates{}
        coordinates.SetLatitude(37.42)
        coordinates.SetLongitude(-122.08)
        fmt.Println(coordinates)
}
```

main.go

Результат.

```
{37.42 -122.08}
```

Ответ на с. 351.

Включение проверки данных в set-методы

Чтобы включить проверку данных в set-методы, придется потрудиться, но все необходимое вы узнали из главы 3. В каждом set-методе мы проверяем, принадлежит ли значение допустимому диапазону. Если значение недействительно, возвращается значение ошибки. Для действительных значений поле структуры `Date` присваивается как обычно, а вместо значения ошибки возвращается `nil`.

Начнем с включения проверки данных в метод `SetYear`. В объявление включается информация о том, что метод возвращает значение типа `error`. В начале блока метода мы проверяем, является ли переданный при вызове параметр `year` любым числом меньше `1`. В таком случае возвращается ошибка с сообщением `"invalid year"`. В остальных случаях задается значение поля `Year` структуры и возвращается `nil` — признак отсутствия ошибки.

В функции `main` мы вызываем `SetYear` и сохраняем возвращаемое значение в переменной с именем `err`. Если переменная `err` отлична от `nil`, это означает, что присваиваемое значение было недействительным, поэтому программа выводит сообщение об ошибке и завершается. В остальных случаях программа переходит к выводу поля `Year` структуры `Date`.

```
package main

import (
        "errors"
        "fmt"
        "log"
)

type Date struct {
        Year  int
        Month int
        Day   int
}

func (d *Date) SetYear(year int) error {
        if year < 1 {
                return errors.New("invalid year")
        }
        d.Year = year
        return nil
}
// SetMonth, SetDay пропущены

func main() {
        date := Date{}
        err := date.SetYear(0)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(date.Year)
}
```

Позволяет создавать значения ошибок.

Для вывода сообщения об ошибке и завершения программы.

Добавляет возвращаемое значение ошибки.

Если год недействителен, возвращается признак ошибки.

В противном случае присваивается значение поля...

...и возвращается ошибка «nil».

Значение недействительно!

Сохраняются любые ошибки.

Если значение недопустимо, программа выводит сообщение об ошибке и завершается.

Сообщение об ошибке.

```
2018/03/17 19:58:02 invalid year
exit status 1
```

Если переданное `SetYear` значение недействительно, программа сообщает об ошибке и завершается. При передаче допустимого значения программа выводит его. Похоже, наш метод `SetYear` работает успешно!

```
date := Date{}
err := date.SetYear(2019)
if err != nil {
        log.Fatal(err)
}
fmt.Println(date.Year)
```

Действительное значение.

```
2019
```

Выводится поле.

Включение проверки данных в set-методы (продолжение)

Код проверки данных в методах SetMonth и SetDay похож на код SetYear.

В SetMonth мы проверяем, выходит ли переданный номер месяца за пределы диапазона от 1 до 12, и если выходит — возвращаем признак ошибки. В противном случае метод присваивает значение поля и возвращает nil.

Метод SetDay проверяет, выходит ли переданный день месяца за пределы диапазона от 1 до 31. Для недопустимых значений возвращается значение ошибки, а для действительных метод изменяет значение поля и возвращает nil.

```
// Директивы package, imports, объявления типов пропущены
func (d *Date) SetYear(year int) error {
	if year < 1 {
		return errors.New("invalid year")
	}
	d.Year = year
	return nil
}
func (d *Date) SetMonth(month int) error {
	if month < 1 || month > 12 {
		return errors.New("invalid month")
	}
	d.Month = month
	return nil
}
func (d *Date) SetDay(day int) error {
	if day < 1 || day > 31 {
		return errors.New("invalid day")
	}
	d.Day = day
	return nil
}

func main() {
	// Сюда подставляются для тестирования
	// фрагменты кода, приведенные ниже
  }
```

Чтобы протестировать set-методы, вставьте приведенные ниже фрагменты кода в блок main...

Если SetMonth передается значение 14, возникает ошибка:

```
date := Date{}
err := date.SetMonth(14)
if err != nil {
	log.Fatal(err)
}
fmt.Println(date.Month)
```

```
2018/03/17 20:17:42
invalid month
exit status 1
```

Но при передаче 5 метод SetMonth работает:

```
date := Date{}
err := date.SetMonth(5)
if err != nil {
	log.Fatal(err)
}
fmt.Println(date.Month)
```

```
5
```

Если SetDay передается значение 50, возникает ошибка:

```
date := Date{}
err := date.SetDay(50)
if err != nil {
	log.Fatal(err)
}
fmt.Println(date.Day)
```

```
2018/03/17 20:30:54
invalid day
exit status 1
```

Но при передаче SetDay значения 27 все работает:

```
date := Date{}
err := date.SetDay(27)
if err != nil {
	log.Fatal(err)
}
fmt.Println(date.Day)
```

```
27
```

Полям все равно можно присвоить недопустимые значения!

Проверка данных в set-методах прекрасно работает... когда пользователи вызывают их. Но люди обращаются к полям структур напрямую и продолжают вводить недопустимые данные!

И верно — любой желающий может напрямую задать значения полей структуры `Date`. В этом случае обходится код проверки данных в set-методах, и полю можно будет присвоить произвольное значение!

```
date := Date{}
date.Year = 2019
date.Month = 14
date.Day = 50
fmt.Println(date)
```

```
{2019 14 50}
```

Поля необходимо как-то защитить, чтобы пользователи типа `Date` могли обновлять их только с использованием set-методов.

В Go предусмотрен такой механизм защиты: тип `Date` можно переместить в другой пакет и запретить экспортирование его полей данных.

До настоящего момента неэкспортируемые переменные, функции и т. д. чаще только мешали. Последний пример встречался в главе 8, когда выяснилось, что даже в экспортированном из пакета `magazine` типа структуры `Subscriber` его поля *не экспортировались* и из-за этого были недоступны за пределами пакета `magazine`.

Имя типа `Subscriber` начинается с буквы верхнего регистра, поэтому к нему можно будет обращаться из пакета `main`. Но здесь мы получаем сообщение об ошибке, в котором сказано, что *поле* `rate` недоступно, потому что *оно* не экспортируется.

```
Shell Edit View Window Help
$ go run main.go
./main.go:10:13: s.rate undefined
(cannot refer to unexported field or method rate)
./main.go:11:25: s.rate undefined
(cannot refer to unexported field or method rate)
```

Даже если тип структуры экспортируется из пакета, его поля *не экспортируются*, если их имена не начинаются с буквы верхнего регистра. Попробуем изменить регистр первой буквы имени `Rate` (как в файле *magazine.go*, так и в файле *main.go*)...

Но в этом конкретном случае мы *не хотим*, чтобы поля были доступными. Неэкспортируемые поля структур — именно то, что нужно!

Переместим тип `Date` в другой пакет, сделаем его поля неэкспортируемыми и посмотрим, решит ли это нашу проблему.

Перемещение типа Date в другой пакет

В каталоге *headfirstgo* в рабочей области Go создайте новый каталог для хранения пакета с именем `calendar`. В каталоге *calendar* создайте файл с именем *date.go*. (Напомним, что имена файлов в каталоге пакета могут быть любыми, все файлы станут частью одного пакета.)

В файле *date.go* добавьте объявление пакета `calendar` и импортируйте пакет `"errors"`. (Единственный пакет, который будет использоваться в коде этого файла.) Затем скопируйте весь старый код типа `Date` и вставьте его в этот файл.

```
package calendar

import "errors"

type Date struct {
       Year  int
       Month int
       Day   int
}

func (d *Date) SetYear(year int) error {
       if year < 1 {
              return errors.New("invalid year")
       }
       d.Year = year
       return nil
}
func (d *Date) SetMonth(month int) error {
       if month < 1 || month > 12 {
              return errors.New("invalid month")
       }
       d.Month = month
       return nil
}
func (d *Date) SetDay(day int) error {
       if day < 1 || day > 31 {
              return errors.New("invalid day")
       }
       d.Day = day
       return nil
}
```

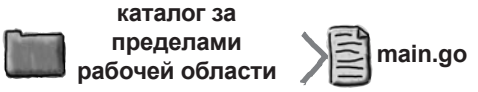
Файл является частью пакета «calendar».

В файле используются только функции из пакета «error».

Скопируйте весь код типа Date в новый файл.

Перемещение типа Date в другой пакет (продолжение)

А теперь создадим программу для тестирования пакета `calendar`. Так как программа предназначена исключительно для экспериментов, мы поступим так, как в главе 8, и сохраним файл *за пределами* рабочей области Go, чтобы он не конфликтовал с другими пакетами. (Для его запуска будет использоваться команда `go run`.) Присвойте файлу имя *main.go*.



(При желании этот код можно будет переместить в рабочую область Go, при условии, что для него будет создан отдельный каталог пакета.)

На данный момент код, добавленный в *main.go*, сможет создать недействительное значение `Date` — либо прямым заданием полей, либо с помощью литерала структуры.

```
package main
```
Используем пакет «main», так как код будет выполняться в виде программы.

```
import (
        "fmt"
        "github.com/headfirstgo/calendar"
)
```
Импортируем новый пакет.

```
func main() {
        date := calendar.Date{}
        date.Year = 2019
        date.Month = 14
        date.Day = 50
        fmt.Println(date)
        date = calendar.Date{Year: 0, Month: 0, Day: -2}
        fmt.Println(date)
}
```
Необходимо указать пакет, из которого импортируется тип.

Создается новое значение Date.

Поля Date задаются напрямую.

Поля другого значения Date задаются с использованием литерала стркутуры.

Указываем пакет.

Если запустить *main.go* в терминале, вы увидите, что оба способа заполнения полей сработали, а программа выводит две недействительные даты.

```
Shell Edit View Window Help
$ cd temp
$ go run main.go
{2019 14 50}
{0 0 -2}
```
Недействительные даты!

Отмена экспортирования полей Date

Попробуем обновить структуру `Date`, чтобы ее поля не экспортировались из пакета. Просто измените имена полей, чтобы они начинались с буквы нижнего регистра, в определении типа и везде, где они встречаются.

Сам тип `Date` при этом должен оставаться экспортируемым, как и все set-методы, потому что нам *потребуется* обращаться к ним за пределами пакета `calendar`.

date.go

```
package calendar

import "errors"

type Date struct {
        year  int
        month int
        day   int
}

func (d *Date) SetYear(year int) error {
        if year < 1 {
                return errors.New("invalid year")
        }
        d.year = year
        return nil
}
func (d *Date) SetMonth(month int) error {
        if month < 1 || month > 12 {
                return errors.New("invalid month")
        }
        d.month = month
        return nil
}
func (d *Date) SetDay(day int) error {
        if day < 1 || day > 31 {
                return errors.New("invalid day")
        }
        d.day = day
        return nil
}
```

Тип Date должен оставаться экспортируемым!

Имена полей изменяются, чтобы запретить их экспортирование.

Имена методов не изменились.

Параметры метода не изменились.

Имя поля изменилось в соответствии с предшествующим объявлением.

Имя поля изменилось в соответствии с предшествующим объявлением.

Имя поля изменилось в соответствии с предшествующим объявлением.

Чтобы протестировать внесенные изменения, обновите имена полей в *main.go* — они должны соответствовать именам полей в *date.go*.

main.go

```
// Директивы package и import пропущены.
func main() {
        date := calendar.Date{}
        date.year = 2019
        date.month = 14
        date.day = 50
        fmt.Println(date)

        date = calendar.Date{year: 0, month: 0, day: -2}
        fmt.Println(date)
}
```

Имена полей изменяются в соответствии с объявлением.

Имена полей изменяются в соответствии с объявлением.

Обращения к неэкспортируемым полям через экспортируемые методы

Как и следовало ожидать, после того как поля `Date` будут преобразованы в неэкспортируемые, попытка обращения к ним из пакета `main` приведет к ошибкам компиляции. Это относится и к попытке прямого присваивания значений полей, и к использованию их в литерале структуры.

К полям не удастся обратиться напрямую.

```
Shell Edit View Window Help
$ cd temp
$ go run main.go
./main.go:10:6: date.year undefined (cannot refer to unexported field or method year)
./main.go:11:6: date.month undefined (cannot refer to unexported field or method month)
./main.go:12:6: date.day undefined (cannot refer to unexported field or method day)
./main.go:15:27: unknown field 'year' in struct literal of type calendar.Date
./main.go:15:37: unknown field 'month' in struct literal of type calendar.Date
./main.go:15:45: unknown field 'day' in struct literal of type calendar.Date
```

Тем не менее к полям можно обращаться опосредованно. К *неэкспортируемым* переменным, полям структур, функциям, методам и т. д. можно обращаться из *экспортируемых* функций и методов того же пакета. Таким образом, когда код пакета `main` вызывает экспортируемый метод `SetYear` для значения `Date`, `SetYear` сможет обновить поле `year` структуры `Date`, хотя это поле и не экспортируется. Экспортируемый метод `SetMonth` может обновить неэкспортируемое поле `month` и т. д.

Изменив *main.go* для использования set-методов, мы сможем обновить поля значения `Date`:

main.go

```
package main

import (
        "fmt"
        "github.com/headfirstgo/calendar"
        "log"
)

func main() {
        date := calendar.Date{}
        err := date.SetYear(2019)
        if err != nil {
                log.Fatal(err)
        }
        err = date.SetMonth(5)
        if err != nil {
                log.Fatal(err)
        }
        err = date.SetDay(27)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(date)
}
```

Используется set-метод.

Используется set-метод.

Используется set-метод.

```
Shell Edit View Window Help
$ cd temp
$ go run main.go
{2019 5 27}
```

Поля можно обновлять set-методами!

Неэкспортируемые переменные, поля структур, функции и методы остаются доступными для экспортируемых функций и методов в том же пакете.

Обращения к неэкспортируемым полям через экспортируемые методы (продолжение)

Если обновить *main.go* для вызова `SetYear` с недопустимым значением, то при запуске вы получите сообщение об ошибке:

main.go

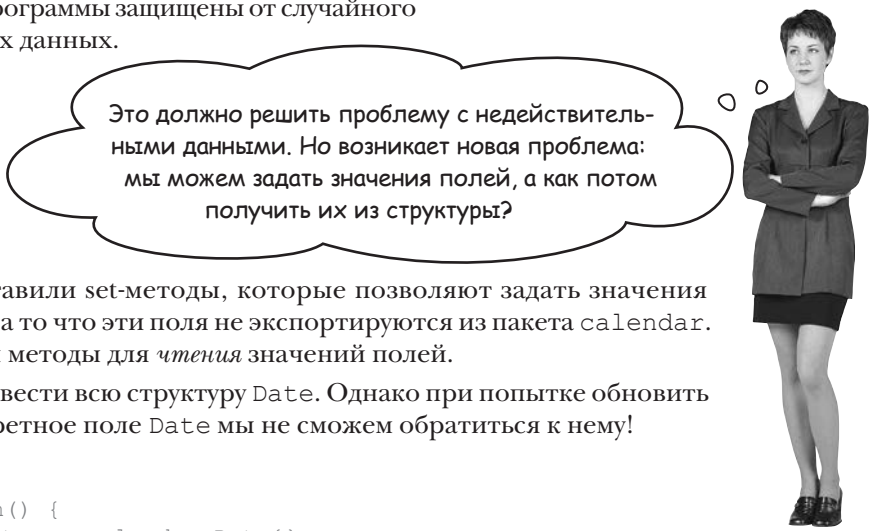
```
func main() {
        date := calendar.Date{}
        err := date.SetYear(0)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(date)
}
```

Вызов set-метода с недопустимым значением.

Сообщает о недопустимом значении!

```
Shell Edit View Window Help
$ cd temp
$ go run main.go
2018/03/23 19:20:17 invalid year
exit status 1
```

Теперь поля значения `Date` могут обновляться только set-методами, поэтому программы защищены от случайного ввода недействительных данных.



Да, верно. Мы предоставили set-методы, которые позволяют задать значения полей `Date`, несмотря на то что эти поля не экспортируются из пакета `calendar`. Но мы не предоставили методы для *чтения* значений полей.

Можно попробовать вывести всю структуру `Date`. Однако при попытке обновить *main.go* и вывести конкретное поле `Date` мы не сможем обратиться к нему!

main.go

```
func main() {
        date := calendar.Date{}
        err := date.SetYear(2019)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(date.year)
}
```

Присваивается допустимый год.

Пытаемся вывести поле year.

Происходит ошибка, так как поле не экспортировано!

```
Shell Edit View Window Help
$ cd temp
$ go run main.go
# command-line-arguments
./main.go:16:18: date.year undefined
(cannot refer to unexported field or method year)
```

Get-методы

Как вы уже знаете, методы, предназначенные для *присваивания* значения поля структуры или переменной, называются *set-методами*. Как и следовало ожидать, методы, предназначенные для *получения* значения поля структуры или переменной, называются **get-методами**, или геттерами.

По сравнению с set-методами добавить get-методы в тип `Date` будет проще. При вызове им не нужно ничего делать, кроме как вернуть значение поля.

По общепринятым соглашениям имя get-метода должно совпадать с именем поля или переменной, к которой он обращается. (Конечно, чтобы метод экспортировался, его имя должно начинаться с буквы верхнего регистра.) Таким образом, типу `Date` понадобится метод `Year` для обращения к полю `year`, метод `Month` для обращения к полю `month`, а метод `Day` для поля `day`.

Get-методам вообще не нужно изменять получателя, поэтому в качестве получателя *можно было* использовать непосредственное значение `Date`. Но если любой метод типа получает указатель на получателя, согласно общепринятым соглашениям *все* методы должны получать указатель для предотвращения путаницы. Так как set-методы используют указатель на получателя, get-методы тоже должны использовать указатель.

После внесения всех изменений в *date.go* обновим файл *main.go*: сначала он задает значения всех полей `Date`, а затем выводит их при помощи get-методов.

main.go

```
// Директивы package и import пропущены
func main() {
       date := calendar.Date{}
       err := date.SetYear(2019)
       if err != nil {
              log.Fatal(err)
       }
       err = date.SetMonth(5)
       if err != nil {
              log.Fatal(err)
       }
       err = date.SetDay(27)
       if err != nil {
              log.Fatal(err)
       }
       fmt.Println(date.Year())
       fmt.Println(date.Month())
       fmt.Println(date.Day())
}
```

date.go

```
package calendar

import "errors"

type Date struct {
       year  int
       month int
       day   int
}

func (d *Date) Year() int {
       return d.year
}
func (d *Date) Month() int {
       return d.month
}
func (d *Date) Day() int {
       return d.day
}
// Set-методы пропущены
```

Используется тип указателя на получателя, для согласованности с set-методами.

Имя совпадает с именем поля (но начинается с буквы верхнего регистра, чтобы поле экспортировалось).

Возвращает значение поля.

```
Shell Edit View Window Help
$ cd temp
$ go run main.go
2019
5
27
```

Значения, возвращаемые get-методами.

Инкапсуляция

Практика сокрытия данных в одной части программы от кода в другой части называется **инкапсуляцией**. Механизм инкапсуляции поддерживается не только в Go. Инкапсуляция полезна прежде всего тем, что помогает защититься от некорректных данных (как вы уже видели). Кроме того, разработчик может изменять инкапсулированную часть программы, не рискуя нарушить работоспособность другого кода, который обращается к этой части из-за возможности прямого доступа.

Многие другие языки программирования инкапсулируют данные в классах. (На концептуальном уровне классы похожи на типы Go, но не идентичны им.) В Go данные инкасулируются в пакетах с применением неэкспортируемых переменных, полей структур, функций или методов.

В других языках инкапсуляция применяется намного чаще, чем в Go. Например, в некоторых языках принято определять get- и set-методы для каждого поля, даже если к нему можно обратиться напрямую. Разработчики обычно применяют инкапсуляцию только при необходимости — например, когда требуется проверить данные поля set-методами. В языке Go, если вы не видите явной необходимости в инкапсуляции поля, обычно бывает проще экспортировать это поле и предоставить прямой доступ к нему.

Часто Задаваемые Вопросы

В: Многие другие языки не разрешают обращаться к инкапсулированным значениями за пределами класса, в котором они определяются. Go разрешает другому коду из того же пакета обращаться к неэкспортируемым полям. Насколько это безопасно?

О: Как правило, весь код пакета пишется одним разработчиком (или группой разработчиков). Кроме того, весь код пакета обычно предназначен для одной цели. Скорее всего, авторам кода из пакета потребуется доступ к неэкспортируемым данным, поэтому с большой вероятностью они будут работать с этими данными корректно. Таким образом, взаимодействие остального кода пакета с неэкспортированными данными обычно безопасно.

С другой стороны, код *за пределами* пакета с большой вероятностью будет написан *другими* разработчиками, но это нормально — ведь неэкспортированные поля скрыты от них, и они не могут случайно изменить их недействительными значениями.

В: Я видел другие языки, в которых имена всех get-методов начинаются с префикса «`Get`» — например, `GetName`, `GetCity` и т. д. А в Go это возможно?

О: Язык Go позволит вам это сделать, но так поступать не стоит. Сообщество Go выработало соглашения, по которым префикс `Get` не указывается в именах get-методов. Включение префикса только запутает ваших коллег-разработчиков!

Для set-методов в Go используется префикс Set, как и во многих других языках, потому что позволяет отличить set-методы от get-методов для того же поля.

Наберитесь терпения — для кода этого упражнения понадобятся две страницы...
Заполните пропуски и внесите следующие изменения в тип `Coordinates`:

- Обновите поля типа, чтобы они не экспортировались.
- Добавьте get-метод для каждого поля. (Не забудьте об общепринятых соглашениях: имя get-метода должно совпадать с именем поля, к которому он обращается, и начинаться в буквы верхнего регистра, если метод должен экспортироваться.)
- Добавьте проверку данных в set-методы. Метод `SetLatitude` должен возвращать ошибку, если переданное значение меньше `-90` или больше `90`. Метод `SetLongitude` должен возвращать ошибку, если новое значение меньше `-180` или больше `180`.


coordinates.go

```
package geo

import "errors"

type Coordinates struct {
       _________  float64
       _________ float64
}

func (c *Coordinates) _________() _______ {
       return c.latitude
}

func (c *Coordinates) _________() _______ {
       return c.longitude
}

func (c *Coordinates) SetLatitude(latitude float64) _____ {
       if latitude < -90 || latitude > 90 {
              return __________("invalid latitude")
       }
       c.latitude = latitude
       return ___
}
func (c *Coordinates) SetLongitude(longitude float64) _____ {
       if longitude < -180 || longitude > 180 {
              return __________("invalid longitude")
       }
       c.longitude = longitude
       return ___
}
```

Упражнение *(Продолжение)*

Теперь измените код пакета `main`, чтобы в нем использовался обновленный тип `Coordinates`.

- Для каждого вызова set-метода сохраните возвращаемое значение `error`.
- Если значение `error` отлично от `nil`, вызовите функцию `log.Fatal`, чтобы вывести сообщение об ошибке и завершить программу.
- Если присваивание значений обошлось без ошибок, вызовите оба get-метода для вывода значений полей.

Законченный код должен выводить показанный результат. (Вызов `SetLatitude` должен быть успешным, но `SetLongitude` передается недопустимое значение, поэтому в этот момент программа должна вывести сообщение об ошибке и завершиться.)

main.go

```
package main

import (
        "fmt"
        "geo"
        "log"
)

func main() {
        coordinates := geo.Coordinates{}
        ___ := coordinates.SetLatitude(37.42)
        if err != ___ {
                log.Fatal(err)
        }
        err = coordinates.SetLongitude(-1122.08)
        if err != ___ {
                log.Fatal(err)
        }
        fmt.Println(coordinates._________())
        fmt.Println(coordinates._________())
}
```

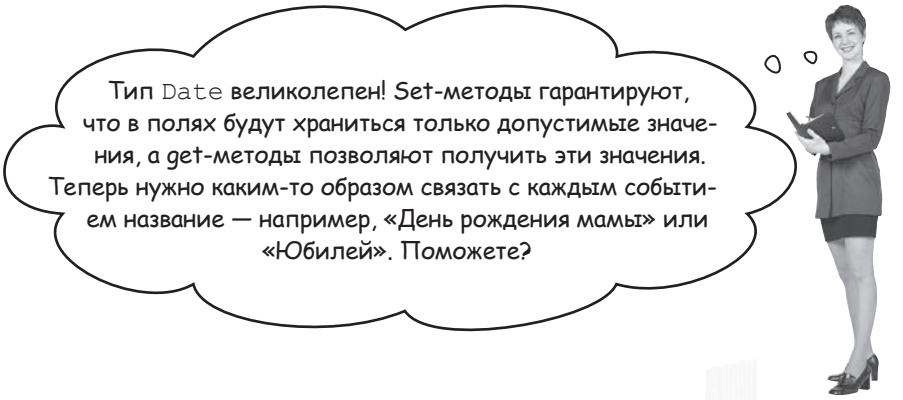
(Недопустимое значение!)

Результат.

```
2018/03/23 20:12:49 invalid longitude
exit status 1
```

Ответ на с. 352.

Встраивание типа Date в тип Event



Особых сложностей быть не должно. Помните, как мы встроили структуру `Address` в два других типа структур в главе 8?

Тип `Address` считается «встроенным», потому что мы использовали анонимное поле (поле, у которого нет имени, а только тип) в другой структуре для его хранения. В результате поля `Address` были повышены до внешней структуры, и мы могли обращаться к полям внутренней структуры так, словно они принадлежат внешней структуре.

```
package magazine

type Subscriber struct {
        Name   string
        Rate   float64
        Active bool
        Address
}

type Employee struct {
        Name   string
        Salary float64
        Address
}

type Address struct {
        // ...
}
```

Установите поля Address так, как если бы они были определены на Subscriber.

```
subscriber.Street = "123 Oak St"
subscriber.City = "Omaha"
subscriber.State = "NE"
subscriber.PostalCode = "68111"
```

ваша рабочая область > **src** > **github.com** > **headfirstgo** > **calendar** > **event.go**

Поскольку эта стратегия отлично работала ранее, давайте определим тип `Event`, в котором `Date` встраивается в виде анонимного поля.

Создайте в папке пакета `calendar` другой файл с именем *event.go*. (Код можно разместить в существующем файле *date.go*, но такая структура получается более элегантной.) В этом файле определите тип `Event` с двумя полями: полем `Title` типа `string` и анонимным полем `Date`.

```
package calendar

type Event struct {
        Title string
        Date
}
```

Значение Date встраивается в виде анонимного поля.

Неэкспортируемые поля не повышаются

Тем не менее встраивание `Date` в тип `Event` *не приводит* к повышению полей `Date` до `Event`. Поля `Date` не экспортируются, а Go не повышает неэкспортируемые поля до внешнего типа. И это логично — мы инкапсулировали поля для того, чтобы к ним можно было обращаться только через set- и get-методы, и не хотим, чтобы инкапсуляцию можно было обойти из-за повышения полей.

event.go

```
package calendar

type Event struct {
        Title string
        Date
}
```

Встраивание с использованием анонимного поля.

В нашем пакете `main` при попытке присвоить значение поля `month` структуры `Date` через внешнюю структуру `Event` произойдет ошибка:

main.go

```
package main

import "github.com/headfirstgo/calendar"

func main() {
       event := calendar.Event{}
       event.month = 5
}
```

Неэкспортируемые поля Date не повышаются до Event!

Ошибка.

```
event.month undefined (type calendar.Event has no field or method month)
```

И конечно, сцепление операторов «точка» для получения поля `Date` и последующего обращения к его полям тоже не сработает. К неэкспортируемым полям значения `Date` невозможно обратиться как для структуры по отдельности, так и в том случае, когда она является частью `Event`.

main.go

```
func main() {
       event := calendar.Event{}
       event.Date.year = 2019
}
```

К полям Date невозможно обратиться напрямую при работе со структурой Date!

Ошибка.

```
event.Date.year undefined (cannot refer to unexported field or method year)
```

Означает ли это, что мы не сможем обратиться к полям типа `Date`, если он встроен в тип `Event`? Не беспокойтесь, есть другой способ!

Экспортируемые методы повышаются так же, как и поля

Если тип с экспортируемыми методами встраивается в тип структуры, его методы повышаются до внешнего типа; это означает, что эти методы можно вызвать так, как если бы они были определены для внешнего типа. (Помните, что при встраивании одного типа структуры в другой поля внутренней структуры повышаются до уровня внешней? Здесь то же самое, но с методами вместо полей.)

В следующем пакете определяются два типа. `MyType` — тип структуры, в который второй тип, `EmbeddedType`, встраивается как анонимное поле.

```
package mypackage

import "fmt"

type MyType struct {
       EmbeddedType
}

type EmbeddedType string

func (e EmbeddedType) ExportedMethod() {
       fmt.Println("Hi from ExportedMethod on EmbeddedType")
}

func (e EmbeddedType) unexportedMethod() {
}
```

Эти типы находятся в отдельном пакете.

MyType объявляется как тип структуры.

EmbeddedType встраивается в MyType.

Объявление встраиваемого типа (неважно, является ли он структурой).

Этот метод будет повышен до MyType.

Этот метод повышаться не будет.

Так как `EmbeddedType` определяет экспортируемый метод (с именем `ExportedMethod`), этот метод повышается до `MyType`, и его можно будет вызывать для значений `MyType`.

```
package main

import "mypackage"

func main() {
       value := mypackage.MyType{}
       value.ExportedMethod()
}
```

Вызывается метод, повышенный с уровня EmbeddedType.

```
Hi from ExportedMethod on EmbeddedType
```

Как и в случае с неэкспортируемыми полями, неэкспортируемые методы *не* повышаются. При попытке вызвать такой метод произойдет ошибка.

```
value.unexportedMethod()
```

Попытка вызова неэкспортируемого метода.

Ошибка.

```
value.unexportedMethod undefined (type mypackage.MyType
has no field or method unexportedMethod)
```

Экспортируемые методы повышаются так же, как и поля (продолжение)

Наши поля `Date` не были повышены до типа `Event`, потому что не экспортировались. Однако get- и set-методы типа `Date` *были* экспортированы и *повышаются* до типа `Event`!

Это означает, что мы можем создать значение `Event`, а затем вызвать get- и set-методы типа `Date` непосредственно для `Event`. Именно это и делается в приведенном ниже обновленном коде *main.go*. Как прежде, экспортируемые методы могут обращаться к неэкспортированным полям `Date` за нас.


```
package main

import (
        "fmt"
        "github.com/headfirstgo/calendar"
        "log"
)

func main() {
        event := calendar.Event{}
        err := event.SetYear(2019)
        if err != nil {
                log.Fatal(err)
        }
        err = event.SetMonth(5)
        if err != nil {
                log.Fatal(err)
        }
        err = event.SetDay(27)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(event.Year())
        fmt.Println(event.Month())
        fmt.Println(event.Day())
}
```

Этот set-метод типа Date был повышен до Event.

Этот set-метод типа Date был повышен до Event.

Этот set-метод типа Date был повышен до Event.

Эти get-методы типа Date были повышены до Event.

```
2019
5
27
```

А если вы предпочитаете синтаксис сцепления операторов «точка» для прямого вызова методов значения `Date`, можете сделать так:

Получаем поле Date типа Event, а затем вызываем для него get-методы.

```
fmt.Println(event.Date.Year())
fmt.Println(event.Date.Month())
fmt.Println(event.Date.Day())
```

```
2019
5
27
```

Инкапсуляция поля Title типа Event

Так как поле `Title` структуры `Event` экспортируется, то к нему можно обращаться напрямую:

main.go

```
// Директивы package и imports пропущены
func main() {
        event := calendar.Event{}
        event.Title = "Mom's birthday"
        fmt.Println(event.Title)
}
```

```
Mom's birthday
```

event.go

```
package calendar

type Event struct {
        Title string
        Date
}
```

Экспортируемое поле.

Однако тут мы сталкиваемся с теми же проблемами, которые возникали с полями `Date`. Например, длина строки `Title` не ограничена:

main.go

```
func main() {
        event := calendar.Event{}
        event.Title = "An extremely long title that is impractical to print"
        fmt.Println(event.Title)
}
```

```
An extremely long title that is impractical to print
```

Пожалуй, поле `title` тоже стоит инкапсулировать, чтобы можно было проверить новые значения. Ниже приведена новая версия типа `Event`, которая делает именно это. Имя поля изменяется на `title`, чтобы оно не экспортировалось, а затем добавляются get- и set-методы. Чтобы проверить, не содержит ли строка слишком много рун (символов), мы используем функцию `RuneCountInString` из пакета `unicode/utf8`.

event.go

```
package calendar

import (
        "errors"
        "unicode/utf8"
)

type Event struct {
        title string
        Date
}

func (e *Event) Title() string {
        return e.title
}
func (e *Event) SetTitle(title string) error {
        if utf8.RuneCountInString(title) > 30 {
                return errors.New("invalid title")
        }
        e.title = title
        return nil
}
```

Этот пакет добавляется для создания значений ошибок.

Добавляем пакет для подсчета количества рун в строке.

Поле изменяется на неэкспортируемое.

Get-метод.

Необходимо использовать указатель.

Set-метод.

Если поле title содержит более 30 символов, возвращается ошибка.

Повышенные методы существуют наряду с методами внешнего типа

После добавления set- и get-методов для поля `title` наши программы будут сообщать об ошибке, если используемое значение `title` длиннее 30 символов. При попытке присваивания 39-символьного значения `title` возвращается признак ошибки:


main.go

```
// Директивы package и imports пропущены
func main() {
        event := calendar.Event{}
        err := event.SetTitle("An extremely long and impractical title")
        if err != nil {
                log.Fatal(err)
        }
}
```

```
2018/03/23 20:44:17 invalid title
exit status 1
```

Методы `Title` и `SetTitle` типа `Event` существуют наряду с методами, повышенными из встроенного типа `Date`. Сторона, импортирующая пакет `calendar`, может рассматривать все методы как принадлежащие типу `Event`, не беспокоясь о том, в каком типе они фактически определены.


main.go

```
// Директивы package и imports пропущены
func main() {
        event := calendar.Event{}
        err := event.SetTitle("Mom's birthday")   <-- Определяется для типа Event.
        if err != nil {
                log.Fatal(err)
        }
        err = event.SetYear(2019)                 <-- Повышается из Date.
        if err != nil {
                log.Fatal(err)
        }
        err = event.SetMonth(5)                   <-- Повышается из Date.
        if err != nil {
                log.Fatal(err)
        }
        err = event.SetDay(27)                    <-- Повышается из Date.
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(event.Title())                <-- Определяется для типа Event.
        fmt.Println(event.Year())                 <-- Определяется для типа Date.
        fmt.Println(event.Month())                <-- Определяется для типа Date.
        fmt.Println(event.Day())                  <-- Определяется для типа Date.
}
```

```
Mom's birthday
2019
5
27
```

Пакет calendar готов!

Теперь мы можем вызывать методы `Title` и `SetTitle` прямо для `Event`, а методы для присваивания year, month и day так, как если бы они принадлежали `Event`. На самом деле они определяются для `Date`, но нам не обязательно об этом помнить. Наша работа завершена!

Повышение методов позволяет легко использовать методы одного типа так, словно они принадлежат другому типу. Этот механизм позволяет создавать типы, объединяющие методы нескольких других типов. В результате вы получаете чистый код, не жертвуя удобством!

Упражнение

Код типа `Coordinates` из предыдущего примера завершен. На этот раз никакие изменения вносить не придется, мы приводим его только для справки. На следующей странице мы встроим его в тип `Landmark` (который также встречался вам в главе 8), чтобы его методы повышались до уровня `Landmark`.

coordinates.go

```
package geo

import "errors"

type Coordinates struct {
       latitude  float64
       longitude float64
}

func (c *Coordinates) Latitude() float64 {
       return c.latitude
}
func (c *Coordinates) Longitude() float64 {
       return c.longitude
}

func (c *Coordinates) SetLatitude(latitude float64) error {
       if latitude < -90 || latitude > 90 {
              return errors.New("invalid latitude")
       }
       c.latitude = latitude
       return nil
}

func (c *Coordinates) SetLongitude(longitude float64) error {
       if longitude < -180 || longitude > 180 {
              return errors.New("invalid longitude")
       }
       c.longitude = longitude
       return nil
}
```

Упражнение (Продолжение)

Ниже приведена обновленная версия типа `Landmark` (из главы 8). Мы хотим, чтобы его поле `name` было инкапсулировано, а обращения могли осуществляться исключительно через get-метод `Name` и set-метод `SetName`. Метод `SetName` возвращает ошибку, если его аргумент является пустой строкой, или задает поле `name` с возвращением ошибки `nil` в противном случае. Тип `Landmark` также должен содержать анонимное поле `Coordinates`, чтобы методы `Coordinates` повышались до уровня `Landmark`.

Заполните пропуски и завершите код типа `Landmark`.

landmark.go

```
package geo

import "errors"

type Landmark struct {
        _____ string
        ___________
}

func (l *Landmark) _____() string {
        return l.name
}

func (l *Landmark) _______(name string) error {
        if name == "" {
                return errors.New("invalid name")
        }
        l.name = name
        return nil
}
```

Если пропуски в коде `Landmark` заполнены правильно, код пакета `main` должен выполняться и выводить показанный результат.

main.go

```
package main
// Директивы imports пропущены
func main() {
        location := geo.Landmark{}
        err := location.SetName("The Googleplex")
        if err != nil {
                log.Fatal(err)
        }
        err = location.SetLatitude(37.42)
        if err != nil {
                log.Fatal(err)
        }
        err = location.SetLongitude(-122.08)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(location.Name())
        fmt.Println(location.Latitude())
        fmt.Println(location.Longitude())
}
```

Результат.

```
The Googleplex
37.42
-122.08
```

Ответ на с. 354.

Ваш инструментарий Go

Глава 10 осталась позади! В ней ваш инструментарий пополнился инкапсуляцией и встраиванием.

ГЛАВА 10

Инкапсуляция

Инкапсуляция — практика сокрытия данных в одной части программы от кода в другой части.

Инкапсуляция может использоваться для защиты от недействительных данных.

Инкапсулированные данные проще изменять. Вы можете быть уверены в том, что это не нарушит работоспособности кода, обращающегося к этим данным.

Встраивание

Тип, хранимый в типе структуры с использованием анонимного поля, называется встроенным в структуру.

Методы встроенного типа повышаются до внешнего типа. Они могут вызываться так, как если бы были определены для внешнего типа.

КЛЮЧЕВЫЕ МОМЕНТЫ

- В Go данные инкапсулируются в пакетах с помощью неэкспортированных переменных пакетов или полей структур.
- К неэкспортированным переменным, полям структур, функциям, методам и т. д. можно обращаться из экспортируемых функций и методов, определенных в том же пакете.
- Практика проверки действительности данных перед их сохранением называется **проверкой данных**.
- Метод, используемый в основном для задания значения неинкапсулируемого поля, называется **set-методом**. Set-методы часто включают логику проверки данных, которая гарантирует, что присваиваемое значение будет допустимым.
- Так как set-методы должны изменять своего получателя, их параметр получателя должен иметь тип указателя.
- Традиционно set-методам присваиваются имена вида `Set`*X*, где *X* — имя присваиваемого поля.
- Метод, предназначенный для получения значения инкапсулированного поля, называется **get-методом**.
- Имена get-методов традиционно задаются в форме *X*, где *X* — имя поля. В некоторых других языках программирования для get-методов была выбрана форма `Get`*X*, но в Go эту форму использовать *не рекомендуется*.
- Методы, определяемые для внешнего типа структуры, существуют на одном уровне с методами, повышенными от встроенного типа.
- Неэкспортируемые методы встроенного типа не повышаются до уровня внешнего типа.

Упражнение
Решение

Необходимо добавить в тип `Coordinates` set-методы для каждого из его полей. Заполните пропуски в файле *coordinates.go*, чтобы код *main.go* выполнялся и выводил показанный результат.

```
package geo

type Coordinates struct {
        Latitude  float64
        Longitude float64
}

func (c *Coordinates) SetLatitude(latitude float64) {
        c.Latitude = latitude
}

func (c *Coordinates) SetLongitude(longitude float64) {
        c.Longitude = longitude
}
```

Должен использоваться указатель типа, чтобы мы могли изменить получатель.

Должен использоваться указатель типа, чтобы мы могли изменить получатель.

coordinates.go

```
package main

import (
        "fmt"
        "geo"
)

func main() {
        coordinates := geo.Coordinates{}
        coordinates.SetLatitude(37.42)
        coordinates.SetLongitude(-122.08)
        fmt.Println(coordinates)
}
```

main.go

Результат.

```
{37.42 -122.08}
```

Ваша задача при обновлении этого кода — инкапсуляция полей типа `Coordinates` и добавление проверки данных в его set-методах.

- Обновите поля типа, чтобы они не экспортировались.
- Добавьте get-метод для каждого поля.

Добавьте проверку данных в set-методы. Метод `SetLatitude` должен возвращать ошибку, если переданное значение меньше `-90` или больше `90`. Метод `SetLongitude` должен возвращать ошибку, если новое значение меньше `-180` или больше `180`.


coordinates.go

```
package geo

import "errors"

type Coordinates struct {
        latitude  float64
        longitude float64
}

func (c *Coordinates) Latitude() float64 {
        return c.latitude
}

func (c *Coordinates) Longitude() float64 {
        return c.longitude
}

func (c *Coordinates) SetLatitude(latitude float64) error {
        if latitude < -90 || latitude > 90 {
                return errors.New("invalid latitude")
        }
        c.latitude = latitude
        return nil
}

func (c *Coordinates) SetLongitude(longitude float64) error {
        if longitude < -180 || longitude > 180 {
                return errors.New("invalid longitude")
        }
        c.longitude = longitude
        return nil
}
```

Поля должны быть неэкспортируемыми.

Имена get-методов с буквы верхнего регистра.

Тот же тип, что и в поле.

Имена get-методов с буквы верхнего регистра.

Тот же тип, что и в поле.

Необходимо вернуть тип ошибки.

Возвращает новое значение ошибки.

Возвращает nil, если нет ошибки.

Необходимо вернуть тип ошибки.

Возвращает новое значение ошибки.

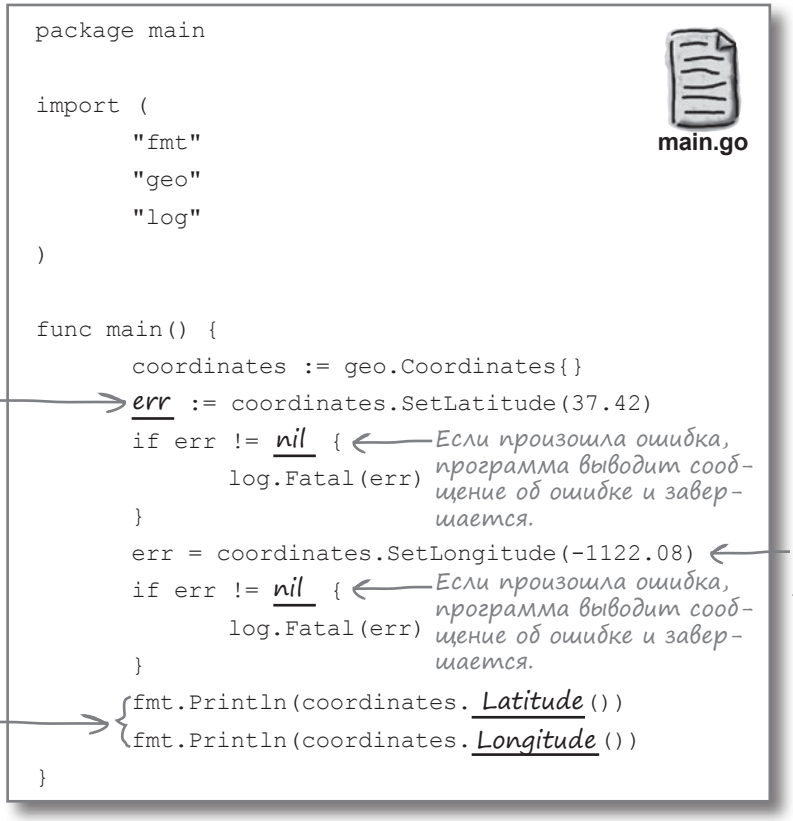
Возвращает nil, если нет ошибки.

Упражнение
Решение
(Продолжение)

Следующей задачей при обновлении кода пакета `main` было использование кода пакета, чтобы в нем использовался обновленный тип `Coordinates`.

- Для каждого вызова set-метода сохраните возвращаемое значение `error`.
- Если значение `error` отлично от `nil`, вызовите функцию `log.Fatal`, чтобы вывести сообщение об ошибке и завершить программу.
- Если присваивание значений обошлось без ошибок, вызовите оба get-метода для вывода значений полей.

Как показано ниже, вызов `SetLatitude` был успешным, но `SetLongitude` передается недопустимое значение, поэтому в этот момент программа выводит сообщение об ошибке и завершается.




```
package main

import (
        "fmt"
        "geo"
        "log"
)

func main() {
        coordinates := geo.Coordinates{}
        err := coordinates.SetLatitude(37.42)
        if err != nil {
                log.Fatal(err)
        }
        err = coordinates.SetLongitude(-1122.08)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(coordinates.Latitude())
        fmt.Println(coordinates.Longitude())
}
```

```
2018/03/23 20:12:49 invalid longitude
exit status 1
```

Ниже приведена обновленная версия типа `Landmark` (из главы 8.) Мы хотим, чтобы его поле `name` было инкапсулировано, а обращения могли осуществляться исключительно через get-метод `Name` и set-метод `SetName`. Метод `SetName` возвращает ошибку, если его аргумент является пустой строкой, или задает поле `name` с возвращением ошибки `nil` в противном случае. Тип `Landmark` также должен содержать анонимное поле `Coordinates`, чтобы методы `Coordinates` повышались до уровня `Landmark`.


```
package geo

import "errors"

type Landmark struct {
        name string
        Coordinates
}

func (l *Landmark) Name() string {
        return l.name
}
func (l *Landmark) SetName(name string) error {
        if name == "" {
                return errors.New("invalid name")
        }
        l.name = name
        return nil
}
```

Поле name не должно экспортироваться, чтобы оно было инкапсулировано.

Встраивание с помощью анонимного поля.

То же имя, что и в поле (но экспортируемое).

То же имя, что и в поле, но с префиксом Set.


```
package main
// Директивы imports пропущены
func main() {
        location := geo.Landmark{}
        err := location.SetName("The Googleplex")
        if err != nil {
                log.Fatal(err)
        }
        err = location.SetLatitude(37.42)
        if err != nil {
                log.Fatal(err)
        }
        err = location.SetLongitude(-122.08)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(location.Name())
        fmt.Println(location.Latitude())
        fmt.Println(location.Longitude())
}
```

Создаем значение Landmark.

Определяется для самого типа Landmark.

Повышается от Coordinates.

Повышается от Coordinates.

Определяется для Landmark.

Повышается от Coordinates.

Результат.

```
The Googleplex
37.42
-122.08
```

Интерфейсы

Иногда конкретный тип значения не важен. Вас не интересует, с чем вы работаете. Вы просто хотите быть уверены в том, что оно может делать то, что нужно вам. Тогда вы сможете вызывать для значения *определенные методы*. Неважно, с каким значением вы работаете — `Pen` или `Pencil`; вам просто нужно нечто, содержащее метод `Draw`. Именно эту задачу решают **интерфейсы** в языке Go. Они позволяют определять переменные и параметры функций, которые могут хранить *любой* тип при условии, что этот тип определяет некоторые методы.

Два разных типа с одинаковыми методами

Помните кассетные магнитофоны? (Хотя, наверное, некоторые читатели их уже не застали.) Это были полезные устройства. Они позволяли легко записать на пленку все ваши любимые песни — даже созданные разными исполнителями. Конечно, магнитофоны были слишком громоздкими, чтобы постоянно носить их с собой. Если вам хотелось взять кассеты в дорогу, обычно для этого использовались плееры на батарейках. Плееры обычно не поддерживали возможности записи. Ах, как же это было здорово — создавать собственные миксы и обмениваться ими с друзьями!

Ностальгия так захватила нас, что мы создали пакет `gadget` для оживления воспоминаний. В него входит тип `TapeRecorder`, представляющий кассетный магнитофон, и тип `TapePlayer`, представляющий плеер.

Тип `TapePlayer` содержит метод `Play`, моделирующий воспроизведение трека, и метод `Stop` для остановки виртуального воспроизведения.

Тип `TapeRecorder` содержит методы `Play` и `Stop`, а также метод `Record`.

```
package gadget

import "fmt"

type TapePlayer struct {
      Batteries string
}
func (t TapePlayer) Play(song string) {
      fmt.Println("Playing", song)
}
func (t TapePlayer) Stop() {
      fmt.Println("Stopped!")
}

type TapeRecorder struct {
      Microphones int
}
func (t TapeRecorder) Play(song string) {
      fmt.Println("Playing", song)
}
func (t TapeRecorder) Record() {
      fmt.Println("Recording")
}
func (t TapeRecorder) Stop() {
      fmt.Println("Stopped!")
}
```

Содержит метод Play, как у TapePlayer.

Содержит метод Stop, как у TapePlayer.

В параметре метода может передаваться только один тип

Ниже приведен пример программы, использующей пакет `gadget`. Мы определяем функцию `playList`, которая получает значение `TapePlayer` и сегмент названий воспроизводимых песен. Функция перебирает все названия в сегменте и передает их методу `Play` значения `TapePlayer`. Завершив воспроизведение списка, функция вызывает `Stop` для `TapePlayer`.

Затем в методе `main` остается создать значение `TapePlayer` и сегмент названий песен и передать их `playList`.

```
package main

import "github.com/headfirstgo/gadget"

func playList(device gadget.TapePlayer, songs []string) {
        for _, song := range songs {
                device.Play(song)
        }
        device.Stop()
}

func main() {
        player := gadget.TapePlayer{}
        mixtape := []string{"Jessie's Girl", "Whip It", "9 to 5"}
        playList(player, mixtape)
}
```

Импортируем пакет.

Перебираем все песни в цикле.

Воспроизведение текущей песни.

Плеер останавливается после завершения.

Создание TapePlayer.

Создается сегмент с названиями песен.

Песни воспроизводятся при помощи TapePlayer.

```
Playing Jessie's Girl
Playing Whip It
Playing 9 to 5
Stopped!
```

Функция `playList` отлично работает со значениями `TapePlayer`. Казалось бы, она также должна работать с `TapeRecorder`. (В конце концов, магнитофон по сути представляет собой плеер с дополнительной функцией записи.) Но первый параметр `playList` имеет тип `TapePlayer`. Попробуем передать аргумент любого другого типа, и сразу получим ошибку компиляции:

```
func main() {
        player := gadget.TapeRecorder{}
        mixtape := []string{"Jessie's Girl", "Whip It", "9 to 5"}
        playList(player, mixtape)
}
```

Создаем значение TapeRecorder вместо TapePlayer.

TapeRecorder передается функции playList.

Ошибка.

```
cannot use player (type gadget.TapeRecorder)
as type gadget.TapePlayer in argument to playList
```

В параметре метода может передаваться только один тип (продолжение)

Плохо... На самом деле функции playList необходимо значение, тип которого определяет методы Play и Stop. Эти методы есть как у TapePlayer, так и у TapeRecorder!

```
func playList(device gadget.TapePlayer, songs []string) {
       for _, song := range songs {
              device.Play(song)
       }
       device.Stop()
}

type TapePlayer struct {
       Batteries string
}
func (t TapePlayer) Play(song string) {
       fmt.Println("Playing", song)
}
func (t TapePlayer) Stop() {
       fmt.Println("Stopped!")
}

type TapeRecorder struct {
       Microphones int
}
func (t TapeRecorder) Play(song string) {
       fmt.Println("Playing", song)
}
func (t TapeRecorder) Record() {
       fmt.Println("Recording")
}
func (t TapeRecorder) Stop() {
       fmt.Println("Stopped!")
}
```

Здесь должно быть значение, содержащее метод Play со строковым параметром.

Здесь должно быть значение, содержащее метод Stop без параметров.

TapePlayer содержит метод Play со строковым параметром.

TapePlayer содержит метод Stop без параметров.

TapePlayer тоже содержит метод Play со строковым параметром.

TapePlayer также содержит метод Stop без параметров.

Похоже, в данном случае безопасность типов языка Go не столько помогает, сколько мешает. Тип TapeRecorder определяет все методы, необходимые функции playList, но мы не можем использовать его, так как playList принимает только значения TapePlayer.

Что же делать? Может, написать вторую, почти идентичную функцию playListWithRecorder, которая получает значение TapeRecorder?

Однако Go предлагает и другое решение...

Интерфейсы

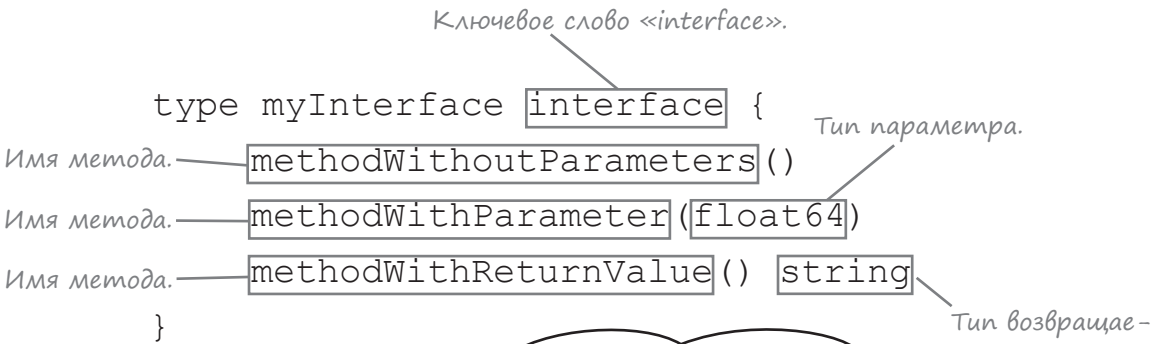
Когда вы устанавливаете программу на свой компьютер, разумно ожидать, что эта программа предоставит какие-то средства для взаимодействия с ней. Текстовый редактор предоставит место для ввода текста. Программа архивации — возможность выбрать сохраняемые файлы. В электронной таблице есть инструменты для вставки столбцов и строк данных. Набор средств, предоставляемых программой для взаимодействия с ней, часто называется ее *интерфейсом*.

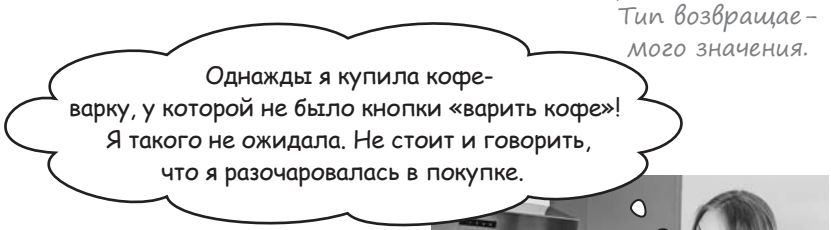
Задумывались вы об этом или нет, можно ожидать, что значения Go тоже предоставят средства для взаимодействия с ними. Как вы чаще всего взаимодействуете со значениями Go? Пожалуй, через их методы.

В Go **интерфейс** определяется как набор методов, которые должны поддерживаться некоторыми значениями. Таким образом, интерфейс представляет набор действий, выполняемых с использованием типа.

Определение типа интерфейса состоит из ключевого слова `interface`, за ним следуют фигурные скобки со списком имен методов, а также параметрами и возвращаемыми значениями, которые должны иметь эти методы.

Интерфейс — набор методов, который должен поддерживаться некоторыми значениями.





Любой тип, который содержит все методы, перечисленные в определении интерфейса, называется **поддерживающим** этот интерфейс. Тип, поддерживающий интерфейс, может использоваться в любом месте, где должен использоваться этот интерфейс.

Имена методов, типы параметров (если они есть) и типы возвращаемых значений (если они есть) должны совпадать с определениями в интерфейсе. Тип может содержать методы *помимо* тех, которые перечислены в интерфейсе, но в нем не могут *отсутствовать* такие методы, иначе тип не будет поддерживать интерфейс.

Тип может поддерживать несколько интерфейсов, а интерфейс может (и обычно должен) поддерживаться несколькими типами.

Определение типа, поддерживающего интерфейс

Приведенный ниже код создает небольшой экспериментальный пакет с именем `mypkg`. Он определяет тип интерфейса с именем `MyInterface`, содержащий три метода. Затем определяет тип с именем `MyType`, поддерживающий `MyInterface`.

Для поддержки `MyInterface` необходимы три метода: `MethodWithoutParameters`, `MethodWithParameter` с параметром `float64` и `MethodWithReturnValue`, возвращающий `string`.

Затем объявляется другой тип `MyType`. Базовый тип `MyType` в данном случае неважен; используется `int`. Мы определили все методы `MyType`, необходимые для поддержки `MyInterface`, а также один дополнительный метод, который не является частью интерфейса.


```
package mypkg

import "fmt"

type MyInterface interface {
       MethodWithoutParameters()
       MethodWithParameter(float64)
       MethodWithReturnValue() string
}
type MyType int

func (m MyType) MethodWithoutParameters() {
       fmt.Println("MethodWithoutParameters called")
}
func (m MyType) MethodWithParameter(f float64) {
       fmt.Println("MethodWithParameter called with", f)
}
func (m MyType) MethodWithReturnValue() string {
       return "Hi from MethodWithReturnValue"
}
func (my MyType) MethodNotInInterface() {
       fmt.Println("MethodNotInInterface called")
}
```

Объявление типа интерфейса.

Тип поддерживает этот интерфейс, если содержит этот метод...

...а также этот метод (с параметром float64)...

...и этот метод (с возвращаемым значением string).

Объявление типа, поддерживающего myInterface.

Первый обязательный метод.

Второй обязательный метод (с параметром float64).

Третий обязательный метод (с возвращаемым значением string).

Тип может поддерживать интерфейс даже в том случае, если содержит другие методы, не входящие в этот интерфейс.

Многие другие языки требуют явно указать, что `MyType` поддерживает `MyInterface`. Но в Go это происходит *автоматически*. Если тип содержит все методы, объявленные в интерфейсе, то может находиться в любом месте, где должен использоваться этот интерфейс, без каких-либо дополнительных объявлений.

Определение типа, поддерживающего интерфейс (продолжение)

Ниже приведена простая программа для тестирования `mypkg`.

Переменная, объявленная с типом интерфейса, может содержать любое значение, тип которого поддерживает этот интерфейс. В программе объявляется переменная `value` с типом `MyInterface`, после чего программа создает значение `MyType` и присваивает его `value`. (Это возможно потому, что `MyType` поддерживает `MyInterface`.) Затем вызываются все методы этого значения, являющиеся частью интерфейса.

```
package main

import (
        "fmt"
        "mypkg"
)

func main() {
        var value mypkg.MyInterface
        value = mypkg.MyType(5)
        value.MethodWithoutParameters()
        value.MethodWithParameter(127.3)
        fmt.Println(value.MethodWithReturnValue())
}
```

Объявление переменной с использованием типа интерфейса.

Значения myType поддерживают myInterface, поэтому это значение может быть присвоено переменной с типом myInterface.

Мы можем вызвать любой метод, входящий в myInterface.

```
MethodWithoutParameters called
MethodWithParameter called with 127.3
Hi from MethodWithReturnValue
```

Конкретные типы и типы интерфейсов

Все типы, которые мы определяли в предшествующих главах, были конкретными. **Конкретный тип** определяет не только то, что могут *делать* его значения (то есть какие методы для них можно вызывать), но и то, чем они *являются*: он определяет базовый тип, используемый для хранения данных значения.

Типы интерфейсов не описывают, чем значение *является*: они ничего не говорят о базовом типе или о том, как хранятся его данные. Они только описывают, что значение может *делать*, то есть какие методы оно содержит.

Предположим, вы хотите написать короткую записку. В ящике стола у вас лежат значения нескольких конкретных типов: Карандаш, Ручка и Фломастер. Каждый из этих конкретных типов определяет метод `Write`, так что вас на самом деле не интересует, какой именно тип будет выбран. Вам нужен предмет, которым можно писать: тип интерфейса, который поддерживается любым конкретным типом с методом `Write`.

Тип интерфейса.

«Мне нужно что-то, чем можно писать».

Конкретные типы.

Присваивание любого типа, поддерживающего интерфейс

Если у вас имеется переменная с типом интерфейса, она может принимать значения любого типа, поддерживающего интерфейс.

Допустим, есть два типа, `Whistle` и `Horn`, каждый из которых содержит метод `MakeSound`. Мы можем создать интерфейс `NoiseMaker`, который представляет любой тип с методом `MakeSound`. Если объявить переменную `toy` с типом `NoiseMaker`, ей можно будет присваивать как значения `Whistle`, так и значения `Horn`. (*Или* любых других типов, которые будут объявлены позднее — при условии, что тип содержит метод `MakeSound`.)

Затем можно будет вызвать метод `MakeSound` для любого значения, присвоенного переменной `toy`. И хотя мы точно не знаем, каким конкретным типом *является* значение переменной `toy`, мы знаем, что с ним можно *делать*: вызывать метод `MakeSound`. Если его тип не содержит метод `MakeSound`, то он не поддерживает интерфейс `NoiseMaker`, и значение нельзя будет присвоить переменной.

```
package main

import "fmt"

type Whistle string

func (w Whistle) MakeSound() {
      fmt.Println("Tweet!")
}

type Horn string

func (h Horn) MakeSound() {
      fmt.Println("Honk!")
}

type NoiseMaker interface {
      MakeSound()
}

func main() {
      var toy NoiseMaker
      toy = Whistle("Toyco Canary")
      toy.MakeSound()
      toy = Horn("Toyco Blaster")
      toy.MakeSound()
}
```

Содержит метод MakeSound.

Также содержит метод MakeSound.

Представляет любой тип с методом MakeSound.

Объявляется переменная NoiseMaker.

Присваивает переменной значение типа, поддерживающего интерфейс NoiseMaker.

Присваивает переменной значение другого типа, поддерживающего интерфейс NoiseMaker.

```
Tweet!
Honk!
```

Параметры функций также могут объявляться с типами интерфейсов. (В конце концов, параметры функций — это по сути те же переменные.) Если объявить функцию `play`, которая получает `NoiseMaker`, вы можете передать любое значение типа, содержащего метод `MakeSound`:

```
func play(n NoiseMaker) {
      n.MakeSound()
}

func main() {
      play(Whistle("Toyco Canary"))
      play(Horn("Toyco Blaster"))
}
```

```
Tweet!
Honk!
```

Вызывать можно только методы, определенные как часть интерфейса

После того как значение будет присвоено переменной (или параметру метода) с типом интерфейса, для нее можно будет вызывать *только* те методы, которые определяются интерфейсом.

Предположим, вы создали тип `Robot`, который в дополнение к методу `MakeSound` также содержит метод `Walk`. Мы добавляем вызов `Walk` в функцию `play` и передаем `play` новое значение `Robot`.

Однако код не компилируется: в сообщении об ошибке говорится, что значения `NoiseMaker` не содержат метода `Walk`.

Почему? Значения `Robot` *содержат* метод `Walk`; определение у вас прямо перед глазами!

Но функции `play` передается *не* значение `Robot`, а `NoiseMaker`. Что, если `play` вместо него будет передано значение `Whistle` или `Horn`? У них нет методов `Walk`!

Если у вас имеется переменная с типом интерфейса, то она гарантированно содержит только те методы, которые определены в интерфейсе. И только эти методы компилятор Go позволит вызвать. (Вообще говоря, вы *можете* получить информацию о конкретном типе значения для вызова более специализированных методов. Эта возможность будет рассмотрена позже.)

```
package main

import "fmt"

type Whistle string

func (w Whistle) MakeSound() {
      fmt.Println("Tweet!")
}

type Horn string

func (h Horn) MakeSound() {
      fmt.Println("Honk!")
}

type Robot string

func (r Robot) MakeSound() {
      fmt.Println("Beep Boop")
}
func (r Robot) Walk() {
      fmt.Println("Powering legs")
}

type NoiseMaker interface {
      MakeSound()
}

func play(n NoiseMaker) {
      n.MakeSound()
      n.Walk()
}

func main() {
      play(Robot("Botco Ambler"))
}
```

Объявление нового типа Robot.

Robot поддерживает интерфейс NoiseMaker.

Дополнительный метод.

Можно: часть интерфейса NoiseMaker.

Нельзя: не принадлежит NoiseMaker!

Ошибка.

```
n.Walk undefined
(type NoiseMaker has no
field or method Walk)
```

Обратите внимание: переменной с типом интерфейса *можно* присвоить тип, содержащий *другие* методы. Пока вы не вызываете эти другие методы, все будет работать.

```
func play(n NoiseMaker) {
      n.MakeSound()
}

func main() {
      play(Robot("Botco Ambler"))
}
```

Вызываются только те методы, которые являются частью интерфейса.

```
Beep Boop
```

Сломай и изучи!

Ниже приведена пара конкретных типов, `Fan` и `CoffeePot`. Также имеется интерфейс `Appliance` с методом `TurnOn`.

`Fan` и `CoffeePot` содержат метод `TurnOn`, поэтому оба типа поддерживают интерфейс `Appliance`.

Благодаря этому в функции `main` мы можем определить переменную `Appliance` и присвоить ей значения обоих типов, `Fan` и `CoffeePot`.

Внесите одно из указанных изменений и попробуйте откомпилировать код. Затем отмените изменение и переходите к следующему. Посмотрите, что из этого выйдет!

```
type Appliance interface {
       TurnOn()
}

type Fan string
func (f Fan) TurnOn() {
       fmt.Println("Spinning")
}

type CoffeePot string
func (c CoffeePot) TurnOn() {
       fmt.Println("Powering up")
}
func (c CoffeePot) Brew() {
       fmt.Println("Heating Up")
}

func main() {
       var device Appliance
       device = Fan("Windco Breeze")
       device.TurnOn()
       device = CoffeePot("LuxBrew")
       device.TurnOn()
}
```

Если...	*...программа не будет работать, потому что...*
Вызвать метод конкретного типа, не определенный в интерфейсе: `device.Brew()`	Если в переменной с типом интерфейса хранится значение, вызывать можно только методы, определенные как часть интерфейса, независимо от того, какие методы содержит реальный тип
Удалить из типа метод, обеспечивающий поддержку интерфейса: ~~`func (c CoffeePot) TurnOn() {`~~ ~~`fmt.Println("Powering up")`~~ ~~`}`~~	Если тип не поддерживает интерфейс, значения этого типа не могут присваиваться переменным, объявленным с типом этого интерфейса

Добавить новое возвращаемое значение или параметр в метод, обеспечивающий поддержку интерфейса:
`func (f Fan) TurnOn() error {`
`fmt.Println("Spinning")`
`return nil`
`}`
	Если количество и типы всех параметров и возвращаемых значений не соответствуют определению метода конкретного типа и определению метода в интерфейсе, то конкретный тип не поддерживает интерфейс

Исправление функции playList с помощью интерфейса

Посмотрим, можно ли воспользоваться интерфейсом, чтобы функция `playList` работала с методами `Play` и `Stop` обоих конкретных типов: `TapePlayer` и `TapeRecorder`.

```
// ...Определение типа TapePlayer...
func (t TapePlayer) Play(song string) {
       fmt.Println("Playing", song)
}
func (t TapePlayer) Stop() {
       fmt.Println("Stopped!")
}
// ...Определение типа TapeRecorder...
func (t TapeRecorder) Play(song string) {
       fmt.Println("Playing", song)
}
func (t TapeRecorder) Record() {
       fmt.Println("Recording")
}
func (t TapeRecorder) Stop() {
       fmt.Println("Stopped!")
}
```

В пакете `main` объявляется интерфейс `Player`. (Его также можно определить в пакете `gadget`, но определение интерфейса в том пакете, где он используется, обеспечивает большую гибкость.) Мы указываем, что интерфейсу необходим как метод `Play` с параметром `string`, так и метод `Stop` без параметров. Это означает, что оба типа — `TapePlayer` и `TapeRecorder` — будут поддерживать интерфейс `Player`.

Мы обновляем функцию `playList` так, чтобы она получала любое значение, поддерживающее `Player` (вместо конкретного `TapePlayer`). Мы также меняем тип переменной `player` с `TapePlayer` на `Player`. Это позволит присвоить `player` как `TapePlayer`, так и `TapeRecorder`. А затем значение любого из этих типов передается `playList`!

```
package main

import "github.com/headfirstgo/gadget"

type Player interface {
       Play(string)
       Stop()
}

func playList(device Player, songs []string) {
       for _, song := range songs {
              device.Play(song)
       }
       device.Stop()
}

func main() {
       mixtape := []string{"Jessie's Girl", "Whip It", "9 to 5"}
       var player Player = gadget.TapePlayer{}
       playList(player, mixtape)
       player = gadget.TapeRecorder{}
       playList(player, mixtape)
}
```

Определяет тип интерфейса.

Должен содержать метод Play с параметром string.

Также необходим метод Stop.

Допустимо любое значение, поддерживающее Player, не только TapePlayer.

Обновляем переменную для хранения любого значения, поддерживающего Player.

TapePlayer передается playList.

TapeRecorder передается playList.

```
Playing Jessie's Girl
Playing Whip It
Playing 9 to 5
Stopped!
Playing Jessie's Girl
Playing Whip It
Playing 9 to 5
Stopped!
```

Будьте осторожны!

Если тип объявляет методы с указателями на получателей, при присваивании переменным с типом интерфейса можно будет использовать только указатели на этот тип.

Метод `toggle` *для типа* `Switch`*, показанный ниже, должен использовать указатель на получателя, чтобы он мог изменить получателя.*

```
package main

import "fmt"

type Switch string
func (s *Switch) toggle() {
       if *s == "on" {
              *s = "off"
       } else {
              *s = "on"
       }
       fmt.Println(*s)
}

type Toggleable interface {
       toggle()
}

func main() {
       s := Switch("off")
       var t Toggleable = s
       t.toggle()
       t.toggle()
}
```

Но это приводит к ошибке при присваивании значения `Switch` *переменной с типом интерфейса* `Toggleable`*:*

```
Switch does not implement Toggleable
(toggle method has pointer receiver)
```

Когда Go принимает решение о том, что значение поддерживает интерфейс, то методы указателей не включаются для непосредственных значений, но включаются для указателей. Таким образом, проблема решается присваиванием переменной `Toggleable` *указателя на* `Switch` *вместо непосредственного значения* `Switch`*:*

```
var t Toggleable = &s
```
← *Присваивание указателя.*

После внесения этого изменения код будет работать правильно.

Часто Задаваемые Вопросы

В: Должны ли имена типов интерфейсов начинаться с буквы верхнего или нижнего регистра?

О: Для имен типов интерфейсов действуют те же правила, что и для любых других типов. Если имя начинается с буквы нижнего регистра, то тип интерфейса будет *неэкспортируемым* и недоступным за пределами текущего пакета. Иногда использовать объявленные интерфейсы в других пакетах не нужно, поэтому интерфейс без экспортирования вас вполне устроит. Но если вы *хотите* использовать интерфейс в других пакетах, его имя должно начинаться с буквы верхнего регистра — это необходимо для экспортирования интерфейса.

Упражнение

Код справа определяет типы `Car` и `Truck`, каждый из которых содержит методы `Accelerate`, `Brake` и `Steer`. Заполните пропуски, чтобы добавить интерфейс `Vehicle`, содержащий эти три метода; код функции `main` должен компилироваться и выдавать показанный результат.

```
package main

import "fmt"

type Car string
func (c Car) Accelerate() {
      fmt.Println("Speeding up")
}
func (c Car) Brake() {
      fmt.Println("Stopping")
}
func (c Car) Steer(direction string) {
      fmt.Println("Turning", direction)
}

type Truck string
func (t Truck) Accelerate() {
      fmt.Println("Speeding up")
}
func (t Truck) Brake() {
      fmt.Println("Stopping")
}
func (t Truck) Steer(direction string) {
      fmt.Println("Turning", direction)
}
func (t Truck) LoadCargo(cargo string) {
      fmt.Println("Loading", cargo)
}
```

Место для вашего кода! →

```
_____________________________
      _________________
      _________
      _________________
_

func main() {
      var vehicle Vehicle = Car("Toyoda Yarvic")
      vehicle.Accelerate()
      vehicle.Steer("left")

      vehicle = Truck("Fnord F180")
      vehicle.Brake()
      vehicle.Steer("right")
}
```

```
Speeding up
Turning left
Stopping
Turning right
```

→ Ответ на с. 382.

Утверждения типа

Мы определили новую функцию `TryOut` для тестирования различных методов типов `TapePlayer` и `TapeRecorder`. `TryOut` имеет один параметр с типом интерфейса `Player`, так что передать можно как `TapePlayer`, так и `TapeRecorder`.

В `TryOut` вызываются методы `Play` и `Stop`, входящие в интерфейс `Player`. Мы также вызываем метод `Record`, который *не является* частью интерфейса `Player`, но *определяется* для типа `TapeRecorder`. Сейчас мы передаем `TryOut` значение `TapeRecorder`, поэтому все должно быть нормально, правда?

К сожалению, нет. Как было показано ранее, если значение конкретного типа присваивается переменной с типом интерфейса (включая параметры функций), то вызывать для него можно только методы, входящие в этот интерфейс, независимо от того, какие еще методы содержит конкретный тип. Внутри функции `TryOut` мы имеем дело не со значением `TapeRecorder` (конкретный тип), а со значением `Player` (тип интерфейса). А в интерфейсе `Player` нет метода `Record`!

```
type Player interface {
        Play(string)
        Stop()
}

func TryOut(player Player) {
        player.Play("Test Track")
        player.Stop()
        player.Record()
}

func main() {
        TryOut(gadget.TapeRecorder{})
}
```

Нормально: эти методы являются частью интерфейса Player.

Не входит в Player!

Функции передается значение типа TapeRecorder (которое поддерживает интерфейс Player).

Ошибка.

```
player.Record undefined (type Player
has no field or method Record)
```

Необходимо каким-то образом получить значение конкретного типа (который *содержит* метод `Record`). Первое, что обычно приходит в голову — попытаться выполнить преобразование типа для преобразования значения `Player` в значение `TapeRecorder`. Но преобразования типов не предназначены для использования с типами интерфейсов, поэтому возникает ошибка. В тексте сообщения рекомендуется воспользоваться чем-то иным:

```
func TryOut(player Player) {
        player.Play("Test Track")
        player.Stop()
        recorder := gadget.TapeRecorder(player)
        recorder.Record()
}
```

Преобразование типа не работает!

Ошибка.

```
cannot convert player (type Player) to type
gadget.TapeRecorder: need type assertion
```

«Утверждение типа» (“type assertion”)? Что это такое?

Утверждения типа (продолжение)

Если у вас имеется значение конкретного типа, присвоенное переменной с типом интерфейса, **утверждение типа** позволяет получить информацию о конкретном типе. Утверждение типа *отчасти* напоминает преобразование типа. Его синтаксис похож на гибрид вызова метода и преобразования типа. После значения с типом интерфейса указывается точка, а за ней — пара круглых скобок с конкретным типом. (А точнее, *предполагаемым* конкретным типом значения.)

```
var noiseMaker NoiseMaker = Robot("Botco Ambler")
var robot Robot = noiseMaker.(Robot)
```

Значение типа интерфейса. *Проверяемый тип.*

На обычном языке такое утверждение типа означает примерно следующее: «Я знаю, что эта переменная использует тип интерфейса `NoiseMaker`, но уверен, что *это* значение `NoiseMaker` в действительности является `Robot`».

После использования утверждения типа для возвращения к значению конкретного типа вы сможете вызывать для него методы, определенные для этого типа, но не являющиеся частью интерфейса.

Этот код присваивает `Robot` переменной с типом интерфейса `NoiseMaker`. Для `NoiseMaker` можно вызвать метод `MakeSound`, потому что он является частью интерфейса. Но чтобы вызвать метод `Walk`, необходимо использовать утверждение типа для получения значения `Robot`. После того как мы получим значение `Robot` (вместо `NoiseMaker`), для него можно будет вызывать `Walk`.

```
type Robot string
func (r Robot) MakeSound() {
       fmt.Println("Beep Boop")
}
func (r Robot) Walk() {
       fmt.Println("Powering legs")
}

type NoiseMaker interface {
       MakeSound()
}
func main() {
       var noiseMaker NoiseMaker = Robot("Botco Ambler")
       noiseMaker.MakeSound()
       var robot Robot = noiseMaker.(Robot)
       robot.Walk()
}
```

Определяем переменную с типом интерфейса...

...и присваиваем значение типа, поддерживающего интерфейс.

Вызов метода, который является частью интерфейса.

Обратное преобразование к конкретному типу с использованием утверждения типа.

Вызов метода, определенного для конкретного типа (не для интерфейса).

```
Beep Boop
Powering legs
```

Ошибки утверждений типа

Раньше наша функция `TryOut` не могла вызвать метод `Record` для значения `Player`, так как он не является частью интерфейса `Player`. Посмотрим, можно ли решить проблему с помощью утверждения типа.

Как и прежде, мы передаем значение `TapeRecorder` функции `TryOut`, где оно присваивается параметру с типом интерфейса `Player`. Для значения `Player` можно вызвать методы `Play` и `Stop`, потому что оба этих метода входят в интерфейс `Player`.

Затем для обратного преобразования `Player` в `TapeRecorder` используется утверждение типа. После этого мы снова вызываем `Record`, но на этот раз для `TapeRecorder`.

```
type Player interface {
        Play(string)
        Stop()
}

func TryOut(player Player) {
        player.Play("Test Track")
        player.Stop()
        recorder := player.(gadget.TapeRecorder)
        recorder.Record()
}

func main() {
        TryOut(gadget.TapeRecorder{})
}
```

Утверждение типа используется для перехода к значению TapeRecorder.

Сохраняем значение TapeRecorder.

Вызов метода, определенного только для конкретного типа.

```
Playing Test Track
Stopped!
Recording
```

Вроде бы все работает нормально... с `TapeRecorder`. Но что произойдет, если передать `TapePlayer` функции `TryOut`? Сработает ли такое преобразование, если учесть, что согласно утверждению типа параметр `TryOut` в действительности является `TapeRecorder`?

```
func main() {
        TryOut(gadget.TapeRecorder{})
        TryOut(gadget.TapePlayer{})
}
```

Также передаем TapePlayer...

Все компилируется успешно, но при попытке запустить программу возникает ситуация паники! Как и следовало ожидать, утверждение о том, что `TapePlayer` в действительности является `TapeRecorder`, ничем хорошим не кончается. (В конце концов, оно просто неверно.)

Паника!

```
Playing Test Track
Stopped!
Recording
Playing Test Track
Stopped!
panic: interface conversion: main.Player
is gadget.TapePlayer, not gadget.TapeRecorder
```

Предотвращение паники при ошибках утверждений типов

Если утверждение типа используется в контексте, который предполагает только одно возвращаемое значение, а исходный тип не совпадает с типом в утверждении, в программе возникнет ситуация паники на стадии выполнения (*не* на стадии компиляции):

```
var player Player = gadget.TapePlayer{}
recorder := player.(gadget.TapeRecorder)
```

Утверждается, что исходным типом является TapeRecorder, хотя на самом деле это TapePlayer...

Паника!

```
panic: interface conversion: main.Player
is gadget.TapePlayer, not gadget.TapeRecorder
```

Если утверждения типа используются в контексте, в котором ожидаются несколько возвращаемых значений, то у них есть второе (необязательное) возвращаемое значение, которое сообщает, успешно ли сработало утверждение или нет. (И в случае неудачи паника не возникает.) Второе значение имеет тип `bool`; оно равно `true`, если исходным типом значения был тип утверждения, или `false`, если нет. Со вторым возвращаемым значением можно делать все что угодно, но по общепринятым соглашениям оно обычно присваивается переменной с именем `ok`.

Это еще одно место для применения идиомы «запятая-OK», которая впервые встретилась вам при рассмотрении ассоциативных массивов в главе 7.

Ниже приведена обновленная версия кода, которая присваивает результаты утверждения типа переменной с типом конкретного значения и второй переменной `ok`. Значение `ok` используется в команде `if` для определения того, безопасно ли вызвать метод `Record` для конкретного значения (потому что значение `Player` имеет исходный тип `TapeRecorder`), или от вызова следует отказаться (потому что `Player` имеет другое конкретное значение).

```
var player Player = gadget.TapePlayer{}
recorder, ok := player.(gadget.TapeRecorder)
if ok {
        recorder.Record()
} else {
        fmt.Println("Player was not a TapeRecorder")
}
```

Второе возвращаемое значение присваивается переменной.

Если исходным типом был тип TapeRecorder, для значения вызывается метод Record.

В противном случае выдается сообщение об ошибке утверждения типа.

```
Player was not a TapeRecorder
```

В данном случае конкретным типом был тип `TapePlayer`, а не `TapeRecorder`, поэтому утверждение типа не может быть выполнено, а переменная `ok` равна `false`. Выполняется секция `else` команды `if` и выводится сообщение о том, что `Player` не является `TapeRecorder`. Ситуация паники во время выполнения предотвращается.

Если при использовании утверждений типов у вас нет полной уверенности в том, какой исходный тип стоит за значением с типом интерфейса, используйте дополнительное значение `ok` — это позволит обработать возможные несоответствия типов и предотвратить панику.

Тестирование TapePlayer и TapeRecorder с утверждениями типов

Посмотрим, удастся ли нам использовать новые знания для решения проблемы функции `TryOut` со значениями `TapePlayer` и `TapeRecorder`. Вместо того чтобы игнорировать второе возвращаемое значение утверждения типов, мы присвоим его переменной `ok`. Переменная `ok` содержит `true`, если утверждение типа отработало успешно (это означает, что переменная `recorder` содержит значение `TapeRecorder`, для которого можно вызвать `Record`), или `false` в противном случае (вызов `Record` *не* безопасен). Вызов метода `Record` заключается в команду `if`, которая гарантирует, что метод будет вызван только в случае правильно выполненного утверждения типа.

```
type Player interface {
        Play(string)
        Stop()
}

func TryOut(player Player) {
        player.Play("Test Track")
        player.Stop()
        recorder, ok := player.(gadget.TapeRecorder)
        if ok {
                recorder.Record()
        }
}

func main() {
        TryOut(gadget.TapeRecorder{})
        TryOut(gadget.TapePlayer{})
}
```

Метод Record вызывается только в том случае, если исходное значение имело тип TapeRecorder.

Второе возвращаемое значение присваивается переменной.

TapeRecorder передается в...

...утверждение типа успешно, вызван Record.

TapePlayer передается в...

...утверждение типа не успешно, Record не вызван.

```
Playing Test Track
Stopped!
Recording
Playing Test Track
Stopped!
```

Как и прежде, в функции `main` мы сначала вызываем `TryOut` со значением `TapeRecorder`. `TryOut` берет полученное значение с типом интерфейса `Player` и вызывает для него методы `Play` и `Stop`. Утверждение о том, что конкретным типом `Player` является `TapeRecorder`, выполняется успешно, и для полученного значения `TapeRecorder` вызывается метод `Record`.

Затем функция `TryOut` снова вызывается для `TapePlayer`. (Это тот самый вызов, который ранее приводил к аварийному завершению программы из-за паники в утверждении типа.) Методы `Play` и `Stop` вызываются так же, как в предыдущем случае. При выполнении утверждения типа происходит ошибка, потому что переменная `Player` содержит тип `TapePlayer`, а не `TapeRecorder`. Но поскольку второе возвращаемое значение сохраняется в переменной `ok`, утверждение типа на этот раз не вызывает паники. Оно просто присваивает `ok` значение `false`, в результате чего код из команды `if` не выполняется, а метод `Record` не вызывается. (И это хорошо, потому что значения `TapePlayer` не содержат метода `Record`.)

Благодаря утверждениям типов наша функция `TryOut` работает со значениями как типа `TapeRecorder`, так и типа `TapePlayer`!

У бассейна

Справа приведен обновленный код из предыдущего упражнения. Мы создаем метод `TryVehicle`, который вызывает все методы из интерфейса `Vehicle`. Затем он должен попытаться применить утверждение типа для получения конкретного значения `Truck`, и в случае успеха — вызвать `LoadCargo` для значения `Truck`.

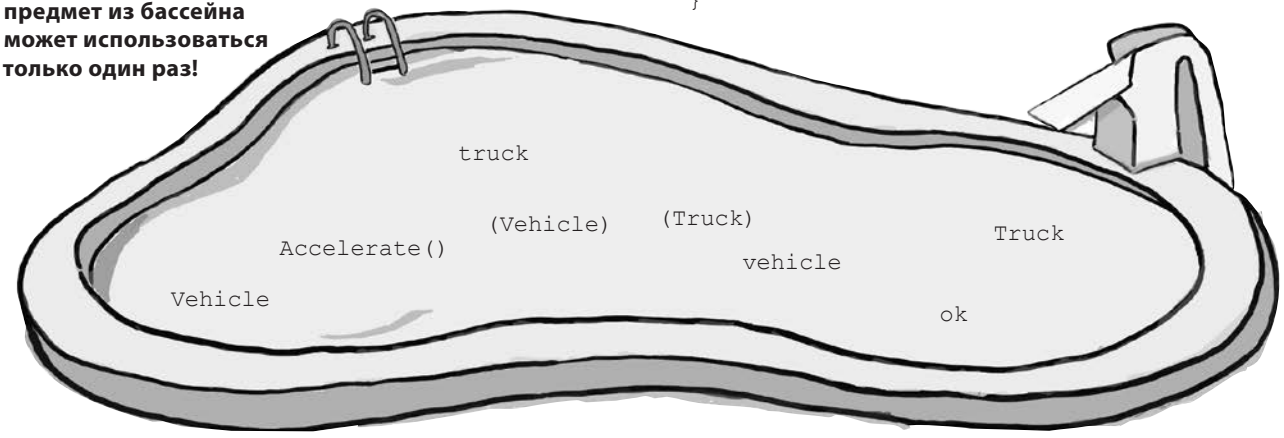
Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: создать программу, которая работает и выводит показанный результат.

```
type Truck string
func (t Truck) Accelerate() {
      fmt.Println("Speeding up")
}
func (t Truck) Brake() {
      fmt.Println("Stopping")
}
func (t Truck) Steer(direction string) {
      fmt.Println("Turning", direction)
}
func (t Truck) LoadCargo(cargo string) {
      fmt.Println("Loading", cargo)
}

type Vehicle interface {
      Accelerate()
      Brake()
      Steer(string)
}

func TryVehicle(vehicle _________) {
      vehicle._________________
      vehicle.Steer("left")
      vehicle.Steer("right")
      vehicle.Brake()
      truck, ___ := vehicle._________
      if ok {
            _____.LoadCargo("test cargo")
      }
}

func main() {
      TryVehicle(Truck("Fnord F180"))
}
```

Результат.

```
Speeding up
Turning left
Turning right
Stopping
Loading test cargo
```

Примечание: каждый предмет из бассейна может использоваться только один раз!



Ответ на с. 382.

Интерфейс error

В завершение этой главы мы рассмотрим несколько интерфейсов, встроенных в Go. Эти интерфейсы не рассматривались явно, но мы уже неоднократно использовали их в программах.

В главе 3 вы узнали, как создавать собственные значения ошибок. Тогда мы сказали: «Значение ошибки — это любое значение с методом `Error`, который возвращает строку».

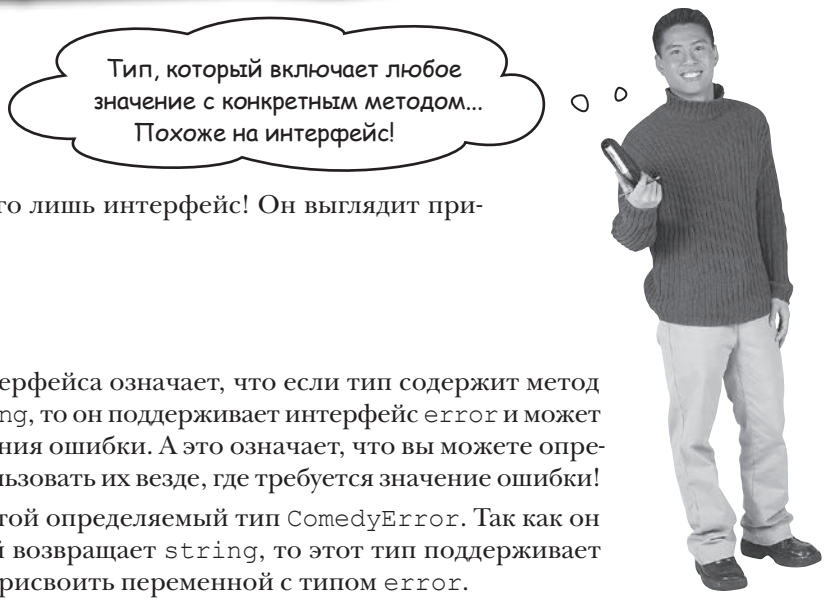
Возвращает значение ошибки.

Выводит сообщение об ошибке.

Также выводит сообщение об ошибке.

```
err := fmt.Errorf("a height of %0.2f is invalid", -2.33333)
fmt.Println(err.Error())
fmt.Println(err)
```

```
a height of -2.33 is invalid
a height of -2.33 is invalid
```



Все верно. Тип `error` — это всего лишь интерфейс! Он выглядит примерно так:

```
type error interface {
      Error() string
}
```

Объявление типа `error` как интерфейса означает, что если тип содержит метод `Error`, который возвращает `string`, то он поддерживает интерфейс `error` и может использоваться в качестве значения ошибки. А это означает, что вы можете определять собственные типы и использовать их везде, где требуется значение ошибки!

Например, ниже приведен простой определяемый тип `ComedyError`. Так как он содержит метод `Error`, который возвращает `string`, то этот тип поддерживает интерфейс `error` и его можно присвоить переменной с типом `error`.

Определяем тип с базовым типом string.

Поддерживает интерфейс error.

Метод Error должен возвращать строку, поэтому выполняется преобразование типа.

Создается переменная с типом error.

ComedyError поддерживает интерфейс error, поэтому переменной можно присвоить ComedyError.

```
type ComedyError string
func (c ComedyError) Error() string {
      return string(c)
}

func main() {
      var err error
      err = ComedyError("What's a programmer's favorite beer? Logger!")
      fmt.Println(err)
}
```

```
What's a programmer's favorite beer? Logger!
```

Интерфейс error (продолжение)

Если кроме значения ошибки требуется еще и отслеживать более подробную информацию о ней в формате, отличном от строки, вы можете создать свой собственный тип, поддерживающий интерфейс `error`, *и* сохранить в нем нужную информацию.

Допустим, вы пишете программу, которая следит за некоторым устройством и предотвращает его перегрев. Ниже приведен тип `OverheatError`, который помогает в решении этой задачи. Он содержит метод `Error` и поэтому поддерживает `error`. Но при этом в качестве базового типа используется `float64`, что позволяет нам отслеживать степень перегрева.

Определяем тип с базовым типом float64.

Поддерживает интерфейс error.

Температура используется в сообщении об ошибке.

```
type OverheatError float64
func (o OverheatError) Error() string {
       return fmt.Sprintf("Overheating by %0.2f degrees!", o)
}
```

Ниже приведена функция `checkTemperature`, которая использует `OverheatError`. В параметрах она получает фактическую температуру системы и температуру, которая считается безопасной. Функция указывает, что она возвращает значение типа `error`, а не конкретно `OverheatError`, но это нормально, потому что `OverheatError` поддерживает интерфейс `error`. Если фактическая температура превышает безопасную, `checkTemperature` возвращает новое значение `OverheatError` с превышением температурного порога.

Указывает, что функция возвращает обычное значение ошибки.

Если фактическая температура превышает безопасную...

...возвращается значение OverheatError с превышением безопасной температуры.

```
func checkTemperature(actual float64, safe float64) error {
       excess := actual - safe
       if excess > 0 {
              return OverheatError(excess)
       }
       return nil
}

func main() {
       var err error = checkTemperature(121.379, 100.0)
       if err != nil {
              log.Fatal(err)
       }
}
```

```
2018/04/02 19:27:44 Overheating by 21.38 degrees!
```

Часто Задаваемые Вопросы

В: Как мы использовали тип интерфейса `error` во всех этих пакетах без импорта? Его имя начинается с буквы нижнего регистра. Разве это не означает, что он не экспортируется из пакета, в котором объявлен? И в каком пакете объявляется `error`, если на то пошло?

О: Тип `error` является **«предварительно объявленным идентификатором»**, как `int` или `string`. И как все остальные предварительно объявленные идентификаторы, он не является частью *никакого* пакета. Он является частью «универсального блока»; это означает, что он доступен везде, независимо от того, какой пакет является текущим.

Помните, что существуют блоки `if` и `for`, которые могут содержаться в блоках функций, которые в свою очередь могут содержаться в блоках пакетов? Так вот, универсальный блок содержит все блоки пакетов. Это означает, что все определенное в универсальном блоке может использоваться в любом пакете без его импорта. И к этой категории относятся ошибки и все остальные предварительно определенные идентификаторы.

Интерфейс Stringer

Помните типы `Gallons`, `Liters` и `Milliliters`, созданные в главе 9 для того, чтобы различать разные единицы измерения объема? Как выяснилось, различать их не так уж просто. Двенадцать галлонов сильно отличаются от 12 литров или 12 миллилитров, но при выводе все значения выглядят одинаково. А если значение содержит слишком много цифр в дробной части, то выглядит громоздко.

```
type Gallons float64
type Liters float64
type Milliliters float64

func main() {
      fmt.Println(Gallons(12.09248342))
      fmt.Println(Liters(12.09248342))
      fmt.Println(Millilite
rs(12.09248342))
}
```

Создаем и выводим значение Gallons.

Создаем и выводим значение Liters.

Создаем и выводим значение Milliliters.

Все три значения выглядят одинаково!

```
12.09248342
12.09248342
12.09248342
```

Можно воспользоваться функцией `Printf` для округления числа и вывода сокращенного обозначения единиц измерения. С другой стороны, если это придется делать при каждом использовании этих типов, это быстро надоест.

Форматирование чисел и добавление сокращений.

```
fmt.Printf("%0.2f gal\n", Gallons(12.09248342))
fmt.Printf("%0.2f L\n", Liters(12.09248342))
fmt.Printf("%0.2f mL\n", Milliliters(12.09248342))
```

```
12.09 gal
12.09 L
12.09 mL
```

Именно для этой цели пакет `fmt` определяет интерфейс `fmt.Stringer`: чтобы любой тип мог решить, как он должен отображаться при выводе. Любой тип легко подготовить для поддержки `Stringer`; достаточно определить метод `String()`, который возвращает строку. Определение интерфейса выглядит так:

```
type Stringer interface {
      String() string
}
```

Любой тип поддерживает fmt.Stringer, если он содержит метод String, который возвращает строку.

Например, здесь мы изменяем тип `CoffeePot` для поддержки `Stringer`:

```
type CoffeePot string
func (c CoffeePot) String() string {
      return string(c) + " coffee pot"
}

func main() {
      coffeePot := CoffeePot("LuxBrew")
      fmt.Println(coffeePot.String())
}
```

Поддерживает интерфейс Stringer.

Метод должен вернуть строку.

```
LuxBrew coffee pot
```

Интерфейс Stringer (продолжение)

Многие функции из пакета `fmt` проверяют, поддерживают ли переданные им значения интерфейс `Stringer`, и если поддерживают — вызывают их методы `String`. В частности, это относится к `Print`, `Println` и `Printf`, а также к другим функциям. Теперь, когда `CoffeePot` поддерживает `Stringer`, мы можем передавать значения `CoffeePot` непосредственно этим функциям, и возвращаемое значение метода `String` значения `CoffeePot` будет использовано при выводе:

Создание значения CoffeePot.

```
coffeePot := CoffeePot("LuxBrew")
fmt.Print(coffeePot, "\n")
fmt.Println(coffeePot)
fmt.Printf("%s", coffeePot)
```

CoffeePot передается различным функциям fmt.

```
LuxBrew coffee pot
LuxBrew coffee pot
LuxBrew coffee pot
```

Возвращаемое значение String используется в выводе.

Перейдем к примеру более серьезного использования типа интерфейса. Сделаем так, чтобы наши типы `Gallons`, `Liters` и `Milliliters` поддерживали `Stringer`. Мы переместим свой код форматирования этих значений в методы `String`, связанные с каждым типом. Вместо `Printf` метод будет вызывать функцию `Sprintf` и возвращать полученное значение.

Чтобы тип Gallons поддерживал Stringer.

```
type Gallons float64
func (g Gallons) String() string {
      return fmt.Sprintf("%0.2f gal", g)
}
```

Чтобы тип Liters поддерживал Stringer.

```
type Liters float64
func (l Liters) String() string {
      return fmt.Sprintf("%0.2f L", l)
}
```

Чтобы тип Milliliters поддерживал Stringer.

```
type Milliliters float64
func (m Milliliters) String() string {
      return fmt.Sprintf("%0.2f mL", m)
}
```

```
func main() {
```

Значения каждого типа передаются Println.

```
fmt.Println(Gallons(12.09248342))
fmt.Println(Liters(12.09248342))
fmt.Println(Millilite
rs(12.09248342))
}
```

```
12.09 gal
12.09 L
12.09 mL
```

Возвращаемые значения каждого типа String используются в выводе.

Теперь каждый раз, когда значения `Gallons`, `Liters` и `Milliliters` передаются `Println` (или большинству других функций `fmt`), будут вызываться их методы `String`, а возвращаемые значения будут использоваться в выводе. Мы определим удобный формат по умолчанию для вывода каждого из этих типов!

Пустой интерфейс

Меня беспокоит один момент. Большинство функций, которые мы до сих пор видели, может вызываться только со значениями конкретных типов. Но некоторые функции `fmt` (например, `fmt.Println`) могут получать значения любого типа! Как это делается?

...передаем число с плавающей точкой...

...строку...

...и логическое значение!

Вызываем fmt.Println...

```
fmt.Println(3.1415, "A string", true)
```

```
3.1415 A string true
```

Хороший вопрос! Выполним команду **`go doc`**, чтобы вывести документацию `fmt.Println`, и посмотрим, с каким типом объявлены его параметры...

Просмотр документации для функции «Println» пакета «fmt».

«...» означает, что функция получает переменное количество аргументов. Но что такое «interface{}»?

```
File  Edit  Window  Help
$ go doc fmt Println
func Println(a ...interface{}) (n int, err error)
    Println formats using the default formats for its operands and writes to
    standard output. Spaces are always added between operands and a newline...
```

Как было показано в главе 6, `...` означает, что это функция с переменным количеством аргументов. Но что это за тип `interface{}`?

Помните: объявление `interface` определяет методы, которые должен содержать тип для поддержки этого интерфейса. Например, наш интерфейс `NoiseMaker` поддерживается любым типом, который содержит метод `MakeSound`.

```
type NoiseMaker interface {
      MakeSound()
}
```

Но что произойдет, если мы объявим тип интерфейса, которому вообще не требуются никакие методы? Он будет поддерживаться *любым* типом! Он будет поддерживаться *всеми* типами!

```
type Anything interface {
}
```

Пустой интерфейс (продолжение)

Тип `interface{}` называется **пустым интерфейсом**, и он используется для передачи значений *любого* типа. Пустой интерфейс не содержит методов, необходимых для его поддержки, поэтому он поддерживается *каждым* типом.

Если вы объявите функцию, которая получает параметр с типом пустого интерфейса, то в ее аргументах могут передаваться значения любого типа:

Получает параметр с типом пустого интерфейса.

```
func AcceptAnything(thing interface{}) {
}

func main() {
    AcceptAnything(3.1415)
    AcceptAnything("A string")
    AcceptAnything(true)
    AcceptAnything(Whistle("Toyco Canary"))
}
```

Все эти типы могут передаваться нашей функции!

Пустому интерфейсу не требуются никакие методы для поддержки, поэтому он поддерживается всеми типами.

Но не торопитесь и не начинайте использовать пустые интерфейсы для всех параметров ваших функций! Если у вас имеется значение с типом пустого интерфейса, *сделать* с ним можно не так уж много.

Многие функции в `fmt` получают значения с типами пустых интерфейсов, поэтому им можно передать все эти значения:

```
func AcceptAnything(thing interface{}) {
    fmt.Println(thing)
}

func main() {
    AcceptAnything(3.1415)
    AcceptAnything(Whistle("Toyco Canary"))
}
```

```
3.1415
Toyco Canary
```

Не пытайтесь вызывать какие-либо методы для значения с типом пустого интерфейса! Помните: если у вас имеется значение с типом интерфейса, то вызвать для него можно только те методы, которые входят в этот интерфейс. А пустой интерфейс *не содержит* никаких методов. Это означает, что у значения с типом пустого интерфейса *нет* методов, которые вы могли бы вызвать!

```
func AcceptAnything(thing interface{}) {
    fmt.Println(thing)
    thing.MakeSound()
}
```

Пытаемся вызвать метод для значения с типом пустого интерфейса...

Ошибка.

```
thing.MakeSound undefined (type interface {} is interface with no methods)
```

Пустой интерфейс (продолжение)

Чтобы вызывать методы для значения с типом пустого интерфейса, необходимо воспользоваться утверждением типа для получения значения конкретного типа.

```
func AcceptAnything(thing interface{}) {
       fmt.Println(thing)
       whistle, ok := thing.(Whistle)
       if ok {
              whistle.MakeSound()
       }
}

func main() {
       AcceptAnything(3.1415)
       AcceptAnything(Whistle("Toyco Canary"))
}
```

Использование утверждения типа для получения Whistle.

Вызов метода для Whistle.

```
3.1415
Toyco Canary
Tweet!
```

А теперь сравните с функцией, которая получает только этот конкретный тип.

```
func AcceptWhistle(whistle Whistle) {
       fmt.Println(whistle)
       whistle.MakeSound()
}
```

Получает Whistle.

Вызов метода. Никакие преобразования типа не нужны.

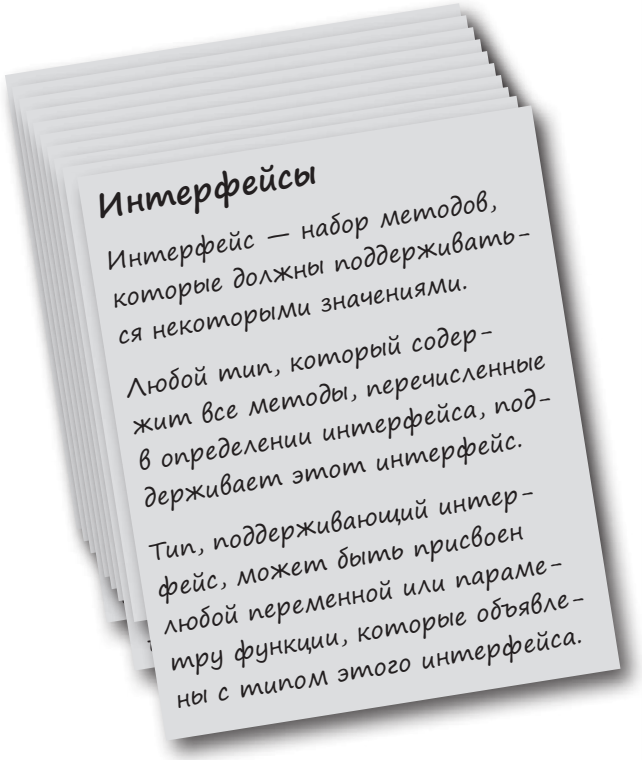
Итак, полезность пустых интерфейсов при определении собственных функций весьма ограниченна. При этом вы будете постоянно использовать пустые интерфейсы с функциями из пакета `fmt` и в других местах тоже. И когда вы в следующий раз увидите параметр `interface{}` в документации функции, вы будете совершенно точно знать, что он означает!

При определении переменных или параметров функций часто бывает известно, с *чем* вы работаете. В таких случаях можно использовать конкретный тип — такой, как `Pen`, `Car` или `Whistle`. В других случаях вас интересует лишь то, что может *делать* значение. И тогда, возможно, стоит определить тип интерфейса — такой, как `WritingInstrument`, `Vehicle` или `NoiseMaker`.

Методы, которые должны вызываться для значения, определяются как часть типа интерфейса. И тогда вы можете выполнять присваивание или вызывать свои функции, не особенно беспокоясь о конкретном типе значений. Если значение содержит нужные методы, вы сможете им пользоваться!

Ваш инструментарий Go

Глава 11 осталась позади! В ней ваш инструментарий пополнился интерфейсами.



КЛЮЧЕВЫЕ МОМЕНТЫ

- Конкретный тип указывает не только то, что могут *делать* значения (какие методы для них можно вызывать), но и то, чем они *являются*: он задает базовый тип, в котором хранятся данные значения.
- Тип интерфейса — абстрактный тип. Интерфейсы не описывают, чем *является* значение: они ничего не говорят о том, какой базовый тип используется или как хранятся его данные. Они описывают только то, что значение может *делать*: какие методы оно содержит.
- Определение интерфейса должно содержать список имен методов со всеми параметрами или возвращаемыми значениями, которые должны иметь такие методы.
- Чтобы поддерживать интерфейс, тип должен содержать все методы, заданные в интерфейсе. Имена методов, типы параметров (если они есть) и типы возвращаемых значений (если они есть) должны совпадать с теми, которые определены в интерфейсе.
- Тип может содержать методы помимо тех, которые перечислены в интерфейсе, но никакие из обязательных методов не могут отсутствовать; в противном случае тип не поддерживает интерфейс.
- Тип может поддерживать сразу несколько интерфейсов, и интерфейс может поддерживаться сразу несколькими типами.
- Поддержка интерфейсов достигается автоматически. В Go не нужно специально объявлять, что конкретный тип поддерживает интерфейс.
- Для переменной с типом интерфейса можно вызывать только те методы, которые определены в интерфейсе.
- Если значение конкретного типа присвоено переменной с типом интерфейса, вы можете воспользоваться **утверждением типа** для получения значения конкретного типа. Только после этого вы сможете вызывать методы, определенные для конкретного типа (но не для интерфейса).
- Утверждения типов возвращают второе логическое значение, которое сообщает, успешно ли было выполнено утверждение.

```
car, ok := vehicle.(Car)
```

Упражнение
Решение

```
type Car string
func (c Car) Accelerate() {
   fmt.Println("Speeding up")
}
func (c Car) Brake() {
   fmt.Println("Stopping")
}
func (c Car) Steer(direction string) {
   fmt.Println("Turning", direction)
}

type Truck string
func (t Truck) Accelerate() {
   fmt.Println("Speeding up")
}
func (t Truck) Brake() {
   fmt.Println("Stopping")
}
func (t Truck) Steer(direction string) {
   fmt.Println("Turning", direction)
}
func (t Truck) LoadCargo(cargo string) {
   fmt.Println("Loading", cargo)
}

type Vehicle interface {
   Accelerate()
   Brake()
   Steer(string)
}

func main() {
   var vehicle Vehicle = Car("Toyoda Yarvic")
   vehicle.Accelerate()
   vehicle.Steer("left")

   vehicle = Truck("Fnord F180")
   vehicle.Brake()
   vehicle.Steer("right")
}
```

Не забудьте указать, что Steer принимает параметр!

```
Speeding up
Turning left
Stopping
Turning right
```

У бассейна. Решение

```
type Truck string
func (t Truck) Accelerate() {
    fmt.Println("Speeding up")
}
func (t Truck) Brake() {
    fmt.Println("Stopping")
}
func (t Truck) Steer(direction string) {
    fmt.Println("Turning", direction)
}
func (t Truck) LoadCargo(cargo string) {
    fmt.Println("Loading", cargo)
}

type Vehicle interface {
    Accelerate()
    Brake()
    Steer(string)
}

func TryVehicle(vehicle Vehicle ) {
    vehicle. Accelerate()
    vehicle.Steer("left")
    vehicle.Steer("right")
    vehicle.Brake()
    truck, ok := vehicle.(Truck)
    if ok {
        truck.LoadCargo("test cargo")
    }
}
func main() {
    TryVehicle(Truck("Fnord F180"))
}
```

Утверждение типа успешно?

Содержит Truck, а не (просто) Vehicle, поэтому мы можем вызвать LoadCargo.

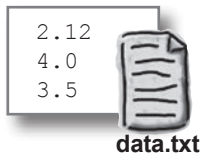
```
Speeding up
Turning left
Turning right
Stopping
Loading test cargo
```

Восстановление после сбоев

Любая программа сталкивается с ошибками. Учтите их при планировании. Иногда обработка ошибки сводится к простому выводу сообщения и завершению программы. Другие ошибки требуют дополнительных действий: например, закрытия открытых файлов или сетевых подключений или иного освобождения ресурсов, чтобы после вашей программы оставался порядок. В этой главе мы покажем, как **отложить** завершающие действия, чтобы они выполнились даже в случае ошибки. Также вы узнаете, как поднять **панику** в тех (редких) ситуациях, когда это уместно, и как **восстановиться** после нее.

Снова о чтении чисел из файла

В книге было немало сказано об обработке ошибок. Но способы, о которых говорилось ранее, подходят не для каждой ситуации. Рассмотрим один из таких сценариев. Напишем программу *sum.go*, которая читает значения `float64` из текстового файла, суммирует их и выводит сумму.




```
Shell Edit View Window Help
$ go run sum.go data.txt
Opening data.txt
Closing file
Sum: 9.62
```

В главе 6 была создана функция `GetFloats`, которая открывает текстовый файл, преобразует каждую строку файла в значение `float64` и возвращает эти значения в виде сегмента.

Здесь мы переместили функцию `GetFloats` в пакет `main` и обновили ее, чтобы для открытия и закрытия текстового файла в ней использовались две новые функции, `OpenFile` и `CloseFile`.

```
package main

import (
    "bufio"
    "fmt"
    "log"
    "os"
    "strconv"
)

func OpenFile(fileName string) (*os.File, error) {
    fmt.Println("Opening", fileName)
    return os.Open(fileName)
}
func CloseFile(file *os.File) {
    fmt.Println("Closing file")
    file.Close()
}

func GetFloats(fileName string) ([]float64, error) {
    var numbers []float64
    file, err := OpenFile(fileName)
    if err != nil {
        return nil, err
    }
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        number, err := strconv.ParseFloat(scanner.Text(), 64)
        if err != nil {
              return nil, err
        }
        numbers = append(numbers, number)
    }
    CloseFile(file)
    if scanner.Err() != nil {
        return nil, scanner.Err()
    }
    return numbers, nil
}
```

Весь этот код был перемещен в пакет «main» в исходном файле sum.go.

Открывает файл и возвращает указатель на него, а также значение обнаруженной ошибки.

Закрывает файл.

Вместо прямого вызова os.Open вызывается функция OpenFile.

Вместо прямого вызова file.Close вызывается CloseFile.

Снова о чтении чисел из файла (продолжение)

Имя файла, из которого будут читаться данные, передается в аргументе командной строки. В главе 6 уже упоминался `os.Args` — сегмент строковых значений, содержащий все аргументы, заданные при запуске программы.

В функции `main` мы получаем имя открываемого файла из первого аргумента командной строки `os.Args[1]`. (Напомним, что элемент `os.Args[0]` содержит имя запускаемой программы; фактические аргументы появляются в `os.Args[1]` и последующих элементах.)

Затем имя файла передается `GetFloats` для чтения данных из файла, и программа получает сегмент значений `float64`.

Если при этом будут обнаружены какие-либо ошибки, они будут возвращены функцией `GetFloats` и сохранены в переменной `err`. Если значение `err` не равно `nil`, значит, произошла ошибка, поэтому программа просто выводит ее и завершается.

В противном случае это означает, что файл был прочитан успешно, поэтому используется цикл `for` для суммирования всех значений в сегменте, после чего выводится полученная сумма.

Первый аргумент командной строки используется как имя файла.

Сохраняется сегмент чисел, прочитанных из файла, со значением ошибки.

Если произошла ошибка, программа выводит сообщение и завершается.

Суммируем все числа в сегменте.

Выводится сумма.

```
func main() {
    numbers, err := GetFloats(os.Args[1])
    if err != nil {
        log.Fatal(err)
    }
    var sum float64 = 0
    for _, number := range numbers {
        sum += number
    }
    fmt.Printf("Sum: %0.2f\n", sum)
}
```

Сохраните весь этот код в файле с именем *sum.go*. Затем создайте обычный текстовый файл с числами, по одному числу в строке. Сохраните файл под именем *data.txt* в одном каталоге с *sum.go*.

Запустите программу командой `go run sum.go data.txt`. Строка `"data.txt"` будет первым аргументом программы *sum.go*, поэтому это имя файла будет передано `GetFloats`.

Мы видим, где вызываются функции `OpenFile` и `CloseFile`, потому что обе функции включают вызовы `fmt.Println`. И в конце вывода мы видим сумму всех чисел в *data.txt*. Похоже, все работает!

data.txt передается в аргументе командной строки.

Здесь вызывается функция OpenFile.

Здесь вызывается функция CloseFile.

Сумма всех чисел в файле.

```
Shell Edit View Window Help
$ go run sum.go data.txt
Opening data.txt
Closing file
Sum: 50.75
```

Любая ошибка помешает закрытию файла!

Если программа *sum.go* получит неправильно отформатированный файл, то возникнут проблемы. Например, файл со строкой, которая не может быть преобразована в `float64`, приводит к возникновению ошибки.

Не преобразуется в float64! →

bad-data.txt

Программа запускается для файла с некорректными данными. →
Здесь вызывается функция OpenFile. →
При чтении файла происходит ошибка... →
Функция CloseFile не будет вызвана! →

```
Shell Edit View Window Help
$ go run sum.go bad-data.txt
Opening data.txt
2018/04/07 21:18:09 strconv.ParseFloat:
parsing "hello": invalid syntax
exit status 1
```

Само по себе это не страшно; любая программа время от времени получает недействительные данные. Но функция `GetFloats` должна вызвать функцию `CloseFile` после завершения работы. Мы не видим в выводе программы сообщения «`Closing file`», а это наводит на мысль, что функция `CloseFile` вообще не вызывается!

Проблема в том, что при вызове `strconv.ParseFloat` для строки, которая не может быть преобразована в `float64`, возвращается ошибка. Наш код устроен так, что в этот момент происходит выход из функции `GetFloats`.

Но возврат происходит *до* вызова `CloseFile`, что означает, что файл никогда не будет закрыт!

```
func GetFloats(fileName string) ([]float64, error) {
    var numbers []float64
    file, err := OpenFile(fileName)
    if err != nil {
        return nil, err
    }
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        number, err := strconv.ParseFloat(scanner.Text(), 64)
        if err != nil {
            return nil, err
        }
        numbers = append(numbers, number)
    }
    CloseFile(file)
    if scanner.Err() != nil {
        return nil, scanner.Err()
    }
    return numbers, nil
}
```

Функция ParseFloat возвращает ошибку, если ей не удается преобразовать текстовую строку в float64... →
...это приводит к тому, что GetFloats возвращает ошибку... →
...а это означает, что функция CloseFile не будет вызвана! →

Отложенные вызовы функций

На первый взгляд в незакрытом файле нет ничего страшного. И наверное, для простой программы, которая просто открывает один файл, это правда. Но каждый файл, который остается открытым, продолжает поглощать ресурсы операционной системы. Со временем незакрытые файлы могут накапливаться и приводить к сбою программы или даже снижать производительность всей системы. Очень важно завести привычку закрывать файлы тогда, когда программа завершила работу с ними.

Но как это сделать? Функция `GetFloats` устроена так, что она немедленно завершает работу при ошибках во время чтения файла, даже если еще не была вызвана функция `CloseFile`!

Если у вас есть вызов функции, который должен быть гарантированно выполнен *в любом случае*, используйте команду `defer`. Ключевое слово `defer` может располагаться перед любым вызовом обычной функции или метода, и Go отложит вызов этой функции вплоть до выхода из текущей функции.

Обычно вызовы функций выполняются сразу же после их обнаружения в программе. В этом коде вызов `fmt.Println("Goodbye!")` располагается перед двумя другими вызовами `fmt.Println`.

```
package main

import "fmt"

func Socialize() {
       fmt.Println("Goodbye!")
       fmt.Println("Hello!")
       fmt.Println("Nice weather, eh?")
}

func main() {
       Socialize()
}
```

```
Goodbye!
Hello!
Nice weather, eh?
```

Но если добавить ключевое слово `defer` перед вызовом `fmt.Println("Goodbye!")`, этот вызов будет выполнен только после того, как отработает весь остальной код в функции `Socialize`, а функция `Socialize` завершится.

```
package main

import "fmt"

func Socialize() {
       defer fmt.Println("Goodbye!")
       fmt.Println("Hello!")
       fmt.Println("Nice weather, eh?")
}

func main() {
       Socialize()
}
```

Перед вызовом функции добавляется ключевое слово «defer».

Первый вызов функции откладывается до завершения Socialize!

```
Hello!
Nice weather, eh?
Goodbye!
```

Восстановление после ошибок

Здорово, конечно, но вы что-то говорили об использовании `defer` для вызовов функций, которые должны выполняться «в любом случае». Объясните?

Ключевое слово `defer` гарантирует, что вызов функции будет выполнен даже в том случае, если вызывающая функция завершится преждевременно — например, из-за ключевого слова `return`.

Ниже приведена обновленная версия функции `Socialize`, которая возвращает ошибку. Выход из `Socialize` происходит перед вызовом `fmt.Println("Nice weather, eh?")`. Но из-за того, что перед вызовом `.Println("Goodbye!")` располагается ключевое слово `defer`, `Socialize` всегда будет выводить сообщение «`Goodbye!`» перед завершением диалога.

Ключевое слово «defer» гарантирует, что вызов функции будет выполнен даже в том случае, если вызывающая функция завершится преждевременно.

```
package main

import (
        "fmt"
        "log"
)

func Socialize() error {
        defer fmt.Println("Goodbye!")
        fmt.Println("Hello!")
        return fmt.Errorf("I don't want to talk.")
        fmt.Println("Nice weather, eh?")
        return nil
}

func main() {
        err := Socialize()
        if err != nil {
                log.Fatal(err)
        }
}
```

Вывод сообщения «Goodbye!» откладывается.

Возвращает ошибку.

Этот код не будет выполнен!

Вызов отложенной функции все еще выполняется, когда возвращается Socialize.

```
Hello!
Goodbye!
2018/04/08 19:24:48 I don't want to talk.
```

Использование отложенного вызова для гарантированного закрытия файлов

Так как ключевое слово `defer` может гарантировать, что вызов функции будет выполнен «в любом случае», обычно оно используется для кода, который должен быть выполнен даже в случае ошибки. Один из типичных примеров такого рода — гарантированное закрытие файлов, открытых в программе.

Именно это и должно происходить в функции `GetFloats` программы *sum.go*. После вызова функции `OpenFile` программа должна вызвать `CloseFile` даже в том случае, если при разборе содержимого файла произойдет ошибка.

Этой цели можно достичь очень просто: переместите вызов `CloseFile` после вызова `OpenFile` (и сопутствующего кода обработки ошибок) и поставьте перед ним ключевое слово `defer`.

Ключевое слово «defer» нужно для того, чтобы функция не выполнялась до выхода из GetFloats.

Ключевое слово `defer` гарантирует, что функция `CloseFile` будет вызвана при выходе из `GetFloats` — как в случае нормального завершения, так и при возникновении ошибок в ходе разбора файла.

Даже если программа *sum.go* получит файл с некорректными данными, она все равно закроет файл перед завершением!

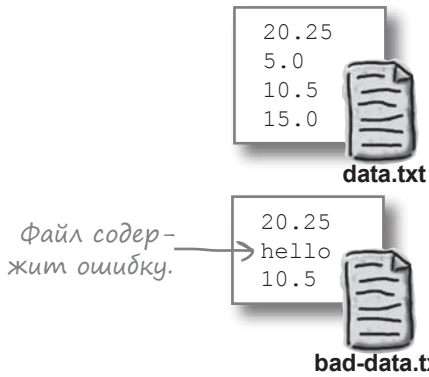
```
func OpenFile(fileName string) (*os.File, error) {
    fmt.Println("Opening", fileName)
    return os.Open(fileName)
}
func CloseFile(file *os.File) {
    fmt.Println("Closing file")
    file.Close()
}

func GetFloats(fileName string) ([]float64, error) {
    var numbers []float64
    file, err := OpenFile(fileName)
    if err != nil {
        return nil, err
    }
    defer CloseFile(file)
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        number, err := strconv.ParseFloat(scanner.Text(), 64)
        if err != nil {
            return nil, err
        }
        numbers = append(numbers, number)
    }
    if scanner.Err() != nil {
        return nil, scanner.Err()
    }
    return numbers, nil
}
```

Перемещается после вызова OpenFile (и сопутствующего кода обработки ошибок).

Даже если здесь будет получена ошибка, функция CloseFile все равно будет вызвана!

Функция CloseFile будет вызвана и в том случае, если здесь была возвращена ошибка!

И конечно, функция CloseFile вызывается и в том случае, если GetFloats завершится нормально!



Файл содержит ошибку.

Выполняется отложенный вызов CloseFile!

```
Shell Edit View Window Help
$ go run sum.go data.txt
Opening data.txt
Closing file
Sum: 50.75
```

Выполняется отложенный вызов CloseFile!

```
Shell Edit View Window Help
$ go run sum.go bad-data.txt
Opening data.txt
Closing file
2018/04/09 21:30:42 strconv.ParseFloat:
parsing "hello": invalid syntax
exit status 1
```

Развлечения с магнитами

Эта программа определяет тип `Refrigerator`, который моделирует холодильник. `Refrigerator` использует в качестве базового типа сегмент строк с названиями продуктов, лежащих в холодильнике. Тип содержит метод `Open`, который моделирует открытие двери, и парный метод `Close` для ее закрытия (никому не захочется понапрасну расходовать энергию). Метод `FindFood` вызывает `Open` для открытия двери, вызывает функцию `find`, написанную для поиска конкретного продукта в базовом сегменте, а затем вызывает `Close` для закрытия двери.

Но с методом `FindFood` возникает проблема. Он возвращает значение ошибки, если искомый продукт не найден. Но когда это происходит, метод возвращает управление до вызова `Close`, и дверь виртуального холодильника остается открытой!

```
func find(item string, slice []string) bool {
       for _, sliceItem := range slice {
              if item == sliceItem {
                     return true
              }
       }
       return false
}

type Refrigerator []string

func (r Refrigerator) Open() {
       fmt.Println("Opening refrigerator")
}
func (r Refrigerator) Close() {
       fmt.Println("Closing refrigerator")
}
func (r Refrigerator) FindFood(food string) error {
       r.Open()
       if find(food, r) {
              fmt.Println("Found", food)
       } else {
              return fmt.Errorf("%s not found", food)
       }
       r.Close()
       return nil
}

func main() {
       fridge := Refrigerator{"Milk", "Pizza", "Salsa"}
       for _, food := range []string{"Milk", "Bananas"} {
              err := fridge.FindFood(food)
              if err != nil {
                     log.Fatal(err)
              }
       }
}
```

(продолжение на следующей странице...)

Возвращает true, если строка найдена в сегменте...

...или false, если строка не найдена.

Тип Refrigerator основан на сегменте строк, в котором хранятся названия продуктов из холодильника.

Моделирует открытие двери холодильника.

Моделирует закрытие двери холодильника.

Если Refrigerator содержит нужный продукт...

...выводится сообщение о том, что продукт найден.

В противном случае возвращается сообщение об ошибке.

Но если будет возвращена ошибка, этот метод не будет вызван!

Дверь холодильника открывается, но не будет закрыта!

```
Opening refrigerator
Found Milk
Closing refrigerator
Opening refrigerator
2018/04/09 22:12:37 Bananas not found
```

Ответ на с. 411.

Развлечения с магнитами (продолжение)

Используйте магниты для создания обновленной версии метода `FindFood`. Она должна откладывать вызов метода `Close`, чтобы он выполнялся при выходе из `FindFood` (независимо от того, был найден продукт или нет).

Метод Close типа Refrigerator должен вызываться в том случае, если продукт найден.

Метод Close также должен вызываться и в том случае, если продукт не найден.

```
Opening refrigerator
Found Milk
Closing refrigerator
Opening refrigerator
Closing refrigerator
2018/04/09 22:12:37 Bananas not found
```

```
defer
```

```
if find(food, r) {
        fmt.Println("Found", food)
} else {
        return fmt.Errorf("%s not found", food)
}
```

```
r.Open()
```

```
r.Close()
```

```
func (r Refrigerator) FindFood(food string) error {
```

```
}
```

```
return nil
```

Часто Задаваемые Вопросы

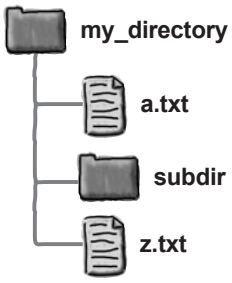
В: Значит, я могу откладывать вызовы функций и методов... А можно откладывать и другие команды — например, циклы или присваивания?

О: Нет, только вызовы функций и методов. Вы можете написать функцию, которая делает то, что вам нужно, а затем отложить вызов этой функции или метода, но само ключевое слово `defer` может использоваться только с вызовом функции или метода.

Получение списка файлов в каталоге

В Go есть и другие средства обработки ошибок. Вскоре мы покажем вам программу, в которой они продемонстрированы. Но в этой программе также используется пара новых приемов, которые нужно объяснить перед рассмотрением программы. Для начала стоит рассказать о том, как прочитать список содержимого каталога.

Создайте каталог с именем *my_directory*, в котором находятся два файла и подкаталог, как показано справа. Программа, приведенная внизу, выводит содержимое *my_directory* с указанием имени каждого найденного элемента и признака того, является ли он файлом или подкаталогом.



Пакет `io/ioutil` включает функцию `ReadDir`, предназначенную для чтения содержимого каталога. Функция `ReadDir` получает имя каталога и возвращает сегмент значений, по одному для каждого файла или подкаталога в каталоге (а также возможные значения ошибок). Каждое значение в сегменте поддерживает интерфейс `FileInfo`, который включает метод `Name`, возвращающий имя файла, и метод `IsDir`, возвращающий значение `true` для каталогов.

Наша программа вызывает функцию `ReadDir`, передавая имя *my_directory* в аргументе. Затем она перебирает значения в полученном сегменте. Если `IsDir` возвращает для значения `true`, то программа выводит `"Directory:"` и имя файла, а если `false` — `"File:"` и имя файла.


```
package main

import (
        "fmt"
        "io/ioutil"
        "log"
)

func main() {
        files, err := ioutil.ReadDir("my_directory")
        if err != nil {
                log.Fatal(err)
        }

        for _, file := range files {
                if file.IsDir() {
                        fmt.Println("Directory:", file.Name())
                } else {
                        fmt.Println("File:", file.Name())
                }
        }
}
```

Получает сегмент с элементами, представляющими содержимое «my_directory».

Для каждого файла в сегменте...

Если файл является каталогом...

...то выводится «Directory» и имя файла.

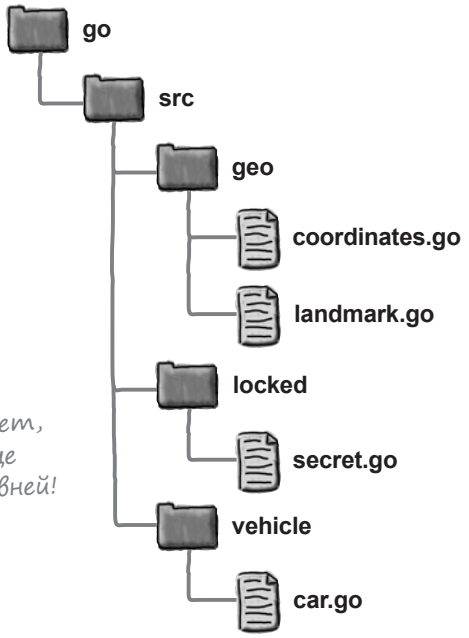
А если нет — выводится «File:» и имя файла.

Сохраните приведенный выше код в файле *files.go* в одном каталоге с *my_directory*. В терминале перейдите в этот каталог и введите команду **`go run files.go`**. Программа выполняется и выводит список файлов и каталогов, находящихся в *my_directory*.

```
Shell Edit View Window Help
$ cd work
$ go run files.go
File: a.txt
Directory: subdir
File: z.txt
```

Получение списка файлов в подкаталогах (более сложная задача)

Программа для чтения содержимого одного каталога не слишком сложна. Но допустим, вы хотите вывести содержимое более сложной иерархии, например каталога рабочей области Go. Такой каталог содержит целое дерево подкаталогов, вложенных в другие подкаталоги — одни подкаталоги содержат файлы, другие нет.



Обычно такая программа получается достаточно сложной. Общая схема выглядит примерно так:

I. Получить список файлов в каталоге.

  A. Получить следующий файл.

  B. Файл является каталогом?

    1. Да: получить список файлов в каталоге.

      a. Получить следующий файл.

      b. Файл является каталогом?

        01. Да: получить список файлов в каталоге...

    2. Нет: просто вывести имя файла.

И кто знает, сколько еще будет уровней!

Голова идет кругом, верно? Писать *такой* код никому не захочется!

Нет ли более простого решения? Логика примерно такая:

I. Получить список файлов в каталоге.

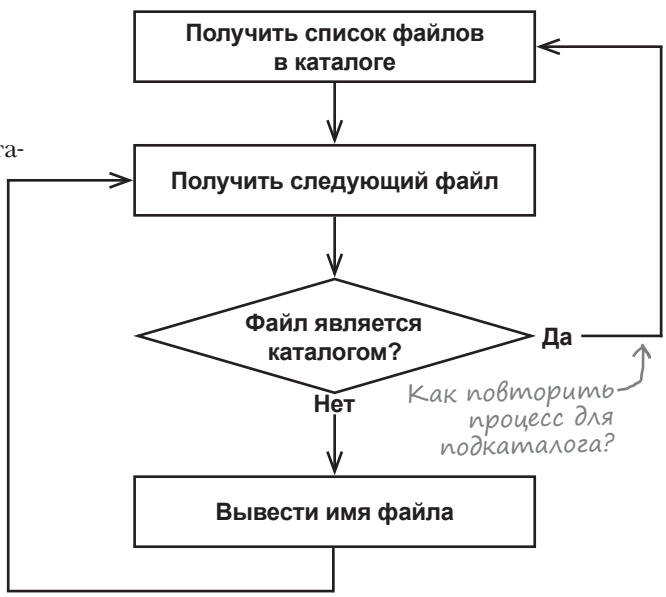
  A. Получить следующий файл.

  B. Файл является каталогом?

    1. Да: повторить с шага **I** для этого каталога.

    2. Нет: просто вывести имя файла.

Но пока неясно, как реализовать этап «повторить с шага **I** для этого каталога». Для этого нам понадобится новая концепция программирования...



Рекурсивные вызовы функций

И тут мы подходим ко второму (и последнему) приему, который мы должны показать вам перед тем, как вернуться к основной теме обработки ошибок.

Go — один из многих языков программирования с поддержкой механизма **рекурсии**, который позволяет функции вызывать саму себя.

Но если действовать неосторожно, то возникнет бесконечный цикл, в котором функция будет вызывать себя снова и снова:

```
package main

import "fmt"

func recurses() {
       fmt.Println("Oh, no, I'm stuck!")
       recurses()
}

func main() {
       recurses()
}
```

Функция recurses вызывает сама себя!

Функция recurses вызывается в первый раз.

```
Oh, no, I'm stuck!
Oh, no, I'm stuck!
Oh, no, I'm stuck!
Oh, no, ^Csignal: interrupt
```

Тот, кто запускает программу, должен нажать Ctrl+C, чтобы прервать бесконечный цикл!

Но если позаботиться о том, чтобы цикл рекурсии в какой-то момент прервался, рекурсивные функции могут оказаться весьма полезными.

Ниже приведена рекурсивная функция `count`, которая ведет отсчет от начального до конечного числа. (Обычно такие задачи эффективнее решаются при помощи циклов, но этот простой пример наглядно показывает, как работает рекурсия.)

```
package main

import "fmt"

func count(start int, end int) {
       fmt.Println(start)
       if start < end {
              count(start+1, end)
       }
}

func main() {
       count(1, 3)
}
```

Вывести текущее начальное число.

Если конечное число еще не достигнуто...

...то функция count вызывает сама себя с начальным числом на 1 больше текущего.

Вызываем count в первый раз и указываем, что отсчет должен вестись от 1 до 3.

```
1
2
3
```

Рекурсивные вызовы функций (продолжение)

Последовательность действий в программе:

1. `main` вызывает `count` с параметром `start`, равным `1`, и параметром `end`, равным `3`.
2. `count` выводит параметр `start`: `1`.
3. `start` (`1`) меньше `end` (`3`), поэтому функция `count` вызывает саму себя с параметром `start`, равным `2`, и параметром `end`, равным `3`.
4. Второй вызов `count` выводит свой новый параметр `start`: `2`.
5. `start` (`2`) меньше `end` (`3`), поэтому функция `count` вызывает саму себя с параметром `start`, равным `3`, и параметром `end`, равным `3`.
6. Третий вызов `count` выводит свой новый параметр `start`: `3`.
7. `start` (`3`) *не* меньше `end` (`3`), поэтому `count` *не* вызывает себя снова, а просто возвращает управление.
8. Предыдущие два вызова `count` также возвращают управление, и программа завершается.

Если добавить вызовы `Printf`, обозначающие каждый вызов `count` и каждый выход из функции, эта последовательность станет чуть более очевидной:

```
package main

import "fmt"

func count(start int, end int) {
       fmt.Printf("count(%d, %d) called\n", start, end)
       fmt.Println(start)
       if start < end {
              count(start+1, end)
       }
       fmt.Printf("Returning from count(%d, %d) call\n", start, end)
}

func main() {
       count(1, 3)
}
```

```
count(1, 3) called
1
count(2, 3) called
2
count(3, 3) called
3
Returning from count(3, 3) call
Returning from count(2, 3) call
Returning from count(1, 3) call
```

Так работает простая рекурсивная функция. Попробуем применить рекурсию в программе *files.go* и посмотрим, как она поможет с выводом содержимого подкаталогов.

Рекурсивный вывод содержимого каталога

Итак, программа *files.go* должна выводить содержимое всех подкаталогов в каталоге рабочей области Go. Для решения этой задачи должна использоваться рекурсивная логика следующего вида:

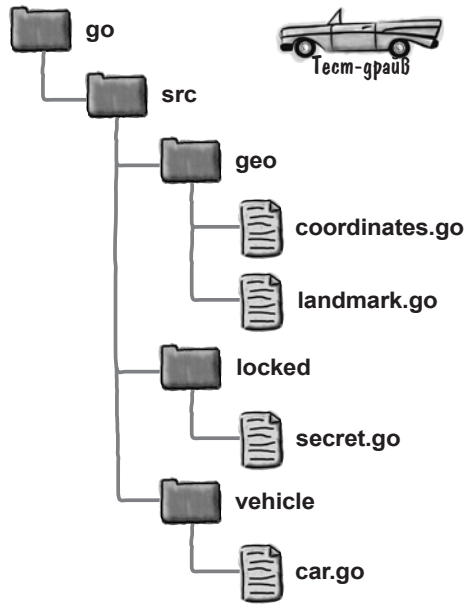
I. Получить список файлв в каталоге.

  A. Получить следующий файл.

  B. Файл является каталогом?

    1. Да: начать с шага **I** для этого каталога.

    2. Нет: просто вывести имя файла.



Из функции `main` был исключен код чтения содержимого каталога; `main` теперь просто вызывает рекурсивную функцию `scanDirectory`. Функция `scanDirectory` получает путь к сканируемому каталогу, поэтому ей передается путь к подкаталогу `"go"`.

`scanDirectory` прежде всего выводит текущий путь, чтобы мы знали, какой каталог обрабатывается в настоящий момент. Затем она вызывает `ioutil.ReadDir` для этого пути, чтобы получить содержимое каталога.

Функция перебирает сегмент значений `FileInfo`, возвращенных `ReadDir`, последовательно обрабатывая каждое значение. Она вызывает функцию `filepath.Join` для объединения текущего пути и текущего имени файла со слешами / (таким образом, `"go"` и `"src"` объединяются в `"go/src"`).

Если текущий файл не является каталогом, `scanDirectory` просто выводит полный путь и переходит к следующему файлу (если в текущем каталоге еще остались необработанные элементы).

Но если текущий файл *является* каталогом, рекурсия вступает в силу: функция `scanDirectory` вызывает сама себя с путем подкаталога. Если этот подкаталог содержит свои подкаталоги, `scanDirectory` вызовет себя для каждого из *этих* подкаталогов и так далее по всему дереву файла.

```
package main

import (
    "fmt"
    "io/ioutil"
    "log"
    "path/filepath"
)

func scanDirectory(path string) error {
    fmt.Println(path)
    files, err := ioutil.ReadDir(path)
    if err != nil {
        return err
    }
    for _, file := range files {
        filePath := filepath.Join(path, file.Name())
        if file.IsDir() {
            err := scanDirectory(filePath)
            if err != nil {
                return err
            }
        } else {
            fmt.Println(filePath)
        }
    }
    return nil
}

func main() {
    err := scanDirectory("go")
    if err != nil {
        log.Fatal(err)
    }
}
```

Рекурсивная функция, которая получает путь для обработки.

Возвращает все обнаруженные ошибки.

Выводит текущий каталог.

Получает сегмент с содержимым каталога.

Соединяет путь каталога и имя файла через символ /.

Если это подкаталог...

...рекурсивно вызывает scanDirectory, но на этот раз с путем подкаталога.

Если это обычный файл, просто вывести его имя с путем.

Запускаем процесс вызовом scanDirectory для каталога верхнего уровня.

Рекурсивный вывод содержимого каталога (продолжение)

Сохраните предыдущий код в файле *files.go* в каталоге рабочей области Go (вероятно, в домашнем каталоге пользователя). В терминале перейдите в этот каталог и запустите программу командой **`go run files.go`**.

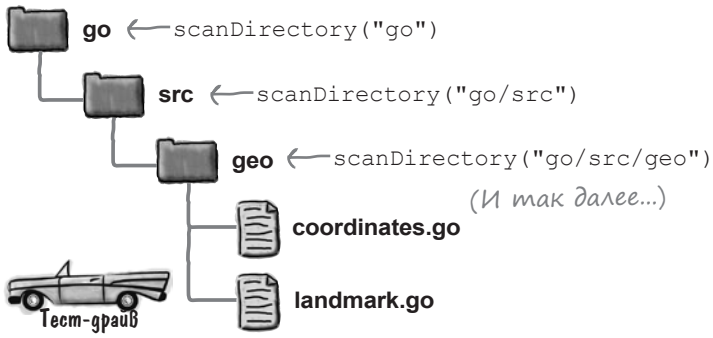
```
Shell Edit View Window Help
$ cd /Users/jay
$ go run files.go
go
go/src
go/src/geo
go/src/geo/coordinates.go
go/src/geo/landmark.go
go/src/locked
go/src/locked/secret.go
go/src/vehicle
go/src/vehicle/car.go
```

И когда вы увидите, как работает функция `scanDirectory`, вы оцените всю элегантность рекурсии. Процесс выглядит примерно так:

1. `main` вызывает `scanDirectory` с путем `"go"`.
2. `scanDirectory` выводит переданный путь `"go"` — текущий каталог, с которым работает функция.
3. Функция вызывает `ioutil.ReadDir` для пути `"go"`.
4. Возвращенный сегмент содержит всего один элемент: `"src"`.
5. Вызов `filepath.Join` для текущего каталога `"go"` и имени файла `"src"` дает новый путь `"go/src"`.
6. *src* является подкаталогом, поэтому `scanDirectory` вызывается снова — на этот раз для пути `"go/src"`. ← *Рекурсия!*
7. `scanDirectory` выводит новый путь: `"go/src"`.
8. Функция вызывает `ioutil.ReadDir` для пути `"go/src"`.
9. Первым элементом возвращенного сегмента является `"geo"`.
10. Вызов `filepath.Join` для текущего каталога `"go/src"` и имени файла `"geo"` дает новый путь `"go/src/geo"`.
11. *geo* является подкаталогом, поэтому `scanDirectory` вызывается снова — на этот раз для пути `"go/src/geo"`. ← *Рекурсия!*
12. `scanDirectory` выводит новый путь: `"go/src/geo"`.
13. Функция вызывает `ioutil.ReadDir` для пути `"go/src/geo"`.
14. Первым элементом возвращенного сегмента является `"coordinates.go"`.
15. *coordinates.go не является* каталогом, поэтому функция просто выводит имя файла.
16. И так далее.

Рекурсивные функции в основном сложнее обычных нерекурсивных функций и часто требуют больше вычислительных ресурсов. Но иногда рекурсивные функции справляются с задачами, которые было бы очень трудно решить другими способами.

Итак, программы *files.go* готовы, и наше отступление можно считать завершенным. А теперь вернемся к обсуждению средств обработки ошибок в Go.



Обработка ошибок в рекурсивной функции

Если функция `scanDirectory` обнаруживает ошибку во время сканирования любого подкаталога (например, если у пользователя отсутствуют разрешения для обращения к каталогу), она возвращает ошибку. И это ожидаемое поведение; программа не контролирует файловую систему, и очень важно сообщать об ошибках, когда они неизбежно возникают в программе.

Но если добавить пару команд `Printf` для вывода возвращаемых ошибок, мы видим, что *способ* обработки ошибок не идеален:

```
Shell Edit View Window Help
$ go run files.go
go
go/src
go/src/geo
go/src/geo/coordinates.go
go/src/geo/landmark.go
go/src/locked
2018/04/09 19:09:21 open
go/src/locked: permission denied
exit status 1
```

```
func scanDirectory(path string) error {
    fmt.Println(path)
    files, err := ioutil.ReadDir(path)
    if err != nil {
        fmt.Printf("Returning error from scanDirectory(\"%s\") call\n", path)
        return err
    }

    for _, file := range files {
        filePath := filepath.Join(path, file.Name())
        if file.IsDir() {
            err := scanDirectory(filePath)
            if err != nil {
                fmt.Printf("Returning error from scanDirectory(\"%s\") call\n", path)
                return err
            }
        } else {
            fmt.Println(filePath)
        }
    }
    return nil
}

func main() {
    err := scanDirectory("go")
    if err != nil {
        log.Fatal(err)
    }
}
```

Вывод отладочной информации для ошибок в вызове ReadDir.

Вывод отладочной информации для ошибок в рекурсивном вызове scanDirectory.

Если ошибка происходит в одном из рекурсивных вызовов `scanDirectory`, эта ошибка должна быть возвращена вверх по всей цепочке, пока не достигнет функции `main`!

```
Shell Edit View Window Help
$ go run files.go
go
go/src
go/src/geo
go/src/geo/coordinates.go
go/src/geo/landmark.go
go/src/locked
Returning error from scanDirectory("go/src/locked") call
Returning error from scanDirectory("go/src") call
Returning error from scanDirectory("go") call
2018/06/11 11:01:28 open go/src/locked: permission denied
exit status 1
```

Запуск паники

Наша функция `scanDirectory` — один из редких примеров, в которых будет уместно инициировать ситуацию паники во время выполнения.

Мы уже встречались с паникой прежде. Вы видели эти ситуации при обращении к несуществующим индексам в массивах и сегментах:

```
notes := [7]string{"do", "re", "mi", "fa", "so", "la", "ti"}
for i := 0; i <= len(notes); i++ {
        fmt.Println(i, notes[i])
}
```

Наибольшее значение, которое будет достигнуто переменной «i», равно 7!

Возвращает длину массива 7.

```
0 do
1 re
2 mi
3 fa
4 so
5 la
6 ti
panic: runtime error: index out of range

goroutine 1 [running]:
main.main()
        /tmp/sandbox094804331/main.go:11 +0x140
```

Доступ к индексу 7 вызывает панику!

Также паника встречалась нам при ошибках утверждений типов (если не использовать необязательное логическое значение `ok`):

```
var player Player = gadget.TapePlayer{}
recorder := player.(gadget.TapeRecorder)
```

Утверждает, что исходным типом является TapeRecorder, хотя в действительности это TapePlayer...

Паника!

```
panic: interface conversion: main.Player
is gadget.TapePlayer, not gadget.TapeRecorder
```

Когда в программе происходит паника, текущая функция перестает выполняться, программа выводит сообщение и аварийно завершается.

Вы также можете вызвать панику самостоятельно, вызвав встроенную функцию `panic`.

```
package main

func main() {
        panic("oh, no, we're going down")
}
```

```
panic: oh, no, we're going down

goroutine 1 [running]:
main.main()
        /tmp/main.go:4 +0x40
```

Функция `panic` рассчитывает получить один аргумент, поддерживающий пустой интерфейс (то есть аргумент любого типа). Этот аргумент преобразуется в строку (при необходимости) и выводится в составе сообщения паники.

Трассировка стека

Каждая вызываемая функция должна вернуть управление функции, из которой была вызвана. Для этого, как и в любом другом языке программирования, в Go поддерживается **стек вызовов** — список вызовов функций, активных на данный момент.

Когда в программе возникает ситуация паники, в вывод включается **трассировка стека**, или перечень содержимого стека вызовов. Это может быть полезно для определения причин, вызвавших аварийное завершение программы.

```
package main

func main() {
      one()
}
func one() {
      two()
}
func two() {
      three()
}
func three() {
      panic("This call stack's too deep for me!")
}
```

Этот вызов функции добавляется в стек.

В стек добавляется еще один вызов.

Добавляется третий вызов.

Паника! Трассировка стека включает все предшествующие вызовы.

```
panic: This call stack's too deep for me!

goroutine 1 [running]:
main.three()
      /tmp/main.go:13 +0x40
main.two()
      /tmp/main.go:10 +0x20
main.one()
      /tmp/main.go:7 +0x20
main.main()
      /tmp/main.go:4 +0x20
```

Трассировка стека включает список вызовов функций.

Отложенные вызовы завершаются перед аварийным завершением

Когда в программе происходит паника, все отложенные вызовы функции все равно будут выполнены. Если существует несколько отложенных вызовов, они выполняются в порядке, обратном тому, в котором были отложены.

Следующий код откладывает два вызова `Println`, после чего инициирует ситуацию паники. Из верхней части вывода программы видно, что два вызова завершаются перед сбоем программы.

```
func main() {
   one()
}
func one() {
   defer fmt.Println("deferred in one()")
   two()
}
func two() {
   defer fmt.Println("deferred in two()")
   panic("Let's see what's been deferred!")
}
```

Этот вызов функции был отложен первым, поэтому будет выполнен последним.

Этот вызов функции был отложен последним, поэтому будет выполнен первым.

Отложенные вызовы завершаются перед сбоем.

```
deferred in two()
deferred in one()
panic: Let's see what's been deferred!

goroutine 1 [running]:
main.two()
   /tmp/main.go:14 +0xa0
main.one()
   /tmp/main.go:10 +0xa0
main.main()
   /tmp/main.go:6 +0x20
```

Использование panic с scanDirectory

Функция `scanDirectory`, показанная справа, изменена так, чтобы вместо возвращения значения ошибки вызвалась функция `panic`. Это сильно упрощает обработку ошибок.

Сначала мы удаляем возвращаемое значение ошибки из объявления `scanDirectory`. Если вызов `ReadDir` возвращает значение ошибки, оно вместо этого передается `panic`. Также код обработки ошибок можно удалить из рекурсивного вызова `scanDirectory` и из вызова `scanDirectory` в `main`.

```
package main

import (
    "fmt"
    "io/ioutil"
    "path/filepath"
)

func scanDirectory(path string) {
    fmt.Println(path)
    files, err := ioutil.ReadDir(path)
    if err != nil {
        panic(err)
    }

    for _, file := range files {
        filePath := filepath.Join(path, file.Name())
        if file.IsDir() {
            scanDirectory(filePath)
        } else {
            fmt.Println(filePath)
        }
    }
}

func main() {
    scanDirectory("go")
}
```

Возвращаемое значение ошибки уже не нужно.

Вместо того чтобы возвращать значение ошибки, мы передаем его panic.

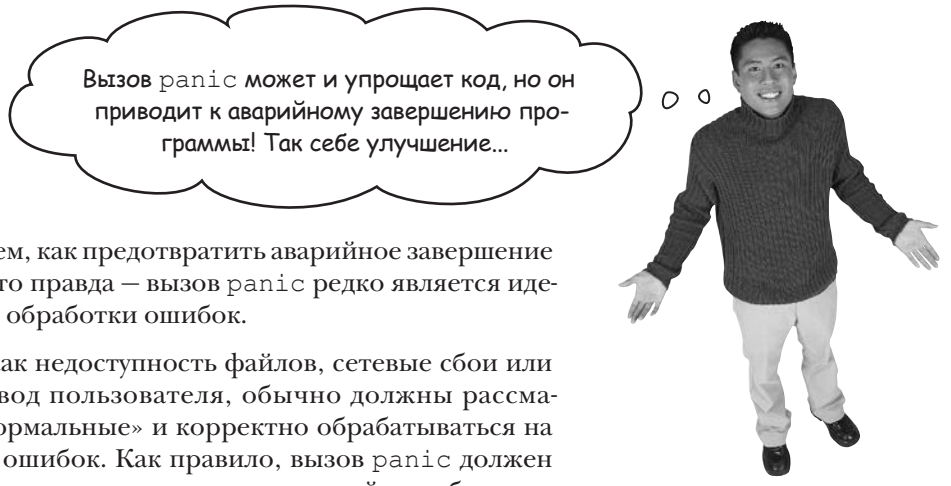
Сохранять или проверять возвращаемое значение ошибки уже не нужно.

Сохранять или проверять возвращаемое значение ошибки уже не нужно.

Теперь при обнаружении ошибки в процессе чтения каталога `scanDirectory` просто инициирует ситуацию паники. Все рекурсивные вызовы `scanDirectory` возвращают управление.

```
Shell Edit View Window Help
$ go run files.go
go
go/src
go/src/geo
go/src/geo/coordinates.go
go/src/geo/landmark.go
go/src/locked
panic: open go/src/locked: permission denied

goroutine 1 [running]:
main.scanDirectory(0xc420014220, 0xd)
        /Users/jay/files.go:37 +0x29a
main.scanDirectory(0xc420014130, 0x6)
        /Users/jay/files.go:43 +0x1ed
main.scanDirectory(0x10c4148, 0x2)
        /Users/jay/files.go:43 +0x1ed
main.main()
        /Users/jay/files.go:52 +0x36
exit status 2
```

Когда стоит паниковать



Вскоре мы покажем, как предотвратить аварийное завершение программы. Но это правда — вызов `panic` редко является идеальным способом обработки ошибок.

Такие факторы, как недоступность файлов, сетевые сбои или некорректный ввод пользователя, обычно должны рассматриваться как «нормальные» и корректно обрабатываться на уровне значений ошибок. Как правило, вызов `panic` должен резервироваться для «невозможных» ситуаций: ошибок, свидетельствующих об ошибке в программе, а не об ошибках со стороны пользователя.

Следующая программа использует `panic` для представления ошибок в программе. Программа выдает приз, спрятанный за одной из трех виртуальных дверей. Переменная `doorNumber` заполняется не значением, введенным пользователем, а случайным числом, выбранным функцией `rand.Intn`. Если `doorNumber` содержит любое другое число, кроме `1`, `2` или `3`, это не ошибка пользователя, а ошибка в программе.

Будет логично вызвать `panic`, если `doorNumber` содержит недопустимое значение. Такого никогда *не должно* быть, но если все же произойдет, программу лучше остановить, пока она не сотворила что-то непредсказуемое.

```
package main

import (
        "fmt"
        "math/rand"
        "time"
)

func awardPrize() {
        doorNumber := rand.Intn(3) + 1
        if doorNumber == 1 {
                fmt.Println("You win a cruise!")
        } else if doorNumber == 2 {
                fmt.Println("You win a car!")
        } else if doorNumber == 3 {
                fmt.Println("You win a goat!")
        } else {
                panic("invalid door number")
        }
}

func main() {
        rand.Seed(time.Now().Unix())
        awardPrize()
}
```

Генерирует случайное число от 1 до 3.

Другие числа генерироваться не должны, но если это все же произойдет — это повод для паники.

```
You win a cruise!
```

Упражнение

Внизу приведен пример кода и результат его выполнения. В выводе встречаются пропуски. Удастся ли вам заполнить их?

```
package main

import "fmt"

func snack() {
       defer fmt.Println("Closing refrigerator")
       fmt.Println("Opening refrigerator")
       panic("refrigerator is empty")
}

func main() {
       snack()
}
```

Результат:

```
Opening refrigerator
Closing refrigerator
panic: refrigerator is empty

goroutine 1 [running]:
main._____()
       /tmp/main.go:8 +0xe0
main.main()
       /tmp/main.go:12 +0x20
```

Ответ на с. 412.

Функция recover

Изменение функции `scanDirectory` для использования `panic` вместо возвращения ошибки сильно упростило бы код обработки ошибок. Однако инициирование паники также привело бы к тому, что программа стала бы аварийно завершаться с выдачей невразумительной трассировки стека. Вместо этого было бы лучше вывести для пользователей сообщение об ошибке.

Go предоставляет встроенную функцию `recover`, которая поможет избежать паники в программе. Необходимо только воспользоваться этой функцией для корректного завершения программы.

При вызове `recover` во время нормального выполнения программы функция просто возвращает `nil` и больше ничего не делает:

```
package main

import "fmt"

func main() {
      fmt.Println(recover())
}
```

Если вызвать recover в программе, в которой не возникла паника...

```
<nil>
```

...будет только возвращено nil.

Если вызвать `recover` для программы, в которой происходит паника, вызов прекратит панику. Но когда вы вызываете функцию `panic` в функции, эта функция перестает выполняться. Таким образом, вызывать `recover` в одной функции с `panic` бессмысленно, потому что паника все равно продолжится:

```
func freakOut() {
      panic("oh no")
      recover()
}
func main() {
      freakOut()
      fmt.Println("Exiting normally")
}
```

Из-за паники оставшаяся часть функции freakOut не выполняется...

...поэтому этот код не будет выполнен!

Все равно происходит сбой!

```
panic: oh no

goroutine 1 [running]:
main.freakOut()
      /tmp/main.go:4 +0x40
main.main()
      /tmp/main.go:8 +0x20
```

Тем не менее *существует* возможность вызвать `recover` во время паники в программе. Даже при панике все отложенные вызовы функций будут завершены. Таким образом, вызов `recover` можно разместить в отдельной функции и использовать `defer` при вызове этой функции перед кодом, в котором возникает паника.

```
func calmDown() {
      recover()
}
func freakOut() {
      defer calmDown()
      panic("oh no")
}
func main() {
      freakOut()
      fmt.Println("Exiting normally")
}
```

Разместить вызов recover в другой функции.

Отложить вызов функции с recover.

Если в программе после этого возникнет паника, отложенный вызов функции позволит провести восстановление!

Программа выходит нормально.

```
Exiting normally
```

Функция recover (продолжение)

Вызов `recover` *не приведет* к продолжению выполнения программы с места возникновения паники, по крайней мере именно с того места. Функция, в которой возникла паника, немедленно возвращает управление, и никакой код в блоке этой функции, следующий за местом возникновения паники, выполняться не будет. Однако после того, как функция, в которой возникла паника, вернет управление, продолжится нормальное выполнение программы.

```
func calmDown() {
      recover()
}
func freakOut() {
      defer calmDown()
      panic("oh no")
      fmt.Println("I won't be run!")
}
func main() {
      freakOut()
      fmt.Println("Exiting normally")
}
```

При восстановлении freakOut возвращает управление в этой точке.

Код после вызова panic не будет выполняться!

Но этот код будет выполнен после возвращения из freakOut.

```
Exiting normally
```

Возвращаемое значение recover

Как упоминалось ранее, при отсутствии паники вызовы `recover` возвращают `nil`.

Если вызвать recover в программе, в которой нет паники...

```
func main() {
      fmt.Println(recover())
}
```

```
<nil>
```

...будет только возвращено nil.

Но если паника *возникла*, `recover` вернет значение, переданное при вызове `panic`. Например, им можно воспользоваться для получения дополнительной информации о причине паники, для восстановления или вывода сообщений об ошибках для пользователя.

```
func calmDown() {
      fmt.Println(recover())
}
func main() {
      defer calmDown()
      panic("oh no")
}
```

Вызывает recover и выводит значение, переданное panic.

Значение, которое будет возвращено из recover.

```
oh no
```

Возвращаемое значение recover (продолжение)

Представляя функцию panic, мы упомянули о том, что ее аргумент имеет тип interface{} (пустой интерфейс), поэтому panic может принимать любое значение. Аналогичным образом возвращаемое значение recover тоже имеет тип interface{}. Возвращаемое значение recover также можно передать функциям fmt — таким, как Println (получающим значения interface{}), но вызывать для него методы напрямую невозможно.

В приведенном ниже коде значение ошибки передается функции panic. При этом ошибка преобразуется в значение interface{}. Когда позднее отложенная функция вызовет recover, будет возвращено это значение interface{}. Таким образом, даже если базовое значение содержит метод Error, попытка вызова Error для значения interface{} приводит к ошибке компиляции.

```
func calmDown() {
       p := recover()
       fmt.Println(p.Error())
}
func main() {
       defer calmDown()
       err := fmt.Errorf("there's an error")
       panic(err)
}
```

Возвращает значение interface{}.

И хотя базовое значение «error» содержит метод Error, у значения interface{} этот метод недоступен!

Вместо строки «panic» передается значение ошибки.

Ошибка компиляции!

```
p.Error undefined (type interface {} is interface with no methods)
```

Чтобы вызывать методы или сделать что-либо еще со значением паники, необходимо преобразовать его обратно к базовому типу с помощью утверждения типа.

Ниже приведена новая версия кода, которая берет возвращаемое значение recover и преобразует его обратно в значение ошибки. После того как это будет сделано, можно будет безопасно вызвать метод Error.

```
func calmDown() {
       p := recover()
       err, ok := p.(error)
       if ok {
              fmt.Println(err.Error())
       }
}
func main() {
       defer calmDown()
       err := fmt.Errorf("there's an error")
       panic(err)
}
```

Утверждается, что фактическим типом значения паники является «error».

Получив значение «error», мы можем вызвать для него метод Error.

```
there's an error
```

Восстановление после паники в scanDirectory

Когда мы расстались с программой *files.go*, добавление вызова `panic` в функции `scanDirectory` позволило отказаться от громоздкого кода обработки ошибок, но программа стала аварийно завершаться. Мы можем воспользоваться всем, что узнали о `defer`, `panic` и `recover`, для вывода сообщения об ошибке и корректного завершения программы.

Для этого в программу добавляется функция `reportPanic`, которая будет вызываться в `main` с ключевым словом `defer`. Это делается *до* вызова `scanDirectory`, который теоретически может вызвать панику.

В `reportPanic` мы вызываем `recover` и сохраняем возвращенное значение. Если в программе возникнет паника, этот вызов ее остановит.

Но при вызове `reportPanic` *неизвестно*, действительно ли в программе произошла паника или нет. Отложенный вызов `reportPanic` будет сделан независимо от того, вызывалась функция `panic` в `scanDirectory` или нет. Итак, прежде всего проверим, равно ли `nil` значение паники, возвращенное из `recover`. Если это так, значит, паники не было, и мы возвращаем управление из `reportPanic`, ничего более не делая.

Но если значение паники *не равно* `nil`, это означает, что в программе возникла ситуация паники и о ней необходимо сообщить.

Так как `scanDirectory` передает значение ошибки `panic`, мы используем утверждение типа для преобразования значения паники `interface{}` в значение ошибки. Если преобразование прошло успешно, программа выводит значение ошибки.

После внесения этих изменений вместо некрасивой трассировки стека пользователь увидит нормальное сообщение об ошибке!

```
Shell Edit View Window Help
$ go run files.go
go
go/src
go/src/geo
go/src/geo/coordinates.go
go/src/geo/landmark.go
go/src/locked
open go/src/locked: permission denied
```

```
package main

import (
    "fmt"
    "io/ioutil"
    "path/filepath"
)

func reportPanic() {
    p := recover()
    if p == nil {
        return
    }
    err, ok := p.(error)
    if ok {
        fmt.Println(err)
    }
}

func scanDirectory(path string) {
    fmt.Println(path)
    files, err := ioutil.ReadDir(path)
    if err != nil {
        panic(err)
    }

    for _, file := range files {
        filePath := filepath.Join(path, file.Name())
        if file.IsDir() {
            scanDirectory(filePath)
        } else {
            fmt.Println(filePath)
        }
    }
}

func main() {
    defer reportPanic()
    scanDirectory("go")
}
```

Добавляем новую функцию.

Вызываем «recover» и сохраняем возвращенное значение.

Если «recover» возвращает nil, паники нет...

...поэтому ничего не делается.

В противном случае получаем базовое значение «error»...

...и выводим его.

Прежде чем вызывать код, в котором может возникнуть паника, откладываем вызов новой функции reportPanic.

Возобновление состояния паники

С `reportPanic` связана одна потенциальная проблема, которую необходимо решить. В данный момент функция перехватывает *любую* панику, даже если она не происходит из `scanDirectory`. А если значение паники не может быть преобразовано в тип `error`, `reportPanic` его не выведет.

Чтобы протестировать эту ситуацию, добавьте в `main` другой вызов `panic` со строковым аргументом:

```
func main() {
        defer reportPanic()
        panic("some other issue")
        scanDirectory("go")
}
```

Создается новая ситуация паники со строковым значением паники.

```
Shell Edit View Window Help
$ go run files.go
$
```

Результата нет!

Функция `reportPanic` восстанавливается от новой паники, но поскольку значение паники не является `error`, оно не будет выведено. И пользователю остается только гадать, почему программа завершилась!

Типичная стратегия для обработки непредвиденной паники, к восстановлению после которой вы не готовы, основана на простом возобновлении состояния паники. Повторная паника обычно уместна, потому что это непредвиденная ситуация.

Справа приведена обновленная версия `reportPanic` для обработки непредвиденной паники. Если утверждение типа для преобразования значения паники в `error` выполняется успешно, мы просто выводим его, как и прежде. Но если оно завершается неудачей, мы просто повторно вызываем `panic` с тем же значением.

Повторный запуск *files.go* показывает, что исправление работает: `reportPanic` восстанавливается после тестового вызова `panic`, но затем снова переходит в состояние паники, когда утверждение типа завершается неудачей. Теперь можно удалить вызов `panic` в `main` и быть уверенным в том, что пользователь будет оповещен о любой непредвиденной ситуации паники!

```
Shell Edit View Window Help
$ go run files.go
panic: some other issue [recovered]
        panic: some other issue

goroutine 1 [running]:
main.reportPanic()
        /Users/jay/files.go:27 +0xd7
panic(0x109ee80, 0x10d1c80)
        /go/.../panic.go:505 +0x229
main.main()
        /Users/jay/files.go:52 +0x55
exit status 2
```

```
func reportPanic() {
    p := recover()
    if p == nil {
        return
    }
    err, ok := p.(error)
    if ok {
        fmt.Println(err)
    } else {
        panic(p)
    }
}
```

Если значение паники не является признаком ошибки, возобновить панику с тем же значением.

```
func scanDirectory(path string) {
    fmt.Println(path)
    files, err := ioutil.ReadDir(path)
    if err != nil {
        panic(err)
    }
    // ...
}
```

Не забудьте удалить эту тестовую панику, когда убедитесь, что reportPanic работает!

```
func main() {
    defer reportPanic()
    panic("some other issue")
    scanDirectory("go")
}
```

Часто Задаваемые Вопросы

В: Я видел другие языки программирования, в которых реализована система «исключений». Похоже, функции `panic` и `recover` работают схожим образом. Можно ли использовать их как исключения?

О: Мы, как и создатели языка Go, настоятельно рекомендуем так не поступать. Можно даже сказать, что сама архитектура языка не способствует использованию `panic` и `recover`. В своем докладе на конференции в 2012 году Роб Пайк (один из создателей Go) описывал `panic` и `recover` как «намеренно неудобные». Это означает, что при проектировании Go его создатели не пытались сделать `panic` и `recover` простыми или приятными в использовании, они хотели, чтобы эти функции использовались *реже*.

Так проектировщики языка Go отреагировали на один из главных недостатков исключений: они существенно усложняют логику программы. Вместо этого разработчикам рекомендуется обрабатывать ошибки точно так же, как они обрабатываются в других частях программы: в командах `if` и `return` наряду со значениями ошибок. Конечно, прямая обработка ошибок внутри функции несколько удлиняет код этой функции, но это лучше, чем полный отказ от обработки ошибок. (Создатели Go обнаружили, что многие разработчики, использующие исключения, просто инициируют исключение, не обеспечивая его должной обработки в будущем.) Прямая обработка ошибок также более наглядно показывает, что происходит при возникновении ошибки — вам не нужно просматривать другую ветвь логики программы, чтобы увидеть код обработки ошибок.

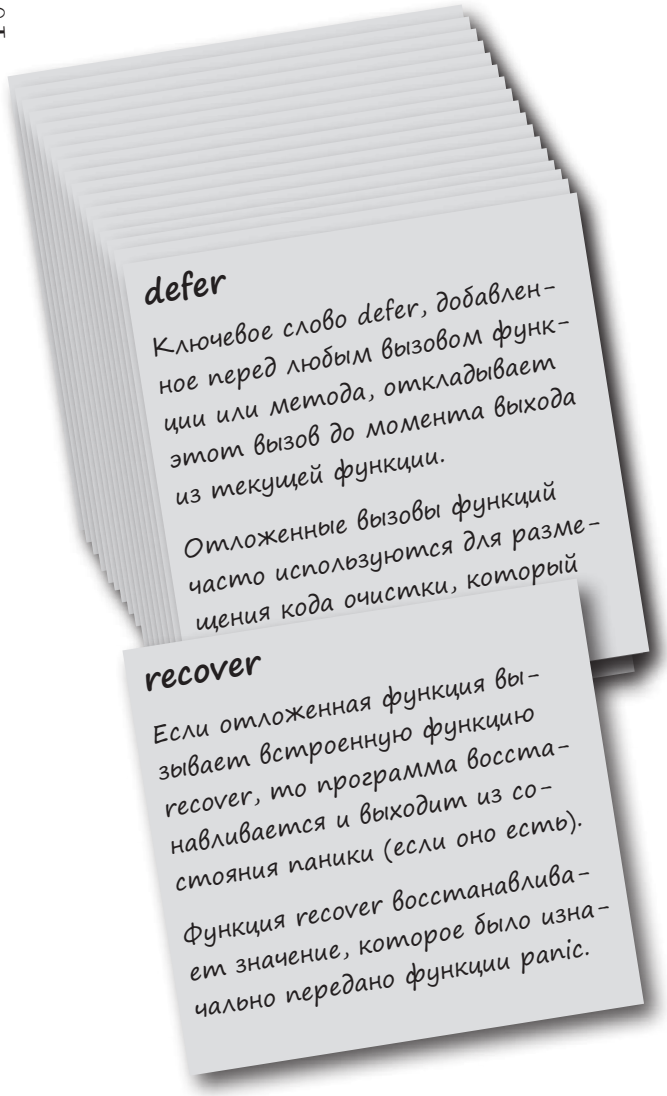
Итак, не ищите эквивалент исключений в Go. Этот механизм был опущен намеренно. Возможно, разработчики с опытом использования исключений не сразу привыкнут к новым средствам обработки ошибок, но создатели Go полагают, что в итоге это приведет к созданию более качественных программных продуктов.

Тезисы доклада Роба Пайка доступны по адресу https://talks.golang.org/2012/splash.article#TOC_16.

Ваш инструментарий Go

Глава 12 подошла к концу! В ней ваш инструментарий пополнился отложенными вызовами функций и восстановлением в ситуации паники.

ГЛАВА 12



КЛЮЧЕВЫЕ МОМЕНТЫ

- Ранний возврат из функции со значением ошибки — хороший механизм оповещения об ошибках, но он может помешать выполнению кода очистки функции, размещенному в конце функции.
- Ключевое слово `defer` позволяет вызвать функцию завершения сразу же после кода, требующего завершающих действий. Код очистки будет выполнен при выходе из текущей функции независимо от того, была ошибка или нет.
- Чтобы создать ситуацию паники в программе, можно вызвать встроенную функцию `panic`.
- Без вызова встроенной функции `recover` программа, в которой возникла ситуация паники, аварийно завершается с выдачей сообщения.
- В аргументе `panic` может передаваться любое значение. Это значение преобразуется в строку и выводится как часть сообщения при аварийном завершении.
- Сообщение при аварийном завершении включает трассировку стека — список всех активных вызовов функций, который может пригодиться для отладки.
- При возникновении паники в программе все отложенные вызовы функций все равно будут выполнены, так что код очистки будет выполнен перед выходом из программы.
- Отложенные функции также могут вызывать встроенную функцию `recover`, что приведет к возобновлению нормального выполнения программы.
- Если функция `recover` вызывается при отсутствии паники, она просто возвращает `nil`.
- Если функция `recover` вызывается во время паники, она возвращает значение, переданное при вызове `panic`.
- В большинстве программ паника должна возникать только в случае непредвиденной ошибки. Всегда учитывайте все возможные ошибки, с которыми может столкнуться ваша программа (например, отсутствие файлов или неправильно отформатированные данные), и обрабатывайте их с использованием значений ошибок.

Развлечения с магнитами. Решение

```
func find(item string, slice []string) bool {
       for _, sliceItem := range slice {
              if item == sliceItem {
                     return true
              }
       }
       return false
}

type Refrigerator []string

func (r Refrigerator) Open() {
       fmt.Println("Opening refrigerator")
}
func (r Refrigerator) Close() {
       fmt.Println("Closing refrigerator")
}

func main() {
       fridge := Refrigerator{"Milk", "Pizza", "Salsa"}
       for _, food := range []string{"Milk", "Bananas"} {
              err := fridge.FindFood(food)
              if err != nil {
                     log.Fatal(err)
              }
       }
}
```

```
func (r Refrigerator) FindFood(food string) error {

        r.Open()

        defer  r.Close()

        if find(food, r) {
               fmt.Println("Found", food)
        } else {
               return fmt.Errorf("%s not found", food)
        }

        return nil

}
```

Close будет вызываться при выходе из FindFood независимо от того, возникла ошибка или нет.

Метод Close типа Refrigerator вызывается, если продукт будет обнаружен.

Метод Close также вызывается и в том случае, если продукт не найден.

```
Opening refrigerator
Found Milk
Closing refrigerator
Opening refrigerator
Closing refrigerator
2018/04/09 22:12:37 Bananas not found
```

Упражнение
Решение

Внизу приведен пример кода и результат его выполнения. В выводе встречаются пропуски. Удастся ли вам заполнить их?

```
package main

import "fmt"

func snack() {
       defer fmt.Println("Closing refrigerator")
       fmt.Println("Opening refrigerator")
       panic("refrigerator is empty")
}

func main() {
       snack()
}
```

Результат:

Этот вызов был отложен, поэтому выполняется до выхода из функции snack (во время паники).

```
Opening refrigerator
Closing refrigerator
panic: refrigerator is empty

goroutine 1 [running]:
main. snack ()
       /tmp/main.go:8 +0xe0
main.main()
       /tmp/main.go:12 +0x20
```

Горутины и каналы

Одновременная работа над одной задачей не всегда быстрее всего приводит к цели. Некоторые большие задачи можно разбить на мелкие. **Горутины (goroutines)** позволяют программам работать над несколькими задачами одновременно. Они координируют свою работу при помощи **каналов**, по которым могут отправлять данные друг другу *и* синхронизировать выполнение, чтобы одна горутина не опережала другую. Горутины позволяют использовать всю мощь многопроцессорных компьютеров, чтобы программы выполнялись как можно быстрее!

Загрузка веб-страниц

Эта глава посвящена тому, как ускорить выполнение работы за счет одновременного выполнения нескольких задач. Но для начала нам понадобится большая задача, которую можно будет разбить на несколько мелких. Поэтому потерпите немного, пока мы подготовим самое необходимое...

Чем меньше веб-страница, тем быстрее она загружается в браузерах посетителей. Нам понадобится инструмент для измерения размера страниц в байтах. К счастью, благодаря стандартной библиотеке Go найти его будет несложно. В следующей программе пакет `net/http` используется для соединения с сайтом и загрузки веб-страницы всего несколькими вызовами функций.

URL-адрес нужного сайта передается функции `http.Get`. Она возвращает объект `http.Response`, а также любые обнаруженные ошибки.

Объект `http.Response` представляет собой структуру с полем `Body`, представляющим содержимое страницы. `Body` поддерживает интерфейс `ReadCloser` уровня пакета, это означает, что оно содержит метод `Read` (для чтения данных страницы) и метод `Close` для освобождения сетевого подключения при завершении работы.

Вызов `Close` откладывается при помощи `defer`, чтобы подключение было освобождено после завершения чтения данных. Затем тело ответа передается функции `ReadAll` пакета `ioutil`, который читает все его содержимое и возвращает в виде сегмента значений `byte`.

Мы еще не рассматривали тип `byte`; это один из базовых типов Go (как `float64` или `bool`), используемый для хранения низкоуровневых данных, например, прочитанных из файла или сетевого подключения. Сегмент значений `byte` не выведет никакой осмысленной информации, если вывести его напрямую, но если выполнить преобразование типа из сегмента значений `byte` в строку, вы получите осмысленный текст. (Конечно, при условии, что данные представляют осмысленный текст.) Итак, программа преобразует тело ответа в строку и выводит ее.

```
package main

import (
        "fmt"
        "io/ioutil"
        "log"
        "net/http"
)

func main() {
        response, err := http.Get("https://example.com")
        if err != nil {
                log.Fatal(err)
        }
        defer response.Body.Close()
        body, err := ioutil.ReadAll(response.Body)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(string(body))
}
```

Вызывает http.Get с URL-адресом нужной страницы.

Освобождает сетевое подключение после выхода из функции «main».

Читает все данные из ответа.

Преобразует данные в строку и выводит ее.

Содержимое HTML-страницы.

```
File Edit Window Help
$ go run temp.go
<!doctype html>
<html>
<head>
    <title>Example Domain</title>
    <meta charset="utf-8" />
...
```

Если сохранить этот код в файле и запустить его командой `go run`, программа загрузит HTML-содержимое страницы *https://example.com* и выведет его в терминале.

Загрузка веб-страниц (продолжение)

Если нужна более подробная информация о функциях и типах, используемых в программе, ее можно вывести командой `go doc` (см. главу 4) в терминале. Попробуйте ввести команду справа, чтобы открыть документацию. (Или при желании просмотрите ее в браузере при помощи своей любимой поисковой системы.)

```
File  Edit  Window  Help
go doc http Get
go doc http Response
go doc io ReadCloser
go doc ioutil ReadAll
```

Документация Go содержит более подробную информацию о работе этой программы!

После этого можно без труда преобразовать программу, чтобы она выводила размеры нескольких страниц.

Код загрузки страницы можно переместить в отдельную функцию `responseSize`, которая получает URL-адрес загрузки в параметре. Загружаемый URL-адрес выводится исключительно для отладочных целей. Код вызова `http.Get`, чтения ответа и освобождения подключения в основном остается неизменным. Наконец, вместо преобразования сегмента байтов из ответа в строку мы просто вызываем `len` для получения длины сегмента. В результате мы получаем длину ответа в байтах, которая выводится программой.

Функция `main` обновляется для вызова `responseSize` с несколькими разными URL-адресами. Запущенная программа выводит URL-адреса и размеры страниц.

```
package main

import (
        "fmt"
        "io/ioutil"
        "log"
        "net/http"
)

func main() {
        responseSize("https://example.com/")
        responseSize("https://golang.org/")
        responseSize("https://golang.org/doc")
}

func responseSize(url string) {
        fmt.Println("Getting", url)
        response, err := http.Get(url)
        if err != nil {
                log.Fatal(err)
        }
        defer response.Body.Close()
        body, err := ioutil.ReadAll(response.Body)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(len(body))
}
```

Получает размеры нескольких страниц.

Получает URL-адрес в параметре.

Код получения страницы перемещается в отдельную функцию.

Выводит загружаемый URL-адрес.

Получает страницу по заданному URL-адресу.

Размер сегмента в байтах равен размеру страницы.

```
Getting https://example.com/
1270
Getting https://golang.org/
8766
Getting https://golang.org/doc
13078
```

URL-адреса и размеры страниц (в байтах).

Многозадачность

А теперь мы подошли к сути главы: ускорению работы программ за счет одновременного выполнения нескольких задач.

Наша программа выдает несколько вызовов `responseSize`, по одному за раз. Каждый вызов `responseSize` устанавливает сетевое подключение к сайту, дожидается ответа сайта, выводит размер ответа и возвращает управление. Только после того, как один вызов `responseSize` вернет управление, сможет начаться следующий вызов. Если у нас будет только одна длинная функция, в которой весь код повторяется три раза, то ее выполнение займет столько же времени, как с тремя нашими вызовами `responseSize`.

Три последовательных вызова responseSize занимают столько времени...

```
fmt.Println("Getting", url)
response, err := http.Get(url)
if err != nil {
        log.Fatal(err)
}
defer response.Body.Close()
body, err := ioutil.ReadAll(
        response.Body)
if err != nil {
        log.Fatal(err)
}
fmt.Println(len(body))

fmt.Println("Getting", url)
response, err := http.Get(url)
if err != nil {
        log.Fatal(err)
}
defer response.Body.Close()
body, err := ioutil.ReadAll(
        response.Body)
if err != nil {
        log.Fatal(err)
}
fmt.Println(len(body))

fmt.Println("Getting", url)
response, err := http.Get(url)
if err != nil {
        log.Fatal(err)
}
defer response.Body.Close()
body, err := ioutil.ReadAll(
        response.Body)
if err != nil {
        log.Fatal(err)
}
fmt.Println(len(body))
```

Но что, если бы все три вызова `responseSize` могли выполняться одновременно? Программа сможет завершиться всего за треть исходного времени!

Если все вызовы responseSize будут выполняться одновременно, программа выполнится намного быстрее!

```
fmt.Println("Getting", url)
response, err := http.Get(url)
if err != nil {
        log.Fatal(err)
}
defer response.Body.Close()
body, err := ioutil.ReadAll(
        response.Body)
if err != nil {
        log.Fatal(err)
}
fmt.Println(len(body))
```

```
fmt.Println("Getting", url)
response, err := http.Get(url)
if err != nil {
        log.Fatal(err)
}
defer response.Body.Close()
body, err := ioutil.ReadAll(
        response.Body)
if err != nil {
        log.Fatal(err)
}
fmt.Println(len(body))
```

```
fmt.Println("Getting", url)
response, err := http.Get(url)
if err != nil {
        log.Fatal(err)
}
defer response.Body.Close()
body, err := ioutil.ReadAll(
        response.Body)
if err != nil {
        log.Fatal(err)
}
fmt.Println(len(body))
```

Конкурентность на базе горутин

Когда `responseSize` вызывает `http.Get`, вашей программе приходится ожидать ответа сайта. Во время ожидания она не делает ничего полезного.

Возможно, другой программе придется дожидаться пользовательского ввода. А еще одной программе придется ждать чтения данных из файла. Существует множество ситуаций, в которых программам приходится просто простаивать в ожидании.

Горутины обеспечивают возможность конкурентности: приостановки одной задачи для работы над другими задачами. А в других ситуациях они позволяют реализовать параллелизм: одновременную работу над несколькими задачами!

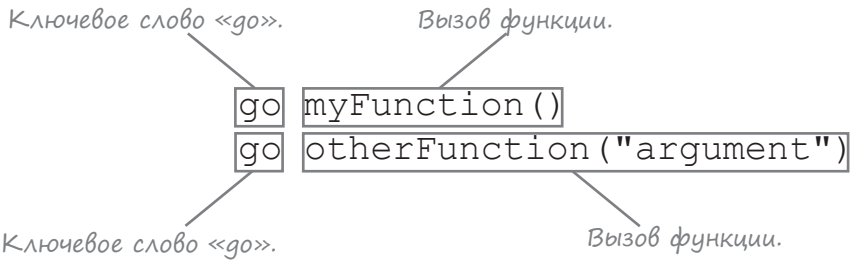
Конкурентность позволяет программе приостановить одну задачу и работать над другими задачами. Программа, ожидающая ввода от пользователя, может выполнять другие действия в фоновом режиме. Во время чтения из файла программа может обновлять индикатор прогресса. Наша программа `responseSize` может выдавать другие сетевые запросы, ожидая завершения первого запроса.

Если программа написана с поддержкой конкурентности, она также может поддерживать **параллельное выполнение**: *одновременное* выполнение задач. Компьютер с одним процессором может выполнять только одну задачу за раз. Тем не менее большинство современных компьютеров содержит несколько процессоров (или один процессор с несколькими ядрами). Ваш компьютер может распределить несколько конкурентных задач между несколькими процессорами, чтобы выполнять их одновременно. (Вам редко приходится управлять ими вручную: операционная система обычно делает все за вас.)

Разбиение больших задач на меньшие подзадачи, которые могут выполняться конкурентно, иногда приводит к существенному приросту скорости ваших программ.

В Go конкурентно выполняемые задачи называются **горутинами**. В других языках программирования существует аналогичная концепция *потоков*, но горутины расходуют меньше компьютерной памяти, чем потоки, а также быстрее запускаются и останавливаются, а это означает, что вы можете запускать больше горутин одновременно.

Кроме того, горутины проще в использовании. Для запуска новой горутины используется `go`-команда — обычный вызов функции или метода, перед которым находится ключевое слово `go`:



Обратите внимание: мы говорим «*другую* горутину». Функция `main` каждой программы Go запускается с помощью горутины, так что в каждой программе Go выполняется по крайней мере одна горутина. Выходит, мы все это время пользовались горутинами, не подозревая об этом!

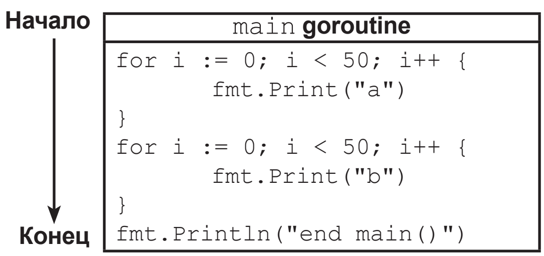
Использование горутин

Следующая программа вызывает функции по одной. В ней используется цикл, который выводит строку "a" 50 раз, а функция b выводит строку "b" 50 раз. Функция main вызывает a, затем b, а потом выводит сообщение при завершении.

```
package main

import "fmt"

func a() {
	for i := 0; i < 50; i++ {
		fmt.Print("a")
	}
}

func b() {
	for i := 0; i < 50; i++ {
		fmt.Print("b")
	}
}

func main() {
	a()
	b()
	fmt.Println("end main()")
}
```

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaabbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbend main()
```

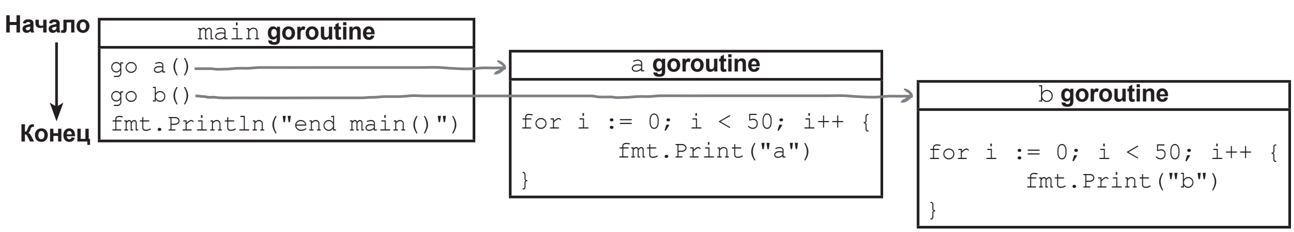
Все происходит так, как если бы функция main содержала весь код функции a, за которым следовал бы весь код функции b и собственный код main:



Чтобы запустить функции a и b в новых горутинах, достаточно добавить ключевое слово go перед вызовами функций:

```
func main() {
	go a()
	go b()
	fmt.Println("end main()")
}
```

Тогда новые горутины будут конкурентно выполняться в функции main:



Использование горутин (продолжение)

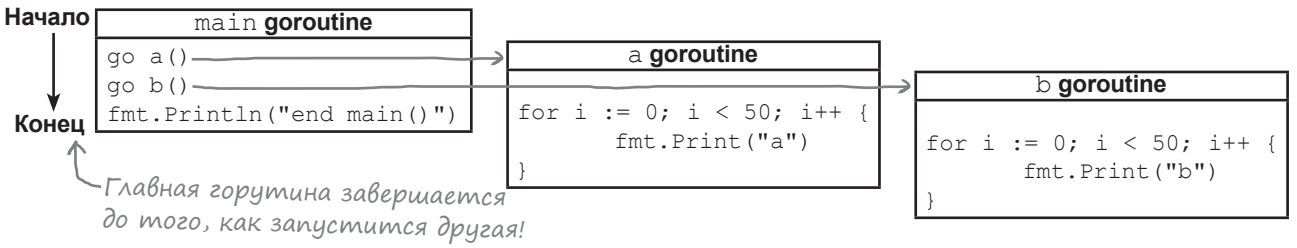
Но если запустить программу сейчас, то единственный результат, который мы увидим, будет выведен функцией `Println` в конце `main`. А вот функции `a` и `b` ничего не выводят!

```
func main() {
        go a()
        go b()
        fmt.Println("end main()")
}
```

Нет вывода функций «a» и «b»?

```
end main()
```

Проблема вот в чем: программы Go перестают выполняться сразу же после завершения горутины `main` (той, которая вызывает функцию `main`), даже если другие горутины продолжают выполняться. Наша функция `main` завершается до того, как код функций `a` и `b` получит возможность выполниться.



Горутина `main` должна продолжать выполнение до того, как горутины функций `a` и `b` смогут завершиться. Чтобы правильно реализовать эту последовательность выполнения, понадобится еще одно средство Go — так называемые *каналы*, но этот механизм будет рассматриваться чуть позже в этой главе. Итак, пока мы просто приостановим горутину `main` на заданный период времени, чтобы смогли выполниться другие горутины.

Для этого мы воспользуемся функцией `Sleep` из пакета `time`. Эта функция приостанавливает текущую горутину на заданный промежуток времени. Вызов `time.Sleep(time.Second)` из функции `main` заставит горутину `main` приостановить выполнение на 1 секунду.

```
func main() {
        go a()
        go b()
        time.Sleep(time.Second)
        fmt.Println("end main()")
}
```

Горутина main приостанавливается на 1 секунду.

Дает другим горутинам достаточно времени для выполнения.

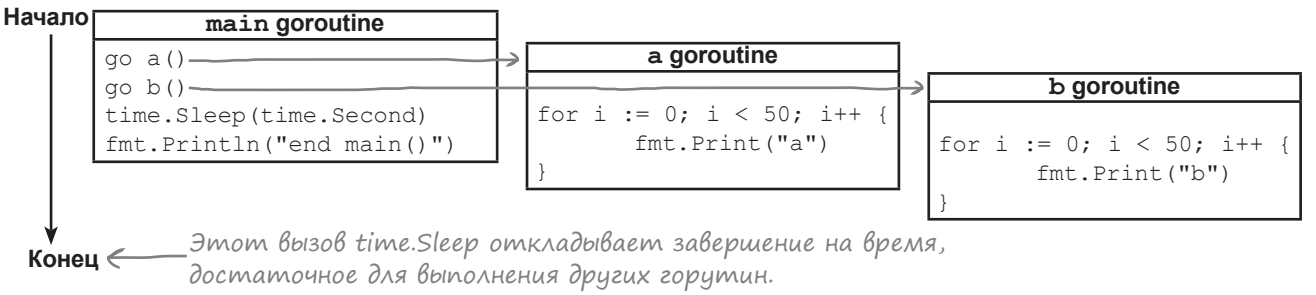
```
aaaaaaaaaaaaaaaaaaaaaaabbbbbaaa
aaaaaaaabbbbbbbbbbbaaaaaaaaaaa
abbaaaaabbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbend main()
```

Когда time.Sleep вернет управление, горутина main завершит выполнение.

Если повторно запустить программу, вы снова увидите выводы функций `a` и `b`, так как их горутины получат возможность выполниться. Вывод этих функций будет чередоваться, так как программа переключается между двумя горутинами. (Закономерность чередования может отличаться от той, которая показана здесь.) Когда Go-функция `main` снова активизируется, она вызовет `fmt.Println` и завершится.

Использование горутин (продолжение)

Вызов `time.Sleep` в главной горутине дает более чем достаточно времени для выполнения функций `a` и `b`.



Использование горутин с функцией responseSize

Программа для вывода размеров веб-страниц легко преобразуется для использования горутин. Для этого понадобится совсем немного — добавить ключевое слово `go` перед каждым вызовом `responseSize`.

Чтобы горутина `main` не завершилась перед тем, как горутины `responseSize` смогут завершиться, также нужно будет добавить вызов `time.Sleep` в функцию `main`.

```
package main

import (
        "fmt"
        "io/ioutil"
        "log"
        "net/http"
        "time"                    ← Добавляется пакет «time».
)

func main() {
        go responseSize("https://example.com/")
        go responseSize("https://golang.org/")
        go responseSize("https://golang.org/doc")
        time.Sleep(5 * time.Second)
}

func responseSize(url string) {
        fmt.Println("Getting", url)
        response, err := http.Get(url)
        if err != nil {
                log.Fatal(err)
        }
        defer response.Body.Close()
        body, err := ioutil.ReadAll(response.Body)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(len(body))
}
```

Вызовы responseSize преобразуются в go-команды.

Пятисекундная пауза.

Впрочем, приостановка всего на 1 секунду может оказаться недостаточной для завершения сетевых запросов. Вызов `time.Sleep(5 * time.Second)` обеспечит приостановку горутины на 5 секунд. (Если вы попробуете запустить эту программу с медленной или перегруженной сетью, возможно, это время придется увеличить.)

Использование горутин с функцией responseSize (продолжение)

Запустив обновленную программу, вы увидите, что она сразу выводит загружаемые URL-адреса, поскольку три горутины `responseSize` запускаются конкурентно.

Три вызова `http.Get` также выполняются конкурентно; программа не дожидается получения ответа, чтобы отправить следующий запрос. В результате три размера ответов с горутинами выводятся намного быстрее, чем в предыдущей, последовательной версии программы. Впрочем, выполнение программы все равно занимает 5 секунд, потому что это время указано в вызове `time.Sleep` для завершения `main`.

Все вызовы Println в начале responseSize выполняются одновременно.

Размеры ответов выводятся после получения ответа от каждого сайта.

```
Getting https://example.com/
Getting https://golang.org/doc
Getting https://golang.org/
1270
8766
13078
```

Мы совершенно не контролируем порядок выполнения вызовов `responseSize`, поэтому при повторном запуске программы может оказаться, что запросы выполняются в другом порядке.

Запросы страниц могут выдаваться в разном порядке.

```
Getting https://golang.org/doc
Getting https://golang.org/
Getting https://example.com/
1270
8766
13078
```

Завершение программы занимает 5 секунд даже в том случае, если все сайты отреагируют быстрее, так что переход на горутины все еще не обеспечивает особого выигрыша по времени. Что еще хуже, 5 секунд может оказаться *недостаточно,* если сайты не успеют ответить за это время. В отдельных случаях программа может завершиться до получения всех ответов.

Вызов time.Sleep может завершиться, а работа программы может закончиться до получения ответов от всех сайтов!

```
Getting https://golang.org/doc
Getting https://golang.org/
Getting https://example.com/
1270
```

Становится ясно, что `time.Sleep` — далеко не идеальный способ ожидания завершения других горутин. Через несколько страниц мы перейдем к рассмотрению каналов — более эффективному механизму для решения таких задач.

Порядок выполнения горутин

Вполне возможно, что горутины `responseSize` будут выполняться в разном порядке при каждом запуске программы:

```
Getting https://example.com/
Getting https://golang.org/doc
Getting https://golang.org/
```

```
Getting https://golang.org/doc
Getting https://golang.org/
Getting https://example.com/
```

Также невозможно предсказать, когда приведенная программа будет переключаться между горутинами `a` и `b`:

```
aaaaaabbbbbbbbbbbbbbb
bbbbbbbaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaab
bbbbbbbbbbbbbbbbbbbbb
bbbbaaaaaaaaend main()
```

```
bbbbbbbbbbbbbbbbbbbaaa
aaaabbbbbbbbbbbbbaaaaa
aaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaabbbbbb
bbbbbbbbbbbbend main()
```

```
aaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaa
aaaaaabbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbend main()
```

В обычных обстоятельствах Go не дает никаких гарантий относительно того, когда и на какое время будет происходить переключение между горутинами. Это позволяет организовать выполнение горутин наиболее эффективным образом. Но если порядок выполнения горутин для вас важен, необходимо синхронизировать их при помощи каналов (о которых мы вскоре расскажем).

Развлечения с магнитами

На холодильнике была выложена программа, использующая горутины. Удастся ли вам расставить фрагменты кода по местам и создать работоспособную программу, которая будет выводить результат, *похожий* на приведенный в примере? (Предсказать порядок выполнения горутин невозможно, так что не беспокойтесь — вывод вашей программы не обязан точно совпадать с приведенным.)

```
(s string)

repeat    repeat

time.Sleep(time.Second)    ()    ("x")

for i := 0; i < 25; i++ {
        fmt.Print(s)
}

go    ("y")

go

package main

import (
        "fmt"
        "time"
)

{    }    {    }

func repeat

func main
```

Один из возможных результатов.

```
yyyyyyyyyyyyyxxxxxxxxxxxxxyyyyyyyxxxxxxxxxxxxxyyyyyxx
```

Ответ на с. 434.

Горутины не могут использоваться с возвращаемыми значениями

С переходом на горутины возникает другая проблема, которую необходимо решить: в go-командах не могут использоваться возвращаемые значения функций. Допустим, вы захотели изменить функцию `responseSize`, чтобы она возвращала размер страницы (вместо вывода):

```
func main() {
        var size int
        size = go responseSize("https://example.com/")
        fmt.Println(size)
        size = go responseSize("https://golang.org/")
        fmt.Println(size)
        size = go responseSize("https://golang.org/doc")
        fmt.Println(size)
        time.Sleep(5 * time.Second)
}

func responseSize(url string) int {
        fmt.Println("Getting", url)
        response, err := http.Get(url)
        if err != nil {
                log.Fatal(err)
        }
        defer response.Body.Close()
        body, err := ioutil.ReadAll(response.Body)
        if err != nil {
                log.Fatal(err)
        }
        return len(body)
}
```

Этот код недействителен!

Добавляет возвращаемое значение.

Возвращает размер ответа вместо того, чтобы выводить его.

Ошибки компиляции.

```
./pagesize.go:13:9: syntax error: unexpected go, expecting expression
./pagesize.go:15:9: syntax error: unexpected go, expecting expression
./pagesize.go:17:9: syntax error: unexpected go, expecting expression
```

Вы получите сообщение об ошибках компиляции. Компилятор не позволяет получать возвращаемое значение из функции, вызванной в go-команде.

И это правильно. Вызывая `responseSize` как часть go-команды, вы тем самым говорите: «Выполни `responseSize` в отдельной горутине. Я собираюсь продолжить выполнение команд в этой функции». Функция `responseSize` не возвращает значение немедленно; она должна дождаться ответа от веб-сайта. Но код в горутине `main` рассчитывает получить возвращаемое значение немедленно, а этого значения еще нет!

Это относится к любым вызовам функций в go-командах, а не только к долго выполняемым функциям вроде `responseSize`. Нет гарантий того, что возвращаемые значения будут готовы вовремя, поэтому компилятор Go блокирует любые попытки их использования.

Означает «Выполни это немедленно; я не собираюсь ждать».

```
size = go responseSize("https://example.com/")
fmt.Println(size)
```

И каким будет возвращаемое значение?

Горутины не могут использоваться с возвращаемыми значениями (продолжение)

Компилятор Go не позволит использовать возвращаемое значение функции, вызванной в горутине, — нет гарантий того, что возвращаемое значение будет готово к моменту, когда мы пытаемся его использовать.

```
func greeting() string {
       return "hi"
}

func main() {
       fmt.Println(go greeting())
}
```

Функция вызывается как горутина.

Немедленно пытается использовать возвращаемое значение функции (которое может быть еще не готово).

Ошибка компиляции.

```
syntax error: unexpected go, expecting expression
```

Тем не менее механизм передачи данных между горутинами *существует*: для этого используются **каналы**. Они не только позволяют передавать значения из одной горутины в другую, но и гарантируют, что отправляющая горутина отправила значение перед тем, как получающая горутина попытается его использовать.

На практике каналы используются только для передачи данных из одной горутины в другую. Чтобы продемонстрировать работу с каналами, необходимо сделать следующее:

- Создать канал.
- Написать функцию, которая получает канал в параметре. Мы выполним эту функцию в отдельной горутине и используем ее для отправки значений по каналу.
- Получить отправленные значения в исходной горутине.

Каждый канал передает значения только заранее определенного типа; например, можно создать один канал для значений `int` и другой канал для значений с типом `struct`. Чтобы объявить переменную для хранения канала, используйте ключевое слово `chan` с типом значений, которые будут передаваться каналом.

Ключевое слово «chan».

Тип значений, передаваемых каналом.

```
var myChannel chan float64
```

Чтобы создать канал, необходимо вызвать встроенную функцию `make` (эта же функция использовалась для создания карт и сегментов). Функции `make` передается тип создаваемого канала (он должен совпадать с типом переменной, которой он присваивается).

```
var myChannel chan float64
myChannel = make(chan float64)
```

Объявление переменной для хранения канала.

Фактическое создание канала.

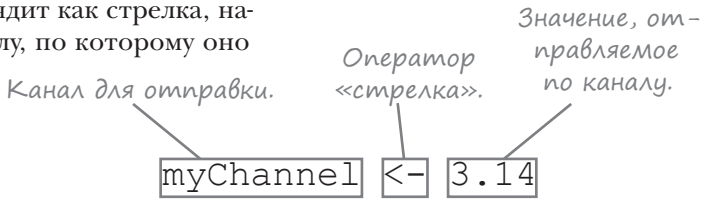
Вместо того чтобы объявлять переменную канала отдельно, обычно бывает проще воспользоваться коротким объявлением переменной:

```
myChannel := make(chan float64)
```

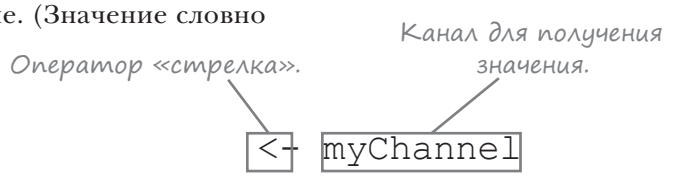
Канал создается одновременно с объявлением переменной.

Отправка и получение значений в каналах

Для отправки значений по каналу используется оператор <- (знак «меньше», за которым следует дефис). Он выглядит как стрелка, направленная от передаваемого значения к каналу, по которому оно передается.



Оператор <- также используется для *получения* значений из каналов, но с другим расположением операндов: стрелка располагается *слева* от канала, из которого принимается значение. (Значение словно «извлекается» из канала.)



Ниже приведена функция `greeting` с предыдущей страницы, переработанная для работы с каналами. Мы добавили в `greeting` параметр `myChannel`, в котором передается канал для передачи строковых значений. Вместо того чтобы возвращать строковое значение, `greeting` теперь отправляет строку по каналу `myChannel`.

В функции `main` канал, который будет передаваться `greeting`, создается встроенной функцией `make`. После этого `greeting` вызывается как новая горутина. Использование отдельной процедуры важно, потому что каналы должны использоваться только для передачи данных *между* горутинами (вскоре объясним почему). Наконец, программа получает значение из канала, переданного `greeting`, и выводит возвращенную строку.

```
                                  Канал передается в параметре.
func greeting(myChannel chan string) {
       myChannel <- "hi"  <-- Значение отправляется по каналу.
}
                          Создание нового канала.
func main() {
       myChannel := make(chan string)
       go greeting(myChannel)  <-- Канал передается функции, выполняющейся в новой горутине.
       fmt.Println(<-myChannel)
}
                  Получение значения из канала.
```

```
hi
```

Значение, полученное из канала, не обязательно передавать `Println`. Получение из канала может использоваться в любом контексте, в котором требуется значение. (То есть в любом месте, где может использоваться переменная или возвращаемое значение функции.) Так, например, полученное значение можно сначала присвоить переменной:

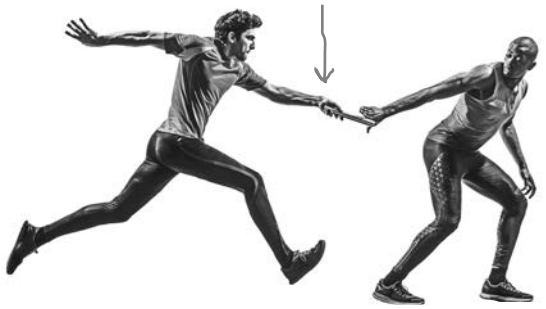
```
receivedValue := <-myChannel  <-- Полученное значение также можно было сохранить в переменной.
fmt.Println(receivedValue)
```

Каналы и синхронизация горутин

Ранее мы упомянули, что каналы также гарантируют, что отправляющая горутина отправила значение, прежде чем получающий канал попытается использовать его. Для этого каналы применяют **блокировку** — все дальнейшие операции в текущей горутине приостанавливаются. Операция отправки блокирует отправляющую горутину до того, как другая горутина выполнит операцию получения с тем же каналом. И наоборот: операция получения блокирует получающую горутину до того, как другая горутина выполнит операцию отправки с тем же каналом. Такое поведение позволяет горутинам **синхронизировать** свои действия, то есть координировать временную последовательность их выполнения.

Ниже приведена программа, которая создает два канала и передает их функциям в двух новых горутинах. Затем горутина `main` получает значения из этих каналов и выводит их. В отличие от нашей программы с горутинами, которая выводила `"a"` и `"b"` с чередованием, результат этой программы можно предсказать: она всегда выводит `"a"`, затем `"d"`, `"b"`, `"e"`, `"c"` и `"f"` именно в таком порядке.

Получающая горутина ожидает, когда другая горутина отправит значение.



```
func abc(channel chan string) {
       channel <- "a"
       channel <- "b"
       channel <- "c"
}

func def(channel chan string) {
       channel <- "d"
       channel <- "e"
       channel <- "f"
}

func main() {
       channel1 := make(chan string)
       channel2 := make(chan string)
       go abc(channel1)
       go def(channel2)
       fmt.Print(<-channel1)
       fmt.Print(<-channel2)
       fmt.Print(<-channel1)
       fmt.Print(<-channel2)
       fmt.Print(<-channel1)
       fmt.Print(<-channel2)
       fmt.Println()
}
```

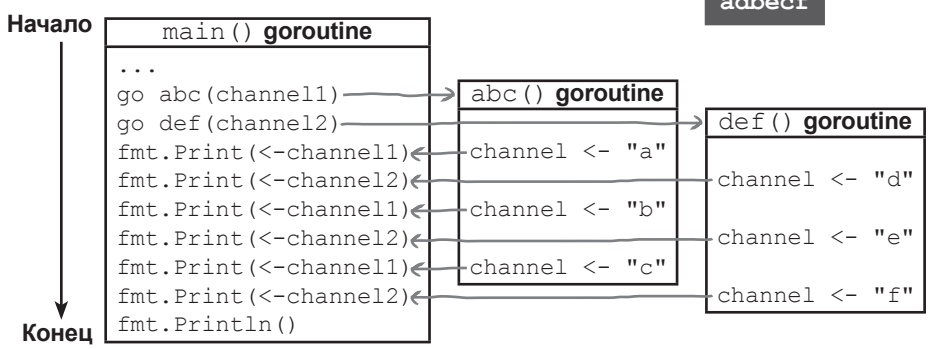
Создаются два канала.

Каждый канал передается функции, выполняемой в новой горутине.

Получает и выводит значения из каналов (по порядку).

Порядок вывода известен, потому что горутина `abc` блокируется каждый раз, когда отправляет значение по каналу, до получения данных горутиной `main`. Горутина `def` делает то же самое. Горутина `main` координирует работу горутин `abc` и `def`, позволяя им продолжать работу только тогда, когда она готова прочитать отправляемые ими значения.

```
adbecf
```



Соблюдение синхронизации в горутине

Горутины `abc` и `def` отправляют свои значения по каналам так быстро, что вам вряд ли удастся уследить за происходящим. Ниже приведена другая программа, которая намеренно замедляет свою работу, чтобы вы увидели, как происходит блокировка.

Начнем с функции `reportNap`, которая заставляет текущую горутину ожидать заданное количество секунд. Каждую секунду своего ожидания горутина выводит соответствующее сообщение.

Мы добавим функцию `send`, которая выполняется в горутине и отправляет по каналу два значения. Но перед отправкой она предварительно вызывает `reportNap`, чтобы горутина ожидала 2 секунды.

Имя приостановленной горутины. *Время приостановки.*

```
func reportNap(name string, delay int) {
        for i := 0; i < delay; i++ {
                fmt.Println(name, "sleeping")
                time.Sleep(1 * time.Second)
        }
        fmt.Println(name, "wakes up!")
}

func send(myChannel chan string) {
        reportNap("sending goroutine", 2)
        fmt.Println("***sending value***")
        myChannel <- "a"
        fmt.Println("***sending value***")
        myChannel <- "b"
}

func main() {
        myChannel := make(chan string)
        go send(myChannel)
        reportNap("receiving goroutine", 5)
        fmt.Println(<-myChannel)
        fmt.Println(<-myChannel)
}
```

Блокируется по этой отправке, пока «main» остается в ожидании.

В горутине `main` мы создаем канал и передаем его функции `send`. Затем функция `reportNap` вызывается снова, чтобы *эта* горутина ожидала 5 секунд (на 3 секунды больше, чем горутина `send`). Наконец, с каналом выполняются две операции получения.

При выполнении этой программы мы видим, что обе горутины ожидают первые 2 секунды. Затем горутина `send` активизируется и отправляет свое значение. После этого она ничего не делает: операция отправки блокирует горутину `send` до того, как горутина `main` получит значение.

Это не происходит немедленно, потому что горутина `main` должна ожидать еще 3 секунды. После активизации она получает значение из канала. Только после этого горутина разблокируется, чтобы отправить второе значение.

```
receiving goroutine sleeping
sending goroutine sleeping
sending goroutine sleeping
receiving goroutine sleeping
receiving goroutine sleeping
sending goroutine wakes up!
***sending value***
receiving goroutine sleeping
receiving goroutine sleeping
receiving goroutine wakes up!
a
***sending value***
b
```

Отправляющая и получающая горутины ожидают.

Отправляющая горутина пробуждается и отправляет значение.

Получающая горутина продолжает ожидание.

Получающая горутина активизируется и получает значение.

И только тогда отправляющая горутина разблокируется, чтобы отправить второе значение.

Сломай и изучи!

Ниже снова приведен код самой ранней и самой простой демонстрации каналов: функции `greeting`, которая выполняется в горутине и отправляет строковое значение горутине `main`.

Внесите одно из указанных изменений и попробуйте запустить программу. Затем отмените изменение и переходите к следующему. Посмотрите, что получится!

```
func greeting(myChannel chan string) {
       myChannel <- "hi"
}

func main() {
       myChannel := make(chan string)
       go greeting(myChannel)
       fmt.Println(<-myChannel)
}
```

Если...	*...программа не будет работать, потому что...*
Отправить по каналу значение из функции `main`: `myChannel <- "hi from main"`	Вы получите сообщение о взаимоблокировке «`all goroutines are asleep - deadlock!`». Это происходит из-за того, что горутина `main` блокируется, ожидая получения данных из канала другой горутиной. Но другая горутина не выполняет операции получения, поэтому горутина `main` остается заблокированной
Убрать ключевое слово `go` перед вызовом `greeting`: ~~`go`~~ `greeting(myChannel)`	Функция `greeting` будет выполняться в горутине `main`. В этом случае также возникнет ситуация взаимоблокировки по той же причине: операция отправки в `greeting` приводит к блокировке горутины `main`, но другой горутины, которая бы выполняла получение, нет, поэтому горутина так и останется заблокированной
Удалить строку, которая отправляет значение по каналу: ~~`myChannel <- "hi"`~~	И в этом случае возникает взаимоблокировка, но по другой причине: горутина `main` пытается *получить* значение, но нет операции, которая бы это значение *отправляла*
Удалить строку, которая получает значение из канала: ~~`fmt.Println(<-myChannel)`~~	Операция отправки в `greeting` приводит к блокировке этой горутины. Но поскольку не существует операции отправки, которая бы также блокировала горутину `main`, последняя завершается немедленно, а программа завершается без выдачи каких-либо результатов

Упражнение

Заполните пропуски в коде, чтобы следующая программа использовала значения, полученные из двух каналов, и выводила показанный результат.

```
package main

import "fmt"

func odd(channel chan int) {
       channel __ 1
       channel __ 3
}

func even(channel chan int) {
       channel __ 2
       channel __ 4
}

func main() {
       channelA := _____________
       channelB := _____________
       __ odd(channelA)
       __ even(channelB)
       fmt.Println(_____________)
       fmt.Println(_____________)
       fmt.Println(_____________)
       fmt.Println(_____________)
}
```

Результат.

```
1
3
2
4
```

Ответ на с. 434.

Использование каналов в программе для вывода размера веб-страниц

В программе для вывода размеров веб-страниц все еще остаются две проблемы:

- Возвращаемое значение функции `responseSize` не может использоваться в go-команде.
- Горутина `main` завершалась до получения размеров ответов, поэтому мы добавили вызов `time.Sleep` с 5-секундным ожиданием. Однако в некоторых случаях 5 секунд слишком много, а в других — слишком мало.

```
func main() {
        var size int
        size = go responseSize("https://example.com/")
        fmt.Println(size)
        size = go responseSize("https://golang.org/")
        fmt.Println(size)
        size = go responseSize("https://golang.org/doc")
        fmt.Println(size)
        time.Sleep(5 * time.Second)
}
```

Получать возвращаемое значение из go-команды запрещено!

Программа может завершиться до того, как будут получены все размеры страниц!

При помощи каналов мы сможем решить сразу обе проблемы!

Для начала удалите пакет `time` из директивы `import`; вызов `time.Sleep` больше не нужен. Затем функция `responseSize` обновляется так, чтобы ей передавался канал для значений `int`. Вместо того чтобы возвращать размер страницы, функция `responseSize` будет отправлять его по каналу.

```
package main

import (
        "fmt"
        "io/ioutil"
        "log"
        "net/http"
)

func responseSize(url string, channel chan int) {
        fmt.Println("Getting", url)
        response, err := http.Get(url)
        if err != nil {
                log.Fatal(err)
        }
        defer response.Body.Close()
        body, err := ioutil.ReadAll(response.Body)
        if err != nil {
                log.Fatal(err)
        }
        channel <- len(body)
}
```

Мы не будем использовать time.Sleep, поэтому пакет «time» удаляется.

Передаем канал responseSize для передачи размеров страниц.

Вместо того чтобы возвращать размер страниц, передает его по каналу.

Использование каналов в программе для вывода размера веб-страниц (продолжение)

В `main` функция `make` вызывается для создания канала для передачи значений `int`. Каждый из вызовов `responseSize` обновляется для добавления канала в аргументе. И наконец, мы выполняем три операции получения с каналом — по одному для каждого значения, отправляемого `responseSize`.

```
func main() {
        sizes := make(chan int)
        go responseSize("https://example.com/", sizes)
        go responseSize("https://golang.org/", sizes)
        go responseSize("https://golang.org/doc", sizes)
        fmt.Println(<-sizes)
        fmt.Println(<-sizes)
        fmt.Println(<-sizes)
}
```

Создаем канал для значений int.

Канал передается при каждом вызове responseSize.

По каналу будут отправляться три значения, поэтому выполняются три операции получения.

При запуске этой программы вы увидите, что программа завершается сразу же после получения ответа от веб-сайтов. Конкретное время может изменяться, но при тестировании время завершения доходило до 1 секунды!

```
Getting https://golang.org/doc
Getting https://example.com/
Getting https://golang.org/
8766
13078
1270
```

Другое возможное улучшение — хранение списка URL-адресов загружаемых страниц в сегменте и вызов `responseSize` в цикле для получения значений из канала. Такой код содержит меньше повторений и его проще адаптировать, если позднее вы захотите добавить новые URL-адреса.

Изменять `responseSize` вообще не придется — только функцию `main`. Мы создаем сегмент строковых значений с нужными URL-адресами, а затем в цикле перебираем элементы сегмента и вызываем `responseSize` с текущим URL-адресом и каналом. Наконец, в программе создается второй отдельный цикл, который выполняется по одному разу для каждого URL-адреса в сегменте, получает и выводит значение из канала. (Важно делать это в отдельном цикле. Если значения будут получаться в том же цикле, в котором запускаются горутины `responseSize`, то горутина `main` будет блокироваться до того, как завершится получение, и программа вернется к последовательному запросу страниц.)

```
func main() {
        sizes := make(chan int)
        urls := []string{"https://example.com/",
                "https://golang.org/", "https://golang.org/doc"}
        for _, url := range urls {
                go responseSize(url, sizes)
        }
        for i := 0; i < len(urls); i++ {
                fmt.Println(<-sizes)
        }
}
```

Переносим сюда URL.

Вызываем responseSize для каждого URL.

Получение данных из канала по одному разу для каждой отправки, выполненной responseSize.

Решение с циклами намного элегантнее, но приводит к тому же результату!

```
Getting https://golang.org/
Getting https://golang.org/doc
Getting https://example.com/
1270
8766
13078
```

Изменение канала для передачи структуры

Осталось решить еще одну проблему, связанную с функцией `responseSize`. Мы понятия не имеем, в каком порядке будут отвечать сайты. А поскольку URL-адрес страницы хранится отдельно от размера ответа, мы не знаем, к какой странице относится тот или иной размер!

```
Getting https://golang.org/
Getting https://golang.org/doc
Getting https://example.com/
1270
8766
13078
```

К какому URL-адресу относится каждый размер?

Но проблема легко решается. Составные типы — такие, как сегменты, ассоциативные массивы и структуры, — могут передаваться по каналам, причем это ничуть не сложнее, чем при передаче основных типов. От нас потребуется лишь создать тип структуры, в котором URL-адрес страницы хранится вместе с ее размером, чтобы они передавались по каналу вместе.

Мы объявим новый тип `Page` с базовым типом `struct`. Тип `Page` содержит поле `URL` для хранения URL-адреса страницы, и поле `Size` для размера страницы.

Параметр функции `responseSize` должен иметь новый тип `Page` (вместо простого типа `int` для размера страницы). Функция `responseSize` будет создавать новое значение `Page` с текущим URL-адресом и размером страницы и передавать его по каналу.

Также в функции `main` следует обновить тип значения канала при вызове `make`. Значение, полученное из канала, имеет тип `Page`; программа выводит его поля `URL` и `Size`.

```
type Page struct {                          <- Объявление типа структуры с нужными полями.
        URL  string
        Size int
}

func responseSize(url string, channel chan Page) {
        // Omitting identical code...
        channel <- Page{URL: url, Size: len(body)}
}

func main() {
        pages := make(chan Page)            <- Изменение типа, передаваемого каналом.
        urls := []string{"https://example.com/",
                "https://golang.org/", "https://golang.org/doc"}
        for _, url := range urls {
                go responseSize(url, pages)
        }
        for i := 0; i < len(urls); i++ {
                page := <-pages             <- Получает Page.
                fmt.Printf("%s: %d\n", page.URL, page.Size)
        }
}
```

Канал, передаваемый responseSize, будет использоваться для передачи Page, а не ints.

Отправляет структуру Page с текущим URL-адресом и размером страницы.

Канал передается responseSize.

URL-адрес выводится вместе с размером.

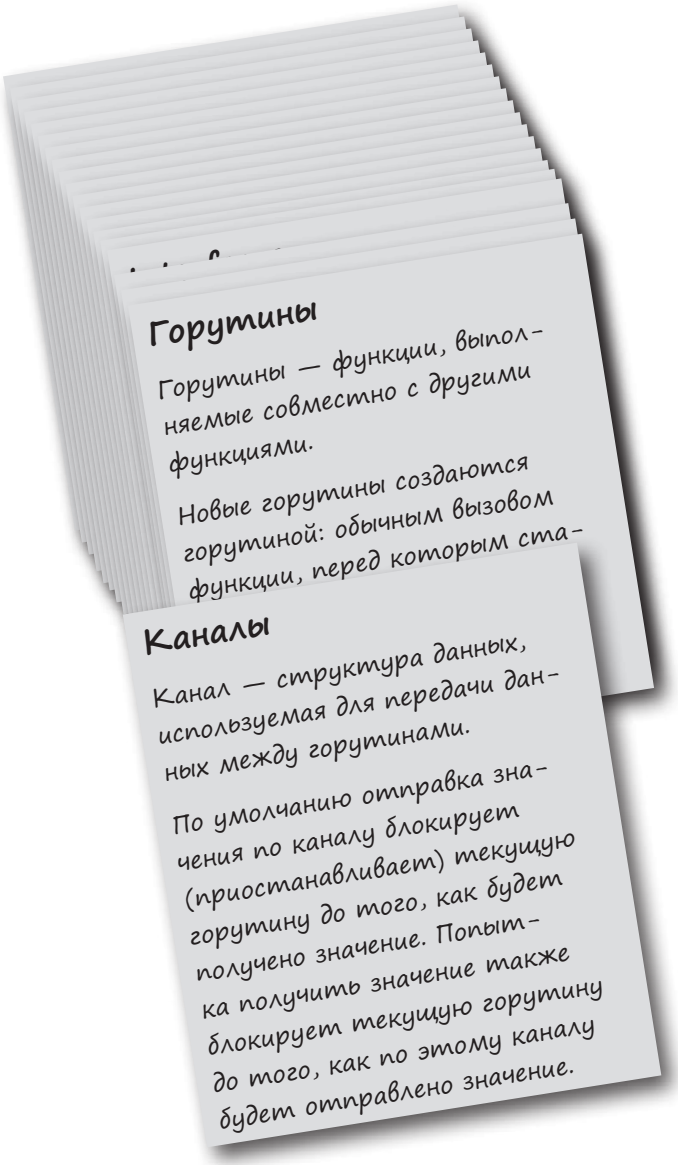
```
https://example.com/: 1270
https://golang.org/: 8766
https://golang.org/doc: 13078
```

Теперь в выходных данных программы размеры страниц выводятся вместе с URL-адресами. Наконец-то стало ясно, к какой странице относится тот или иной размер.

Ранее нашей программе приходилось запрашивать страницы по одной. Горутины позволяют перейти к обработке следующего запроса в то время, пока программа ожидает ответа от предыдущего сайта. Программа завершается за треть исходного времени!

Ваш инструментарий Go

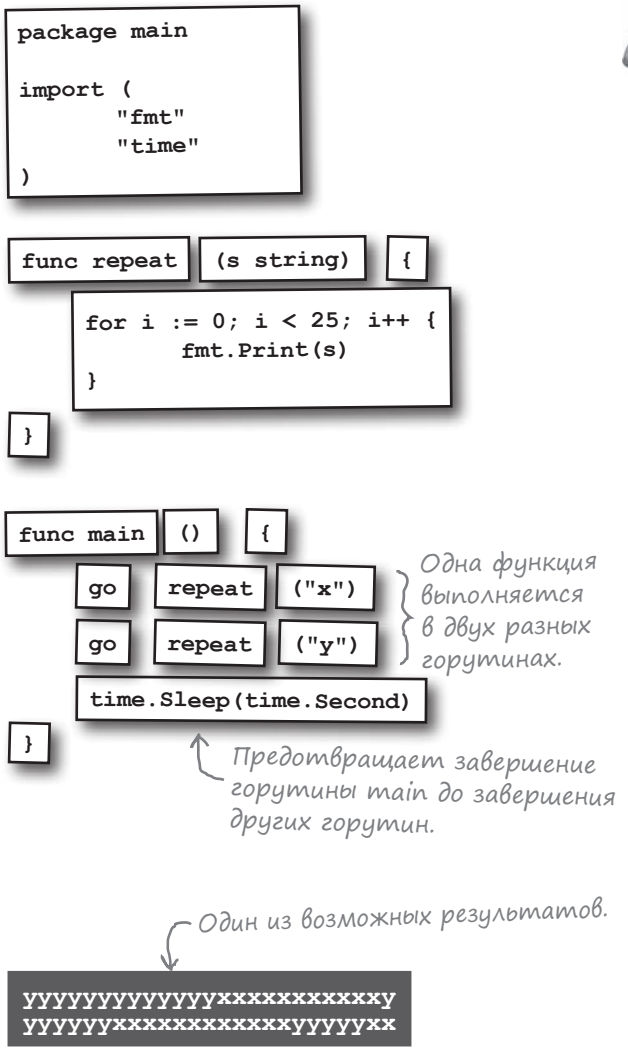
Глава 13 осталась позади! В ней ваш инструментарий пополнился горутинами и каналами.



КЛЮЧЕВЫЕ МОМЕНТЫ

- Все программы Go содержат по крайней мере одну горутину: ту, которая вызывает функцию `main` при запуске программы.
- Программы Go завершаются при остановке горутины `main`, даже если другие горутины еще не завершили свою работу.
- Функция `time.Sleep` приостанавливает текущую горутину на заданный промежуток времени.
- Go не дает никаких гарантий относительно того, когда происходит переключение между горутинами или сколько времени будет выполняться одна горутина. Это позволяет горутинам выполняться более эффективно, но с другой стороны, нельзя рассчитывать на то, что операции будут выполняться в определенном порядке.
- Возвращаемые значения функций не могут использоваться в горутинах — в частности из-за того, что возвращаемое значение не будет готово к тому моменту, когда вызывающая функция попробует использовать его.
- Если вам понадобится значение от горутины, то понадобится канал для отправки значения.
- Каналы создаются вызовом встроенной функции `make`.
- Каждый канал передает значения одного типа; этот тип указывается при создании канала.
  `myChannel := make(chan MyType)`
- Для отправки значений по каналу используется оператор `<-`:
  `myChannel <- "a value"`
- Оператор `<-` также используется для получения значений из канала:
  `value := <-myChannel`

Развлечения с магнитами. Решение



Упражнение
Решение

```
package main

import "fmt"

func odd(channel chan int) {
        channel <- 1
        channel <- 3
}

func even(channel chan int) {
        channel <- 2
        channel <- 4
}

func main() {
        channelA := make(chan int)
        channelB := make(chan int)
        go odd(channelA)
        go even(channelB)
        fmt.Println( <-channelA )
        fmt.Println( <-channelA )
        fmt.Println( <-channelB )
        fmt.Println( <-channelB )
}
```

1
3
2
4

Один канал передает значения от функции «odd», другой — от «even».

Автоматизация тестирования

А вы уверены, что ваша программа работает правильно? Точно уверены? Прежде чем рассылать новую версию пользователям, вы, скорее всего, опробовали ее новые возможности и убедились, что все работает. Но опробовали ли вы *старые* возможности? А вдруг что-нибудь сломалось в процессе доработки? *Все* старые возможности? Если от этого вопроса вам стало слегка не по себе, значит, вашим программам необходимо **автоматизированное тестирование**. Автоматизированные тесты гарантируют, что компоненты программы работают правильно даже после того, как вы внесете изменения в код. Средства тестирования Go и команда `go test` упрощают написание автотестов.

Автотесты обнаруживают ошибки до того, как их обнаружат пользователи

Разработчик А сталкивается в ресторане с разработчиком Б, где они оба часто бывают...

Разработчик А:

Как твоя новая работа?

Сочувствую. Как *оно* проникло на сервер выставления счетов?

Ого, так давно... И ваши тесты ее не обнаружили?

Да, автотесты. Они проходили и после внесения ошибки?

Что?!

Разработчик Б:

Так себе. Собираюсь вернуться в офис после обеда. Обнаружился баг, из-за которого некоторые клиенты получают счета вдвое чаще, чем должно быть.

Мы подозреваем, что ошибка появилась пару месяцев назад. Тогда один из наших разработчиков вносил изменения в код выставления счетов.

Тесты?

Вообще-то у нас ничего такого нет.

Ваши клиенты зависят от вашего кода. Когда в нем обнаруживаются ошибки, последствия могут быть катастрофическими. Страдает репутация компании. И *вам* придется задерживаться на работе вечерами в поисках ошибки.

Именно по этой причине были изобретены автотесты. **Автоматизированный тест** представляет собой отдельную программу, которая выполняет компоненты основной программы и убеждается в том, что их поведение соответствует ожиданиям.

Нет, если только вы хотите протестировать все старые возможности и убедиться в том, что от изменений ничего не сломалось. Автотесты экономят время по сравнению с ручным тестированием и обычно обеспечивают более тщательный контроль.

Функция, для которой нужны автотесты

Рассмотрим пример ошибки, которую можно было бы обнаружить при помощи автотеста. Здесь имеется простой пакет с функцией, которая объединяет несколько строк в одну строку, пригодную для использования в английском предложении. Две строки соединяются словом *and* (например, «apple and orange»). Если строк больше двух, то между ними добавляются запятые (например, «apple, orange and pear»).

И последний замечательный пример, позаимствованный из книги Head First Ruby (в которой также имеется глава, посвященная тестированию)!

Необходимо для использования функции strings.Join.

```
package prose

import "strings"

func JoinWithCommas(phrases []string) string {
       result := strings.Join(phrases[:len(phrases)-1], ", ")
       result += " and "
       result += phrases[len(phrases)-1]
       return result
}
```

Получает сегмент строк для объединения.

Возвращает объединенную строку.

Разделяет все строки, кроме последней, запятыми.

Вставляет слово «and» перед последней строкой.

Добавляет последнюю строку.

В коде используется функция `strings.Join`, которая получает сегмент строк и строку-разделитель. Функция `Join` возвращает одну строку, содержащую все элементы из сегмента, в которой в позиции между объединенными элементами вставлен разделитель.

Сегмент строк, которые должны быть объединены в одну строку.

Строка, используемая как разделитель при объединении.

```
fmt.Println(strings.Join([]string{"05", "14", "2018"}, "/"))
fmt.Println(strings.Join([]string{"state", "of", "the", "art"}, "-"))
```

```
05/14/2018
state-of-the-art
```

В коде `JoinWithCommas` оператор сегмента выбирает все элементы сегмента, кроме последнего. Эти элементы передаются `strings.Join` для формирования одной строки с разделением соседних элементов запятой и пробелом. Затем мы добавляем слово *and* (окруженное пробелами) и завершаем итоговую строку последним элементом.

```
[]string{"apple", "orange", "pear", "banana"}
```

```
apple, orange, pear and banana
```

Все строки, кроме последней, разделяются запятыми.

Перед последней строкой вставляется «and».

Функция, для которой нужны автотесты (продолжение)

Ниже приведена короткая программа, в которой используется новая функция. Мы импортируем наш пакет `prose` и передаем пару сегментов `JoinWithCommas`.


```
package main

import (
       "fmt"
       "github.com/headfirstgo/prose"
)

func main() {
       phrases := []string{"my parents", "a rodeo clown"}
       fmt.Println("A photo of", prose.JoinWithCommas(phrases))
       phrases = []string{"my parents", "a rodeo clown", "a prize bull"}
       fmt.Println("A photo of", prose.JoinWithCommas(phrases))
}
```

```
A photo of my parents and a rodeo clown
A photo of my parents, a rodeo clown and a prize bull
```

Программа работает, но с результатами есть небольшая проблема. Может, дело в плохом чувстве юмора, но некоторые пользователи наверняка поймут этот текст неправильно. Впрочем, такое форматирование списков может породить и другие недоразумения.

Чтобы устранить любое недопонимание, обновим код пакета, чтобы он вставял дополнительную запятую перед *and* (например, «apple, orange, and pear»):

```
func JoinWithCommas(phrases []string) string {
       result := strings.Join(phrases[:len(phrases)-1], ", ")
       result += ", and " ←— Вставляем запятую перед «and».
       result += phrases[len(phrases)-1]
       return result
}
```

Если снова запустить программу, вы увидите, что в обеих полученных строках перед *and* стоит запятая. Теперь никаких проблем быть не должно.

```
A photo of my parents, and a rodeo clown
A photo of my parents, a rodeo clown, and a prize bull
```

Новая запятая!

Мы внесли ошибку!

Постойте! Новый код правильно работает для **трех** элементов в списке, а не для **двух** элементов. Вы внесли ошибку!

И верно! Ранее функция возвращала "`my parents and a rodeo clown`" для списка из двух элементов, но теперь она также добавляет лишнюю запятую! Мы так увлеклись исправлением списков из *трех* элементов, что случайно внесли ошибку для списков из *двух* элементов...

Запятая здесь не нужна!

```
A photo of my parents, and a rodeo clown
```

А если бы для функции существовали автотесты, проблемы можно было бы избежать.

Автотест выполняет ваш код с заданным набором входных значений и проверяет полученный результат. Если результат совпадает с ожидаемым значением, то тест «проходит».

Но предположим, вы случайно внесли в свой код ошибку (как с лишней запятой в нашем примере). Тогда результат выполнения кода уже не будет совпадать с ожидаемым значением, и тест «не проходит». Вы сразу же узнаете об ошибке.

Проходит ☑ For `[]slice{"apple", "orange", "pear"}`, `JoinWithCommas` should return `"apple, orange, and pear"`.

Ошибка! ☒ For `[]slice{"apple", "orange"}`, `JoinWithCommas` should return `"apple and orange"`.

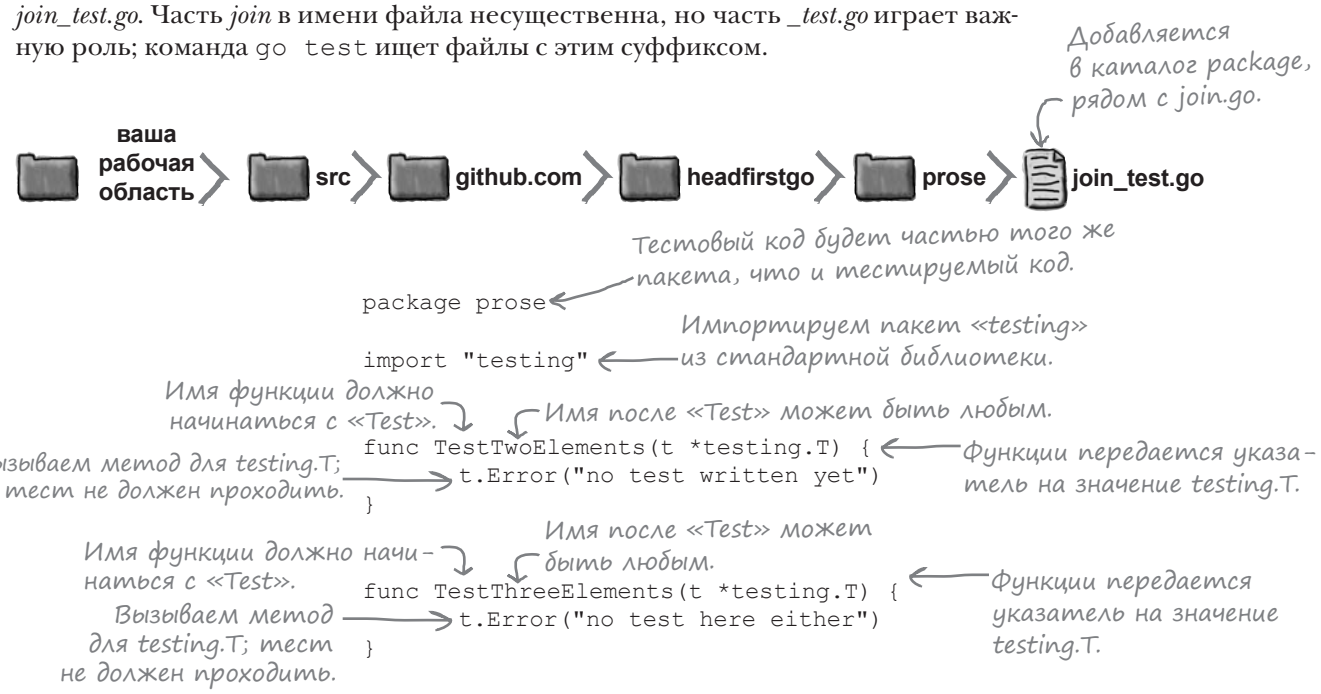
Наличие таких автотестов фактически означает, что код будет автоматически проверяться на наличие ошибок при внесении каждого изменения!

Написание тестов

Go включает пакет `testing`, который может использоваться для написания автотестов, и команду `go test`, предназначенную для выполнения этих тестов.

Начнем с написания простого теста. На первых порах мы не будем использовать реальные тесты, а просто покажем, как работает механизм тестирования. Потом воспользуемся тестами для решения проблемы с функцией `JoinWithCommas`.

В каталоге пакета *prose*, непосредственно рядом с файлом *join.go*, создайте файл *join_test.go*. Часть *join* в имени файла несущественна, но часть *_test.go* играет важную роль; команда `go test` ищет файлы с этим суффиксом.



Код в тестовом файле состоит из обычных функций Go, но для работы с командой `go test` в нем должны соблюдаться некоторые соглашения:

- Ваши тесты не обязаны принадлежать тому же пакету, что и тестируемый код, но это необходимо, если вы хотите обращаться к неэкспортируемым типам или функциям из пакета.
- В тестах используется тип из пакета `testing`, поэтому этот пакет должен импортироваться в начале каждого тестового файла.
- Имена тестовых функций должны начинаться с `Test`. (Остаток имени может быть любым, но имена должны начинаться с буквы верхнего регистра.)
- Тестовые функции должны получать один параметр: указатель на значение `testing.T`.
- Чтобы сообщить о том, что тест не проходит, вы вызываете различные методы (такие, как `Error`) для значения `testing.T`. Многие методы получают строку с сообщением, которое объясняет, почему тест не проходит.

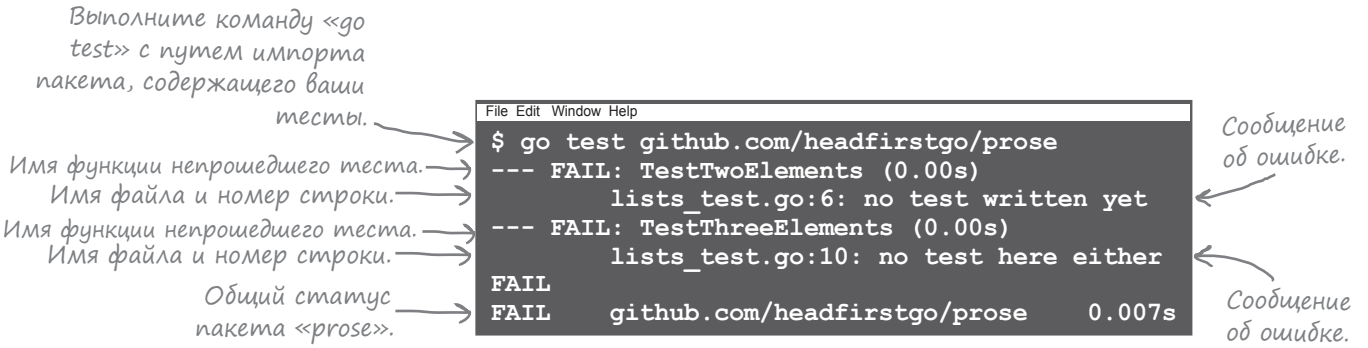
Выполнение тестов командой «go test»

Для выполнения тестов используется команда `go test`. Команда получает пути импорта одного или нескольких пакетов (как и команды `go install` или `go doc`). Она находит в каталогах этих пакетов все файлы, имена которых завершаются суффиксом *_test.go*, и выполняет все функции в этих файлах, имена которых начинаются с `Test`.

Выполним тесты, только что добавленные в пакет `prose`. В терминале введите следующую команду:

```
go test github.com/headfirstgo/prose
```

Тестовые функции выполняются и выводят свои результаты.



Так как обе тестовые функции вызывают метод `Error` для переданного им значения `testing.T`, оба теста не проходят. Выводится имя каждой непрошедшей тестовой функции, номер строки с вызовом `Error` и сообщение об ошибке.

В конце вывода указан статус всего пакета `prose`. Если любые тесты в пакете завершаются неудачей (как в нашем случае), будет выведен статус «`FAIL`» для всего пакета в целом.

Если же удалить вызовы метода `Error` в тестах...

```
func TestTwoElements(t *testing.T) {
}
```
← Удаляем вызов t.Error.

```
func TestThreeElements(t *testing.T) {
}
```
← Удаляем вызов t.Error.

...то при повторном выполнении той же команды `go test` все тесты пройдут успешно. Так как все тесты прошли, `go test` выводит статус «`ok`» для всего пакета `prose`.

Тестирование возвращаемых значений

Теперь вы можете сделать так, чтобы тесты проходили или не проходили. Напишем несколько тестов, которые пригодятся при отладке функции `JoinWithCommas`.

Обновим функцию `TestTwoElements`, чтобы она выводила возвращаемое значение, *ожидаемое* от функции `JoinWithCommas`, при вызове с сегментом из двух элементов. То же самое будет сделано для функции `TestThreeElements` с сегментом из трех элементов. Выполним тесты и убедимся в том, что тест `TestTwoElements` не проходит, а `TestThreeElements` проходит.

После того как тесты будут настроены так, как вам нужно, мы изменим функцию `JoinWithCommas`, чтобы обеспечить прохождение всех тестов. И тогда можно быть уверенными в том, что код рабочий!

В `TestTwoElements` функции `JoinWithCommas` будет передаваться сегмент из двух элементов, `[]string{"apple", "orange"}`. Если результат не равен `"apple and orange"`, тест не проходит. Аналогично в `TestThreeElements` будет передаваться сегмент из трех элементов, `[]string{"apple", "orange", "pear"}`. Если результат не равен `"apple, orange, and pear"`, значит, тест не проходит.

```
func TestTwoElements(t *testing.T) {
      list := []string{"apple", "orange"}
      if JoinWithCommas(list) != "apple and orange" {
            t.Error("didn't match expected value")
      }
}

func TestThreeElements(t *testing.T) {
      list := []string{"apple", "orange", "pear"}
      if JoinWithCommas(list) != "apple, orange, and pear" {
            t.Error("didn't match expected value")
      }
}
```

Передается список из двух элементов.

Если JoinWithCommas не возвращает ожидаемую строку...

...тест не проходит.

Передается список из трех элементов.

Если JoinWithCommas не возвращает ожидаемую строку...

...тест не проходит.

При повторном запуске тест `TestThreeElements` проходит, но тест `TestTwoElements` — нет.

Не проходит только тест TestTwoElements.

```
File  Edit  Window  Help
$ go test github.com/headfirstgo/prose
--- FAIL: TestTwoElements (0.00s)
        lists_test.go:13: didn't match expected value
FAIL
FAIL    github.com/headfirstgo/prose    0.006s
```

Тестирование возвращаемых значений (продолжение)

И это *хорошо*: результат совпадает с тем, что мы ожидаем получить на основании вывода программы `join`. А следовательно, наши тесты служат верным признаком того, что функция `JoinWithCommas` работает так, как должна!

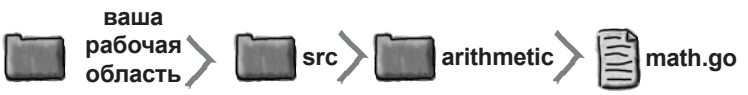
Проходит. ☑ For `[]slice{"apple", "orange", "pear"}`, `JoinWithCommas` should return `"apple, orange, and pear"`.

Не проходит! ☒ For `[]slice{"apple", "orange"}`, `JoinWithCommas` should return `"apple and orange"`.

Неправильно. → `A photo of my parents, and a rodeo clown`
Правильно. → `A photo of my parents, a rodeo clown, and a prize bull`

Упражнение

Заполните пропуски в приведенном ниже коде теста.




```
package arithmetic

func Add(a float64, b float64) float64 {
       return a + b
}
func Subtract(a float64, b float64) float64 {
       return a - b
}
```


```
package _____

import _____

func _____Add(t _____) {
       if _____(1, 2) != 3 {
              _____("1 + 2 did not equal 3")
       }
}

func _____Subtract(t _____) {
       if _____(8, 4) != 4 {
              _____("8 - 4 did not equal 4")
       }
}
```

Ответ на с. 457.

Метод «Errorf» и подробные сообщения о непрохождении тестов

Наше сообщение об отказе теста не принесло особой пользы для диагностики проблемы. Мы знаем, что ожидалось какое-то значение, а возвращаемое значение `JoinWithCommas` отличалось от него, но не знаем, какими были эти значения.

```
--- FAIL: TestTwoElements (0.00s)
        lists_test.go:13: didn't match expected value
FAIL
FAIL    github.com/headfirstgo/prose    0.006s
```

«Ожидаемое значение»? А какое именно?

Параметр `testing.T` тестовой функции также содержит метод `Errorf`, который можно вызвать. В отличие от `Error`, `Errorf` получает строку с глаголами форматирования, как и функции `fmt.Printf` и `fmt.Sprintf`. Метод `Errorf` позволяет включить в сообщения об отказе тестов дополнительную информацию: аргументы, переданные функции, полученное возвращаемое значение и ожидаемое значение.

Здесь приведена обновленная версия тестов, которая использует `Errorf` для генерирования более подробных сообщений об отказе. Чтобы не повторять строки в каждом тесте, мы добавим переменную `want` для хранения значения, которое должна вернуть функция `JoinWithCommas`. Также будет добавлена переменная `got` для хранения фактического возвращаемого значения. Если переменная `got` не равна `want`, мы вызовем метод `Errorf` и сгенерируем в нем сообщение об ошибке, которое включает сегмент, переданный `JoinWithCommas` (используется глагол форматирования `%#v`, чтобы сегмент выводился в том виде, в котором он отображается в коде Go), полученное возвращаемое значение и ожидаемое возвращаемое значение.

Ожидаемое возвращаемое значение.

Полученное возвращаемое значение.

```
func TestTwoElements(t *testing.T) {
    list := []string{"apple", "orange"}
    want := "apple and orange"
    got := JoinWithCommas(list)
    if got != want {
        t.Errorf("JoinWithCommas(%#v) = \"%s\", want \"%s\"", list, got, want)
    }
}
```

Выводит сегмент, переданный JoinWithCommas, в отладочном формате.

Включает возвращаемое значение, полученное для сегмента.

Включает возвращаемое значение, ожидаемое для этого сегмента.

Ожидаемое возвращаемое значение.

Полученное возвращаемое значение.

```
func TestThreeElements(t *testing.T) {
    list := []string{"apple", "orange", "pear"}
    want := "apple, orange, and pear"
    got := JoinWithCommas(list)
    if got != want {
        t.Errorf("JoinWithCommas(%#v) = \"%s\", want \"%s\"", list, got, want)
    }
}
```

Выводит сегмент, переданный JoinWithCommas, в отладочном формате.

Включает возвращаемое значение, полученное для сегмента.

Включает возвращаемое значение, ожидаемое для этого сегмента.

При повторном запуске тестов мы видим, что именно не прошло в тесте.

```
--- FAIL: TestTwoElements (0.00s)
    lists_test.go:15: JoinWithCommas([]string{"apple", "orange"}) =
                      "apple, and orange", want "apple and orange"
FAIL
FAIL    github.com/headfirstgo/prose    0.006s
```

«Вспомогательные» тестовые функции

Файлы *_test.go* могут содержать не только тестовые функции. Можно исключить повторяющийся код из тестов, переместив его в другие «вспомогательные» функции в тестовом файле. Команда `go test` использует только функции, имена которых начинаются с `Test`, поэтому если вспомогательным функциям присвоены любые другие имена, все будет нормально.

В нашей программе есть громоздкий вызов `t.Errorf`, повторяющийся в функциях `TestTwoElements` и `TestThreeElements` (а с добавлением новых тестов объем дублирующегося кода может увеличиться). Одно из возможных решений — перемещение генерирования строки в отдельную функцию `errorString`, которая может вызываться в тестах.

Функция `errorString` получает сегмент, переданный `JoinWithCommas`, значение `got` и значение `want`. Тогда вместо вызова `Errorf` для значения `testing.T` функция `errorString` будет вызывать `fmt.Sprintf` для генерирования (идентичной) строки ошибки. Сам тест сможет вызвать `Error` с возвращаемой строкой, чтобы сообщить об отказе теста. Такой под получается более элегантным, но возвращает тот же результат.

```
import (
    "fmt"
    "testing"
)

func TestTwoElements(t *testing.T) {
    list := []string{"apple", "orange"}
    want := "apple and orange"
    got := JoinWithCommas(list)
    if got != want {
        t.Error(errorString(list, got, want))
    }
}

func TestThreeElements(t *testing.T) {
    list := []string{"apple", "orange", "pear"}
    want := "apple, orange, and pear"
    got := JoinWithCommas(list)
    if got != want {
        t.Error(errorString(list, got, want))
    }
}

func errorString(list []string, got string, want string) string {
    return fmt.Sprintf("JoinWithCommas(%#v) = \"%s\", want \"%s\"", list, got, want)
}
```

Пакет «fmt» необходим для вызова fmt.Sprintf.

Вместо t.Errorf вызывается новая вспомогательная функция.

Вместо t.Errorf вызывается новая вспомогательная функция.

Имя функции не начинается с «Test», поэтому она не интерпретируется как тест.

```
--- FAIL: TestTwoElements (0.00s)
        lists_test.go:18: JoinWithCommas([]string{"apple", "orange"}) =
                          "apple, and orange", want "apple and orange"
FAIL
FAIL    github.com/headfirstgo/prose    0.006s
```

Тот же результат.

Прохождение тестов

Теперь наши тесты выводят полезные сообщения об отказах, и мы сможем использовать их для исправления основного кода.

У нас есть два теста с функцией `JoinWithCommas`. Тест, получающий сегмент с тремя элементами, проходит, а тест, получающий сегмент с двумя элементами, не проходит.

Это происходит из-за того, что `JoinWithCommas` в настоящее время включает запятую даже при возвращении списка всего с двумя элементами.

Проходит. ☑ Для `[]slice{"apple", "orange", "pear"}` функция `JoinWithCommas` должна возвращать `"apple, orange, and pear"`.

Не проходит! ☒ Для `[]slice{"apple", "orange"}` функция `JoinWithCommas` должна возвращать `"apple and orange"`.

Запятая здесь не нужна!

```
A photo of my parents, and a rodeo clown
```

Изменим функцию `JoinWithCommas`, чтобы решить эту проблему. Если сегмент строк содержит всего два элемента, мы просто объединяем их связкой `" and "`, после чего возвращаем полученную строку. В противном случае выполняется та же логика, которая была реализована изначально.

Если сегмент содержит всего два элемента, они просто объединяются связкой «and».

```
func JoinWithCommas(phrases []string) string {
        if len(phrases) == 2 {
                return phrases[0] + " and " + phrases[1]
        } else {
                result := strings.Join(phrases[:len(phrases)-1], ", ")
                result += ", and "
                result += phrases[len(phrases)-1]
                return result
        }
}
```

В противном случае используется тот же код, что и раньше.

Код был изменен, но правильно ли он работает? Тесты немедленно ответят на этот вопрос! Если снова выполнить тесты, `TestTwoElements` пройдет успешно; таким образом, все тесты проходят.

Все тесты проходят!

```
File Edit Window Help
$ go test github.com/headfirstgo/prose
ok      github.com/headfirstgo/prose    0.006s
```

Проходит. ☑ Для `[]slice{"apple", "orange", "pear"}` функция `JoinWithCommas` должна возвращать `"apple, orange, and pear"`.

Не проходит! ☑ Для `[]slice{"apple", "orange"}` функция `JoinWithCommas` должна возвращать `"apple and orange"`.

Прохождение тестов (продолжение)

Можно с уверенностью сказать, что `JoinWithCommas` теперь работает с сегментами из двух строк, так как соответствующий модульный тест теперь успешно проходит. Беспокоиться о том, работает ли он с сегментами из трех строк, не нужно: есть модульный тест, который проверяет и это.

Изменения отражены и в выводе программы `join`. При повторном запуске вы увидите, что оба сегмента отформатированы правильно!

```
func main() {
        phrases := []string{"my parents", "a rodeo clown"}
        fmt.Println("A photo of", prose.JoinWithCommas(phrases))
        phrases = []string{"my parents", "a rodeo clown", "a prize bull"}
        fmt.Println("A photo of", prose.JoinWithCommas(phrases))
}
```

Без лишней запятой с двумя элементами.

Также работает с тремя элементами.

```
A photo of my parents and a rodeo clown
A photo of my parents, a rodeo clown, and a prize bull
```

Разработка через тестирование

После того как у вас появится опыт модульного тестирования, вы, вероятно, начнете использовать цикл *разработки через тестирование*:

1. **Написать тест:** вы пишете тест для *нужной* возможности, даже если она еще не существует. После этого вы выполняете тест и убеждаетесь в том, что он *не проходит*.
2. **Добиться его прохождения:** вы реализуете новую возможность в основном коде. Не беспокойтесь, если написанный код будет неэффективным или громоздким; ваша единственная цель — добиться того, чтобы он работал. Затем вы запускаете тест и убеждаетесь в том, что он *проходит*.
3. **Провести рефакторинг:** теперь можно переходить к *рефакторингу* кода, то есть его изменению и усовершенствованию. Ранее вы видели, что тест изначально *не проходил*, поэтому вы знаете, что при нарушении работоспособности кода он снова не будет проходить. Вы видели, что тест *проходит*, поэтому знаете, что он будет проходить при условии, что код работает правильно.

✗ Написать тест!

✓ Добиться его прохождения!

✓ Провести рефакторинг!

Эта возможность *изменять* код, не беспокоясь о нарушении его работоспособности, — главная причина, по которой так важны модульные тесты. Каждый раз, когда вы видите возможность сделать код более компактным или понятным, принимайтесь за дело без колебаний. Когда работа будет завершена, вы просто запустите тесты и убедитесь, что все работает правильно.

Еще одна ошибка

Функция `JoinWithCommas` также может вызываться с сегментом, содержащим только один элемент. Но в этом случае она работает не идеально: один элемент интерпретируется так, словно располагается в конце списка:

```
phrases = []string{"my parents"}
fmt.Println("A photo of", prose.JoinWithCommas(phrases))
```

Функция интерпретирует элемент так, будто он в конце списка!

```
A photo of , and my parents
```

Что *должна* возвращать функция `JoinWithCommas` в этом случае? Для списка из одного элемента не нужны ни запятые, ни связка *and*... вообще ничего. Достаточно вернуть строку с этим одним элементом.

```
A photo of my parents
```

Список из одного элемента должен выглядеть примерно так.

Представим этот случай в форме нового теста в файле *join_test.go*. Добавим новую тестовую функцию с именем `TestOneElement` к уже существующим тестам `TestTwoElements` или `TestThreeElements`. Новый тест будет выглядеть так же, как и другие, но мы передаем `JoinWithCommas` сегмент всего с одной строкой и ожидаем, что ее возвращаемое значение будет состоять из этой строки.

```
func TestOneElement(t *testing.T) {
        list := []string{"apple"}
        want := "apple"
        got := JoinWithCommas(list)
        if got != want {
                t.Error(errorString(list, got, want))
        }
}
```

Передается сегмент всего с одной строкой.

Ожидается возвращаемое значение, которое содержит только эту строку.

```
--- FAIL: TestOneElement (0.00s)
        lists_test.go:13: JoinWithCommas([]string{"apple"}) =
                          ", and apple", want "apple"
FAIL
FAIL    github.com/headfirstgo/prose    0.006s
```

Как и следовало ожидать (раз мы знаем, что код содержит ошибку), тест не проходит: он показывает, что `JoinWithCommas` возвращает строку `", and apple"` вместо `"apple"`.

Еще одна ошибка (продолжение)

Исправить ошибку в `JoinWithCommas` будет несложно. Мы проверяем, содержит ли полученный сегмент всего одну строку, и если содержит — возвращаем только ее.

```
func JoinWithCommas(phrases []string) string {
      if len(phrases) == 1 {
            return phrases[0]
      } else if len(phrases) == 2 {
            return phrases[0] + " and " + phrases[1]
      } else {
            result := strings.Join(phrases[:len(phrases)-1], ", ")
            result += ", and "
            result += phrases[len(phrases)-1]
            return result
      }
}
```

Если после исправления кода снова выполнить тест, вы увидите, что на этот раз все работает нормально.

Все тесты проходят!

```
File Edit Window Help
$ go test github.com/headfirstgo/prose
ok      github.com/headfirstgo/prose    0.006s
```

А если вы используете функцию `JoinWithCommas` в своем коде, она ведет себя так, как и должна.

```
phrases = []string{"my parents"}
fmt.Println("A photo of", prose.JoinWithCommas(phrases))
```

Теперь все работает правильно!

```
A photo of my parents
```

Часто Задаваемые Вопросы

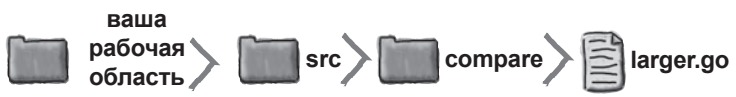
В: Разве тестовый код не увеличит объем программы и не замедлит ее работу?

О: Не беспокойтесь! По аналогии с тем, как команда `go test` работает только с файлами, имена которых заканчиваются суффиксом *_test.go*, другие команды (такие, как `go build` и `go install`) *игнорируют* файлы с суффиксом *_test.go*. Компилятор преобразует код вашей программы в исполняемый файл, но проигнорирует тестовый код, даже если он хранится в том же каталоге пакета.

Развлечения с магнитами

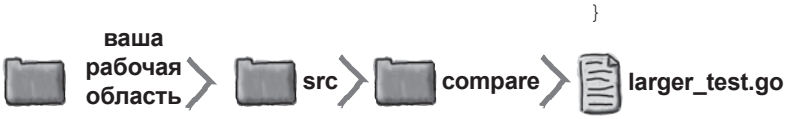
Мы создали пакет `compare` с функцией `Larger`, которая должна возвращать большее из двух переданных ей целых чисел. Но из-за ошибки в сравнении `Larger` возвращает *меньшее* из двух чисел!

Для упрощения диагностики мы взялись за написание тестов. Сможете ли вы восстановить фрагменты кода, чтобы получились работоспособные тесты, которые выдают показанный результат? Создайте вспомогательную функцию, возвращающую строку с сообщением об отказе теста, а затем включите два вызова вспомогательной функции в тесты.




```
package compare

func Larger(a int, b int) int {
        if a < b {
                return a
        } else {
                return b
        }
}
```

Ошибка – обратное сравнение!

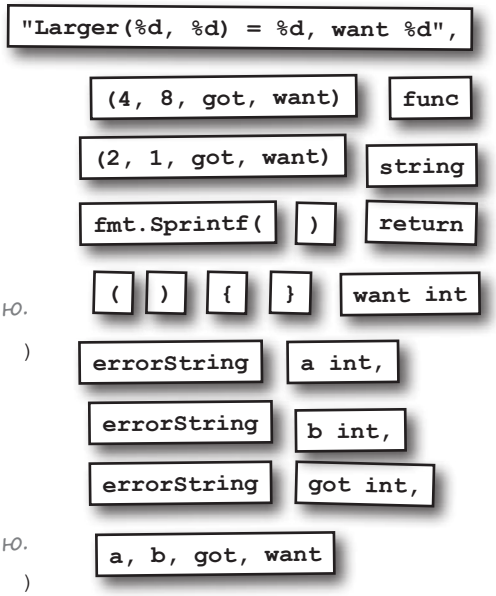



```
package compare

import (
        "fmt"
        "testing"
)

func TestFirstLarger(t *testing.T) {
        want := 2
        got := Larger(2, 1)
        if got != want {
                t.Error(
        }
}

func TestSecondLarger(t *testing.T) {
        want := 8
        got := Larger(4, 8)
        if got != want {
                t.Error(
        }
}
```

Вызовите здесь свою вспомогательную функцию.

Вызовите здесь свою вспомогательную функцию.

Определите здесь свою вспомогательную функцию.




```
File Edit Window Help
$ go test compare
--- FAIL: TestFirstLarger (0.00s)
        larger_test.go:12:
        Larger(2, 1) = 1, want 2
--- FAIL: TestSecondLarger (0.00s)
        larger_test.go:20:
        Larger(4, 8) = 4, want 8
FAIL
FAIL    compare 0.007s
```

Ответ на с. 458.

Выполнение определенных наборов тестов

Иногда бывает нужно выполнить несколько определенных тестов вместо всего набора. Команда `go test` поддерживает пару флагов командной строки, которые помогут в этом. **Флаг** представляет собой аргумент командной строки: обычно это дефис (`-`), за которым следует одна или несколько букв. Флаги изменяют поведение программы.

Первый флаг команды `go test`, который стоит запомнить, — флаг `-v` — означает расширенный вывод («verbose»). Если добавить его в любую команду `go test`, команда выведет имя и статус каждой выполняемой тестовой функции. Обычно проходящие тесты исключаются из вывода, чтобы не загромождать его, но в расширенном режиме `go test` выводит информацию даже о проходящих тестах.

Добавить флаг «-v» в команду.

Имя и статус каждого теста.

```
File  Edit  Window  Help
$ go test github.com/headfirstgo/prose -v
=== RUN   TestOneElement
--- PASS: TestOneElement (0.00s)
=== RUN   TestTwoElements
--- PASS: TestTwoElements (0.00s)
=== RUN   TestThreeElements
--- PASS: TestThreeElements (0.00s)
PASS
ok      github.com/headfirstgo/prose    0.007s
```

Если вы знаете имена одного или нескольких тестов (либо из вывода `go test -v`, либо от просмотра файлов с кодом тестов), добавьте аргумент `-run` для ограничения набора выполняемых тестов. Если указать после `-run` часть имени функции (или полное имя), будут выполнены только тестовые функции, имена которых содержат заданную вами строку.

Например, если добавить `-run Two` в команду `go run`, будут выполнены только тестовые функции, в имени которых есть `Two`. В нашем случае это означает, что будет выполнен только тест `TestTwoElements`. (`-run` можно использовать с флагом `-v` или без него, но на наш взгляд, флаг `-v` помогает однозначно определить, какие именно тесты выполняются).

```
File  Edit  Window  Help
$ go test github.com/headfirstgo/prose -v -run Two
=== RUN   TestTwoElements
--- PASS: TestTwoElements (0.00s)
PASS
ok      github.com/headfirstgo/prose    0.007s
```

Выполняются тесты, в именах которых есть строка «Two».

Если вместо этого добавить аргумент `-run Elements`, то будут выполнены тесты `TestTwoElements` и `TestThreeElements`. (Но не `TestOneElement`, потому что в конце имени должна быть буква `s`.)

```
File  Edit  Window  Help
$ go test github.com/headfirstgo/prose -v -run Elements
=== RUN   TestTwoElements
--- PASS: TestTwoElements (0.00s)
=== RUN   TestThreeElements
--- PASS: TestThreeElements (0.00s)
PASS
ok      github.com/headfirstgo/prose    0.007s
```

Выполняются тесты, в именах которых есть строка «Elements».

Табличные тесты

Три тестовые функции содержат немало повторяющегося кода. Фактически между тестами изменяются только сегменты, передаваемые `JoinWithCommas`, и строка, которую должна возвращать функция.

```
func TestOneElement(t *testing.T) {
       list := []string{"apple"}
       want := "apple"
       got := JoinWithCommas(list)
       if got != want {
              t.Error(errorString(list, got, want))
       }
}

func TestTwoElements(t *testing.T) {
       list := []string{"apple", "orange"}
       want := "apple and orange"
       got := JoinWithCommas(list)
       if got != want {
              t.Error(errorString(list, got, want))
       }
}

func TestThreeElements(t *testing.T) {
       list := []string{"apple", "orange", "pear"}
       want := "apple, orange, and pear"
       got := JoinWithCommas(list)
       if got != want {
              t.Error(errorString(list, got, want))
       }
}

func errorString(list []string, got string, want string) string {
       return fmt.Sprintf("JoinWithCommas(%#v) = \"%s\", want \"%s\"", list, got, want)
}
```

Повторяющийся код.

Чтобы не поддерживать раздельные тестовые функции, можно построить «таблицу» входных данных и ожидаемых результатов, а затем использовать одну тестовую функцию для проверки всех записей таблицы.

Стандартного формата таких таблиц не существует, но в одном из распространенных решений определяется новый тип (специально для ваших тестов), в котором хранятся входные данные и ожидаемый результат для каждого теста. Ниже приведен тип `testData`, который можно было бы использовать для нашего примера; в поле `list` хранится сегмент строк, передаваемый `JoinWithCommas`, а в поле `want` хранится соответствующая строка, которая должна возвращаться для этого сегмента.

```
type testData struct {
       list []string
       want string
}
```

Сегмент, который передается JoinWithCommas.

Строка, которую должна вернуть функция JoinWithCommas для этого сегмента.

Табличные тесты (продолжение)

Тип `testData` можно определить прямо в файле *lists_test.go*, где он будет использоваться.

Три тестовые функции можно объединить в одну функцию `TestJoinWithCommas`. В самом начале создается сегмент `tests`, а значения переменных `list` и `want` из старых функций `TestOneElement`, `TestTwoElements` и `TestThreeElements` перемещаются в значения `testData` в сегменте `tests`.

Затем в цикле перебираются все значения `testData` в сегменте. Мы передаем сегмент `list` функции `JoinWithCommas`, а возвращенная строка сохраняется в переменной `got`. Если переменная `got` не равна строке в поле `want` структуры `testData`, мы вызываем функцию `Errorf` и используем ее для форматирования сообщения об отказе теста, как это было сделано во вспомогательной функции `errorString`. (И поскольку функция `errorString` становится лишней, ее можно удалить.)

```
import "testing"

type testData struct {
    list []string
    want string
}

func TestJoinWithCommas(t *testing.T) {
    tests := []testData{
        testData{list: []string{"apple"}, want: "apple"},
        testData{list: []string{"apple", "orange"}, want: "apple and orange"},
        testData{list: []string{"apple", "orange", "pear"}, want: "apple, orange, and pear"},
    }
    for _, test := range tests {
        got := JoinWithCommas(test.list)
        if got != test.want {
            t.Errorf("JoinWithCommas(%#v) = \"%s\", want \"%s\"", test.list, got, test.want)
        }
    }
}
```

Тип testData можно определить прямо в тестовом файле.

Эта функция заменит три старые функции.

Данные из TestOneElement.

Данные из TestTwoElements.

Данные из TestThreeElements.

Создание сегмента значений testData.

Обрабатывает каждое значение testData в сегменте.

Сегмент передается JoinWithCommas.

Если полученное возвращаемое значение не равно ожидаемому...

Отформатировать сообщение об ошибке и установить признак отказа теста.

Обновленный код получается куда более компактным и содержит меньше повторений, но тесты в таблице проходят точно так же, как проходили для отдельных тестовых функций!

```
File  Edit  Window  Help
$ go test github.com/headfirstgo/prose
ok      github.com/headfirstgo/prose    0.006s
```

Решение проблемы с паникой при помощи тестов

Но самое замечательное в табличных тестах — это простота добавления новых тестов при необходимости. Предположим, вы не уверены в том, как поведет себя функция `JoinWithCommas` при передаче пустого сегмента. Чтобы узнать это, достаточно добавить новую структуру `testData` в сегмент `tests`. Укажем, что при передаче `JoinWithCommas` пустого сегмента должна возвращаться пустая строка:

```
func TestJoinWithCommas(t *testing.T) {
    tests := []testData{
        testData{list: []string{}, want: ""},
        testData{list: []string{"apple"}, want: "apple"},
        testData{list: []string{"apple", "orange"}, want: "apple and orange"},
        testData{list: []string{"apple", "orange", "pear"}, want: "apple, orange, and pear"},
    }
    // ...
}
```

Добавляем новое значение testData, которое будет передавать JoinWithCommas пустой сегмент.

Похоже, мы не зря беспокоились. Если запустить тест, возникает ситуация паники с выдачей трассировки стека:

```
--- FAIL: TestJoinWithCommas (0.00s)
panic: runtime error: slice bounds out of range [recovered]
        panic: runtime error: slice bounds out of range

goroutine 5 [running]:
testing.tRunner.func1(0xc4200a20f0)
        /usr/go/1.10/libexec/src/testing/testing.go:742 +0x29d
panic(0x110a480, 0x11d6fd0)
        /usr/go/1.10/libexec/src/runtime/panic.go:505 +0x229
github.com/headfirstgo/prose.JoinWithCommas(0x11fa400, 0x0, 0x0, 0x10afead, 0x11ae270)
        /Users/jay/go/src/github.com/headfirstgo/prose/lists.go:11 +0x1bf
github.com/headfirstgo/prose.TestJoinWithCommas(0xc4200a20f0)
        /Users/jay/go/src/github.com/headfirstgo/prose/lists_test.go:20 +0x250
...
FAIL    github.com/headfirstgo/prose    0.009s
```

Очевидно, какой-то код попытался обратиться к индексу, выходящему за границы массива (то есть к несуществующему элементу).

```
panic: runtime error: slice bounds out of range
```

Из трассировки стека видно, что паника возникла в строке 11 файла *lists.go*, в функции `JoinWithCommas`:

Ошибка произошла в функции JoinWithCommas.

```
github.com/headfirstgo/prose.JoinWithCommas(0x11fa400, 0x0, 0x0, 0x10afead, 0x11ae270)
        /Users/jay/go/src/github.com/headfirstgo/prose/lists.go:11 +0x1bf
```

Ошибка в строке 11 файла lists.go.

Решение проблемы с паникой при помощи тестов (продолжение)

Итак, паника возникла в строке 11 файла *lists.go*... Здесь мы обращаемся ко всем элементам сегмента, кроме последнего, и объединяем их через запятую. Но поскольку передаваемый сегмент `phrases` пуст, нет *ни одного* элемента, к которому можно было бы обратиться.

```
func JoinWithCommas(phrases []string) string {
      if len(phrases) == 1 {
            return phrases[0]
      } else if len(phrases) == 2 {
            return phrases[0] + " and " + phrases[1]
      } else {
            result := strings.Join(phrases[:len(phrases)-1], ", ")
            result += ", and "
            result += phrases[len(phrases)-1]
            return result
      }
}
```

Паника возникает в этом месте при попытке обратиться к элементам пустого сегмента.

Если сегмент `phrases` пуст, не следует пытаться обращаться *ни к одному* элементу из него. Объединять нечего, поэтому достаточно вернуть пустую строку. Добавим в команду `if` еще одно условие, которое возвращает пустую строку в том случае, если значение `len(phrases)` равно `0`.

```
func JoinWithCommas(phrases []string) string {
      if len(phrases) == 0 {
            return ""
      } else if len(phrases) == 1 {
            return phrases[0]
      } else if len(phrases) == 2 {
            return phrases[0] + " and " + phrases[1]
      } else {
            result := strings.Join(phrases[:len(phrases)-1], ", ")
            result += ", and "
            result += phrases[len(phrases)-1]
            return result
      }
}
```

Если сегмент пуст, просто вернуть пустую строку.

После этого при повторном запуске все проходит несмотря на то, что тесты вызывают `JoinWithCommas` с пустым сегментом!

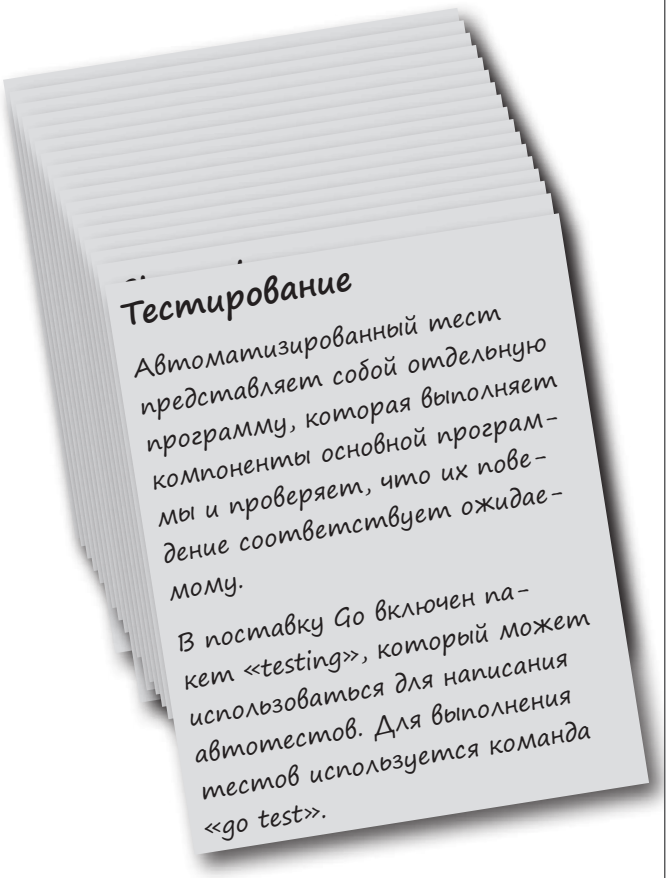
```
File Edit Window Help
$ go test github.com/headfirstgo/prose
ok      github.com/headfirstgo/prose    0.006s
```

Вероятно, вы сможете придумать другие изменения и улучшения для функции `JoinWithCommas`. Действуйте! Это можно делать без малейшего риска что-либо сломать. Если вы будете запускать тесты после каждого изменения, то будете уверены в том, что все работает как положено (или нет — но тогда вы будете четко представлять, что именно нужно исправить!).

Ваш инструментарий Go

Глава 14 осталась позади! В ней ваш инструментарий пополнился средствами тестирования.

ГЛАВА 14



КЛЮЧЕВЫЕ МОМЕНТЫ

- Автотест выполняет код для заданного набора входных данных и проверяет результат. Если он совпадает с ожидаемым значением, тест «проходит»; в противном случае — «не проходит».
- Команда `go test` используется для выполнения тестов. Она ищет в заданном пакете файлы, имена которых заканчиваются суффиксом *_test.go*.
- Тесты не обязательно делать частью того же пакета, что и тестируемый код, но это позволит вам обращаться к неэкспортируемым типам или функциям этого пакета.
- Тесты должны использовать тип из пакета `testing`, поэтому этот пакет должен импортироваться в начале каждого тестового файла.
- Файл *_test.go* может содержать одну или несколько тестовых функций, имена которых начинаются с `Test`. Остаток имени выбирается произвольно.
- Тестовые функции должны получать один параметр: указатель на значение `testing.T`.
- Код теста может вызывать функции и методы вашего пакета, а потом проверять, что возвращаемые значения соответствуют ожидаемым. Если значения не совпадают, тест должен устанавливать признак отказа.
- Чтобы сообщить о том, что тест не прошел, можно вызывать методы (например, `Error`) значения `testing.T`. Большинство методов получает строку с сообщением, объясняющим причину отказа.
- Метод `Errorf` работает аналогично `Error`, но он получает строку форматирования, как и функция `fmt.Printf`.
- Функции в файлах *_test.go*, имена которых не начинаются с `Test`, не запускаются командой `go test`. Они могут использоваться тестами в качестве «вспомогательных» функций.
- **Табличные тесты** обрабатывают «таблицы» с входными данными и ожидаемым выводом. Они передают каждый набор входных данных тестируемому коду и проверяют, соответствуют ли результаты ожидаемым значениям.

Упражнение
Решение

math.go

```
package arithmetic

func Add(a float64, b float64) float64 {
       return a + b
}

func Subtract(a float64, b float64) float64 {
       return a - b
}
```

math_test.go

Тот же пакет, что и тестируемый код.

```
package arithmetic
```

Для типа testing.T должен быть импортирован этот пакет.

```
import "testing"
```

Функции test должны получить *testing.T.

```
func Test Add(t *testing.T) {
    if Add(1, 2) != 3 {
        t.Error("1 + 2 did not equal 3")
    }
}
```

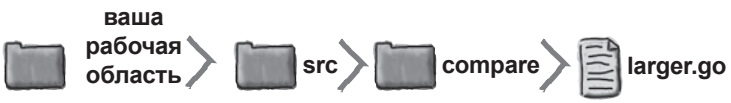
Вызывает тестируемый код. Если возвращаемое значение отличается от ожидаемого, тест не проходит.

Функции test должны получить *testing.T.

```
func Test Subtract(t *testing.T) {
    if Subtract(8, 4) != 4 {
        t.Error("8 - 4 did not equal 4")
    }
}
```

Вызывает тестируемый код. Если возвращаемое значение отличается от ожидаемого, тест не проходит.

Развлечения с магнитами. Решение




```
package compare

func Larger(a int, b int) int {
        if a < b {
                return a
        } else {
                return b
        }
}
```

Наоборот!

```
File Edit Window Help
$ go test compare
--- FAIL: TestFirstLarger (0.00s)
        larger_test.go:12: Larger(2, 1) = 1, want 2
--- FAIL: TestSecondLarger (0.00s)
        larger_test.go:20: Larger(4, 8) = 4, want 8
FAIL
FAIL    compare 0.007s
```


```
package compare

import (
        "fmt"
        "testing"
)

func TestFirstLarger(t *testing.T) {
        want := 2
        got := Larger(2, 1)
        if got != want {
                t.Error( errorString (2, 1, got, want) )
        }
}

func TestSecondLarger(t *testing.T) {
        want := 8
        got := Larger(4, 8)
        if got != want {
                t.Error( errorString (4, 8, got, want) )
        }
}
```

Вызывает вспомогательную функцию и использует ее возвращаемое значение в качестве сообщения об отказе теста.

Вызывает вспомогательную функцию и использует ее возвращаемое значение в качестве сообщения об отказе теста.

```
func errorString ( a int, b int, got int, want int ) string {
        return fmt.Sprintf( "Larger(%d, %d) = %d, want %d", a, b, got, want )
}
```

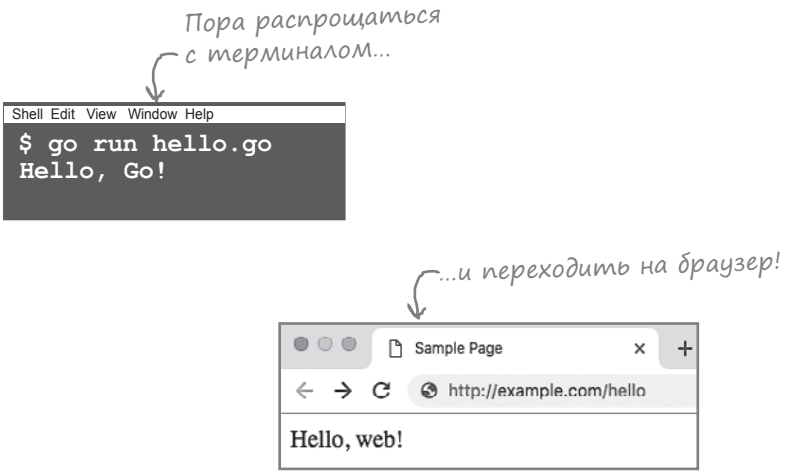
Веб-приложения

Мы живем в XXI веке. Пользователям нужны веб-приложения. Go поможет и в этом! Стандартная библиотека Go включает пакеты, при помощи которых можно размещать собственные веб-приложения и делать их доступными для любого веб-браузера. Последние две главы этой книги будут посвящены построению веб-приложений. Первое, что необходимо веб-приложению — способность отвечать на запросы, отправляемые браузером. В этой главе вы узнаете, как реализовать эту функцию с использованием пакета `net/http`.

Написание веб-приложений на языке Go

Приложение, работающее в терминале, — отличная штука... для вас. Но рядовые пользователи избалованы интернетом и не желают осваивать терминал, чтобы работать с вашим приложением. Они не хотят устанавливать ваше приложение. Им нужно, чтобы приложение было готово к работе, когда они щелкают на ссылке в браузере.

Не беспокойтесь! Go поможет в написании веб-приложений.



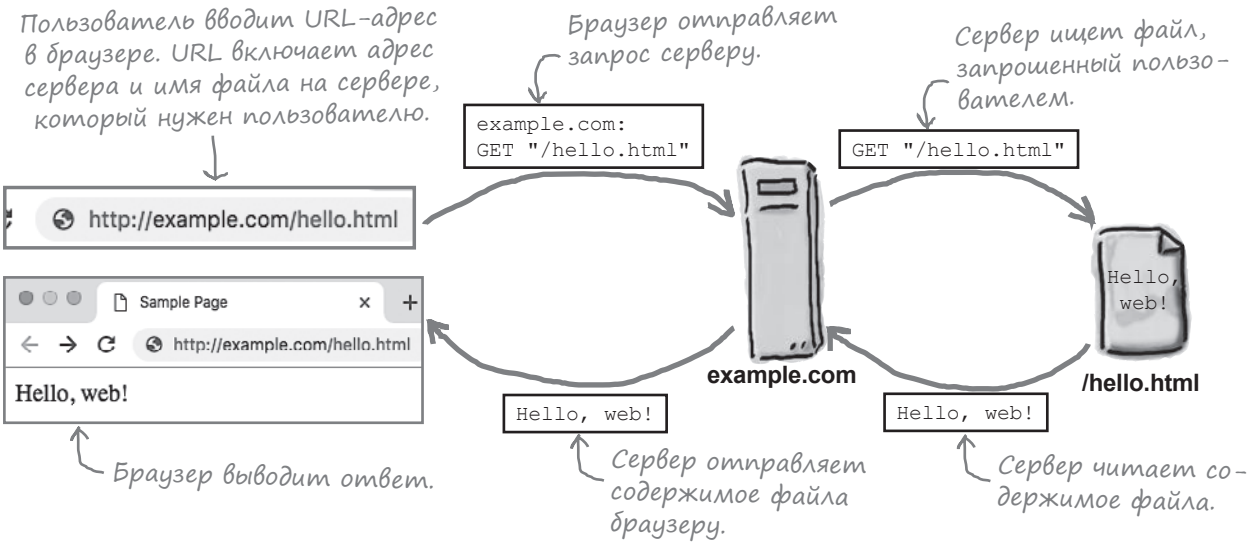
Честно признаем: написание веб-приложения — не самая простая задача. Она потребует от вас всего, что вы узнали до настоящего момента, а также некоторых новых навыков. Но в Go существуют отличные пакеты, которые сильно упростят эту задачу!

К их числу относится пакет `net/http`. Сокращением HTTP (от «**H**yper**T**ext **T**ransfer **P**rotocol», то есть «протокол передачи гипертекста») обозначается протокол, используемый для обмена данными между веб-браузерами и веб-серверами. С помощью `net/http` вы сможете использовать Go для создания собственных веб-приложений!

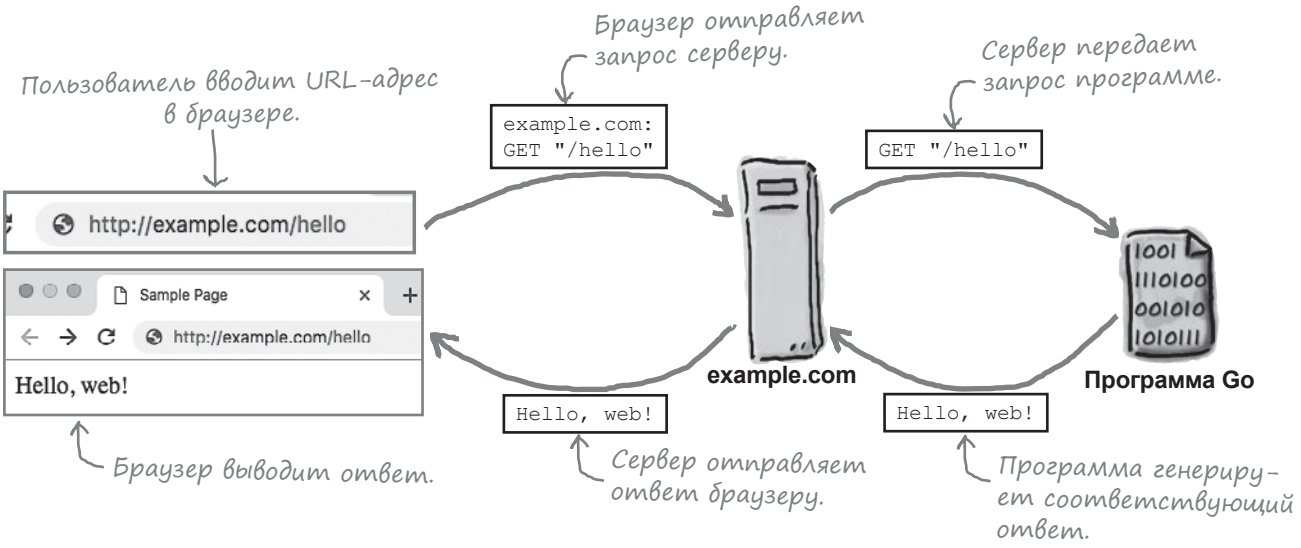
Браузеры, запросы, серверы и ответы

Когда вы вводите URL-адрес в своем браузере, фактически вы отправляете *запрос* на получение веб-страницы. Запрос передается *серверу*. Задача сервера — получить соответствующую страницу и вернуть ее браузеру в *ответе*.

На заре существования Всемирной паутины сервер обычно читал содержимое HTML-файла на своем жестком диске и отправлял эту разметку HTML-браузеру.



Но в наши дни сервер намного чаще взаимодействует с *программой* для выполнения запросов (вместо чтения из файла). Такая программа может быть написана практически на любом языке, в том числе и на Go!



Простое веб-приложение

Обработка запроса от браузера требует немалой работы. К счастью, делать все это самостоятельно вам не придется. В главе 13 мы использовали пакет `net/http` для выдачи запросов к веб-серверам. Пакет `net/http` также включает небольшой веб-сервер, который также может *отвечать* на запросы. От нас потребуется лишь написать код, который заполняет эти запросы данными.

В следующей программе пакет `net/http` используется для предоставления простых ответов браузеру. И хотя программа получается короткой, в ней происходит много всего, в том числе и незнакомого вам. Сначала мы запустим программу, а потом вернемся и разберем ее шаг за шагом.

```
package main

import (
       "log"
       "net/http"
)

func viewHandler(writer http.ResponseWriter, request *http.Request) {
       message := []byte("Hello, web!")
       _, err := writer.Write(message)
       if err != nil {
              log.Fatal(err)
       }
}

func main() {
       http.HandleFunc("/hello", viewHandler)
       err := http.ListenAndServe("localhost:8080", nil)
       log.Fatal(err)
}
```

Импортируем пакет «net/http».

Значение для обновления ответа, которое будет отправлено браузеру.

Значение, представляющее запрос от браузера.

Добавляем строку «Hello, web!» в ответ.

Если был получен запрос для URL-адреса, завершающегося суффиксом «/hello»...

...то вызывается функция viewHandler, которая генерирует ответ.

Обрабатывает запрос браузера и отвечает на него.

Сохраните приведенный код в файле и запустите его в терминале командой **`go run`**:

Запуск сервера.

```
File Edit Window Help
$ go run hello.go
```

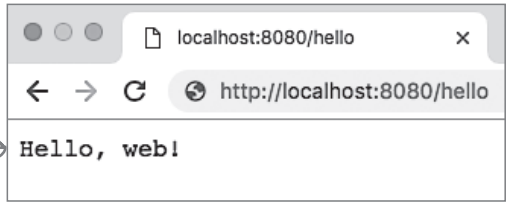
Мы запустили собственное веб-приложение! А теперь необходимо связаться с ним в браузере и протестировать его. Откройте браузер и введите следующий URL-адрес в адресной строке. (Если он кажется вам немного странным, не беспокойтесь: позже мы объясним, что все это значит.)

```
http://localhost:8080/hello
```

Браузер отправляет запрос приложению, которое отвечает строкой «`Hello, web!`». Мы только что отправили свой первый ответ браузеру!

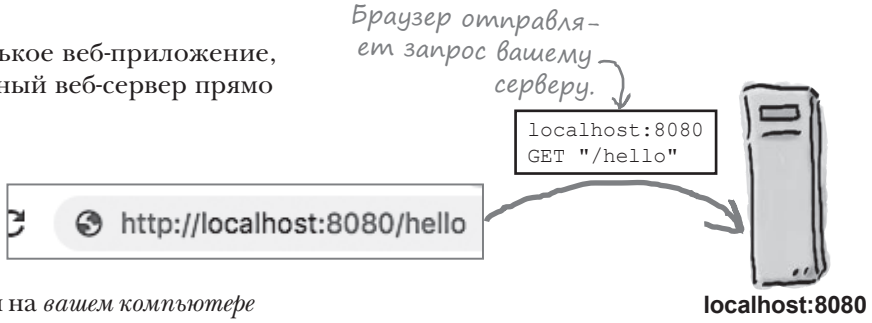
Приложение продолжает прослушивать запросы, пока мы его не остановим. Когда вы закончите работу со страницей, нажмите Ctrl+C в терминале, чтобы дать команду программе завершить работу.

Ответ нашего приложения.



Ваш компьютер общается сам с собой

Когда мы запустили свое маленькое веб-приложение, то оно запустило свой собственный веб-сервер прямо на вашем компьютере.



Так как приложение выполняется на *вашем компьютере* (а не на удаленном хосте в интернете), в URL используется специальное имя хоста `localhost`. Оно сообщает браузеру о том, что он должен установить соединение *с* вашего компьютера на *тот же* компьютер.

```
http://localhost:8080/hello
       Хост.  Порт.
```

Также необходимо указать порт как часть URL-адреса. (*Порт* – номер канала сетевых коммуникаций, на котором приложение ведет прослушивание запросов.) В своем коде мы указали, что сервер должен использовать для прослушивания порт 8080, поэтому мы включили номер порта в URL за именем хоста.

```
                                  Номер порта.
http.ListenAndServe("localhost:8080", nil)
```

Часто Задаваемые Вопросы

В: Я получаю сообщение об ошибке, в котором говорится, что браузер не смог установить связь!

О: Возможно, ваш сервер на самом деле не запустился. Поищите сообщения об ошибках в терминале. Также проверьте имя хоста и номер порта в браузере — возможно, вы ошиблись при вводе.

В: Почему в URL-адресе нужно указывать номер порта? На других веб-сайтах это делать необязательно!

О: Большинство веб-серверов использует для прослушивания запросов HTTP порт 80, так как этот порт по умолчанию используется браузерами для выдачи запросов HTTP. Но во многих операционных системах для запуска служб, ведущих прослушивание с портом 80, по соображениям безопасности требуются специальные разрешения. По этой причине мы настроили сервер для прослушивания с номером порта 8080.

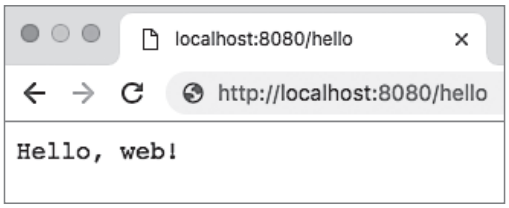
В: Мой браузер выводит сообщение «404 page not found».

О: Это ответ от сервера, что уже неплохо, но он означает, что запрашиваемый вами ресурс не был найден. Проверьте, что URL-адрес заканчивается суффиксом */hello*, и проверьте код серверной программы на возможные ошибки.

В: При попытке запустить приложение я получил сообщение об ошибке, в котором говорится, что адрес уже используется!

О: Ваша программа пытается вести прослушивание с портом, уже занятым другой программой (чего ваша ОС не разрешает). Вы запускали серверную программу более одного раза? А если запускали, то нажали ли Ctrl+C в терминале, чтобы остановить ее после завершения работы? Не забывайте, что старый сервер необходимо остановить перед тем, как запускать новый.

Простое веб-приложение: шаг за шагом



А теперь подробно изучим разные части нашего маленького веб-приложения.

В функции `main` вызывается функция `http.HandleFunc` со строкой `"/hello"` и функцией `viewHandler`. (В Go поддерживаются *функции первого класса*; это означает, что функции могут передаваться другим функциям. Вскоре мы поговорим об этом подробнее.) Приложению дается команда вызывать `viewHandler` при получении запроса URL-адреса, завершающегося суффиксом */hello*.

Затем вызывается функция `http.ListenAndServe`, которая запускает веб-сервер. Ей передается строка `"localhost:8080"`, чтобы веб-сервер принимал запросы только от вашего компьютера через порт 8080. (Когда вы будете готовы открыть приложение для запросов с других компьютеров, используйте строку `"0.0.0.0:8080"`. Также можно заменить номер порта и использовать другое значение вместо 8080.) Значение `nil` во втором аргументе означает, что запросы будут обрабатываться с использованием функций, заданных при помощи `HandleFunc`.

(Если позднее вас заинтересуют другие способы обработки запросов, поищите в документации описания функции «ListenAndServe», интерфейса «Handler» и типа «ServeMux» из пакета «http».)

Функция `ListenAndServe` вызывается *после* `HandleFunc`, потому что `ListenAndServe` будет работать бесконечно в цикле, пока не произойдет ошибка. В этом случае функция вернет эту ошибку, сообщение о которой будет выведено перед завершением программы. Если же ошибок не будет, программа просто будет выполняться бесконечно, пока вы не прервете ее нажатием клавиш Ctrl+C в терминале.

Если будет получен запрос для URL-адреса, завершающегося «/hello»...

...для генерирования ответа вызывается функция viewHandler.

```
func main() {
        http.HandleFunc("/hello", viewHandler)
        err := http.ListenAndServe("localhost:8080", nil)
        log.Fatal(err)
}
```

Прослушивать запрос браузера и отвечать на них.

По сравнению с `main` в функции `viewHandler` нет ничего необычного. Сервер передает `viewHandler` значение `http.ResponseWriter`, которое используется для записи данных в ответ браузера, и указатель на значение `http.Request`, представляющее запрос браузера. (Значение `Request` не используется в этой программе, но функция-обработчик все равно должна получать его.)

Значение для изменения ответа, отправляемого браузеру.

Значение, представляющее запрос от браузера.

```
func viewHandler(writer http.ResponseWriter, request *http.Request) {
        ...
}
```

Простое веб-приложение: шаг за шагом (продолжение)

Внутри `viewHandler` мы добавляем данные в ответ на вызов метода `Write` для `ResponseWriter`. Метод `Write` не принимает строки, но может получить сегмент значений `byte`, поэтому строка `"Hello, web!"` преобразуется в `[]byte`, а затем передается `Write`.

```
message := []byte("Hello, web!")
_, err := writer.Write(message)
```

Строка «Hello, web!» преобразуется в сегмент байтов.

Строка «Hello, web!» добавляется в ответ.

Возможно, вы помните значения `byte` из главы 13. Функция `ioutil.Readall` возвращала сегмент значений `byte` при вызове для ответа, полученного при помощи функции `http.Get`.

Мы еще не рассматривали тип `byte`; это один из базовых типов Go (как `float64` или `bool`), используемый для хранения низкоуровневых данных — например, прочитанных из файла или сетевого подключения. Сегмент значений `byte` не выведет никакой осмысленной информации, если вывести его напрямую, но если выполнить преобразование типа из сегмента значений `byte` в строку, вы получите осмысленный текст. (Конечно, при условии, что данные представляют осмысленный текст.) Итак, программа преобразует тело ответа в строку и выводит ее.

```
func main() {
        response, err := http.Get("https://example.com")
        if err != nil {
                log.Fatal(err)
        }
        defer response.Body.Close()
        body, err := ioutil.ReadAll(response.Body)
        if err != nil {
                log.Fatal(err)
        }
        fmt.Println(string(body))
}
```

Сетевое подключение закрывается при выходе из функции «main».

Читает все данные в ответе.

Преобразует данные в строку и выводит ее.

Как было показано в главе 13, `[]byte` можно преобразовать в строку:

```
fmt.Println(string([]byte{72, 101, 108, 108, 111}))
```

```
Hello
```

Как вы только что видели в простом веб-приложении, строка может быть преобразована в `[]byte`.

```
fmt.Println([]byte("Hello"))
```

```
[72 101 108 108 111]
```

Метод `Write` значения `ResponseWriter` возвращает количество успешно записанных байтов и ошибку, если она была обнаружена. Ничего полезного с количеством байтов не сделать, поэтому мы его игнорируем. Но если произошла ошибка, то программа выводит сообщение и завершается.

```
_, err := writer.Write(message)
if err != nil {
        log.Fatal(err)
}
```

Пути к ресурсам

Когда мы вводим URL-адрес в браузере, чтобы обратиться к веб-приложению, он обязательно должен завершаться суффиксом */hello*. Но почему это важно?

```
http://localhost:8080/hello
```

У сервера есть множество различных ресурсов, которые он может отправлять в браузер, включая HTML-страницы, изображения и прочее.

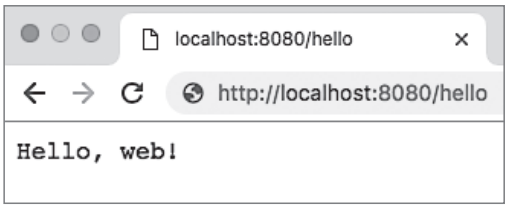
Часть URL-адреса, следующая за адресом хоста и портом, содержит *путь* к ресурсу. Она сообщает серверу, с каким из его многочисленных ресурсов вы хотите работать. Сервер `net/http` извлекает путь из завершающей части URL и использует его для обработки запроса.

```
http://localhost:8080/hello
                      Путь.
```

Когда мы вызываем `http.HandleFunc` в своем веб-приложении, то передаем строку `"/hello"` и функцию `viewHandler`. Строка используется как путь к искомому ресурсу запроса. В дальнейшем каждый раз, когда будет получен запрос с путем `/hello`, приложение будет вызывать функцию `viewHandler`. После этого функция `viewHandler` должна сгенерировать ответ, соответствующий полученному запросу.

```
Если получен запрос для URL-адреса, завершающегося суффиксом «/hello»...
                                  ...для генерирования ответа вызывается функция viewHandler.
http.HandleFunc("/hello", viewHandler)
```

В данном случае это означает ответ с текстом «Hello, web!».



Впрочем, ваше приложение не может просто отвечать строкой «Hello, web!» на каждый полученный запрос. В большинстве приложений должны выдаваться разные ответы для разных путей запросов.

Одно из возможных решений — вызвать `HandleFunc` для каждого пути, который вы хотите обрабатывать, и предоставить отдельную функцию для обработки каждого пути. В таком случае ваше приложение сможет отвечать на запросы по всем этим путям.

Разные ответы для разных путей к ресурсам

Ниже приведена новая версия приложения, которая выдает приветствия на трех языках. Функция `HandleFunc` вызывается три раза. Для запросов с путем `"/hello"` вызывается функция `englishHandler`, для запросов с путем `"/salut"` — функция `frenchHandler`, и для запросов с путем `"/namaste"` — функция `hindiHandler`. Каждая из функций-обработчиков передает свои значения `ResponseWriter` и строку новой функции `write`, которая записывает строку в ответ.

```
package main

import (
       "log"
       "net/http"
)

func write(writer http.ResponseWriter, message string) {
       _, err := writer.Write([]byte(message))
       if err != nil {
              log.Fatal(err)
       }
}

func englishHandler(writer http.ResponseWriter, request *http.Request) {
       write(writer, "Hello, web!")
}
func frenchHandler(writer http.ResponseWriter, request *http.Request) {
       write(writer, "Salut web!")
}
func hindiHandler(writer http.ResponseWriter, request *http.Request) {
       write(writer, "Namaste, web!")
}

func main() {
       http.HandleFunc("/hello", englishHandler)
       http.HandleFunc("/salut", frenchHandler)
       http.HandleFunc("/namaste", hindiHandler)
       err := http.ListenAndServe("localhost:8080", nil)
       log.Fatal(err)
}
```

ResponseWriter от функции-обработчика.

Сообщение, добавляемое в ответ.

Строка преобразуется в сегмент байтов, как и прежде, и записывается в ответ.

Строка записывается в ответ.

Строка записывается в ответ.

Строка записывается в ответ.

Для запросов с путем «/hello» вызывается englishHandler.

Для запросов с путем «/salut» вызывается frenchHandler.

Для запросов с путем «/namaste» вызывается hindiHandler.

Упражнение

Ниже приведен код простого веб-приложения и несколько возможных ответов. Рядом с каждым ответом запишите URL-адрес, который нужно ввести в браузере для генерирования этого ответа.

```
package main

import (
       "log"
       "net/http"
)

func write(writer http.ResponseWriter, message string) {
       _, err := writer.Write([]byte(message))
       if err != nil {
              log.Fatal(err)
       }
}

func d(writer http.ResponseWriter, request *http.Request) {
       write(writer, "z")
}
func e(writer http.ResponseWriter, request *http.Request) {
       write(writer, "x")
}
func f(writer http.ResponseWriter, request *http.Request) {
       write(writer, "y")
}

func main() {
       http.HandleFunc("/a", f)
       http.HandleFunc("/b", d)
       http.HandleFunc("/c", e)
       err := http.ListenAndServe("localhost:4567", nil)
       log.Fatal(err)
}
```

Ответ: *URL-адрес для генерирования ответа:*

x .....

y .....

z .....

Ответы на с. 476.

Функции первого класса

При вызове `http.HandleFunc` с функциями-обработчиками мы не вызываем функцию-обработчик с передачей результата `HandleFunc`. Мы передаем `HandleFunc` *саму функцию*. Эта функция сохраняется для того, чтобы быть вызванной в будущем, при получении запроса с подходящим путем.

```
func main() {
        http.HandleFunc("/hello", englishHandler)
        http.HandleFunc("/salut", frenchHandler)
        http.HandleFunc("/namaste", hindiHandler)
        err := http.ListenAndServe("localhost:8080", nil)
        log.Fatal(err)
}
```

Функция englishHandler передается HandleFunc.

Функция frenchHandler передается HandleFunc.

Функция hindiHandler передается HandleFunc.

В языке Go поддерживаются **функции первого класса**.

В языках программирования с функциями первого класса программа может присвоить функцию переменной, а затем вызвать ее через переменную.

В приведенном ниже коде сначала определяется функция `sayHi`. В функции `main` объявляется переменная `myFunction` с типом `func()`: это означает, что в переменной может храниться функция.

Затем `myFunction` присваивается функция `sayHi`. Обратите внимание на отсутствие круглых скобок (мы не используем запись `sayHi()`), потому что в этом случае функция `sayHi` будет *вызвана*. Вводится только имя функции без скобок:

```
myFunction = sayHi
```

При такой записи функция `sayHi` будет присвоена переменной `myFunction`.

Но в следующей строке за именем переменной `myFunction` *находятся* круглые скобки:

```
myFunction()
```

С таким синтаксисом вызывается функция, хранящаяся в переменной `myFunction`.

Функция объявляется обычным образом.

```
func sayHi() {
        fmt.Println("Hi")
}

func main() {
        var myFunction func()
        myFunction = sayHi
        myFunction()
}
```

```
Hi
```

Объявляется переменная с типом «func()». В такой переменной может храниться функция.

Переменной присваивается функция sayHi.

Вызов функции, хранящейся в переменной.

Передача функций другим функциям

Языки программирования с функциями первого класса также позволяют передавать функции в аргументах других функций. В этом коде определяются простые функции `sayHi` и `sayBye`. Также определяется функция `twice`, которая получает другую функцию в параметре с именем `theFunction`. Затем функция `twice` дважды вызывает функцию, хранящуюся в `theFunction`.

В функции `main` мы вызываем `twice` и передаем функцию `sayHi` в аргументе, в результате чего `sayHi` вызывается дважды. Затем `twice` вызывается с функцией `sayBye`, в результате чего дважды вызывается `sayBye`.

```
func sayHi() {
       fmt.Println("Hi")
}
func sayBye() {
       fmt.Println("Bye")
}

func twice(theFunction func()) {
       theFunction()
       theFunction()
}

func main() {
       twice(sayHi)
       twice(sayBye)
}
```

Функция «twice» получает другую функцию в параметре.

Вызываем переданную функцию.

Вызываем переданную функцию (снова).

Функция «sayHi» передается функции «twice».

Функция «sayBye» передается функции «twice».

```
Hi
Hi
Bye
Bye
```

Функции как типы

Впрочем, вы не сможете просто использовать любую функцию как аргумент при вызове любой другой функции. При попытке передать функцию `sayHi` в аргументе `http.HandleFunc` компилятор выдаст сообщение об ошибке:

```
func sayHi() {
       fmt.Println("Hi")
}

func main() {
       http.HandleFunc("/hello", sayHi)
       err := http.ListenAndServe("localhost:8080", nil)
       log.Fatal(err)
}
```

Пытаемся назначить sayHi функцией-обработчиком запросов HTTP.

Ошибка компиляции.

```
cannot use sayHi (type func()) as type func(http.ResponseWriter, *http.Request)
in argument to http.HandleFunc
```

Функции как типы (продолжение)

Параметры и возвращаемое значение функции являются частью ее типа. Для переменной, в которой хранится функция, должны быть указаны параметры и возвращаемые значения этой функции. В переменной могут храниться только функции, у которых количество и тип параметров соответствуют заданному типу.

В этом коде определяется переменная `greeterFunction` с типом `func()`: она предназначена для функций, которые не получают параметров и не возвращают значений. Затем определяется переменная `mathFunction` с типом `func(int, int) float64`: она предназначена для функций, которые получают два целочисленных параметра и возвращают значение `float64`.

В коде также определяются функции `sayHi` и `divide`. Если присвоить `sayHi` переменной `greeterFunction`, а `divide` — переменной `mathFunction`, программа успешно компилируется и работает:

```
func sayHi() {
       fmt.Println("Hi")
}
func divide(a int, b int) float64 {
       return float64(a) / float64(b)
}

func main() {
       var greeterFunction func()
       var mathFunction func(int, int) float64
       greeterFunction = sayHi
       mathFunction = divide
       greeterFunction()
       fmt.Println(mathFunction(5, 2))
}
```

В переменной хранится функция без параметров и без возвращаемого значения.

В этой переменной может храниться функция с двумя параметрами int и возвращаемым значением float64.

Функция «sayHi» присваивается переменной greeterFunction.

Функция «divide» присваивается переменной mathFunction.

```
Hi
2.5
```

Но если попытаться поменять их местами, компилятор снова выдает сообщение об ошибке:

```
       greeterFunction = divide
       mathFunction = sayHi
```

Ошибка компиляции.

```
cannot use divide (type func(int, int) float64) as type func() in assignment
cannot use sayHi (type func()) as type func(int, int) float64 in assignment
```

Функция `divide` получает два параметра `int` и возвращает значение `float64`, поэтому она не может храниться в переменной `greeterFunction` (в которой должна храниться функция без параметров и возвращаемого значения). А функция `sayHi` не получает параметров и не возвращает значения, поэтому не может храниться в переменной `mathFunction` (предназначенной для функций с двумя параметрами `int` и возвращаемым значением `float64`).

Функции как типы (продолжение)

У функций, которые получают функцию в параметре, также должны быть указаны параметры и возвращаемые типы, которыми должна обладать передаваемая функция.

Ниже приведена функция `doMath` с параметром `passedFunction`. Передаваемая функция должна получать два параметра `int` и возвращать одно значение `float64`.

Мы также определяем функции `divide` и `multiply`; обе эти функции получают два параметра `int` и возвращают одно значение `float64`. Любая из этих функций может быть передана `doMath`.

```
func doMath(passedFunction func(int, int) float64) {
      result := passedFunction(10, 2)
      fmt.Println(result)
}
func divide(a int, b int) float64 {
      return float64(a) / float64(b)
}
func multiply(a int, b int) float64 {
      return float64(a * b)
}

func main() {
      doMath(divide)
      doMath(multiply)
}
```

Функция doMath получает другую функцию в параметре. Переданная функция должна получать два целых числа и возвращать float64.

Вызов переданной функции.

Выводит возвращаемое значение переданной функции.

Функция, которая может передаваться doMath.

Другая функция, которая может передаваться doMath.

Функция «divide» передается doMath.

Функция «multiply» передается doMath.

```
5
20
```

Функция, которая не соответствует заданному типу, не может передаваться `doMath`.

```
func main() {
      doMath(sayHi)
}
```

Функция sayHi не получает параметров и не имеет возвращаемого значения.

Ошибка компиляции.

```
cannot use sayHi (type func()) as type func(int, int) float64 in argument to doMath
```

Становится понятно, почему мы получаем ошибки компиляции при передаче неподходящей функции `http.HandleFunc`. `HandleFunc` ожидает, что переданная функция будет получать в параметрах `ResponseWriter` и указатель на `Request`. Стоит вам передать что-нибудь еще — вы получите ошибку компиляции.

И это хорошо. Функция, которая не может проанализировать запрос и записать ответ, пожалуй, будет бесполезной для обработки запросов браузера. При попытке передать функцию с неправильным типом Go сообщит вам о проблеме даже до того, как программа откомпилируется.

```
http.HandleFunc("/hello", sayHi)
```

Ошибка компиляции.

```
cannot use sayHi (type func()) as type func(http.ResponseWriter, *http.Request)
in argument to http.HandleFunc
```

У бассейна

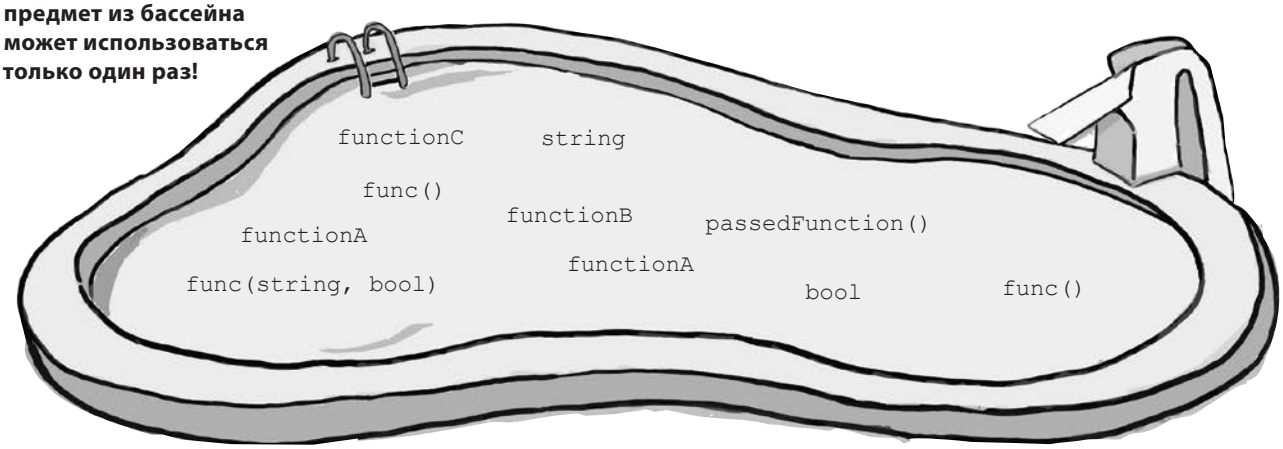
Выловите из бассейна фрагменты кода и разместите их в пустых строках кода. Каждый фрагмент может использоваться **только один** раз; использовать все фрагменты не обязательно. Ваша **задача**: создать программу, которая работает и выводит показанный результат.

```
func callFunction(passedFunction _________) {
      passedFunction()
}
func callTwice(passedFunction _________) {
      passedFunction()
      passedFunction()
}
func callWithArguments(passedFunction ___________________) {
      passedFunction("This sentence is", false)
}
func printReturnValue(passedFunction func() string) {
      fmt.Println(_____________________)
}

func functionA() {
      fmt.Println("function called")
}
func functionB() _________ {
      fmt.Println("function called")
      return "Returning from function"
}
func functionC(a string, b bool) {
      fmt.Println("function called")
      fmt.Println(a, b)
}

func main() {
      callFunction(____________)
      callTwice(____________)
      callWithArguments(functionC)
      printReturnValue(functionB)
}
```

Результат.

```
function called
function called
function called
function called
This sentence is false
function called
Returning from function
```

Примечание: каждый предмет из бассейна может использоваться только один раз!



Ответ на с. 477.

Что дальше

Теперь вы знаете, как получить запрос от браузера и как отправить ответ. Самое сложное позади!

```
package main

import (
       "log"
       "net/http"
)

func viewHandler(writer http.ResponseWriter, request *http.Request) {
       message := []byte("Hello, web!")
       _, err := writer.Write(message)
       if err != nil {
              log.Fatal(err)
       }
}

func main() {
       http.HandleFunc("/hello", viewHandler)
       err := http.ListenAndServe("localhost:8080", nil)
       log.Fatal(err)
}
```

Значение для обновления ответа, который будет отправлен браузеру.

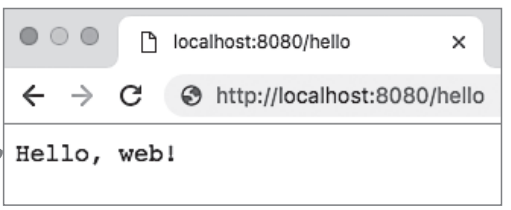
Значение, представляющее запрос от браузера.

В ответ добавляется текст «Hello, web!».

Если получен запрос для URL-адреса, завершающегося суффиксом «/hello»...

...для генерирования ответа вызывается функция viewHandler.

Прослушивание запросов браузера и генерирование ответов на них.



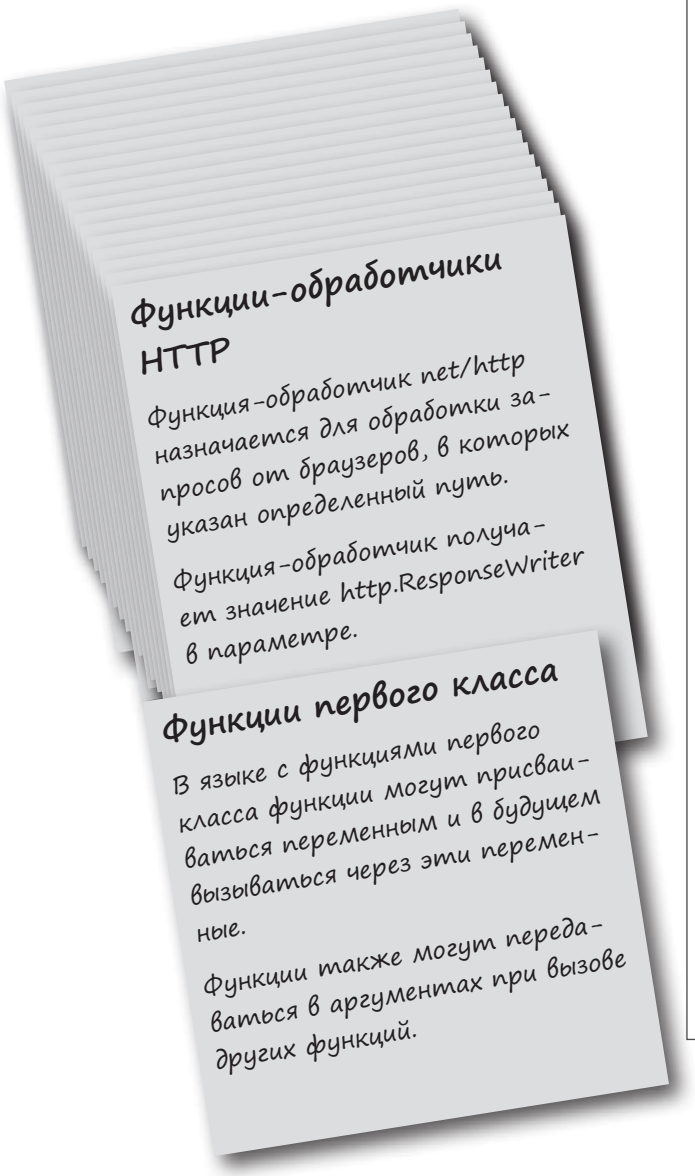
Ответ нашего приложения!

В последней главе книги мы воспользуемся этими знаниями для построения более сложных приложений.

До сих пор в наших ответах использовался простой текст. Мы будем использовать язык HTML для структурирования страницы. В частности, вы научитесь пользоваться пакетом `html/template` для вставки данных в разметку HTML перед их отправкой браузеру. До скорой встречи!

Ваш инструментарий Go

Глава 15 осталась позади! В ней ваш инструментарий пополнился функциями-обработчиками HTTP и функциями первого класса.



КЛЮЧЕВЫЕ МОМЕНТЫ

- Функция `ListenAndServe` из пакета `net/http` запускает веб-сервер с прослушиванием запросов с использованием указанного порта.
- Имя хоста `localhost` обозначает подключения с вашего компьютера на ваш компьютер.
- Каждый запрос HTTP включает путь к ресурсу, который указывает, какой из многочисленных ресурсов сервера запрашивается браузером.
- Функция `HandleFunc` получает строку пути и функцию, которая будет обрабатывать запросы для этого пути.
- Многократные вызовы `HandleFunc` позволяют назначить разные функции-обработчики для разных путей.
- Функции-обработчики должны получать в параметрах значение `http.ResponseWriter` и указатель на значение `http.Request`.
- Если вызвать метод `Write` для `http.ResponseWriter` с сегментом байтов, эти данные будут добавлены в ответ, отправляемый браузеру.
- Переменные, в которых могут храниться функции, имеют тип функции.
- Тип функции включает количество и тип параметров, которые получает функция (если они есть), и количество и тип значений, которые возвращаются функцией (если они есть).
- Если переменная `myVar` содержит функцию, то вы можете вызвать эту функцию, поставив круглые скобки (с аргументами, которые должны передаваться функции) после имени переменной.

Упражнение
Решение

Ниже приведен код простого веб-приложения и несколько возможных ответов. Рядом с каждым ответом запишите URL-адрес, который нужно ввести в браузере для генерирования этого ответа.

```
package main

import (
       "log"
       "net/http"
)

func write(writer http.ResponseWriter, message string) {
       _, err := writer.Write([]byte(message))
       if err != nil {
              log.Fatal(err)
       }
}

func d(writer http.ResponseWriter, request *http.Request) {
       write(writer, "z")
}
func e(writer http.ResponseWriter, request *http.Request) {
       write(writer, "x")
}
func f(writer http.ResponseWriter, request *http.Request) {
       write(writer, "y")
}

func main() {
       http.HandleFunc("/a", f)
       http.HandleFunc("/b", d)
       http.HandleFunc("/c", e)
       err := http.ListenAndServe("localhost:4567", nil)
       log.Fatal(err)
}
```

Обратите внимание: здесь задан другой порт!

Ответ:	*URL-адрес для генерирования ответа:*
x	*http://localhost:4567/c*
y	*http://localhost:4567/a*
z	*http://localhost:4567/b*

У бассейна. Решение

```
func callFunction(passedFunction func() ) {
      passedFunction()
}
func callTwice(passedFunction func() ) {
      passedFunction()
      passedFunction()
}
func callWithArguments(passedFunction func(string, bool) ) {
      passedFunction("This sentence is", false)
}
func printReturnValue(passedFunction func() string) {
      fmt.Println( passedFunction() )
}

func functionA() {
      fmt.Println("function called")
}
func functionB() string {
      fmt.Println("function called")
      return "Returning from function"
}
func functionC(a string, b bool) {
      fmt.Println("function called")
      fmt.Println(a, b)
}

func main() {
      callFunction( functionA )
      callTwice( functionA )
      callWithArguments(functionC)
      printReturnValue(functionB)
}
```

Из тела callFunction видно, что переданная функция не получает параметров.

Из тела callTwice видно, что переданная функция не получает параметров.

Из тела функции callWithArguments видно, что переданная функция должна принимать эти типы параметров.

Вызвать переданную функцию и вывести ее возвращаемое значение.

Если функция functionB должна передаваться printReturnValue, она должна возвращать строку.

Только функция functionA имеет правильный набор параметров (и правильный результат).

```
function called
function called
function called
function called
This sentence is false
function called
Returning from function
```

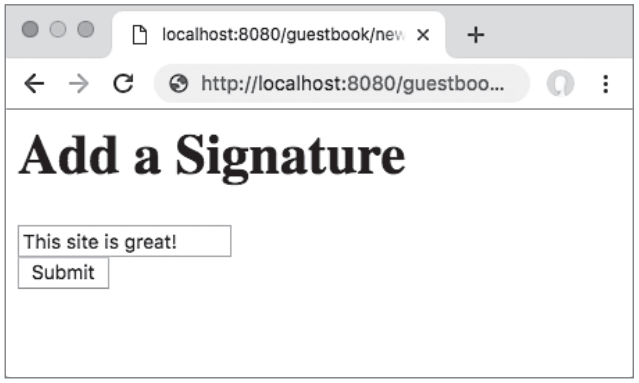
Шаблоны HTML

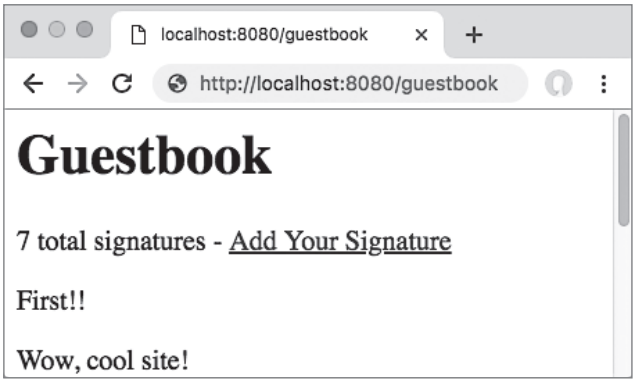
Веб-приложение должно выдавать ответ с разметкой HTML, а не с простым текстом. Для сообщений электронной почты и постов в соцсетях достаточно простого текста. Тем не менее ваши страницы должны быть отформатированы с выделением заголовков и разбиением на абзацы. На них должны быть формы, в которых пользователь сможет ввести данные для приложения. Для решения таких задач вам понадобится разметка HTML.

Рано или поздно в разметку HTML потребуется вставлять данные. Для этого в Go существует пакет `html/template` — мощный инструмент для вставки данных в HTML-ответы приложения. Шаблоны играют ключевую роль при построении более масштабных и качественных веб-приложений, и в последней главе книги мы научим вас ими пользоваться!

Гостевая книга

А теперь применим на практике все, что вы узнали в главе 15. Мы собираемся построить простое приложение — гостевую книгу для веб-сайта. Ваши посетители получат возможность вводить на форме сообщения, которые затем будут сохраняться в файле. Пользователю также необходимо предоставить средства для просмотра списка предыдущих записей.





Прежде чем приложение заработает, придется основательно потрудиться. Но не беспокойтесь — мы разобьем этот процесс на несколько простых этапов. Давайте посмотрим, что нам предстоит сделать.

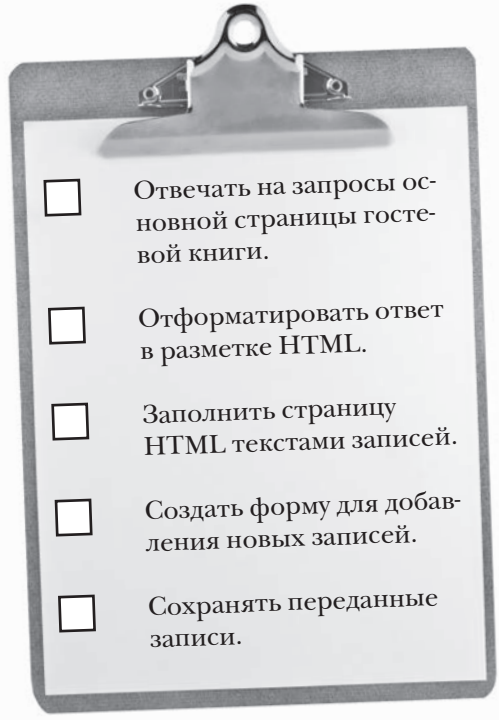
Мы должны создать приложение и добиться того, чтобы оно отвечало на запросы основной страницы гостевой книги. С этой частью особых сложностей не будет; все, что необходимо знать, уже было рассказано в предыдущей главе.

Затем необходимо включить разметку HTML в ответ. Мы ограничимся простой страницей с несколькими тегами HTML, которая будет сохранена в файле. Приложение будет загружать разметку HTML из файла и использовать ее в своем ответе.

Затем нужно взять записи, введенные посетителями, и встроить их в разметку HTML. Мы покажем, как сделать это с помощью пакета `html/template`.

На следующем шаге мы создадим отдельную страницу с формой для добавления новой записи. Это относительно легко делается средствами HTML.

Наконец, когда пользователь отправляет данные формы, содержимое формы необходимо сохранить как новую запись. Данные сохраняются в текстовом файле вместе со всеми ранее введенными записями, чтобы их можно было загрузить позднее.



Функции обработки запроса и проверки ошибок

Первая задача — отображение основной страницы гостевой книги. В предыдущей главе вы уже потренировались в создании веб-приложений, так что особых сложностей быть не должно. В функции `main` мы вызовем `http.HandleFunc` и подготовим приложение, чтобы оно вызывало функцию с именем `viewHandler` для любого запроса с путем `"/guestbook"`. Затем приложение запускает сервер вызовом `http.ListenAndServe`.

Пока функция `viewHandler` почти не отличается от функций-обработчиков из предыдущих примеров. Она получает `http.ResponseWriter` и указатель на `http.Request`, как и предыдущие обработчики. Мы преобразуем строку ответа в `[]byte` и вызовем метод `Write` значения `ResponseWriter`, чтобы добавить ее в ответ.

Функция `check` — единственная по-настоящему новая часть кода. В этом приложении будет много возвращаемых признаков потенциальных ошибок, и повторять код проверки и вывода во всех возможных местах было бы неразумно. По этой причине все ошибки передаются новой функции `check`. Если ошибка равна nil, функция `check` не делает ничего, но в остальных случаях она выводит сообщение об ошибке и завершает программу.

guestbook.go

```
package main

import (
       "log"
       "net/http"
)

func check(err error) {
       if err != nil {
              log.Fatal(err)
       }
}

func viewHandler(writer http.ResponseWriter, request *http.Request) {
       placeholder := []byte("signature list goes here")
       _, err := writer.Write(placeholder)
       check(err)
}

func main() {
       http.HandleFunc("/guestbook", viewHandler)
       err := http.ListenAndServe("localhost:8080", nil)
       log.Fatal(err)
}
```

Код сообщений об ошибках перемещается в эту функцию.

Как обычно, функциям-обработчикам передается значение ResponseWriter...

...а также указатель на значение Request.

Строка преобразуется в сегмент байтов...

...и добавляется в ответ методом Write.

Затем вызывается функция «check» для вывода информации об ошибке (если она была).

Функция viewHandler вызывается для любых запросов с путем «/guestbook».

Как обычно, сервер запускается для прослушивания с портом 8080.

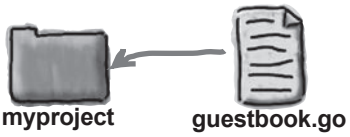
Ошибка никогда не равна nil, поэтому мы не вызываем для нее «check».

Вызов `Write` для `ResponseWriter` может вернуть ошибку, но может и не вернуть, поэтому возвращаемое значение ошибки передается `check`. Обратите внимание: мы *не передаем* возвращаемое значение `http.ListenAndServe`. Дело в том, что `ListenAndServe` всегда возвращает ошибку. (Если ошибки нет, то `ListenAndServe` не возвращает управление.) Так как мы знаем, что значение ошибки никогда не будет равно `nil`, мы просто вызываем для него `log.Fatal`.

Создание каталога проекта и пробный запуск

Мы создадим для этого проекта несколько файлов, которые будут храниться в новом каталоге (необязательно хранить его в каталоге вашей рабочей области Go.) Сохраните приведенный код в новом каталоге в файле с именем *guestbook.go*.

Создайте каталог для хранения проекта и сохраните код в файле с именем guestbook.go.



Теперь попробуем запустить его. В терминале перейдите в каталог, в котором находится файл *guestbook.go*, и запустите его командой **`go run`**.

Перейдите в каталог, в котором был сохранен файл guestbook.go.

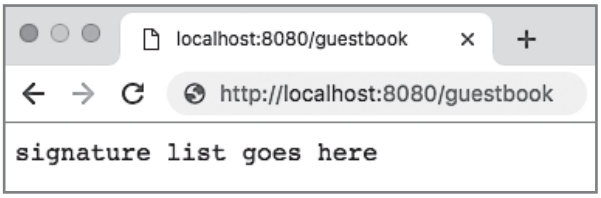
Запускаем приложение.

```
File  Edit  Window  Help
$ cd myproject
$ go run guestbook.go
```

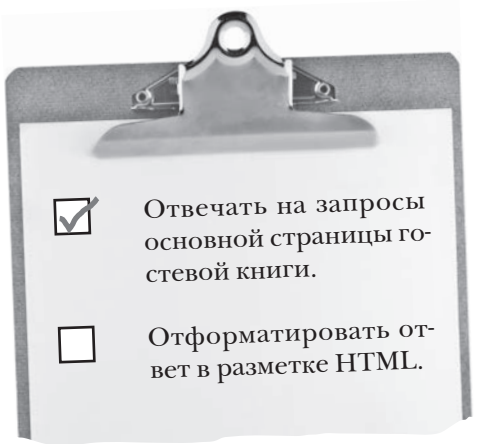
Откройте следующий URL-адрес в браузере:

http://localhost:8080/guestbook

URL-адрес выглядит так же, как и в предыдущих приложениях, но с суффиксом */guestbook*. Браузер выдает запрос к приложению, которое отвечает текстом-заполнителем:



Теперь приложение отвечает на запросы. Первый пункт выполнен!



Впрочем, пока запрос состоит из простого текста. На следующем шаге мы отформатируем ответ в разметке HTML.

Создание списка записей в HTML

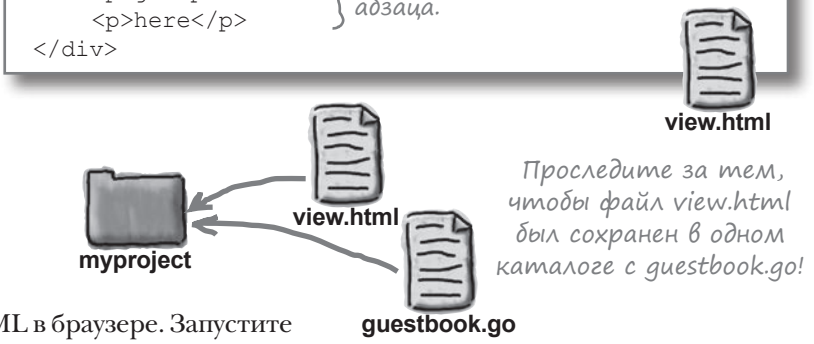
До настоящего момента мы отправляли браузеру фрагменты простого текста. Чтобы применить форматирование к странице, потребуется разметка HTML, которая использует теги для форматирования текста. Если вы еще не писали разметку HTML, не беспокойтесь — основные принципы работы с HTML будут изложены во время работы над приложением! Сохраните приведенную ниже разметку HTML в одном каталоге с файлом *guestbook.go*, в файле с именем *view.html*.

Основные элементы разметки HTML в этом файле:

- `<h1>`: Заголовок первого уровня. Обычно выводится крупным жирным шрифтом.
- `<div>`: Элемент раздела. Не виден на экране, но используется для логического деления страницы на части.
- `<p>`: Абзац текста. Мы будем рассматривать каждую запись как отдельный абзац.
- `<a>`: Якорный элемент (сокращение от «anchor»). Создает ссылку.

```
<h1>Guestbook</h1>
<div>
    X total signatures -
    <a href="/guestbook/new">Add Your Signature</a>
</div>

<div>
    <p>Signatures</p>
    <p>go</p>
    <p>here</p>
</div>
```

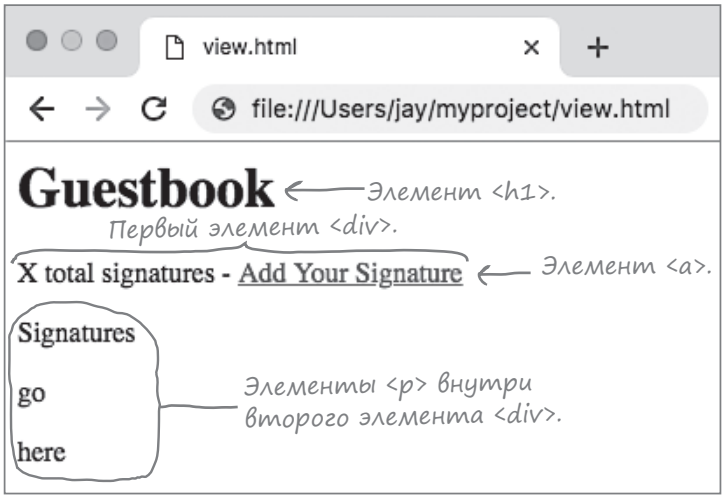
Заголовок первого уровня.
Раздел страницы.
Простой текст.
Ссылка.
Другой раздел страницы.
Элементы абзаца.



Попробуем просмотреть разметку HTML в браузере. Запустите свой любимый браузер, выберите в меню команду «Открыть файл...» и откройте только что созданный файл HTML.

Обратите внимание на соответствие элементов страницы с разметкой HTML. У каждого элемента имеется открывающий тег (`<h1>`, `<div>`, `<p>` и т. д.) с парным закрывающим тегом (`</h1>`, `</div>`, `</p>` и т. д.). Любой текст между открывающим и закрывающим тегом рассматривается как содержимое элемента на странице. Элементы также могут содержать другие элементы (как элементы `<div>` на этой странице).

При желании можно щелкнуть на ссылке, но сейчас она выдаст только ошибку «Страница не найдена». Прежде чем решать эту проблему, необходимо понять, как выдавать эту разметку в веб-приложении...



Как заставить приложение отвечать разметкой HTML

Наша разметка HTML работает, если загрузить ее напрямую в браузере из файла *view.html*, но она должна предоставляться веб-приложением. Обновим код *guestbook.go*, чтобы он отвечал созданной нами разметке HTML.

Go предоставляет пакет `html/template`, который загружает разметку HTML из файла *и* сможет вставить в нее записи гостевой книги за нас. Для начала просто загрузим содержимое *view.html* в исходном виде: вставка записей станет следующим шагом.

Измените директиву `import` и включите в нее пакет `html/template`. Все остальные изменения относятся к функции `viewHandler`. Мы вызовем функцию `template.ParseFiles` и передадим ей имя загружаемого файла: `"view.html"`. Функция использует содержимое *view.html* для создания значения `Template`. `ParseFiles` вернет указатель на значение `Template` и, возможно, значение ошибки, которое мы передадим функции `check`.

Чтобы получить вывод от значения `Template`, мы вызовем метод `Execute` с двумя аргументами. Значение `ResponseWriter` передается для записи вывода. Во втором аргументе передаются данные, которые необходимо вставить в шаблон, но поскольку пока таких данных еще нет, мы просто передадим `nil`.

```
// ...
import (
       "html/template"
       "log"
       "net/http"
)

func check(err error) {
       // Код пропущен...
}

func viewHandler(writer http.ResponseWriter, request *http.Request) {
       html, err := template.ParseFiles("view.html")
       check(err)
       err = html.Execute(writer, nil)
       check(err)
}
// Код пропущен...
```

Импортировать пакет «html/template».

Содержимое view.html используется для создания нового значения Template.

Сообщает об ошибках.

Содержимое шаблона записывается в ResponseWriter.

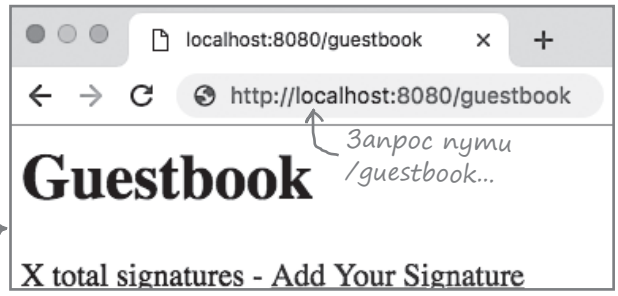
Сообщает об ошибках.

Чуть позже мы изучим пакет `html/template` более подробно, а пока просто посмотрим, как он работает. В терминале введите команду *guestbook.go*. (Убедитесь в том, что текущим является каталог проекта, иначе функция `ParseFiles` не найдет файл *view.html*.)

Введите в браузере следующий URL-адрес:

http://localhost:8080/guestbook

Вместо временного текста вы увидите разметку HTML из *view.html*.



Запрос пути /guestbook...

Приложение загружает содержимое файла view.html и выдает в качестве ответа.

Пакет «text/template»

- [x] Отвечать на запросы основной страницы гостевой книги.
- [x] Отформатировать ответ в разметке HTML.
- [ ] Заполнить страницу HTML текстами записей.

Наше приложение отвечает разметкой HTML. Вторая задача выполнена!

Пока в приложении выводится временный список записей, который был жестко «вписан» в программу. Наша следующая задача — использовать пакет `html/template` для вставки списка записей в HTML (чтобы вид страницы изменялся при изменении списка).

Пакет `html/template` базируется на пакете `text/template`. Принципы работы с двумя пакетами почти не отличаются, но `html/template` содержит дополнительные средства безопасности, необходимые для работы с HTML. Для начала посмотрим, как использовать пакет `text/template`, а затем применим все наши знания к пакету `html/template`.

Следующая программа разбирает и выводит строку шаблона при помощи пакета `text/template`. Результат выводится в терминале, поэтому для тестирования браузер не понадобится.

В функции `main` вызывается функция `New` пакета `text/template`, которая возвращает указатель на новое значение `Template`. Затем программа вызывает метод `Parse` для `Template` и передает ему строку `"Here's my template!\n"`. Функция `Parse` использует свой строковый аргумент как текст шаблона — в отличие от функции `ParseFiles`, которая загружает текст шаблона из файлов. `Parse` возвращает шаблон и значение ошибки. Шаблон сохраняется в переменной `tmpl`, а ошибка передается функции `check` (идентичной одноименной функции из *guestbook.go*) для вывода информации о любых ошибках, отличных от `nil`.

Затем мы вызываем метод `Execute` для значения `Template` из переменной `tmpl`, как это делалось в *guestbook.go*. Однако вместо `http.ResponseWriter` мы передаем `os.Stdout` как приемник для записи вывода. В результате строка шаблона `"Here's my template!\n"` выводится как результат запуска программы.

```
package main

import (
        "log"
        "os"
        "text/template"
)

func check(err error) {
        if err != nil {
                log.Fatal(err)
        }
}

func main() {
        text := "Here's my template!\n"
        tmpl, err := template.New("test").Parse(text)
        check(err)
        err = tmpl.Execute(os.Stdout, nil)
        check(err)
}
```

Этот пакет понадобится для обращения к os.Stdout.

Импортируем text/template вместо html/template.

Идентично предыдущей функции «check».

Текст шаблона.

Создает новое значение Template на базе текста.

Записывает текст шаблона.

Вместо ответа HTTP выводит шаблон в терминале.

```
Here's my template!
```

Использование интерфейса io.Writer с методом шаблона Execute

Что это за значение — os.Stdout? И как может быть, что методу Execute значения Template могут передаваться как значение http.ResponseWriter, так и os.Stdout?

```
func viewHandler(writer http.ResponseWriter, request *http.Request) {
       html, err := template.ParseFiles("view.html")
       check(err)
       err = html.Execute(writer, nil)
       // ...
```

Содержимое шаблона записывается в ResponseWriter.

```
text := "Here's my template!\n"
tmpl, err := template.New("test").Parse(text)
check(err)
err = tmpl.Execute(os.Stdout, nil)
check(err)
```

Содержимое шаблона записывается в терминал.

Значение os.Stdout является частью пакета os (Stdout означает «стандартный вывод»). Оно работает как файл, но любые данные, записанные в него, выводятся в терминале, а не сохраняются на диске. (Такие функции, как fmt.Println, fmt.Printf и т. д., незаметно для вас записывают данные в os.Stdout.)

Как http.ResponseWriter и os.Stdout могут одновременно быть допустимыми аргументами для Template.Execute? Обратимся к документации и посмотрим...

```
File Edit Window Help
$ go doc text/template Template.Execute
func (t *Template) Execute(wr io.Writer, data interface{}) error
    Execute applies a parsed template to the specified data object, and writes
    the output to wr. If an error occurs executing the template or writing its
    ...
```

Хм, из документации следует, что первым аргументом Execute должно быть значение io.Writer. Что это такое? Обратимся к документации пакета io:

```
File Edit Window Help
$ go doc io Writer
type Writer interface {
        Write(p []byte) (n int, err error)
}
    Writer is the interface that wraps the basic Write method.
    ...
```

Похоже, io.Writer — это интерфейс! Он поддерживается любым типом с методом Write, который получает сегмент значений byte и возвращает значение int с количеством записанных байтов и значением ошибки.

ResponseWriter и os.Stdout поддерживают io.Writer

Вы уже видели, что значения `http.ResponseWriter` содержат метод `Write`. Этот метод использовался в нескольких предыдущих примерах:

```
func viewHandler(writer http.ResponseWriter, request *http.Request) {
       placeholder := []byte("signature list goes here")
       _, err := writer.Write(placeholder)
       check(err)
}
```

Строка преобразуется в сегмент байтов...

...и добавляется в ответ методом Write.

Оказывается, значение `os.Stdout` тоже содержит метод `Write`! Если передать ему сегмент значений `byte`, эти данные будут записаны в терминал:

Данные записываются в терминал.

```
func main() {
       _, err := os.Stdout.Write([]byte("hello"))
       check(err)
}
```

```
hello
```

Это означает, что значения `http.ResponseWriter` и `os.Stdout` поддерживают интерфейс `io.Writer` и могут передавать методу `Execute` значения `Template`. Вызов `Execute` записывает шаблон, вызывая метод `Write` для переданного значения.

Если передается значение `http.ResponseWriter`, это означает, что шаблон будет записываться в ответ HTTP. А при передаче `os.Stdout` шаблон будет записываться для вывода в терминал:

```
func main() {
       tmpl, err := template.New("test").Parse("Here's my template!\n")
       check(err)
       err = tmpl.Execute(os.Stdout, nil)
       check(err)
}
```

Выводит текст шаблона.

Шаблон отправляется в терминал.

```
Here's my template!
```

Вставка данных в шаблоны при помощи действий

Во втором параметре метода `Execute` значения `Template` передаются данные для вставки в шаблон. Его типом является пустой интерфейс, это означает, что передавать можно значение любого типа.

```
File Edit Window Help
$ go doc text/template Template.Execute
func (t *Template) Execute(wr io.Writer, data interface{}) error
    Execute applies a parsed template to the specified data object, and writes
    the output to wr. If an error occurs executing the template or writing its
    ...
```

До сих пор в наших шаблонах не было мест для вставки данных, поэтому вместо значения данных мы просто передаем `nil`:

Шаблон не содержит мест для вставки данных.

```
func main() {
        tmpl, err := template.New("test").Parse("Here's my template!\n")
        check(err)
        err = tmpl.Execute(os.Stdout, nil)
        check(err)
}
```

Просто передаем «nil» вместо вставляемых данных.

```
Here's my template!
```

Чтобы вставить данные в шаблон, следует добавить **действия** (actions) в текст шаблона. Действия обозначаются двойными фигурными скобками `{{ }}`. В двойных фигурных скобках указываются данные, которые необходимо вставить, или операция, которая должна быть выполнена шаблоном. Каждый раз, когда в шаблоне обнаруживается действие, происходит обработка его содержимого, а результат вставляется в текст шаблона на место действия.

Внутри действия вы можете обращаться к значению данных, переданных методу `Execute`, в форме "." (точка).

В следующем фрагменте создается шаблон с одним действием. Затем для шаблона несколько раз вызывается метод `Execute` с разными значениями данных. Перед тем как записывать результат в `os.Stdout`, `Execute` заменяет действие значением данных.

Действие, которое вставляет значение данных.

```
func main() {
        templateText := "Template start\nAction: {{.}}\nTemplate end\n"
        tmpl, err := template.New("test").Parse(templateText)
        check(err)
        err = tmpl.Execute(os.Stdout, "ABC")
        check(err)
        err = tmpl.Execute(os.Stdout, 42)
        check(err)
        err = tmpl.Execute(os.Stdout, true)
        check(err)
}
```

Один шаблон выполняется с разными значениями данных.

Значения вставляются в шаблон на место действия.

```
Template start
Action: ABC
Template end
Template start
Action: 42
Template end
Template start
Action: true
Template end
```

Вставка данных в шаблоны при помощи действий (продолжение)

С действиями шаблонов можно сделать еще много чего интересного. Давайте создадим функцию `executeTemplate`, которая упростит эксперименты с шаблонами. Функция будет получать строку шаблона, которая передастся `Parse` для создания нового шаблона, и значение данных, которое будет передаваться методу `Execute` этого шаблона. Как и прежде, каждый шаблон записывается в `os.Stdout`.

Создаем шаблон на основе строки.

Это значение данных передается методу Execute шаблона.

```
func executeTemplate(text string, data interface{}) {
       tmpl, err := template.New("test").Parse(text)
       check(err)
       err = tmpl.Execute(os.Stdout, data)
       check(err)
}
```

Переданный текст разбирается для создания шаблона.

Переданное значение данных используется в действиях шаблонов.

Как упоминалось ранее, символ "." (точка) используется для обозначения текущего значения данных, с которым работает шаблон. И хотя значение "." может изменяться в шаблоне в зависимости от контекста, изначально оно относится к значению, переданному `Execute`.

```
func main() {
       executeTemplate("Dot is: {{.}}!\n", "ABC")
       executeTemplate("Dot is: {{.}}!\n", 123.5)
}
```

```
Dot is: ABC!
Dot is: 123.5!
```

Необязательные действия «if» в шаблонах

Часть шаблона между действием `{{if}}` и соответствующим маркером `{{end}}` включается только в том случае, если условие истинно. В следующем примере один текст шаблона выполняется дважды: в одном случае значение "." истинно, а в другом ложно. Благодаря действию `{{if}}` текст «Dot is true!» включается в результат только в том случае, если значение "." истинно.

Эта часть шаблона включается в вывод только в том случае, если значение "." истинно.

```
executeTemplate("start {{if .}}Dot is true!{{end}} finish\n", true)
executeTemplate("start {{if .}}Dot is true!{{end}} finish\n", false)
```

```
start Dot is true! finish
start  finish
```

Повторение частей шаблонов в действиях «range»

Часть шаблона между действием `{{range}}` и соответствующим маркером `{{end}}` повторяется для каждого значения в массиве, сегменте, ассоциативном массиве или канале. Любые действия в этом разделе также будут повторяться.

Внутри повторяемой части значению "." присваивается текущий элемент коллекции, что позволяет включить в результат каждый элемент коллекции или каким-то образом обработать его.

Следующий шаблон включает действие `{{range}}` для вывода всех элементов в массиве. До и после цикла значением "." будет сам сегмент. Но *внутри* цикла "." обозначает текущий элемент сегмента. Этот факт отражен в выводе.

Эта часть шаблона повторяется для каждого элемента в сегменте.

```
templateText := "Before loop: {{.}}\n{{range .}}In loop: {{.}}\n{{end}}After loop: {{.}}\n"
```

До цикла "." содержит весь сегмент.

В цикле "." обозначает текущее значение из сегмента.

После цикла "." снова содержит весь сегмент.

```
executeTemplate(templateText, []string{"do", "re", "mi"})
```

Данные передаются в форме сегмента.

```
Before loop: [do re mi]
In loop: do
In loop: re
In loop: mi
After loop: [do re mi]
```

Этот шаблон работает с сегментом значений `float64`, которые будут отображаться как список цен.

Эта часть шаблона повторяется для каждого элемента в сегменте.

```
templateText = "Prices:\n{{range .}}${{.}}\n{{end}}"
executeTemplate(templateText, []float64{1.25, 0.99, 27})
```

```
Prices:
$1.25
$0.99
$27
```

Если действию `{{range}}` передается пустое значение или `nil`, цикл не будет выполняться вообще:

```
templateText = "Prices:\n{{range .}}${{.}}\n{{end}}"
executeTemplate(templateText, []float64{}) ← Передается пустой сегмент.
executeTemplate(templateText, nil) ← Передается nil.
```

```
Prices: ← Часть внутри цикла не включается.
Prices: ← Часть внутри цикла не включается.
```

Вставка полей структуры в шаблон

Впрочем, простые типы обычно не могут вместить разнообразную информацию, необходимую для заполнения шаблона. На практике при выполнении шаблона чаще используются структурные типы.

Если значение "." представляет собой структуру, то действие из точки, за которой следует имя поля, вставляет в шаблон значение этого поля. В следующем примере мы определяем тип структуры `Part`, а затем создаем шаблон для полей `Name` и `Count` значения `Part`:

Значение поля Name структуры Part вставляется в шаблон.

Значение поля Count структуры Part вставляется в шаблон.

```
type Part struct {
       Name  string
       Count int
}
templateText := "Name: {{.Name}}\nCount: {{.Count}}\n"
executeTemplate(templateText, Part{Name: "Fuses", Count: 5})
executeTemplate(templateText, Part{Name: "Cables", Count: 2})
```

```
Name: Fuses
Count: 5
Name: Cables
Count: 2
```

Наконец, ниже объявлен тип структуры `Subscriber` и шаблон для его вывода. Поле `Name` выводится в любом случае, но действие `{{if}}` используется для вывода поля `Rate` только в том случае, если поле `Active` содержит `true`.

```
type Subscriber struct {
       Name   string
       Rate   float64
       Active bool
}
templateText = "Name: {{.Name}}\n{{if .Active}}Rate: ${{.Rate}}\n{{end}}"
subscriber := Subscriber{Name: "Aman Singh", Rate: 4.99, Active: true}
executeTemplate(templateText, subscriber)
subscriber = Subscriber{Name: "Joy Carr", Rate: 5.99, Active: false}
executeTemplate(templateText, subscriber)
```

Эта часть шаблона будет выводиться только в том случае, если значение поля Active структуры Subscriber равно true.

Раздел Rate пропускается для неактивных подписчиков.

```
Name: Aman Singh
Rate: $4.99
Name: Joy Carr
```

Шаблоны предоставляют массу других возможностей, и у нас нет времени рассмотреть их все. Если вам захочется узнать больше, обращайтесь к документации пакета `text/template`:

```
File  Edit  Window  Help
$ go doc text/template
package template // import "text/template"

Package template implements data-driven templates for generating textual
output.

To generate HTML output, see package html/template, which has the same
interface as this package but automatically secures HTML output against
certain attacks.
...
```

Чтение сегмента записей из файла

Временные «записи» гостевой книги сохраняются в файле в каталоге проекта.

Теперь вы знаете, как вставить данные в шаблон, и почти все готово к вставке подписей в страницу гостевой книги. Но сначала нам понадобятся записи, которые можно было бы вставить.

В каталоге проекта сохраните несколько строк в текстовом файле с именем *signatures.txt*. Пока они будут играть роль «записей».

Теперь эти записи нужно загрузить в приложение. Добавьте в файл *guestbook.go* новую функцию `getStrings`. Эта функция работает практически так же, как функция `datafile.GetStrings`, написанная в главе 7: она читает файл, присоединяет каждую строку к сегменту, а затем возвращает сегмент.

Впрочем, есть и пара отличий. Во-первых, новая функция `getStrings` использует функцию `check` для проверки ошибок (а не возвращает их).

Во-вторых, если файл не существует, `getStrings` просто возвращает `nil` вместо сегмента строк (а не сообщает об ошибке). Для этого любое значение ошибки, полученное от `os.Open`, передается функции `os.IsNotExist`, которая возвращает `true`, если ошибка указывает на то, что файл не существует.

```
import (
        "bufio"
        "fmt"
        "html/template"
        "log"
        "net/http"
        "os"
)

// ...

func getStrings(fileName string) []string {
        var lines []string
        file, err := os.Open(fileName)
        if os.IsNotExist(err) {
                return nil
        }
        check(err)
        defer file.Close()
        scanner := bufio.NewScanner(file)
        for scanner.Scan() {
                lines = append(lines, scanner.Text())
        }
        check(scanner.Err())
        return lines
}

// ...
```

Используется функцией getStrings.

Вскоре будет использоваться внутри viewHandler.

Используется функцией getStrings.

Открывает файл.

Если возвращается ошибка, указывающая на то, что файл не существует...

...вернуть nil вместо сегмента строк.

Для любой другой разновидности ошибки сообщить о ней и завершить работу.

После выхода из функции позаботиться о том, чтобы файл был закрыт.

Сообщить о любых ошибках обработки файла и завершить работу.

Чтение сегмента записей из файла (продолжение)

Мы также внесем небольшое изменение в функцию `viewHandler`: в нее будет добавлен вызов `getStrings` и временный вызов `fmt.Printf`, который покажет, что было загружено из файла.

```
func viewHandler(writer http.ResponseWriter, request *http.Request) {
        signatures := getStrings("signatures.txt")   <-- Добавляем вызов getStrings.
        fmt.Printf("%#v\n", signatures)               <-- Выводим загруженные записи.
        html, err := template.ParseFiles("view.html")
        check(err)
        err = html.Execute(writer, nil)
        check(err)
}
```

Опробуем функцию `getStrings`. В терминале перейдите в каталог проекта и запустите файл *guestbook.go*. Откройте в браузере страницу *http://localhost:8080/guestbook*, чтобы была вызвана функция `viewHandler`. Она вызывает функцию `getStrings`, которая загружает и возвращает сегмент с содержимым *signatures.txt*.

При загрузке страницы выводится сегмент с записями гостевой книги.

```
File  Edit  Window  Help
$ cd myproject
$ go run guestbook.go
[]string{"First signature", "Second signature", "Third signature"}
```

Часто Задаваемые Вопросы

В: Что произойдет, если файл *signatures.txt* не существует и `getStrings` вернет `nil`? Не создаст ли это проблем с рендерингом шаблона?

О: Беспокоиться не о чем. Как и функция `append`, другие функции Go обычно интерпретируют `nil`-сегменты и карты так, как если бы они были пустыми. Например, функция `len` просто возвращает `0` при передаче `nil`-сегмента:

```
var mySlice []string          <-- Так как сегмент не присвоен, значение mySlice будет равно nil.
fmt.Printf("%#v, %d\n", mySlice, len(mySlice))
```

```
[]string(nil), 0
```

Но «len» возвращает 0, как будто была передана пустая строка!

Действия шаблонов тоже рассматривают `nil`-сегменты и ассоциативные массивы так, как если бы они были пустыми. Например, как вы узнали, действие `{{range}}` просто пропускает свое содержимое при передаче значения `nil`. Итак, возвращение функцией `getStrings` `nil` вместо сегмента не создаст проблем: если ни одна запись не будет загружена из файла, шаблон просто пропустит вывод записей.

Структура для хранения записей и количества записей

Теперь мы можем просто передать сегмент записей методу `Execute` нашего шаблона HTML, и записи будут вставлены в шаблон. Но мы также хотим, чтобы на главной странице гостевой книги выводилось *количество* полученных записей (наряду с самими записями).

Однако методу `Execute` может передаваться только одно значение. Следовательно, нужно будет создать тип структуры, в котором будет храниться как общее количество записей, так и сегмент, содержащий сами записи.

Добавьте в начало файла *guestbook.go* объявление нового типа структуры `Guestbook`. Структура должна состоять из двух полей: в поле `SignatureCount` хранится количество записей, а в поле `Signatures` — сегмент с самими записями.

```
type Guestbook struct {
        SignatureCount int
        Signatures     []string
}
```

Новый тип определяется в начале guestbook.go.

Теперь необходимо обновить код `viewHandler`, чтобы в нем создавалась новая структура `Guestbook` и передавалась шаблону. Прежде всего вызов `fmt.Printf` для вывода содержимого сегмента записей нам больше не понадобится, поэтому ее можно удалить. (Также следует удалить `"fmt"` из директивы `import`.) Затем создайте новое значение `Guestbook`. Присвойте полю `SignatureCount` длину сегмента записей, а полю `Signatures` — сам сегмент. Наконец, данные необходимо передать шаблону. Замените значение данных, передаваемое во втором аргументе метода `Execute`, с `nil` на новое значение `Guestbook`.

```
func viewHandler(writer http.ResponseWriter, request *http.Request) {
        signatures := getStrings("signatures.txt")
        html, err := template.ParseFiles("view.html")
        check(err)
        guestbook := Guestbook{
                SignatureCount: len(signatures),
                Signatures:     signatures,
        }
        err = html.Execute(writer, guestbook)
        check(err)
}
```

Создаем новую структуру Guestbook.

В поле SignatureCount сохраняется длина сегмента записей.

В поле Signatures сохраняется сам сегмент записей.

Структура передается методу Execute значения Template.

Обновление шаблона для включения записей

Обновим текст шаблона в *view.html* для вывода списка записей.

Структура `Guestbook` передается методу `Execute` шаблона, так что в шаблоне "." представляет эту структуру. В первом элементе `div` замените временное обозначение `X` в `"X total signatures"` действием, которое вставляет поле `SignatureCount` структуры `Guestbook`: `{{.SignatureCount}}`.

Второй элемент `div` содержит серию элементов `p` (абзацев), по одному для каждой записи. Действие `range` используется для перебора всех записей в сегменте `Signatures`: `{{range .Signatures}}`. (Не забудьте поставить соответствующий маркер `{{end}}` перед концом элемента `div`.) В действии `range` включите элемент HTML `p` с действием, которое выводит ".": `<p>{{.}}</p>`. Напомним, что "." последовательно присваивается каждый элемент сегмента, так что для каждой записи в сегменте будет выводиться элемент `p`, содержимым которого является текст записи.

```
<h1>Guestbook</h1>

<div>
       {{.SignatureCount}} total signatures -
       <a href="/guestbook/new">Add Your Signature</a>
</div>

<div>
       {{range .Signatures}}
              <p>{{.}}</p>
       {{end}}
</div>
```

Вставляется количество записей из структуры Guestbook.

Повторить для каждой строки в сегменте Signatures.

Вставляется элемент <p>, содержащий текущую запись.

Наконец-то мы можем протестировать шаблон с включенными данными! Перезапустите приложение *guestbook.go* и снова откройте *http://localhost:8080/guestbook* в браузере. В ответ должен быть включен ваш шаблон. Общее количество записей выводится наверху, а каждая запись выводится в отдельном элементе <p>!

Часто Задаваемые Вопросы

В: Вы говорили о пакете `html/template`, который содержит некоторые «средства безопасности». О чем речь?

О: Пакет `text/template` вставляет записи в шаблон в неизменном виде независимо от его содержания. Но это означает, что посетители могут ввести в поле записи разметку HTML, и эта разметка будет рассматриваться как часть разметки HTML-страницы.

Вы можете попробовать сделать это самостоятельно. В файле *guestbook.go* замените импортируемый пакет `html/template` на `text/template`. (Изменять другой код не придется, потому что имена всех функций в двух пакетах идентичны.) Затем добавьте следующую новую строку в файл *signatures.txt*:

```
<script>alert("hi!");</script>
```

Это тег HTML, содержащий код JavaScript. Если вы попробуете запустить приложение и перезагрузите страницу записей, появится раздражающее всплывающее окно, все потому, что пакет `text/template` включает разметку в страницу в неизменном виде.

Теперь вернитесь к файлу *guestbook.go*, снова исправьте директиву для импорта `html/template` и перезапустите приложение. Перезагрузите страницу — вместо всплывающего окна вы увидите на странице текст, который выглядит как приведенный выше тег.

Дело в том, что пакет `html/template` автоматически «экранирует» HTML, заменяя символы, из-за которых вставленный текст интерпретируется как HTML, специальными кодами, которые отображаются в тексте страницы (где они безвредны). Вот какой текст будет на самом деле вставлен в ответ:

```
&lt;script&gt;alert(&#34;hi!&#34;);&lt;/script&gt;
```

Вставка таких тегов — всего лишь один из многих способов, которыми злоумышленники могут вставлять вредоносный код в ваши веб-страницы. Пакет `html/template` позволяет легко защититься от этих и многих других угроз!

Упражнение

Приведенная ниже программа загружает шаблон HTML из файла и выводит его в терминале. Заполните пропуски в файле *bill.html*, чтобы программа запускалась и выводила показанный результат.

```
type Invoice struct {
    Name     string
    Paid     bool
    Charges  []float64
    Total    float64
}

func main() {
    html, err := template.ParseFiles("bill.html")
    check(err)
    bill := Invoice{
       Name:    "Mary Gibbs",
       Paid:    true,
       Charges: []float64{23.19, 1.13, 42.79},
       Total:   67.11,
    }
    err = html.Execute(os.Stdout, bill)
    check(err)
}
```

bill.go

```
<h1>Invoice</h1>

<p>Name: _____</p>

{{if _____}}
<p>Paid - Thank you!</p>
_____

<h1>Fees</h1>

{{range .Charges}}
<p>$_____</p>
{{end}}

<p>Total: $_____</p>
```

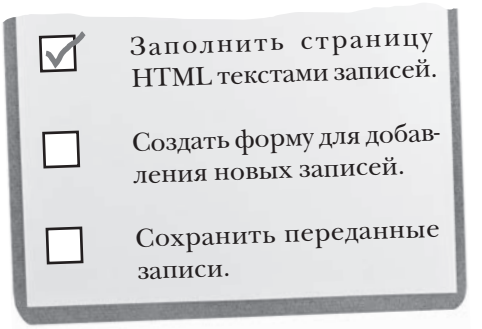
bill.html

Результат.

```
<h1>Invoice</h1>

<p>Name: Mary Gibbs</p>

<p>Paid - Thank you!</p>

<h1>Fees</h1>

<p>$23.19</p>
<p>$1.13</p>
<p>$42.79</p>

<p>Total: $67.11</p>
```

Ответ на с. 512.

Ввод данных в формах HTML



Еще одна задача выполнена. Мы постепенно продвигаемся к финишу: осталось всего две задачи!

На следующем шаге нужно дать посетителям возможность вводить новые записи. Для этого необходимо создать *форму* HTML, в которой вводится текст записи. Форма обычно содержит одно или несколько полей, где пользователь вводит данные, и кнопку отправки данных для передачи данных серверу.

В каталоге проекта создайте файл *new.html*, содержащий приведенную ниже разметку HTML. В ней есть теги, которые ранее еще не встречались:

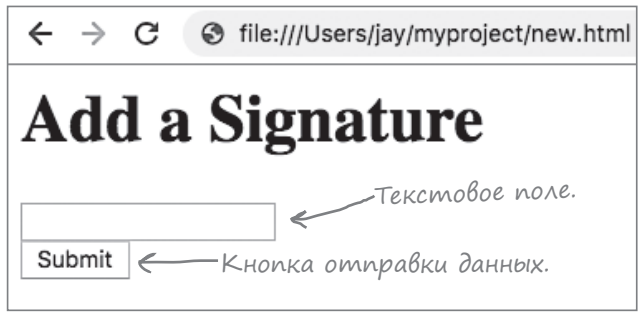
- **`<form>`**: этот элемент содержит все остальные компоненты формы.
- **`<input>` с атрибутом `type`, равным `"text"`**: текстовое поле, в котором пользователь вводит строку. Его атрибут `name` используется для пометки значения поля в данных, передаваемых серверу (что-то вроде ключа в карте).
- **`<input>` с атрибутом `type`, равным `"submit"`**: создает кнопку, нажатием которой пользователь отправляет данные формы.

```
<h1>Add a Signature</h1>

<form>
  <div><input type="text" name="signature"></div>
  <div><input type="submit"></div>
</form>
```

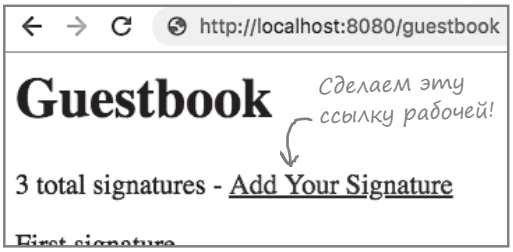
Текстовое поле «signature».

Кнопка для отправки данных формы.

Если вы загрузите эту разметку HTML в браузере, то результат будет выглядеть примерно так:



Включение формы HTML в ответ



В файле *view.html* уже присутствует ссылка «Add Your Signature», которая указывает на путь */guestbook/new*. Щелчок на этой ссылке открывает новый путь на том же сервере; происходит то же самое, как если бы вы ввели следующий URL-адрес:

http://localhost:8080/guestbook/new

Но если вы попытаетесь открыть этот путь сейчас, то получите лишь ошибку «404 страница не найдена». Необходимо настроить приложение так, чтобы при щелчке по ссылке оно выдавало ответ с формой из *new.html*.

В файле *guestbook.go* добавьте функцию `newHandler`. Эта функция очень похожа на ранние версии нашей функции `viewHandler`. Как и `viewHandler`, `newHandler` получает в параметрах значение `http.ResponseWriter` и указатель на `http.Request`. Функция должна вызвать `template.ParseFiles` для файла *new.html*. После этого она вызывает `Execute` для полученного шаблона, чтобы содержимое *new.html* было записано в ответ HTTP. Пока никакие данные в этот шаблон не вставляются, поэтому вместо значения данных при этом вызове `Execute` передается `nil`.

Затем нужно сделать так, чтобы функция `newHandler` вызывалась при щелчке по ссылке «Add Your Signature». Добавьте в функцию `main` еще один вызов `http.HandleFunc` и укажите `newHandler` как функцию-обработчик для запросов к пути */guestbook/new*.

```
// Код пропущен...

func newHandler(writer http.ResponseWriter, request *http.Request) {
        html, err := template.ParseFiles("new.html")
        check(err)
        err = html.Execute(writer, nil)
        check(err)
}

// ...

func main() {
        http.HandleFunc("/guestbook", viewHandler)
        http.HandleFunc("/guestbook/new", newHandler)
        err := http.ListenAndServe("localhost:8080", nil)
        log.Fatal(err)
}
```

Добавляем другую функцию-обработчик с такими же параметрами, как у viewHandler.

Загружает содержимое new.html как текст шаблона.

Записываем шаблон в ответ (вставлять данные пока не нужно).

Назначаем функцию newHandler для обработки запросов с путем «/guestbook/new».

Если сохранить приведенный код и перезапустить *guestbook.go*, а затем щелкнуть по ссылке «Add Your Signature», произойдет переход по пути */guestbook/new*. Будет вызвана функция `newHandler`, которая загрузит форму HTML из файла *new.html* и включит ее в ответ.

Запросы на отправку данных формы

- [x] Создать форму для добавления новых записей.
- [ ] Сохранять переданные записи.

Еще одна задача осталась позади. Остается последний пункт!

Когда пользователь посещает путь */guestbook/new*— либо вводя его напрямую, либо щелкая по ссылке, открывается форма для ввода записи гостевой книги. Но если заполнить форму и щелкнуть по кнопке отправки данных, ничего полезного не произойдет.

Браузер выдает еще один запрос к пути */guestbook/new*. Содержимое поля формы `"signature"` добавляется в уродливый параметр в конце URL-адреса. А поскольку функция `newHandler` не знает, как сделать что-то осмысленное с данными формы, эти данные попросту игнорируются.

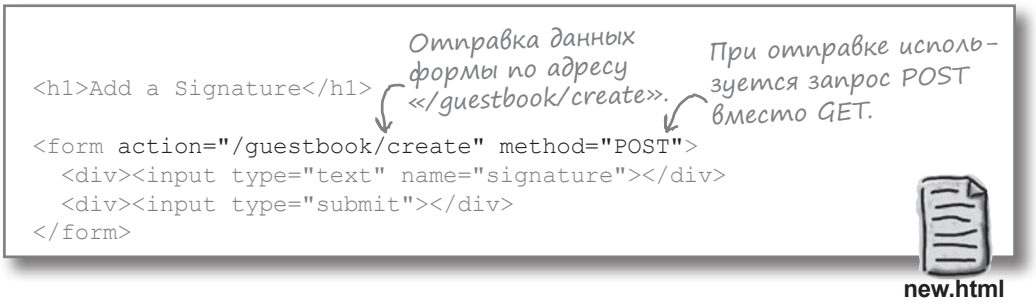
Наше приложение отвечает на запросы вывода формы, но форма пока не отправляет данные приложению. Эту проблему необходимо решить перед тем, как мы сможем сохранять записи посетителей.

Путь и метод HTTP для отправки данных формы

Отправка данных формы на самом деле требует *двух* запросов к серверу: для *получения* формы и для *отправки* данных, введенных пользователем, обратно серверу. Обновим разметку HTML формы и укажем, где и как должен отправляться этот второй запрос.

Отредактируйте файл *new.html* и добавьте в элемент `form` разметки HTML два новых атрибута. Первый атрибут, `action`, задает путь, который должен использоваться для запроса на отправку. Мы не будем использовать путь по умолчанию */guestbook/new*, а укажем новый путь: */guestbook/create*.

Также потребуется второй атрибут с именем `method`, который должен иметь значение `"POST"`.



Атрибут `method` заслуживает пояснений. В HTTP определяются *методы*, которые могут использоваться запросом. Не путайте их с методами значений Go, хотя по смыслу они чем-то похожи. Наиболее распространенными являются методы GET и POST:

- **GET**: используется, когда браузер должен *получить* что-то от сервера — обычно из-за того, что вы ввели URL-адрес или щелкнули по ссылке. Это может быть страница HTML, изображение или другой ресурс.
- **POST**: используется, когда браузеру нужно *добавить* данные на сервер — обычно из-за отправки формы с новыми данными.

Мы добавляем на сервер новые данные: новую запись гостевой книги. Похоже, данные должны отправляться запросом POST.

Однако по умолчанию формы отправляются запросами GET. Именно по этой причине в элемент `form` нужно было добавить атрибут `method` со значением `"POST"`.

Если теперь перезагрузить страницу */guestbook/new* и заново отправить форму, запрос вместо этого будет использовать путь */guestbook/create*. Возникнет ошибка «404 страница не найдена», но это объясняется тем, что мы еще не назначили обработчик для пути */guestbook/create*.

Также мы видим, что данные формы уже не добавляются в конец URL-адреса. Это объясняется тем, что форма отправляется запросом POST.

Перезагрузка и повторная отправка данных формы...

Обработчика для этого пути еще нет, но это нормально.

Некрасивый параметр в конце URL-адреса исчез.

Получение значений полей формы из запроса

Теперь вместо отправки данных форм при помощи запроса POST данные формы встраиваются в сам запрос (вместо присоединения к пути запроса в параметре).

Решим проблему с ошибкой «404 страница не найдена», которая появляется при отправке формы для пути */guestbook/create*. Когда проблема будет решена, вы также увидите, как обращаться к данным форм из запроса POST.

Как обычно, для этого будет добавлена функция-обработчик запроса. В функции `main` из файла *guestbook.go* вызовите `http.HandleFunc` и назначьте запросы с путем `"/guestbook/create"` новой функции `createHandler`.

Затем добавьте определение самой функции `createHandler`. Функция должна получать `http.ResponseWriter` и указатель на `http.Request`, как и все остальные функции-обработчики.

Впрочем, в отличие от других функций-обработчиков, функция `createHandler` предназначена для работы с данными формы. К этим данным можно обращаться через указатель на `http.Request`, передаваемый функции-обработчику. (Все верно: ранее мы игнорировали значение `http.Request`, а сейчас наконец-то воспользуемся им!)

Для начала посмотрим, какие данные содержит запрос. Вызовите метод `FormValue` для `http.Request` и передайте ему строку `"signature"`. Он вернет строку со значением поля `"signature"` формы. Сохраните его в переменной с именем `signature`.

Запишем значение в ответ, чтобы его можно было просмотреть в браузере. Вызовите метод `Write` для `http.ResponseWriter` и передайте ему переменную `signature` (но сначала, конечно, значение нужно преобразовать в сегмент байтов). Как обычно, `Write` вернет количество записанных байтов и значение ошибки. Мы проигнорируем количество байтов, присвоив его пустому идентификатору `_`, и вызовем `check` для значения ошибки.

Определяет другую функцию-обработчик запроса с аналогичными параметрами.

Получает значение поля «signature» формы.

Записывает значение поля в ответ.

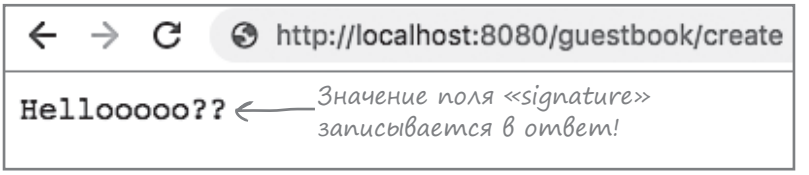
Вызывает createHandler для запросов с путем «/guestbook/create».

```
func createHandler(writer http.ResponseWriter, request *http.Request) {
        signature := request.FormValue("signature")
        _, err := writer.Write([]byte(signature))
        check(err)
}

func main() {
        http.HandleFunc("/guestbook", viewHandler)
        http.HandleFunc("/guestbook/new", newHandler)
        http.HandleFunc("/guestbook/create", createHandler)
        err := http.ListenAndServe("localhost:8080", nil)
        log.Fatal(err)
}
```

Получение значений полей формы из запроса (продолжение)

Посмотрим, доходят ли отправленные данные до функции `createHandler`. Перезапустите *guestbook.go*, откройте страницу */guestbook/new* и снова отправьте форму.

Происходит переход по пути */guestbook/create*, и вместо ошибки «404 страница не найдена» приложение ответит значением, введенным в поле `"signature"`!



При желании вы можете щелкнуть по кнопке **Назад** в браузере, чтобы вернуться к странице */guestbook/new* и опробовать другие записи. Любой введенный вами текст будет воспроизводиться в браузере.

Создание обработчика для отправки форм HTML — важное достижение. Мы постепенно приближаемся к конечной цели!

Сохранение данных формы

Наша функция `createHandler` получает запрос с данными формы и извлекает из них текст записи в гостевой книге. После этого остается добавить запись в файл *signatures.txt*. Мы сделаем это внутри самой функции `createHandler`.

Для начала избавимся от вызова метода `Write` для `ResponseWriter`; он был нужен только для того, чтобы проверять доступность поля формы.

Теперь добавьте код, приведенный ниже. Функция `os.OpenFile` вызывается несколько необычным образом, и эти особенности не связаны напрямую с программированием веб-приложений, поэтому мы не станем полностью описывать его. (За дополнительной информацией обращайтесь к приложению А.) А пока достаточно знать, что этот код выполняет три основные операции:

1. Открывает файл *signatures.txt* (и создает его, если файл не существует).
2. Добавляет строку текста в конец файла.
3. Закрывает файл.

```
import (
    // ...
    "fmt"
    // ...
)

// Code omitted...

func createHandler(writer http.ResponseWriter, request *http.Request) {
    signature := request.FormValue("signature")
    options := os.O_WRONLY | os.O_APPEND | os.O_CREATE
    file, err := os.OpenFile("signatures.txt", options, os.FileMode(0600))
    check(err)
    _, err = fmt.Fprintln(file, signature)
    check(err)
    err = file.Close()
    check(err)
}
```

Снова импортирует пакет «fmt».

(За полным описанием os.OpenFile обращайтесь к приложению А.)

Параметры открытия файла.

Открывает файл.

Записывает текст в новой строке файла.

Закрывает файл.

Функция `fmt.Fprintln` добавляет строку текста в файл. В аргументах она получает файл, в который должны записываться данные, и записываемую строку (преобразовывать ее в `[]byte` не нужно). По аналогии с методами `Write`, представленными ранее в этой главе, `Fprintln` возвращает количество байтов, успешно записанных в файл (пока мы игнорируем это значение), и любые обнаруженные ошибки (которые будут передаваться функции `check`).

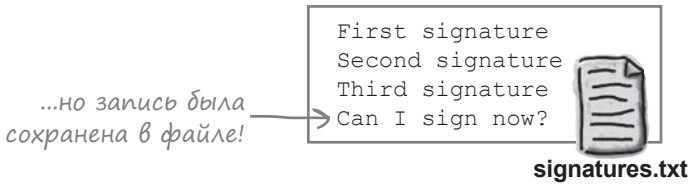
Наконец, для файла вызывается метод `Close`. Возможно, вы заметили, что мы *не* использовали ключевое слово `defer`. Это связано с тем, что мы записываем данные в файл, а не читаем из него. Вызов `Close` для файла, в который идет запись, может привести к ошибкам; эти ошибки необходимо обработать, а при использовании `defer` это не так легко. Таким образом, мы просто вызываем `Close` как часть обычной последовательности выполнения программы и передаем возвращаемое значение функции `check`.

Сохранение данных формы (продолжение)

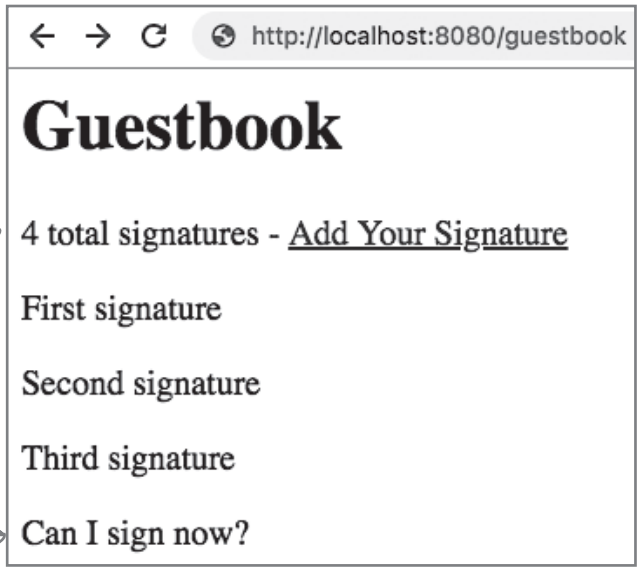
Сохраните приведенный ранее код и перезапустите *guestbook.go*. Заполните и отправьте форму на странице */guestbook/go*.

Браузер загрузит путь */guestbook/create*, по которому сейчас отображается совершенно пустая страница (потому что `createHandler` уже не может ничего записать в `http.ResponseWriter`).

Но если просмотреть файл *signatures.txt*, вы увидите, что в конце была сохранена новая запись!



При посещении списка записей в */guestbook* мы видим, что счетчик записей увеличился на 1, а в конце списка появилась новая запись!



(Кстати, если при создании файла signatures.txt вы не нажали Enter после последней строки, ваша новая запись будет добавлена в конец предыдущей записи. Ничего страшного! Отредактируйте файл signatures.txt, чтобы исправить ошибку, и все будущие записи станут сохраняться в отдельных строках.)

Перенаправления HTTP

Функция `createHandler` для сохранения новых записей готова. Осталось решить последнюю проблему: когда пользователь отправляет данные формы, браузер загружает путь */guestbook/create*, по которому отображается пустая страница.

Вообще по пути */guestbook/create* ничего полезного выводиться и не должно: он предназначен исключительно для приема запросов на добавление новой записи. Вместо этого лучше загрузить в браузере путь */guestbook*, чтобы пользователь увидел свою новую запись в гостевой книге.

В конце функции `createHandler` мы добавим вызов `http.Redirect`, который отправляет ответ браузеру и дает команду загрузить другой ресурс, отличный от запрошенного. `Redirect` получает в первых двух аргументах `http.ResponseWriter` и `*http.Request`, поэтому достаточно передать значения параметров `writer` и `request` вызова `createHandler`. Затем `Redirect` понадобится строка с путем для перенаправления браузера: мы используем путь `"/guestbook"`.

Последний аргумент `Redirect` должен содержать код состояния для передачи браузеру. Каждый ответ HTTP должен содержать код состояния. До сих пор наши ответы содержали коды, которые устанавливались за нас автоматически: успешные ответы имеют код состояния 200 («OK»), а запросам несуществующих страниц присваивается код состояния 404 («Not found»). Мы должны указать код для `Redirect`, поэтому используем константу `http.StatusFound`, из-за которой ответ перенаправления имеет код состояния 302 («Found»).

```
func createHandler(writer http.ResponseWriter, request *http.Request) {
        signature := request.FormValue("signature")
        options := os.O_WRONLY | os.O_APPEND | os.O_CREATE
        file, err := os.OpenFile("signatures.txt", options, os.FileMode(0600))
        check(err)
        _, err = fmt.Fprintln(file, signature)
        check(err)
        err = file.Close()
        check(err)
        http.Redirect(writer, request, "/guestbook", http.StatusFound)
}
```

Путь для перенаправления.

Этот код ответа означает, что запрос был успешным.

Необходимо передать Redirect значение ResponseWriter...

...а также исходный запрос.

После добавления вызова `Redirect` отправка формы для ввода записи должна работать примерно так:

1. Браузер отправляет запрос HTTP POST для пути */guestbook/create*.
2. Приложение отвечает перенаправлением на путь */guestbook*.
3. Браузер отправляет запрос GET для пути */guestbook*.

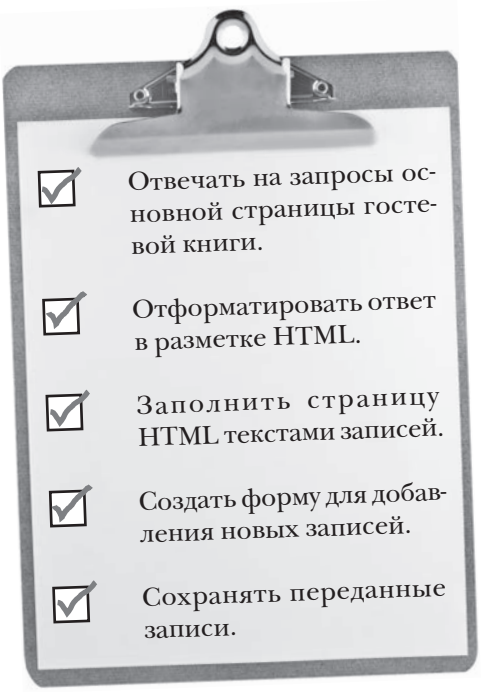
А теперь посмотрим, что получилось!

Давайте посмотрим, работает ли перенаправление! Перезапустите программу *guestbook.go* и откройте путь */guestbook/new*. Заполните поля формы и отправьте ее.

Приложение сохраняет содержимое формы в файле *signatures.txt*, после чего немедленно перенаправляет браузер на путь */guestbook*. Когда браузер запрашивает путь */guestbook*, приложение загружает файл *signatures.txt*, а пользователь видит свою запись в списке!

Наше приложение сохраняет записи, отправленные с формы, и выводит их в списке с остальными записями. Весь функционал готов!

Чтобы приложение заработало, пришлось решить целый ряд задач, но зато теперь у вас есть работоспособное веб-приложение!



Полный код приложения

Код приложения получился довольно длинным, но все это время мы рассматривали его по частям. Давайте напоследок взглянем на этот код как на единое целое!

Файл *guestbook.go* содержит основную часть кода приложения. (В приложениях, предназначенных для массового использования, часть кода можно было бы разбить на несколько пакетов и файлов с исходным кодом в каталоге рабочей области Go; при желании вы можете сделать это самостоятельно.) Мы добавили комментарии, документирующие тип `Guestbook` и каждую из функций.


```
package main

import (
    "bufio"
    "fmt"
    "html/template"
    "log"
    "net/http"
    "os"
)

// Guestbook - структура, используемая при отображении view.html.
type Guestbook struct {
    SignatureCount int
    Signatures     []string
}

// check вызывает log.Fatal для любых ошибок, отличных от nil.
func check(err error) {
    if err != nil {
       log.Fatal(err)
    }
}

// viewHandler читает записи гостевой книги и выводит их
// вместе со счетчиком записей.
func viewHandler(writer http.ResponseWriter, request *http.Request) {
    signatures := getStrings("signatures.txt")
    html, err := template.ParseFiles("view.html")
    check(err)
    guestbook := Guestbook{
       SignatureCount: len(signatures),
       Signatures:     signatures,
    }
    err = html.Execute(writer, guestbook)
    check(err)
}
```

Методу Render значения Template может передаваться только одно значение, поэтому эта структура будет содержать все необходимые данные.

Для хранения общего количества записей.

Для хранения самих записей.

Будет вызываться, когда потребуется проверить значение ошибки, возвращаемое функцией или методом.

В большинстве случаев значение будет равно nil, но если нет...

...программа выводит информацию об ошибке и завершается.

*Как и все функции-обработчики HTTP, получает http.ResponseWriter и *http.Request.*

Читает записи из файла.

Создает шаблон на основании содержимого view.html.

Для хранения количества записей.

Для хранения самих записей.

Данные структуры Guestbook вставляются в шаблон, а результат записывается в ResponseWriter.

guestbook.go (continued)

```
// newHandler отображает форму для ввода записи.
func newHandler(writer http.ResponseWriter, request *http.Request) {
    html, err := template.ParseFiles("new.html")
    check(err)
    err = html.Execute(writer, nil)
    check(err)
}

// createHandler получает запрос POST с добавляемой записью
// и присоединяет ее к файлу signatures.
func createHandler(writer http.ResponseWriter, request *http.Request) {
    signature := request.FormValue("signature")
    options := os.O_WRONLY | os.O_APPEND | os.O_CREATE
    file, err := os.OpenFile("signatures.txt", options, os.FileMode(0600))
    check(err)
    _, err = fmt.Fprintln(file, signature)
    check(err)
    err = file.Close()
    check(err)
    http.Redirect(writer, request, "/guestbook", http.StatusFound)
}
// getStrings возвращает сегмент строк, прочитанный из fileName,
// по одной строке на каждую строку файла.
func getStrings(fileName string) []string {
    var lines []string
    file, err := os.Open(fileName)
    if os.IsNotExist(err) {
       return nil
    }
    check(err)
    defer file.Close()
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
       lines = append(lines, scanner.Text())
    }
    check(scanner.Err())
    return lines
}

func main() {
    http.HandleFunc("/guestbook", viewHandler)
    http.HandleFunc("/guestbook/new", newHandler)
    http.HandleFunc("/guestbook/create", createHandler)
    err := http.ListenAndServe("localhost:8080", nil)
    log.Fatal(err)
}
```

Загружает форму HTML из шаблона.

Записывает шаблон в ResponseWriter (нет данных для вставки).

Получает значение поля формы «signature».

Открывает файл для записи. Если файл существует, то данные присоединяются к нему, а если нет — он создается.

Добавляет содержимое поля формы в файл.

Закрывает файл.

Перенаправляет браузер на основную страницу гостевой книги.

Каждая строка файла присоединяется к сегменту в виде отдельного элемента.

Открывает файл.

Если будет получена ошибка, указывающая на то, что файл не существует...

...возвращает nil вместо сегмента.

Все остальные ошибки должны проверяться обычным образом.

Создается сканер для содержимого файла.

Для каждой строки файла...

...ее текст присоединяется к сегменту.

Сообщает о любых ошибках, обнаруженных в процессе сканирования.

Возвращает сегмент строк.

Запросы на просмотр списка записей будут обрабатываться функцией viewHander.

Запросы на получение формы HTML будут обрабатываться функцией newHandler.

Запросы на отправку формы будут обрабатываться функцией createHandler.

В бесконечном цикле запросы HTTP передаются соответствующим функциям для обработки.

Файл *view.html* содержит шаблон HTML для списка записей. Действия шаблонов предоставляют место для вставки счетчика записей, а также всего списка записей.

```
<h1>Guestbook</h1>
<div>
    {{.SignatureCount}} total signatures -
    <a href="/guestbook/new">Add Your Signature</a>
</div>

<div>
    {{range .Signatures}}
      <p>{{.}}</p>
    {{end}}
</div>
```

Заголовок первого уровня для верхней части страницы.

Значение "." соответствует структуре Guestbook.

Сюда вставляется ее поле SignatureCount.

Ссылка на путь, представляющий форму HTML.

Берет сегмент из поля Signatures структуры Guestbook и повторяет обработку для каждой содержащейся в нем строки.

Повторяется для каждого элемента сегмента. "." присваивается текущая строка записи, а в вывод вставляется элемент абзаца HTML с текстом записи.

view.html

Файл *new.html* содержит форму HTML для ввода новых записей. Данные в разметку HTML не вставляются, поэтому нет и действий шаблона.

```
<h1>Add a Signature</h1>
<form action="/guestbook/create" method="POST">
  <div><input type="text" name="signature"></div>
  <div><input type="submit"></div>
</form>
```

Заголовок первого уровня для верхней части страницы.

Форма HTML.

Отправка следует по пути «/guestbook/create».

При отправке используется метод POST.

Текстовое поле, к данным которого мы обращаемся по имени «signature».

Кнопка для отправки данных формы.

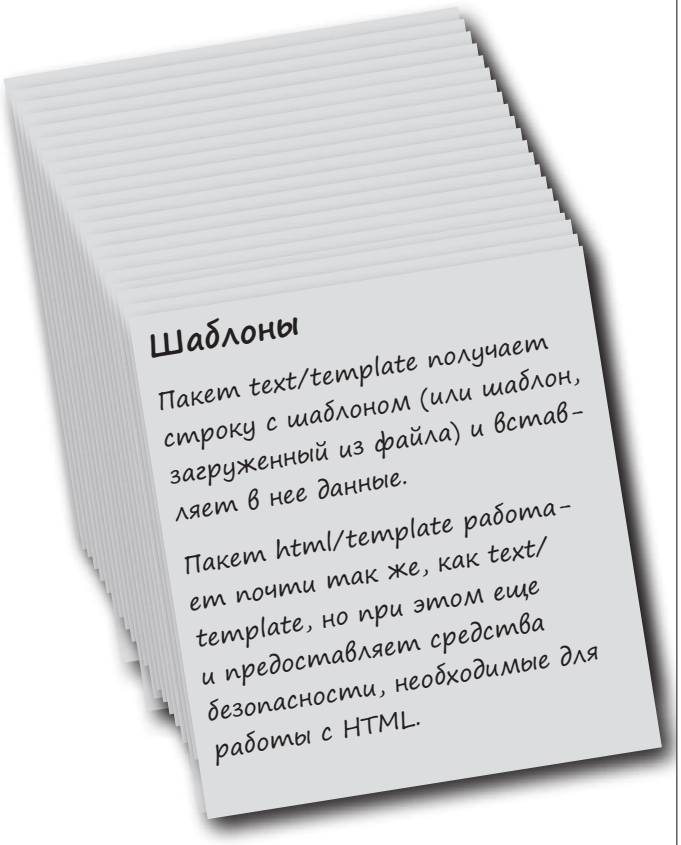
new.html

Вот и все — мы построили законченное веб-приложение, которое может сохранять записи, введенные пользователем, и снова загружать их в будущем!

Программирование веб-приложений — не самое простое занятие, но пакеты `net/http` и `html/template` упрощают для вас этот процесс!

Ваш инструментарий Go

Глава 16 осталась позади! В ней ваш инструментарий пополнился шаблонами.



КЛЮЧЕВЫЕ МОМЕНТЫ

- Строка шаблона содержит текст, который будет выводиться в исходном виде. Внутри этого текста можно вставлять различные **действия**, содержащие простой код для вычисления. Действия могут использоваться для вставки данных в текст шаблона.
- Метод `Execute` значения `Template` получает значение, поддерживающее интерфейс `io.Writer`, и значение данных, к которому можно обращаться из действий внутри шаблона.
- Действия шаблонов обращаются к значению данных, переданному `Execute`, в форме `{{.}}`. Значение "." (точка) может изменяться в различных контекстах внутри шаблона.
- Раздел шаблона между действием `{{if}}` и соответствующим маркером `{{end}}` включается только при выполнении некоторого условия.
- Раздел шаблона между действием `{{range}}` и соответствующим маркером `{{end}}` повторяется для каждого значения в массиве, сегменте, карте или канале. Любые действия внутри этого раздела тоже будут повторяться.
- В разделе `{{range}}` значение "." изменяется и обозначает текущий элемент коллекции.
- Если "." обозначает значение с типом структуры, для обращения к полям этой структуры может использоваться запись `{{.FieldName}}`.
- Запросы HTTP GET обычно используются в тех случаях, когда браузер должен получить данные от сервера.
- Запросы HTTP POST используются в тех случаях, когда браузер должен отправить новые данные на сервер.
- Для обращения к данным форм из запросов используется метод `FormValue` значения `http.Request`.
- Функция `http.Redirect` может использоваться для перенаправления браузера на другой путь.

Упражнение
Решение

Приведенная ниже программа загружает шаблон HTML из файла и выводит его в терминале. Заполните пропуски в файле *bill.html*, чтобы программа запускалась и выводила показанный результат.

```
type Invoice struct {
    Name     string
    Paid     bool
    Charges  []float64
    Total    float64
}

func main() {
    html, err := template.ParseFiles("bill.html")
    check(err)
    bill := Invoice{
       Name:    "Mary Gibbs",
       Paid:    true,
       Charges: []float64{23.19, 1.13, 42.79},
       Total:   67.11,
    }
    err = html.Execute(os.Stdout, bill)
    check(err)
}
```

bill.go

```
<h1>Invoice</h1>

<p>Name: {{.Name}} </p>

{{if .Paid }}
<p>Paid - Thank you!</p>
{{end}}

<h1>Fees</h1>

{{range .Charges}}
<p>$ {{.}} </p>
{{end}}

<p>Total: $ {{.Total}} </p>
```

bill.html

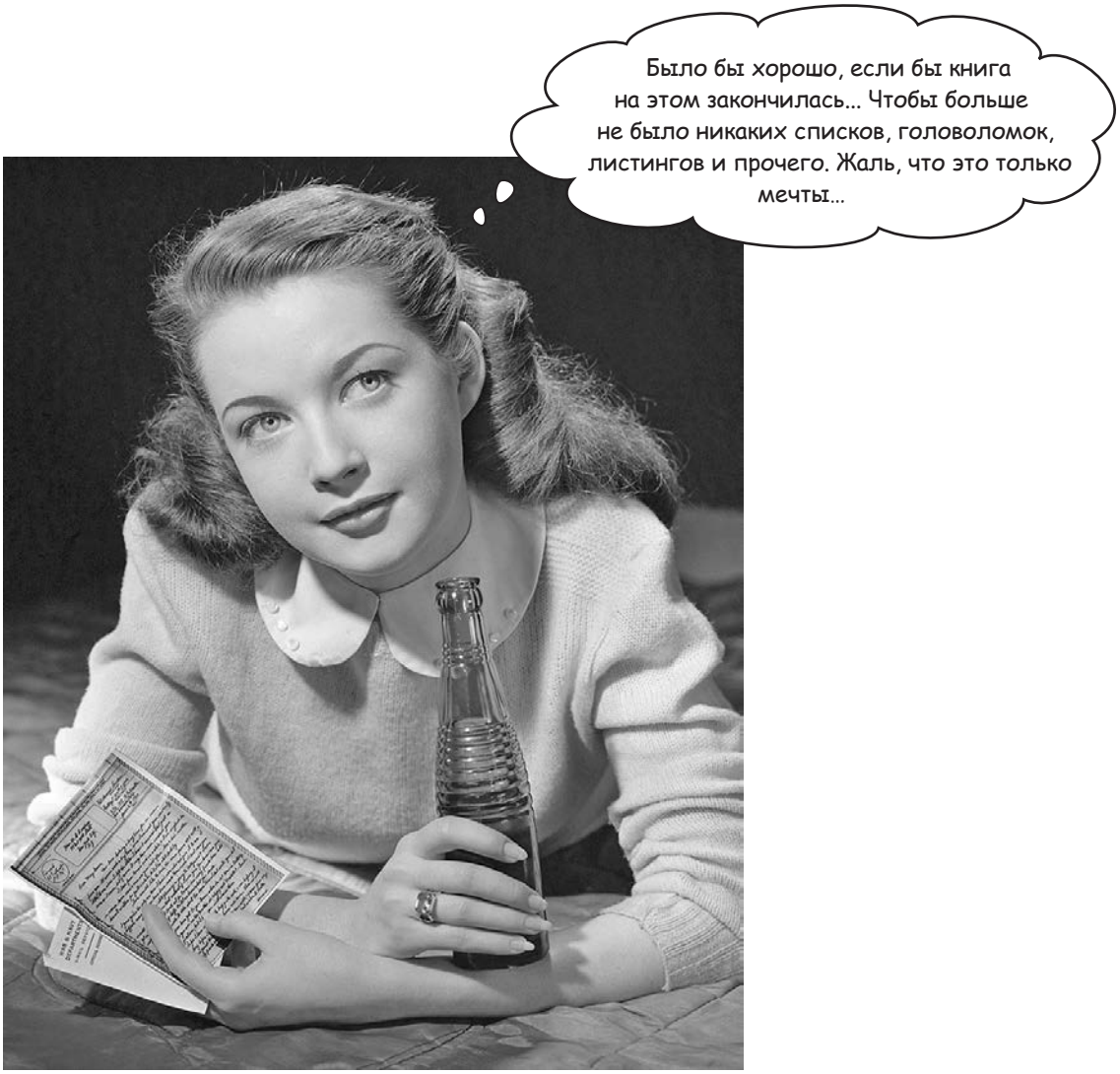
Поле Paid структуры Invoice содержит true?

Конец действия «if».

Выводит элемент <p> для каждого элемента в сегменте Charges.

Результат.

```
<h1>Invoice</h1>

<p>Name: Mary Gibbs</p>

<p>Paid - Thank you!</p>

<h1>Fees</h1>

<p>$23.19</p>
<p>$1.13</p>
<p>$42.79</p>

<p>Total: $67.11</p>
```



Поздравляем!

Вы добрались до конца.

Хотя еще есть два приложения.
И веб-сайт…
В общем, вам от нас никуда не деться.

Приложение A. Открытие файлов

Некоторые программы не только читают данные, но и записывают их в файлы. Когда в этой книге мы собирались поработать с файлами, приходилось создавать их в текстовом редакторе, чтобы программа могла прочитать данные. Но некоторые программы *генерируют* данные, и когда это происходит, программа должна иметь возможность *записать* данные в файл. Функция `os.OpenFile` уже использовалась для открытия файла для записи. Но у нас не было возможности объяснить, как она работает. В этом приложении вы узнаете все, что необходимо знать для эффективного использования `os.OpenFile`!

Как работает os.OpenFile

В главе 16 функция `os.OpenFile` использовалась для открытия файла для записи. При этом использовался довольно странного вида код:

```
options := os.O_WRONLY | os.O_APPEND | os.O_CREATE
file, err := os.OpenFile("signatures.txt", options, os.FileMode(0600))
```

Параметры открытия файла.

Открывает файл.

Тогда мы занимались разработкой веб-приложения, и возможности подробно объяснять, как работает `os.OpenFile`, не было. Но вам почти наверняка придется еще не раз использовать эту функцию в своей карьере программиста Go, и мы добавили это приложение, чтобы вы познакомились с ней поближе.

Чтобы разобраться в том, как работает функция, всегда полезно начать с документации. В терминале выполните команду **`go doc os OpenFile`** (или просмотрите документацию пакета `"os"` в браузере).

```
File Edit Window Help
$ go doc os OpenFile
func OpenFile(name string, flag int, perm FileMode) (*File, error)
    OpenFile is the generalized open call; most users will use Open or Create
    instead. It opens the named file with specified flag (O_RDONLY etc.) and
    ...
```

Аргументы функции — строка с именем файла, целое число «flag» и «perm» типа `os.FileMode`. Понятно, что в первом аргументе передается имя открываемого файла. Для начала посмотрим, что означает «flag», а затем вернемся к `os.FileMode`.

Чтобы не делать примеры в этом приложении слишком длинными, предполагается, что во всех программах присутствует функция `check` (наподобие той, которая была приведена в главе 16). Функция получает значение ошибки, проверяет, не равно ли оно `nil`, и если не равно — сообщает об ошибке и завершает программу.

```
func check(err error) {
      if err != nil {
            log.Fatal(err)
      }
}
```

Предполагается, что все программы, приведенные в этом приложении, включают функцию «check».

Передача констант флагов функции os.OpenFile

В описании упоминается, что одним из возможных значений флага является `os.O_RDONLY`. Поищем информацию и посмотрим, что это означает...

```
File Edit Window Help
$ go doc os O_RDONLY
const (
        // Exactly one of O_RDONLY, O_WRONLY, or O_RDWR must be specified.
        O_RDONLY int = syscall.O_RDONLY // open the file read-only.
        O_WRONLY int = syscall.O_WRONLY // open the file write-only.
        O_RDWR   int = syscall.O_RDWR   // open the file read-write.
        // The remaining values may be or'ed in to control behavior.
        O_APPEND int = syscall.O_APPEND // append data to the file when writing.
        O_CREATE int = syscall.O_CREAT  // create a new file if none exists.
        ...
)
    Flags to OpenFile wrapping those of the underlying system. Not all flags may
    be implemented on a given system.
```

Из документации следует, что `os.O_RDONLY` — одна из нескольких констант типа `int`, предназначенных для передачи функций `os.OpenFile` и управляющих поведением функции.

Попробуем вызвать `os.OpenFile` с некоторыми из этих констант и посмотрим, что произойдет.

Для начала понадобится файл для экспериментов. Создайте простой текстовый файл, содержащий всего одну строку текста. Сохраните его в любом каталоге под именем *aardvark.txt*.

Создайте этот файл в текстовом редакторе.

Затем в том же каталоге создайте программу Go, которая содержит функцию `check` с предыдущей страницы и приведенную ниже функцию `main`. В `main` функция `os.OpenFile` вызывается с передачей константы `os.O_RDONLY` во втором аргументе. (На третий аргумент пока не обращайте внимания, о нем мы поговорим позднее.) Затем мы создадим значение `bufio.Scanner` и воспользуемся им для вывода содержимого файла.

```
func main() {
        file, err := os.OpenFile("aardvark.txt", os.O_RDONLY, os.FileMode(0600))
        check(err)
        defer file.Close()
        scanner := bufio.NewScanner(file)
        for scanner.Scan() {
                fmt.Println(scanner.Text())
        }
        check(scanner.Err())
}
```

Открывает файл для чтения.

Выводит все строки из файла.

В терминале перейдите в каталог, в котором был сохранен файл *aardvark.txt* и ваша программа, и запустите программу командой **go run**. Команда открывает файл *aardvark.txt* и выводит его содержимое.

```
File Edit Window Help
$ cd work
$ go run openfile.go
Aardvarks are...
```

Передача констант флагов функции os.OpenFile (продолжение)

Теперь попробуем записать данные в файл. Обновите функцию `main` приведенным ниже кодом. (Также не забудьте удалить неиспользуемые пакеты из команды `import`.) На этот раз функции `os.OpenFile` будет передаваться константа `os.O_WRONLY`, чтобы файл открывался для записи. Затем для файла вызывается метод `Write` с сегментом байтов, которые должны быть записаны в файл.

```
func main() {
        file, err := os.OpenFile("aardvark.txt", os.O_WRONLY, os.FileMode(0600))
        check(err)
        _, err = file.Write([]byte("amazing!\n"))
        check(err)
        err = file.Close()
        check(err)
}
```

Файл открывается для записи.

Записывает данные в файл.

Если запустить программу, она не выдаст никакого результата, но обновит файл *aardvark.txt*. Но открыв файл *aardvark.txt*, мы увидим, что вместо присоединения текста в конец файла программа перезаписала часть файла!

Программа вставила новый текст в начало файла и перезаписала данные, которые там находились!

Это совсем не то, чего мы ожидали от своей программы. Что же делать?

К счастью, пакет `os` содержит несколько других констант, которые могут нам помочь. К их числу относится флаг `os.O_APPEND`, который заставляет программу дописать данные в конец файла вместо того, чтобы перезаписывать его текущее содержимое.

```
File Edit Window Help
$ go doc os O_RDONLY
...
        // The remaining values may be or'ed in to control behavior.
        O_APPEND int = syscall.O_APPEND // append data to the file when writing.
        O_CREATE int = syscall.O_CREAT  // create a new file if none exists.
        ...
```

Однако передать функции `os.OpenFile` только один флаг `os.O_APPEND` не получится: если вы попытаетесь это сделать, происходит ошибка.

Пытаемся присоединить данные к существующему файлу.

Ошибка времени выполнения!

```
file, err := os.OpenFile("aardvark.txt", os.O_APPEND, os.FileMode(0600))
```

```
write aardvark.txt:
bad file descriptor
```

В документации говорится о том, что константы `os.O_APPEND` и `os.O_CREATE` «могут объединяться операцией or». Речь идет о *двоичном операторе OR*. Чтобы разобраться в том, как он работает, нам понадобится несколько страниц...

Двоичная запись

На самом низком уровне для представления информации в компьютерах используются простые переключатели, которые могут находиться в одном из двух состояний: «включенном» или «выключенном». Если один переключатель будет использоваться для представления числа, то с его помощью можно будет представить только значения `0` («выключен») или `1` («включен»). В информатике эта единица называется *битом*.

(Если вам захочется узнать больше, введите строку «двоичные данные» в своей любимой поисковой системе.)

Для представления больших чисел потребуется группа из нескольких битов. Эта идея заложена в основу *двоичной* системы счисления. В повседневной жизни мы обычно используем десятичную систему с цифрами от 0 до 9. Но в двоичной системе для представления чисел используются только цифры 0 и 1.

Для просмотра двоичного представления различных чисел (битов, из которых состоят числа) можно воспользоваться функцией `fmt.Printf` с глаголом форматирования `%b`:

Выводит число в десятичной системе.

Выводит число в двоичной системе.

```
fmt.Printf("%3d: %08b\n", 0, 0)
fmt.Printf("%3d: %08b\n", 1, 1)
fmt.Printf("%3d: %08b\n", 2, 2)
fmt.Printf("%3d: %08b\n", 3, 3)
fmt.Printf("%3d: %08b\n", 4, 4)
fmt.Printf("%3d: %08b\n", 5, 5)
fmt.Printf("%3d: %08b\n", 6, 6)
fmt.Printf("%3d: %08b\n", 7, 7)
fmt.Printf("%3d: %08b\n", 8, 8)
fmt.Printf("%3d: %08b\n", 16, 16)
fmt.Printf("%3d: %08b\n", 32, 32)
fmt.Printf("%3d: %08b\n", 64, 64)
fmt.Printf("%3d: %08b\n", 128, 128)
```

```
  0: 00000000
  1: 00000001
  2: 00000010
  3: 00000011
  4: 00000100
  5: 00000101
  6: 00000110
  7: 00000111
  8: 00001000
 16: 00010000
 32: 00100000
 64: 01000000
128: 10000000
```

Побитовые операторы

Вы уже видели операторы `+`, `-`, `*` и `/`, выполняющие математические операции с числами. Но в Go также существуют **поразрядные операторы** для работы с отдельными битами, из которых состоит число. Два самых распространенных оператора такого рода — `&` (побитовый оператор И) и `|` (побитовый оператор ИЛИ).

Оператор	Название
`&`	Побитовый оператор И
`|`	Побитовый оператор ИЛИ

Побитовый оператор И

Ранее вам уже встречался оператор `&&`. Этот логический оператор возвращает значение `true` только в том случае, если оба операнда — левый *и* правый — равны `true`:

```
fmt.Printf("false && false == %t\n", false && false)
fmt.Printf("true  && false == %t\n", true  && false)
fmt.Printf("true  && true  == %t\n", true  && true)
```

```
false && false == false
true  && false == false
true  && true  == true
```

С другой стороны, оператор `&` (с одним знаком) является *побитовым* оператором. Он устанавливает бит в состояние `1` только в том случае, если соответствующие биты левого *и* правого операнда содержат 1. Для чисел `0` и `1`, для представления которых достаточно одного бита, все происходящее вполне очевидно.

```
fmt.Printf("%b & %b == %b\n", 0, 0, 0&0)
fmt.Printf("%b & %b == %b\n", 0, 1, 0&1)
fmt.Printf("%b & %b == %b\n", 1, 1, 1&1)
```

```
0 & 0 == 0
0 & 1 == 0
1 & 1 == 1
```

Ни один бит не равен 1.

Только один бит равен 1.

Оба бита равны 1.

Однако для больших чисел результат выглядит довольно странно!

```
fmt.Println(170 & 15)
fmt.Println( 10 &  7)
fmt.Println(100 & 45)
```

```
10
2
36
```

Что означают ЭТИ результаты?

И только если вы посмотрите на значения отдельных битов, побитовые операции обретают смысл. Оператор `&` присваивает 1 биту результата только в том случае, если бит в той же позиции левого числа *и* бит в той же позиции правого числа одновременно равны `1`.

```
fmt.Printf("%02b\n", 1)
fmt.Printf("%02b\n", 3)
fmt.Printf("%02b\n", 1&3)
```

```
01
11
01
```

Второй бит равен 0.

Второй бит равен 1.

Второй бит результата равен 0.

Первый бит равен 1.

Первый бит равен 1.

Первый бит результата равен 1.

```
fmt.Printf("%02b\n", 2)
fmt.Printf("%02b\n", 3)
fmt.Printf("%02b\n", 2&3)
```

```
10
11
10
```

Второй бит равен 1.

Второй бит равен 1.

Второй бит результата равен 1.

Первый бит равен 0.

Первый бит равен 1.

Первый бит результата равен 0.

Это относится к числам любого размера. Биты двух значений, к которым применяется оператор `&`, определяют биты в тех же позициях результирующего значения.

```
fmt.Printf("%08b\n", 170)
fmt.Printf("%08b\n", 15)
fmt.Printf("%08b\n", 170&15)
```

```
10101010
00001111
00001010
```

Если бит в данной позиции в первом числе равен 1...

...а бит в той же позиции во втором числе равен 1...

...то бит в той же позиции результирующего значения равен 1.

Побитовый оператор ИЛИ

Оператор || тоже вам уже встречался. Это логический оператор, который возвращает значение `true` только в том случае, если *хотя бы один* из двух операндов равен `true`.

```
fmt.Printf("false || false == %t\n", false || false)
fmt.Printf("true  || false == %t\n", true  || false)
fmt.Printf("true  || true  == %t\n", true  || true)
```

```
false || false == false
true  || false == true
true  || true  == true
```

Оператор | устанавливает бит в состояние 1 в том случае, если соответствующий бит левого значения *или* соответствующий бит правого значения равен 1.

```
fmt.Printf("%b | %b == %b\n", 0, 0, 0|0)
fmt.Printf("%b | %b == %b\n", 0, 1, 0|1)
fmt.Printf("%b | %b == %b\n", 1, 1, 1|1)
```

```
0 | 0 == 0
0 | 1 == 1
1 | 1 == 1
```

Ни один бит не равен 1.

Только один бит равен 1.

Оба бита равны 1.

Как и в случае с побитовым оператором И, поразрядный оператор ИЛИ определяет значение бита в некоторой позиции результата по значениям битов в той же позиции двух операндов.

```
fmt.Printf("%02b\n", 1)
fmt.Printf("%02b\n", 0)
fmt.Printf("%02b\n", 1|0)
```

Второй бит равен 0.

Второй бит равен 0.

Второй бит результата равен 0.

```
01
00
01
```

Первый бит равен 1.

Первый бит равен 0.

Первый бит результата равен 1.

```
fmt.Printf("%02b\n", 2)
fmt.Printf("%02b\n", 0)
fmt.Printf("%02b\n", 2|0)
```

Второй бит равен 1.

Второй бит равен 0.

Второй бит результата равен 1.

```
10
00
10
```

Первый бит равен 0.

Первый бит равен 0.

Первый бит результата равен 0.

Это относится к числам любого размера. Биты двух значений, к которым применяется оператор |, определяют биты в тех же позициях результирующего значения.

Побитовый оператор ИЛИ и константы пакета «os»

Все это хорошо... в теории. Я пока не вижу, как эти операторы помогут мне с использованием констант `os.O_APPEND` и `os.O_CREATE`!

Мы рассказали все это, потому что для объединения констант будем использовать побитовый оператор ИЛИ!

Когда в документации говорится, что значения `os.O_APPEND` и `os.O_CREATE` «могут объединяться операцией ИЛИ» со значениями `os.O_RDONLY`, `os.O_WRONLY` или `os.O_RDWR`, это означает, что к ним должен применяться побитовый оператор ИЛИ.

На самом деле все эти константы представляют собой обычные значения `int`:

```
fmt.Println(os.O_RDONLY, os.O_WRONLY, os.O_RDWR, os.O_CREATE, os.O_APPEND)
```

```
0 1 2 64 1024
```

Обратившись к двоичным представлениям этих значений, мы видим, что в каждом из них только один бит содержит `1`, а все остальные биты равны `0`:

```
fmt.Printf("%016b\n", os.O_RDONLY)
fmt.Printf("%016b\n", os.O_WRONLY)
fmt.Printf("%016b\n", os.O_RDWR)
fmt.Printf("%016b\n", os.O_CREATE)
fmt.Printf("%016b\n", os.O_APPEND)
```

```
0000000000000000
0000000000000001
0000000000000010
0000000001000000
0000010000000000
```

Это означает, что значения можно объединять побитовым оператором ИЛИ, и биты не будут путаться друг с другом:

```
fmt.Printf("%016b\n", os.O_WRONLY|os.O_CREATE)
fmt.Printf("%016b\n", os.O_WRONLY|os.O_CREATE|os.O_APPEND)
```

```
0000000001000001
0000010001000001
```

Чтобы определить, должен ли файл быть доступным только для записи, функция `os.OpenFile` может проверить, равен ли первый бит `1`. Если седьмой бит равен `1`, то функция `OpenFile` узнает, что файл нужно создать, если он не существует. А если 11-й бит равен `1`, `OpenFile` будет присоединять данные к файлу.

Будьте осторожны!

Используйте в своем коде только имена констант и никогда — их значения `int`!

Если вы будете использовать в своем коде вместо констант их значения (например, `1` или `1024`), в ближайшее время такое решение будет работать. Но если в будущем команда разработки Go изменит значения констант, ваш код перестанет работать. Обязательно используйте только имена констант — такие, как `os.O_WRONLY` и `os.O_APPEND`, и все будет нормально.

Решение проблемы с параметрами os.OpenFile

Когда мы передавали `os.OpenFile` только константу `os.O_WRONLY`, это приводило к перезаписи части данных, хранившихся в файле. Попробуем объединить параметры так, чтобы новые данные вместо этого добавлялись в конец файла.

Для начала отредактируйте файл *aardvark.txt*, чтобы он снова содержал только одну строку.

Отредактируйте текстовый файл, чтобы он вернулся к исходному виду.

Теперь обновим программу, чтобы константы `os.O_WRONLY` и `os.O_APPEND` объединялись в одно значение побитовым оператором ИЛИ. Результат передается функции `os.OpenFile`.

Два значения объединяются побитовым оператором ИЛИ.

Результат передается функции os.OpenFile.

```
func main() {
        options := os.O_WRONLY | os.O_APPEND
        file, err := os.OpenFile("aardvark.txt", options, os.FileMode(0600))
        check(err)
        _, err = file.Write([]byte("amazing!\n"))
        check(err)
        err = file.Close()
        check(err)
}
```

Снова запустите программу и просмотрите содержимое файла. В конце файла должна появиться новая строка текста.

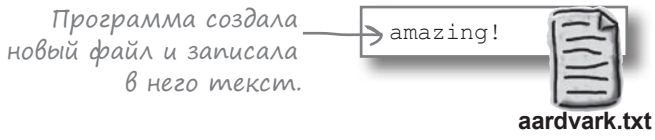
Также попробуем использовать константу `os.O_CREATE`, которая дает команду `os.OpenFile` создать заданный файл, если он не существует. Начните с удаления файла *aardvark.txt*.

Затем обновите программу и добавьте `os.O_CREATE` в набор параметров, передаваемых `os.OpenFile`.

Значение os.O_CREATE добавляется при помощи поразрядного оператора ИЛИ.

```
        options := os.O_WRONLY | os.O_APPEND | os.O_CREATE
        file, err := os.OpenFile("aardvark.txt", options, os.FileMode(0600))
        // ...
```

Запустите программу. Она создает новый файл *aardvark.txt* и записывает в него данные.



Разрешения доступа к файлам в стиле Unix

До настоящего момента мы ограничивались вторым аргументом функции `os.OpenFile`, который управляет чтением, записью, созданием и присоединением данных к файлу. При этом ни слова не было сказано о третьем аргументе, управляющем *разрешениями доступа* к файлу, то есть каким пользователям будет разрешено выполнять операции чтения и записи в файл после его создания программой.

Этот аргумент управляет «разрешениями» для доступа к новым файлам.

```
file, err := os.OpenFile("aardvark.txt", options, os.FileMode(0600))
```

```
File Edit Window Help
$ go doc os OpenFile
func OpenFile(name string, flag int, perm FileMode) (*File, error)
    OpenFile is the generalized open call; most users will use Open or Create
    instead. It opens the named file with specified flag (O_RDONLY etc.) and
    ...
```

Когда разработчики говорят о разрешениях доступа к файлам, они обычно имеют в виду разрешения, реализованные в системах семейства Unix — таких, как macOS и Linux. В Unix существуют три основных разрешения для операций с файлами, которые могут предоставляться пользователю:

Сокращение	Разрешение
`r`	Пользователю разрешено читать содержимое файла (**r**ead).
`w`	Пользователю разрешено записывать содержимое в файл (**w**rite).
`x`	Пользователю разрешено исполнять файл (только для файлов, содержащих программный код (e**x**ecute)).

Например, если пользователь не имеет разрешения на чтение файла, то любая запущенная им программа при попытке обратиться к содержимому файла получит ошибку от операционной системы:

```
File Edit Window Help
$ cat locked.txt
cat: locked.txt: Permission denied
```

Если пользователь не имеет разрешений на исполнение файла, то не сможет выполнить содержащийся в нем код. (Файлы, которые не содержат исполняемого кода, *не должны* помечаться как исполняемые, потому что попытка запустить их может привести к непредсказуемым результатам.)

```
File Edit Window Help
$ ./hello
-bash: ./hello: Permission denied
```

Будьте осторожны!

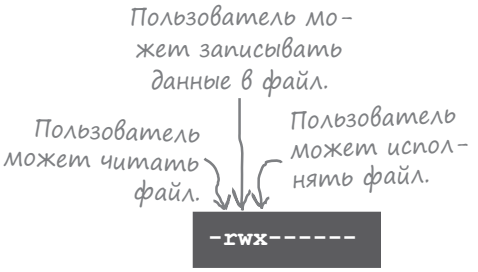
В Windows аргумент с разрешениями игнорируется.

Windows работает с разрешениями доступа к файлам не так, как системы семейства Unix, поэтому что бы вы ни делали, в Windows файлы создаются с разрешениями по умолчанию. Но при запуске на машинах семейства Unix та же программа не будет игнорировать разрешения. Очень важно разобраться в том, как работают разрешения, и по возможности тестировать вашу программу для разных операционных систем, в которых она должна выполняться.

Представление разрешений с типом os.FileMode

Пакет Go `os` использует тип `FileMode` для представления разрешений доступа к файлам. Если файл еще не существует, то переданное `os.OpenFile` значение `FileMode` определяет, с какими разрешениями файл будет создан, а следовательно, какие права доступа к нему получат различные пользователи.

Значение `FileMode` содержит метод `String`, поэтому при передаче `FileMode` функциям пакета `fmt` (например, `fmt.Println`) вы получите специальное строковое представление значения. Эта строка представляет разрешения, хранящиеся в `FileMode`, в формате, аналогичном формату Unix-утилиты `ls`.

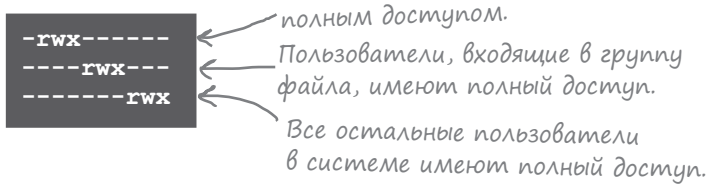



```
fmt.Println(os.FileMode(0700))
```

Каждый файл имеет три набора разрешений для трех разных классов пользователей. Первый набор разрешений действует только для пользователя, который является владельцем файла. (По умолчанию ваша учетная запись пользователя является владельцем всех созданных вами файлов.) Второй набор разрешений действует для группы пользователей, с которой связывается файл. А третий — действует для остальных пользователей в системе, которые не являются владельцами и не входят в группу, связанную с файлом.

(Если вам потребуется дополнительная информация, введите строку «разрешения файлов Unix» в поисковой системе.)

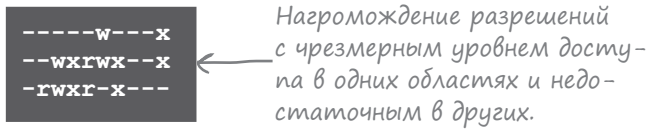
```
fmt.Println(os.FileMode(0700))
fmt.Println(os.FileMode(0070))
fmt.Println(os.FileMode(0007))
```



Тип `FileMode` основан на типе `uint32`, то есть «32-разрядное целое без знака». Этот базовый тип еще не упоминался в книге. Беззнаковый тип не может содержать отрицательные значения, но зато в нем могут храниться числа большие, чем те, которые обычно помещаются в 32 битах памяти.

Так как тип `FileMode` основан на `uint32`, (почти) любое неотрицательное целое число можно преобразовать в `FileMode`. Тем не менее разобраться в результате преобразования может быть непросто:

```
fmt.Println(os.FileMode(17))
fmt.Println(os.FileMode(249))
fmt.Println(os.FileMode(1000))
```



Восьмеричная система счисления

Целые числа для преобразования в значения `FileMode` удобнее записывать в **восьмеричной системе**. Вам хорошо знакома десятичная система счисления, в которой используются 10 цифр: от 0 до 9. Вы видели двоичную систему, в которой используются всего две цифры: 0 и 1. В восьмеричной системе используются восемь цифр: от 0 до 7.

Для просмотра восьмеричного представления различных чисел можно воспользоваться функцией `fmt.Printf` с глаголом форматирования `%0`:

```
for i := 0; i <= 19; i++ {
        fmt.Printf("%3d: %04o\n", i, i)
}
```

Выводит число в десятичной системе.

Выводит число в восьмеричной системе.

```
  0: 0000
  1: 0001
  2: 0002
  3: 0003
  4: 0004
  5: 0005
  6: 0006
  7: 0007
  8: 0010
  9: 0011
 10: 0012
 11: 0013
 12: 0014
 13: 0015
 14: 0016
 15: 0017
 16: 0020
 17: 0021
 18: 0022
 19: 0023
```

В первой позиции цифры возрастают до 7...

...затем первая позиция обнуляется, а вторая позиция увеличивается до 1.

Снова до 7 в первой позиции...

...после чего первая позиция обнуляется, а вторая увеличивается до 2. И так далее.

В отличие от двоичных чисел, Go позволяет записывать числа в восьмеричной системе прямо в коде программы. Любая последовательность цифр, начинающаяся с `0`, интерпретируется как восьмеричное число.

На первых порах разобраться в этом непросто. Десятичное число `10` — не то же самое, что восьмеричное `10`, а десятичное число `100` сильно отличается от восьмеричного `0100`!

```
fmt.Printf("Decimal   1: %3d Octal   01: %2d\n",   1,   01)
fmt.Printf("Decimal  10: %3d Octal  010: %2d\n",  10,  010)
fmt.Printf("Decimal 100: %3d Octal 0100: %2d\n", 100, 0100)
```

```
Decimal   1:   1 Octal   01:  1
Decimal  10:  10 Octal  010:  8
Decimal 100: 100 Octal 0100: 64
```

В восьмеричных числах разрешены только цифры от 0 до 7. Если в восьмеричном числе встречается цифра 8 или 9, вы получите ошибку компиляции.

```
fmt.Println(089)
```

```
illegal octal number
```

Ошибка компиляции!

Преобразование восьмеричных значений в FileMode

Так зачем использовать эту (странную на первый взгляд) восьмеричную систему для записи файловых разрешений? Потому что каждая цифра восьмеричного числа может быть представлена всего тремя битами памяти.

```
fmt.Printf("%09b\n", 0007)   000000111
fmt.Printf("%09b\n", 0070)   000111000
fmt.Printf("%09b\n", 0700)   111000000
```

3 3 3 бита бита бита

Ровно столько же памяти — три бита — необходимо для хранения разрешений одного класса пользователей («пользователь», «группа» или «остальные»). А значит, любая комбинация разрешений для класса пользователей может быть представлена одной восьмеричной цифрой!

Цифра для разрешений группы.

Цифра для разрешений пользователя.

Цифра для разрешений других пользователей.

```
os.FileMode(0777)
```

Обратите внимание на сходство между двоичным представлением восьмеричных чисел, приведенных ниже, и результатом преобразования `FileMode` для того же числа. Если бит в двоичном представлении равен `1`, то соответствующее разрешение включено.

Выводит двоичное представление восьмеричного числа.

Выводит строку для преобразования того же числа в FileMode.

Если бит равен 1, соответствующее разрешение включено.

```
fmt.Printf("%09b %s\n", 0000, os.FileMode(0000))   000000000 ----------
fmt.Printf("%09b %s\n", 0111, os.FileMode(0111))   001001001 ---x--x--x
fmt.Printf("%09b %s\n", 0222, os.FileMode(0222))   010010010 --w--w--w-
fmt.Printf("%09b %s\n", 0333, os.FileMode(0333))   011011011 --wx-wx-wx
fmt.Printf("%09b %s\n", 0444, os.FileMode(0444))   100100100 -r--r--r--
fmt.Printf("%09b %s\n", 0555, os.FileMode(0555))   101101101 -r-xr-xr-x
fmt.Printf("%09b %s\n", 0666, os.FileMode(0666))   110110110 -rw-rw-rw-
fmt.Printf("%09b %s\n", 0777, os.FileMode(0777))   111111111 -rwxrwxrwx
```

По этой причине утилита Unix `chmod` (сокращение от «change mode») использует восьмеричные цифры для установки разрешения доступа для десятичных цифр.

File Edit Window Help

```
$ chmod 0000 allow_nothing.txt
$ chmod 0100 execute_only.sh
$ chmod 0200 write_only.txt
$ chmod 0300 execute_write.sh
$ chmod 0400 read_only.txt
$ chmod 0500 read_execute.sh
$ chmod 0600 read_write.txt
$ chmod 0700 read_write_execute.sh
$ chmod 0124 user_execute_group_write_other_read.sh
$ chmod 0777 all_read_write_execute.sh
```

Поддержка восьмеричной записи в Go позволяет использовать ту же схему в вашем коде!

Восьмеричная цифра	Разрешение
0	Разрешения отсутствуют
1	Исполнение
2	Запись
3	Запись, исполнение
4	Чтение
5	Чтение, исполнение
6	Запись, чтение
7	Запись, чтение, исполнение

Подробный анализ вызова os.OpenFile

Итак, теперь вы понимаете, как работают побитовые операторы и восьмеричная система, и мы наконец-то можем объяснить, что происходит при вызове `os.OpenFile`!

Например, следующий фрагмент присоединяет новые данные к существующему файлу. Пользователь, который является владельцем файла, сможет читать и записывать данные в файл. Все остальные пользователи смогут только читать данные из файла.

```
options := os.O_WRONLY | os.O_APPEND
file, err := os.OpenFile("log.txt", options, os.FileMode(0644))
```

Файл открывается для записи, с добавлением новых данных в конец файла.

Владелец сможет читать и записывать данные в файл, а все остальные смогут только читать данные.

А этот фрагмент создает файл, если он не существует, а затем присоединяет к нему данные. Полученный файл будет доступен для чтения и записи для владельца, но недоступен для всех остальных пользователей.

```
options := os.O_WRONLY | os.O_APPEND | os.O_CREATE
file, err := os.OpenFile("log.txt", options, os.FileMode(0600))
```

Если файл не существует, то будет создан. Файл открывается для записи с присоединением новых данных в конец файла.

Файл доступен для чтения и записи для своего владельца. Для всех остальных он будет недоступен.

Будьте осторожны!

Если функции os.Open или os.Create делают то, что вам нужно, лучше пользоваться ими.

Функция `os.Open` *может открывать файлы только для чтения. Но если этого достаточно, возможно, будет проще использовать ее, чем* `os.OpenFile`. *Функция* `os.Create` *может создавать только файлы, доступные для чтения и записи для любого пользователя. Но если этого достаточно, возможно, будет проще использовать ее, чем* `os.OpenFile`. *Иногда менее мощные функции позволяют получить более понятный код.*

Часто Задаваемые Вопросы

В: Восьмеричная система, побитовые операторы — все очень сложно! И для чего это все придумали?

О: Для экономии памяти! Эти соглашения уходят корнями к системе Unix, которая создавалась в те времена, когда оперативная память и дисковое пространство имели меньший объем и стоили дороже. Но даже в наши дни, когда жесткий диск может содержать миллионы файлов, упаковка файловых разрешений в несколько битов (вместо нескольких байтов) может ощутимо экономить память (и ускорять работу системы). Поверьте, игра стоит свеч!

В: Что это за дефис в начале строки `FileMode`? (См. страницу 525.)

О: Дефис в этой позиции означает, что элемент является обычным файлом, но возможны и другие значения. Например, если значение `FileMode` представляет каталог, здесь будет находиться буква `d`.

Получение статистики для файла или каталога. Информацию вы найдете в документации!

```
fileInfo, err := os.Stat("my_directory")
if err != nil {
      log.Fatal(err)
}
fmt.Println(fileInfo.Mode())
```

Выводит информацию FileMode для директории.

```
drwxr-xr-x
```

Приложение Б. Напоследок

Позади большой путь, и книга почти подошла к концу. Мы будем скучать, но, пожалуй, было бы неправильно отпускать вас в свободное плавание без *небольшой* дополнительной подготовки. Мы приберегли шесть важных тем для этого приложения.

№ 1. Команды инициализации для «if»

Ниже приведена функция `saveString`, которая возвращает одно значение ошибки (или `nil`, если ошибки не было). В функции `main` возвращаемое значение можно сохранить в переменной `err` перед тем, как обработать его:

```
func saveString(fileName string, str string) error {
       err := ioutil.WriteFile(fileName, []byte(str), 0600)
       return err
}
```

(Чтобы больше узнать о функции WriteFile, введите команду «go doc io/ioutil WriteFile».)

Вызывает saveString и сохраняет возвращаемое значение.

Сообщает о любых обработанных ошибках.

```
func main() {
       err := saveString("hindi.txt", "Namaste")
       if err != nil {
              log.Fatal(err)
       }
}
```

А теперь предположим, что мы добавили в `main` еще один вызов `saveString`, который также использует переменную `err`. Следует помнить о том, что при первом использовании переменной должно применяться короткое объявление переменной, а во всех последующих — присваивания. В противном случае компилятор выдает сообщение об ошибке повторного объявления переменной.

В этом коде также используется переменная с именем «err».

Если вы забудете преобразовать короткое объявление в исходном коде в присваивание...

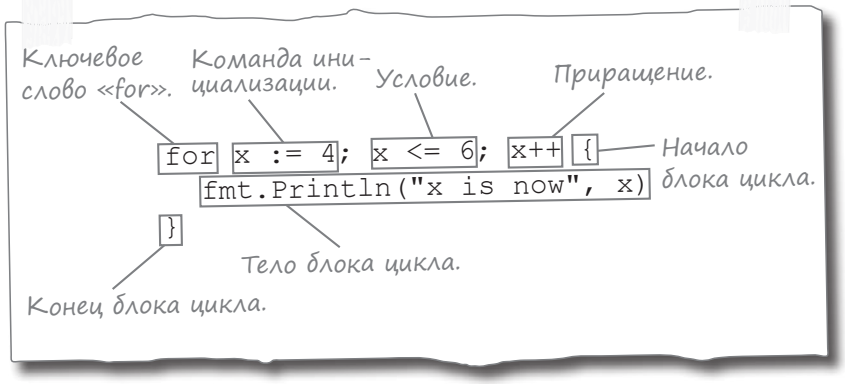
```
func main() {
       err := saveString("english.txt", "Hello")
       if err != nil {
              log.Fatal(err)
       }
       err := saveString("hindi.txt", "Namaste")
       if err != nil {
              log.Fatal(err)
       }
}
```

Ошибка компиляции!

```
no new variables on left side of :=
```

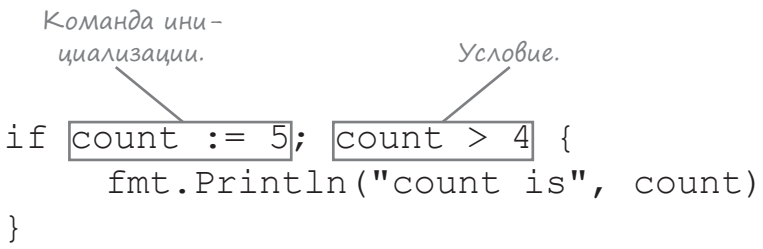
Но на самом деле переменная `err` используется только внутри команды `if` и ее блока. Нельзя ли как-то ограничить область видимости переменной, чтобы каждое появление рассматривалось как отдельная переменная?

Помните наше знакомство с циклами `for` в главе 2? Мы тогда сказали, что они могут включать команду инициализации, в которой инициализируются переменные. Область видимости таких переменных ограничивается блоком цикла `for`.

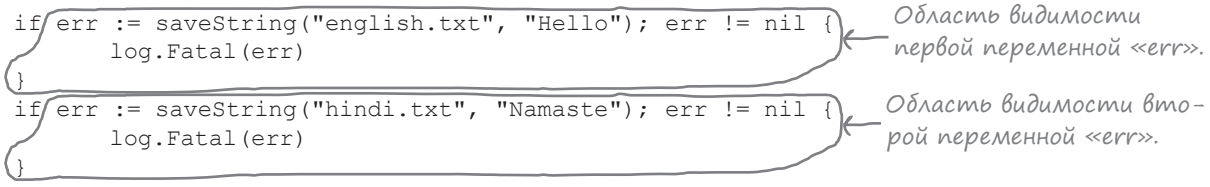


№ 1. Команды инициализации для «if» (продолжение)

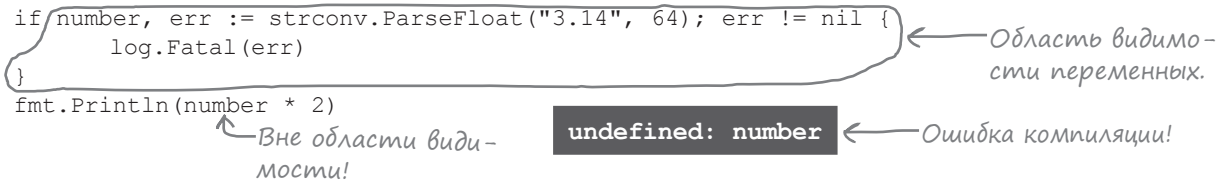
По аналогии с циклами `for`, Go позволяет добавить команду инициализации перед условием в команде `if`. Команды инициализации обычно используются для инициализации одной или нескольких переменных, используемых внутри блока `if`.



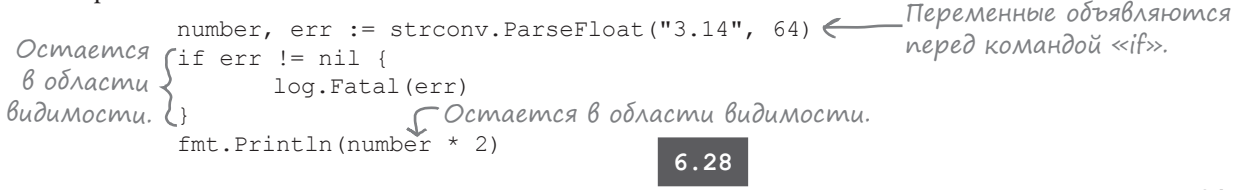
Область видимости переменных, объявленных в командах инициализации, ограничивается условием этой команды `if` и ее блоком. Если переписать предыдущий пример, чтобы в нем использовались команды инициализации `if`, область видимости каждой переменной `err` будет ограничена условием и блоком команды `if`; это означает, что в программе существуют две разные переменные `err`. И нам не нужно беспокоиться о том, какая переменная определяется первой.



Впрочем, ограничение области видимости — палка о двух концах. Если функция имеет несколько возвращаемых значений и одно из них должно использоваться *внутри* команды `if`, а другое — *вне* ее, скорее всего, вы не сможете вызвать эту функцию в команде инициализации. Если же вы попытаетесь это сделать, то значение, которое должно использоваться вне блока `if`, оказывается вне области видимости.



Вместо этого необходимо вызвать функцию до команды `if`, как обычно, чтобы ее возвращаемые значения находились в области видимости как внутри команды `if`, так *и* за ее пределами:



№ 2. Команда switch

Если программа должна выбрать одно из нескольких действий в зависимости от значения выражения, код быстро превращается в мешанину из команд `if` и условий `else`. Команда `switch` позволяет более эффективно представить конструкцию выбора.

Она состоит из ключевого слова `switch`, за которым следует условное выражение. Далее добавляются несколько выражений `case` с возможными значениями, которые может принимать условие. Программа выбирает первую секцию `case`, значение которой совпадает с результатом условного выражения, и выполняет содержащийся в ней код. Другие выражения `case` игнорируются. Также команда может содержать секцию `default`, которая выполняется в том случае, если ни одна из альтернатив не подходит.

Ниже приведена повторная реализация примера с командами `if` и `else` из главы 12. Эта версия получается гораздо более компактной. В условии `switch` выбирается случайное число от 1 до 3. Мы предоставляем выражения `case` для каждого из этих значений, и в каждом случае выводится отдельное сообщение. Чтобы оповестить пользователя о теоретически невозможных ситуациях, в которых не совпадает ни один из вариантов, мы также определяем секцию `default`, которая инициирует ситуацию паники.

```
import (
       "fmt"
       "math/rand"
       "time"
)

func awardPrize() {
       switch rand.Intn(3) + 1 {
       case 1:
              fmt.Println("You win a cruise!")
       case 2:
              fmt.Println("You win a car!")
       case 3:
              fmt.Println("You win a goat!")
       default:
              panic("invalid door number")
       }
}

func main() {
       rand.Seed(time.Now().Unix())
       awardPrize()
}
```

Условное выражение.

Если результат равен 1...

...выводится это сообщение.

Если результат равен 2...

...выводится это сообщение.

Если результат равен 3...

...выводится это сообщение.

Если результат не равен ни одному из приведенных значений...

...запускается паника, потому что в коде произошло что-то совершенно непредвиденное.

```
You win a goat!
```

Часто задаваемые вопросы

В: В некоторых языках каждая секция `case` должна завершаться командой «break», в противном случае будет выполнен код следующей секции `case`. В Go это не обязательно?

О: В других языках разработчики часто забывали включать команду «break», что приводило к ошибкам. Чтобы избавиться от этой проблемы, Go автоматически выходит из команды `switch` в конце кода `case`.

Если же вы *хотите*, чтобы после текущей секции `case` выполнялся код следующей, для этого существует ключевое слово `fallthrough`.

№ 3. Другие базовые типы

В Go существуют и другие базовые типы, о которых из-за нехватки места мы не говорили. Вероятно, у вас не будет особых причин использовать их в своих проектах, но эти типы встречаются в некоторых библиотеках, поэтому лучше знать об их существовании.

Типы	Описание

`int8`
`int16`
`int32`
`int64`
	Эти типы предназначены для хранения целых чисел, как и `int`, но занимают блок памяти определенного размера (число в имени типа определяет размер в битах). С меньшим количеством битов тип занимает меньше памяти или другого пространства; большее количество битов позволяет хранить большие числа. Используйте `int`, если только у вас нет веских причин для выбора одного из этих типов, решение с `int` будет более эффективно

`uint`	Тип похож на `int`, но подходит для хранения только целых чисел *без знака*; в нем не могут храниться отрицательные числа. Это означает, что в той же области памяти можно будет хранить большие числа, но вы должны следить за тем, чтобы эти значения не были отрицательными

`uint8`
`uint16`
`uint32`
`uint64`
	Типы также предназначены для хранения целых чисел без знака, но как и другие разновидности `int`, они занимают конкретное количество битов в памяти

`float32`	Тип `float64` предназначен для хранения чисел с плавающей точкой и занимает 64 бита памяти. Этот тип является его 32-разрядным «родственником». (У чисел с плавающей точкой нет 8- или 16-разрядных разновидностей.)

№ 4. О рунах

Руны были кратко представлены в главе 1, и с тех пор мы к ним не возвращались. Но прежде чем завершать книгу, хотелось бы рассказать о них чуть подробнее...

До появления современных операционных систем в большей части вычислений использовался латинский алфавит из 26 букв (верхнего и нижнего регистра) без диакритических знаков. Символов было так мало, что их можно было представить одним байтом (причем 1 бит оставался свободным). Чтобы одно значение байта интерпретировалось как одна и та же буква в разных системах, был разработан специальный стандарт, который назывался ASCII.

Но конечно, латинский алфавит — не единственная система письменности в мире. Существует много других систем, причем некоторые состоят из тысяч разных символов. Создатели стандарта Юникод постарались создать набор *4-байтовых* значений, которые позволяют представить любой символ всех этих разных алфавитов (а также многих других символов).

Для представления символов Юникода в Go используются значения типа `rune`, то есть руны. Обычно одна руна представляет один символ. (Есть и исключения, но они выходят за рамки темы этой книги.)

№ 4. О рунах (продолжение)

Go использует UTF-8-стандарт, в котором представления символов Юникода занимают от 1 до 4 байт. Символы из старого набора ASCII, как и прежде, представляются 1 байтом, другие символы могут занимать от 2 до 4 байт.

Перед вами две строки: одна состоит из букв латинского алфавита, а другая — из букв алфавита русского языка.

```
asciiString := "ABCDE"
utf8String := "БГДЖИ"
```

Все эти символы принадлежат набору символов ASCII, поэтому они занимают 1 байт каждый.

Эти символы Юникода занимают по 2 байта каждый.

Обычно не приходится досконально разбираться в том, как хранятся символы... во всяком случае *до того*, как вам потребуется преобразовать строки в составляющие их байты и наоборот. Например, при попытке вызвать функцию `len` для этих двух строк будут получены разные результаты:

```
fmt.Println(len(asciiString))
fmt.Println(len(utf8String))
```

```
5
10
```

Строка занимает 5 байт.

Строка занимает 10 байт.

Когда строка передается функции `len`, функция возвращает длину в *байтах*, а не в *рунах*. Строка латинского алфавита занимает 5 байт — для каждой руны требуется всего 1 байт, потому что символы входят в старый набор ASCII. Но строка букв русского алфавита занимает 10 байт — для хранения каждой руны требуются 2 байта.

Если вас интересует длина строки в *символах*, используйте функцию `RuneCountInString` из пакета `unicode/utf8`. Эта функция возвращает правильное количество символов независимо от количества байтов, используемых для хранения отдельных символов.

```
fmt.Println(utf8.RuneCountInString(asciiString))
fmt.Println(utf8.RuneCountInString(utf8String))
```

```
5
5
```

Строка содержит пять рун.

Эта строка тоже содержит пять рун.

Для безопасной работы с неполными строками их необходимо преобразовать в руны, а не в байты.

№ 4. О рунах (продолжение)

В этой книге нам уже приходилось преобразовывать строки в сегменты байтов, чтобы записать их в ответ HTTP или вывести на терминал. Все отлично работает, пока записываются *все* байты полученного сегмента. Но если вы попытаетесь работать только с *частью* байтов, то напрашиваетесь на неприятности.

Следующий код пытается исключить первые три символа из строк. Каждая строка преобразуется в сегмент байтов, после чего оператор сегмента используется для получения всех данных от четвертого элемента до конца сегмента. Затем программа снова преобразует неполные сегменты байтов в строки и выводит их.

```
asciiBytes := []byte(asciiString)
utf8Bytes := []byte(utf8String)
asciiBytesPartial := asciiBytes[3:]
utf8BytesPartial := utf8Bytes[3:]
fmt.Println(string(asciiBytesPartial))
fmt.Println(string(utf8BytesPartial))
```

Строки преобразуются в сегменты байтов.

Из каждого сегмента исключаются первые 3 байта.

```
DE
□ДЖИ
```

При исключении первых 3 байтов исключаются первые 3 символа.

При исключении первых 3 байтов исключается первая руна и 1 байт из второй руны!

Такое решение хорошо работает с символами латинского алфавита, каждый из которых занимает 1 байт. Но каждый символ русского алфавита занимает *2 байта*. При исключении первых 3 байтов из этой строки удаляется только первый символ и «половина» второго, что приводит к появлению символа, не имеющего печатного представления.

Go поддерживает преобразование строк в сегменты значений `rune` и из сегментов рун обратно в строки. Чтобы работать с неполными строками, сначала преобразуйте их в сегмент значений `rune` вместо сегмента значений `byte`. Тем самым предотвращается риск того, что часть байтов будет интерпретирована как руна.

Ниже приведена обновленная версия предыдущего кода, которая преобразует строки в сегменты рун вместо сегмента байтов. Операторы сегмента теперь исключают из каждого сегмента первые 3 *руны* вместо первых *3 байтов*. Когда мы преобразуем неполные сегменты в строки и выводим их, то получаем только последние два (полных) символа из каждой строки.

```
asciiRunes := []rune(asciiString)
utf8Runes := []rune(utf8String)
asciiRunesPartial := asciiRunes[3:]
utf8RunesPartial := utf8Runes[3:]
fmt.Println(string(asciiRunesPartial))
fmt.Println(string(utf8RunesPartial))
```

Строки преобразуются в сегменты рун.

Из каждого сегмента исключаются первые три руны.

Сегмент рун преобразуется в строку.

```
DE
ЖИ
```

Первые три руны исключены.

Первые три руны исключены.

№ 4. О рунах (продолжение)

Аналогичные проблемы возникают и в том случае, если вы попытаетесь использовать сегмент байтов для обработки каждого символа строки. Обработка по 1 байту работает, если все символы входят в набор ASCII. Но как только вам встретится символ, занимающий два и более байта, вы снова будете работать только с частью байтов руны.

В следующем коде используется цикл `for ... range` для вывода символов латинского алфавита, по 1 байту на символ. Затем программа пытается сделать то же самое с символами русского алфавита, по 1 байту на символ — и эта попытка завершается неудачей, потому что каждый из этих символов занимает 2 байта.

```
for index, currentByte := range asciiBytes {
      fmt.Printf("%d: %s\n", index, string(currentByte))
}
for index, currentByte := range utf8Bytes {
      fmt.Printf("%d: %s\n", index, string(currentByte))
}
```

Обрабатывает каждый байт в сегменте.

Преобразует байт в строку и выводит результат.

Обрабатывает каждый байт в сегменте.

Преобразует байт в строку и выводит его.

Результат имеет печатное представление для ASCII...

```
0: A
1: B
2: C
3: D
4: E
0: Ð
1: 
2: Ð
3: 
4: Ð
5: 
6: Ð
7: 
8: Ð
9: 
```

...но не для Юникод!

В Go цикл `for...range` может использоваться со строками; в этом случае строка обрабатывается по *рунам*, а не по *байтам*. Такое решение намного безопаснее. Первой переменной будет присвоен индекс текущего байта (не индекс руны) внутри строки. Второй переменной будет присвоена текущая руна.

Ниже приведена обновленная версия кода, в которой цикл `for...range` используется для обработки самих строк, а не их байтовых представлений. Из индексов в выводе видно, что символы латинского алфавита обрабатываются по 1 байту, но символы русского алфавита обрабатываются по *2 байта*.

```
for position, currentRune := range asciiString {
      fmt.Printf("%d: %s\n", position, string(currentRune))
}
for position, currentRune := range utf8String {
      fmt.Printf("%d: %s\n", position, string(currentRune))
}
```

Обрабатывает каждую руну в строке.

Преобразует руну в строку и выводит ее.

Обрабатывает каждую руну в строке.

Преобразует руну в строку и выводит ее.

Все символы имеют печатное представление.

```
0: A
1: B
2: C
3: D
4: E
0: Б
2: Г
4: Д
6: Ж
8: И
```

Руны Go позволяют легко и удобно работать с неполными строками, не беспокоясь о том, содержат они символы Юникода или нет. Запомните: каждый раз, когда вам приходится работать с частью строки, преобразуйте ее в руны, а не в байты!

№ 5. Буферизованные каналы

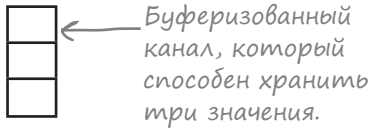
В Go существуют две разновидности каналов: *буферизованные* и *небуферизованные*.

Все каналы, упоминавшиеся до настоящего момента, были небуферизованными. Когда горутина отправляет значение в небуферизованный канал, она немедленно блокируется до того, как значение будет получено другой горутиной. С другой стороны, буферизованные каналы могут вмещать определенное количество значений, прежде чем отправляющая горутина будет заблокирована. В подходящей ситуации это может улучшить быстродействие программы.

Чтобы создать буферизованный канал, передайте `make` второй аргумент с количеством значений, которые должны помещаться в буфере канала.

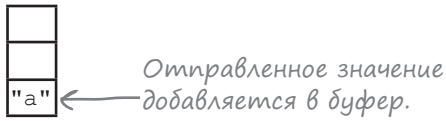
```
channel := make(chan string, 3)
```

Аргумент определяет размер канала буфера.



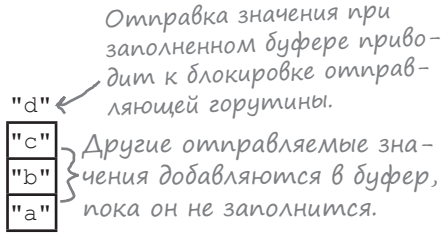
Когда горутина отправляет значение по каналу, это значение добавляется в буфер. Отправляющая горутина не блокируется, а продолжает работу.

```
channel <- "a"
```



Отправляющая горутина может продолжить отправлять значения по каналу, пока буфер не будет заполнен; и только тогда следующая операция отправки приведет к блокировке горутины.

```
channel <- "b"
channel <- "c"
channel <- "d"
```



Когда другая горутина получает значение из канала, она извлекает из буфера значение, которое было добавлено раньше других.

```
fmt.Println(<-channel)
```

"a"

Дальнейшие операции получения продолжают опустошать буфер, а дальнейшие операции отправки снова заполняют его.

```
fmt.Println(<-channel)
```

"b"

№ 5. Буферизованные каналы (продолжение)

Попробуем запустить программу с небуферизованным каналом, а затем переключимся на буферизованный канал, чтобы вы увидели различия. Ниже определяется функция `sendLetters`, которая должна выполняться в форме горутины. Она отправляет по каналу четыре значения, делая паузу продолжительностью в 1 секунду перед каждым значением. В `main` мы создаем небуферизованный канал и передаем его `sendLetters`. Затем горутина `main` приостанавливается на 5 секунд.

```
func sendLetters(channel chan string) {
        time.Sleep(1 * time.Second)
        channel <- "a"
        time.Sleep(1 * time.Second)
        channel <- "b"
        time.Sleep(1 * time.Second)
        channel <- "c"
        time.Sleep(1 * time.Second)
        channel <- "d"
}

func main() {
        fmt.Println(time.Now())
        channel := make(chan string)
        go sendLetters(channel)
        time.Sleep(5 * time.Second)
        fmt.Println(<-channel, time.Now())
        fmt.Println(<-channel, time.Now())
        fmt.Println(<-channel, time.Now())
        fmt.Println(<-channel, time.Now())
        fmt.Println(time.Now())
}
```

Получает канал в параметре.

Отправляет четыре значения, ожидая 1 секунду перед каждым.

Выводит время запуска программы.

Создает небуферизованный канал так, как это делалось ранее.

Запускает sendLetters в новой горутине.

Приостанавливает горутину main на 5 секунд.

Получает и выводит четыре значения с текущим временем.

Выводит время завершения программы.

Здесь запускается программа.

Первое значение ожидает отправки, пока запускается горутина main.

Но горутина sendLetters блокируется, пока получено первое значение. Нужно ждать, пока не будет получено позднее значение.

```
2018-07-21 11:36:20.676155577 -0700 MST m=+0.000255509
a 2018-07-21 11:36:25.677846276 -0700 MST m=+5.001810208
b 2018-07-21 11:36:26.677931968 -0700 MST m=+5.001895900
c 2018-07-21 11:36:27.679233609 -0700 MST m=+6.003129541
d 2018-07-21 11:36:28.680125059 -0700 MST m=+7.004020991
2018-07-21 11:36:28.680236070 -0700 MST m=+7.004132001
```

Программа выполняется 8 секунд.

Когда горутина `main` пробуждается, она получает четыре значения из канала. Но горутина `sendLetters` была заблокирована, ожидая, пока `main` получит первое значение. Таким образом, горутина `main` должна ждать 1 секунду между каждым оставшимся значением, пока горутина `sendLetters` перехватит его.

№ 5. Буферизованные каналы (продолжение)

Чтобы немного ускорить выполнение программы, можно добавить в канал буфера еще одно значение.

Для этого достаточно добавить второй аргумент при вызове `make`. В остальном взаимодействия с каналом остаются неизменными, так что вносить другие изменения в код не придется.

Теперь когда горутина `sendLetters` отправит свое первое значение в канал, она не блокируется до того момента, как это значение будет получено горутиной `main`. Отправленное значение вместо этого попадает в буфер канала. Только при отправке второго значения (при том, что ни одного значения еще не было получено) буфер канала заполняется, а горутина `sendLetters` блокируется. Добавление буфера, рассчитанного на одно значение, сокращает время выполнения программы на 1 секунду.

```
func main() {
        channel := make(chan string, 1)
        // Далее без изменений
}
```

Создает буферизованный канал, в котором может храниться одно значение.

Первый отправленный элемент попадает в очередь буферизованного канала.

После того как очередь будет заполнена, следующая отправка приводит к блокировке горутины sendLetters.

```
2018-07-21 15:29:10.709656836 -0700 MST m=+0.000318261
a 2018-07-21 15:29:15.710058943 -0700 MST m=+5.000584368
b 2018-07-21 15:29:15.710105511 -0700 MST m=+5.000630936
c 2018-07-21 15:29:16.712044927 -0700 MST m=+6.002502352
d 2018-07-21 15:29:17.716495 -0700 MST m=+7.006883143
2018-07-21 15:29:17.716615312 -0700 MST m=+7.007004737
```

Выполнение программы занимает всего 7 секунд.

Увеличение размера буфера до 3 позволяет горутине `sendLetters` отправить три значения без блокировки. Она блокируется при последней отправке, но это происходит после завершения всех ее 1-секундных вызовов `Sleep`. Таким образом, когда горутина `main` активизируется через 5 секунд, она немедленно получает 3 значения, ожидающие в буферизованном канале, а также значение, отправка которого привела к блокировке `sendLetters`.

```
channel := make(chan string, 3)
```

Создает буферизованный канал, который может вместить три значения перед блокировкой.

Три значения ожидают в буферизованном канале.

Это значение приводит к блокировке sendLetters, но только после того, как она завершится.

```
2018-07-21 17:02:20.062202682 -0700 MST m=+0.000341112
a 2018-07-21 17:02:25.066350665 -0700 MST m=+5.004353095
b 2018-07-21 17:02:25.066574585 -0700 MST m=+5.004577015
c 2018-07-21 17:02:25.066583453 -0700 MST m=+5.004585883
d 2018-07-21 17:02:25.066588589 -0700 MST m=+5.004591019
2018-07-21 17:02:25.066593481 -0700 MST m=+5.004595911
```

Выполнение программы занимает всего 5 секунд.

Программа завершается всего за 5 секунд!

№ 6. Дополнительные ресурсы

Книга подошла к концу. Тем не менее это всего лишь начало вашего пути как программиста Go. Мы хотим порекомендовать несколько ресурсов, которые вам помогут.

Веб-сайт Head First Go
https://headfirstgo.com/

Официальный веб-сайт этой книги. Здесь вы сможете загрузить все примеры кода, поэкспериментировать с дополнительными упражнениями и получить информацию по новым темам.

A Tour of Go
https://tour.golang.org

Интерактивный учебный курс основных возможностей языка Go. В него включена большая часть материала книги, но также имеются некоторые дополнительные темы. Примеры можно редактировать и запускать из браузера (так же, как в среде Go Playground).

Effective Go
https://golang.org/doc/effective_go.html

Руководство по написанию идиоматичного кода Go (то есть кода, следующего общепринятым соглашениям), сопровождением которого занимается команда разработки Go.

The Go Blog
https://blog.golang.org

Официальный блог Go. Содержит полезные статьи об использовании Go, а также анонсы о новых версиях и возможностях Go.

Документация пакетов
https://golang.org/pkg/

Документация по всем стандартным пакетам. Здесь находится та же документация, которую можно прочитать командой `go doc`, но все библиотеки объединены в один удобный список для просмотра. Например, можно начать с пакетов `encoding/json`, `image` и `io/ioutil`.

The Go Programming Language
https://www.gopl.io/

Эта книга — единственный небесплатный ресурс на этой странице, однако она того стоит. Книга хорошо известна и востребована разработчиками.

Все книги по языкам программирования можно условно разделить на две категории: учебники (вы как раз читаете такую книгу) и справочники (как *The Go Programming Language*). И это отличный справочник: в нем рассмотрены многие темы, на которые нам не хватило места. Если вы собираетесь продолжить работать на Go, эта книга обязательна для прочтения.

Джей Макгаврен

Head First. Изучаем Go

Перевел с английского *Е. Матвеев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*М. Петруненко*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Б. Файзуллин*
Верстка	*Н. Лукьянова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 02.2020. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 17.02.20. Формат 84×108/16. Бумага писчая. Усл. п. л. 57,120. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает
профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Qiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы,(адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**