# GO от первой строки до книжного магазина

**JUNIOR · MIDDLE · SENIOR**

Практическая книга: один пример,
который вырастает в работающий магазин

SHANRAQ.ORG

Баймурзин Даулет Абузарович

# Go: от первой строки до книжного магазина

Один проект: от основ Go до работающего магазина цифровых книг

Junior · Middle · Senior

Рабочая редакция 0.4-draft

shanraq.org
12 сентября 2026 года

# Выходные сведения

**Баймурзин Даулет Абузарович**
Go: от первой строки до книжного магазина: практическая книга. — shanraq.org, 2026. — Электронное издание. Рабочая редакция 0.4-draft.

Практическая книга о Go. Читатель шаг за шагом создаёт магазин цифровых книг: от первой программы и каталога до заказов, выдачи файлов и эксплуатации приложения. Материал организован в три уровня и использует опорные образы, проверяемые примеры и самостоятельные задания.

Дополнительный учебный материал для студентов бакалавриата ИТ-университетов, технических и многопрофильных вузов по направлениям «Программная инженерия», «Компьютерные науки», «Информационные системы», «Вычислительная техника и программное обеспечение». Отдельные разделы полезны студентам информационной безопасности и прикладной информатики, преподавателям практических дисциплин и разработчикам, изучающим Go самостоятельно.

**Автор:** Баймурзин Даулет Абузарович.
По образованию — инженер-технолог по ОПИ, по призванию — инженер-программист.
**Издательский проект:** shanraq.org.
**Дата этой редакции:** 12 сентября 2026 года.
**Язык:** русский.
**Форматы:** PDF, EPUB, HTML.

Этот файл содержит предисловие и главы 1–8. Полная книга ещё готовится. Дата относится к рабочей редакции, а не к завершённому коммерческому выпуску. ISBN и библиотечные классификационные индексы в этой редакции не указаны.



**Сайт:** shanraq.org.
**Код книги:** github.com/dake-edu/gopher-shop.

# Оглавление

# Предисловие. Магазин начинается с одной книги

Вы открыли книгу о Go. К её последней главе у вас будет магазин электронных книг. Первым товаром станет та самая книга, которую вы сейчас читаете.

Это удобно для учёбы: у каждой новой возможности есть понятная причина. Строка хранит название. Срез собирает каталог. Функция считает стоимость. Сервер показывает страницу. База помнит заказ после перезапуска. Проверка доступа решает, кому можно отдать файл.

Но магазин начинается раньше сервера. Он начинается с обещания: человек выбирает книгу, оплачивает её и получает возможность читать. Наша работа — написать программу, которая выполняет это обещание даже тогда, когда что-то идёт не по плану.

## Зачем учиться, если ИИ уже пишет код

Возможно, перед покупкой вы задали себе именно этот вопрос. Он разумный. Если помощник за несколько секунд выдаёт целый файл, зачем разбираться в каждой скобке?

Представьте, что вы попросили его сделать кнопку «Купить». Кнопка появилась. Вы нажали её — открылось «Спасибо за покупку». Всё готово? А если деньги не поступили? Если покупатель нажал дважды? Если по ссылке можно скачать чужую книгу? Красивый экран ещё не отвечает на эти вопросы. Чтобы принять работу, нужно понимать, какое поведение вы заказали и чем можете его проверить.

Мы пишем эту книгу с убеждением: развитие ИИ даёт человеку, умеющему программировать, больше способов воплощать собственные идеи. Можно задумать небольшой сервис для знакомой вам задачи, собрать первую версию, показать людям и узнать, нужна ли она им. Вам уже не обязательно ограничиваться ролью заказчика, который умеет лишь описать желаемый экран. Вы можете участвовать в создании продукта и менять его сами.

Из таких небольших попыток могут вырасти новые компании. Некоторые найдут решение удобнее существующего. Но заранее обещать успех каждому стартапу или пожизненную занятость каждому программисту было бы нечестно. Важны сама задача, люди, которым она нужна, качество решения и способность продолжать работу после первой неудачи. Заголовки об увольнениях не позволяют заранее решить, как сложится ваш путь.

Поэтому наша цель конкретнее обещания профессии «навсегда»: дать вам возможность самостоятельно создавать, проверять и развивать работающие программы. ИИ может участвовать в этой работе. Умение читать код поможет заметить неверное предположение, задать точный вопрос и понять предложенное исправление.

Попробуйте такой порядок: сначала предскажите поведение маленького примера сами, затем попросите помощника объяснить его, после чего проверьте оба ответа запуском. Если помощник предлагает сразу заменить десять файлов, попросите выделить одно изменение и показать проверку. Если объяснение вам непонятно, это хороший повод уменьшить пример. Не нужно соглашаться с кодом только потому, что он написан уверенным тоном.

К концу нашего пути у вас будет собственный магазин — предмет разговора с пользователем, а не только строка в резюме. Начать можно с одной книги и одной честно выполненной покупки.

## Образ, который можно унести с собой

Закройте на секунду текст и представьте небольшой книжный стол: на нём лежит книга, рядом стоит ценник, под столом хранится тетрадь заказов. Покупатель выбирает книгу; продавец проверяет оплату и отдаёт экземпляр. В этой картине уже есть будущие данные, действия и правила нашего магазина.

Мы будем возвращаться к знакомым предметам: ценнику, карточке, полке, записанному адресу. Это опорные образы. Их задача — помочь вам вспомнить связь между частями программы. Адрес на бумаге, например, напоминает, что копирование адреса не создаёт вторую квартиру. В главе об указателях мы проверим похожее различие на данных книги.

Образ всегда имеет границу. Компьютерная память не устроена как квартира, а срез Go не является деревянной полкой. После знакомства с образом мы назовём точное правило, запустим пример и найдём случай, в котором бытовая подсказка уже недостаточна.

Так мы используем приёмы опорных сигналов В. Ф. Шаталова: сначала видим целое, потом разбираем детали, сжимаем их до понятной опоры и восстанавливаем смысл самостоятельно. Маленький рисунок на полях полезен только тогда, когда по нему вы можете объяснить действие. Художественные способности для этого не нужны: прямоугольник, стрелка и два слова вполне подойдут.

## Что мы построим

Посетитель увидит каталог и откроет описание. Покупатель создаст заказ и перейдёт к оплате. После подтверждения книга появится в его библиотеке. Он сможет вернуться завтра и скачать её снова.

У автора будет возможность добавить книгу, исправить описание и выпустить новую версию файла. При этом изменение сегодняшней цены не должно переписать вчерашнюю покупку. А человек, который подобрал адрес чужого заказа, не должен получить чужую библиотеку.

Сначала всё будет происходить на вашем компьютере. Для первых опытов нам не нужны настоящие деньги. Позже появится тестовая платёжная система: мы научимся обрабатывать успех, отказ и ситуацию, когда ответа нет. Подключение реальной оплаты требует отдельной настройки выбранного провайдера; учебный ответ «успешно» не означает, что деньги поступили.

## Три ступени одной работы

На первой ступени вы освоите основы языка и соберёте каталог. Мы будем останавливаться на вещах, которые опытный разработчик делает автоматически: где находится файл, что означает сообщение компилятора, почему изменение значения не всегда меняет исходные данные.

На второй ступени приложение начнёт хранить данные и различать пользователей. Здесь появятся заказы, сессии, платежи и файлы. Вам потребуется объяснять не только удачный путь, но и отказ: что произойдёт, если запрос пришёл второй раз или база не ответила.

На третьей ступени мы будем проверять решения под нагрузкой и при сбоях. Научимся находить причину задержки, завершать работу без брошенных задач, восстанавливать данные и выпускать изменения. Сложность здесь определяется последствиями решений, а не количеством установленных технологий.

Эти ступени называются Junior, Middle и Senior. Это ориентиры глубины работы. Прочитанная глава сама по себе не присваивает квалификацию. Более полезная проверка — сумеете ли вы объяснить решение и повторить его в задаче, которую ещё не видели.

## Кому адресована книга

Книга предназначена для самостоятельного изучения Go и использования в качестве дополнительного учебного материала на практических занятиях и в учебных проектах вуза. Начать можно без опыта программирования: первая ступень объясняет инструменты, запись языка и работу небольших программ. Если вы уже пишете на другом языке, первые главы помогут связать знакомые задачи с правилами Go.

В высшем образовании книга прежде всего адресована студентам бакалавриата ИТ-университетов, технических университетов и ИТ-факультетов многопрофильных вузов. Особенно близки нашему проекту следующие направления:

| Направление или специальность | Для каких учебных задач пригодится книга |
|---|---|
| Программная инженерия / Software Engineering | Разработка приложения, разделение ответственности, тестирование, совместная работа с кодом |
| Компьютерные науки / Computer Science | Практика языка, структуры данных, работа с памятью и конкурентное выполнение |
| Информационные системы | Проектирование каталога и заказов, базы данных, серверные интерфейсы и права доступа |
| Вычислительная техника и программное обеспечение | Создание и сопровождение серверных программ, сетевое взаимодействие, инструменты сборки |
| Прикладная информатика | Перевод требований пользователя в данные и поведение работающего продукта |
| Информационная безопасность / кибербезопасность | Практика проверки входных данных, управления доступом и безопасной разработки веб-приложений |

Таблица описывает назначение полной программы книги. В этой рабочей редакции доступны основы языка и первые шаги каталога; разделы о базе данных, доступе и эксплуатации ещё готовятся. Книга не заменяет весь учебный план: математические основы, общие алгоритмы, теория вычислений и специальные дисциплины изучаются также по профильным источникам.

### Примеры образовательных программ в Казахстане

Для ориентира сопоставим проект книги с реально существующими программами. Названия и коды проверены по официальным страницам вузов 12 сентября 2026 года; оценка учебной пользы в последнем столбце — авторская.

| Вуз | Программа | Возможное применение книги |
|---|---|---|
| Astana IT University | 6B06102 Software Engineering | Практикум по разработке ПО и веб-приложений; сквозной проект магазина |
| Международный университет информационных технологий — МУИТ / IITU | 6B06105 «Информационные системы» | Проектирование и реализация информационной системы с каталогом, заказами и хранением данных |
| Satbayev University | Computer Science | Практика программирования на Go и инженерная часть учебного проекта |

Этот перечень можно продолжить другими вузами с близкими направлениями. Включение книги в конкретную дисциплину и требования к работе определяет преподаватель или кафедра.

Для курса основ программирования подойдут первые контрольные точки и обязательные задания. Для проектного семинара — развитие магазина с объяснением принятых решений. В дипломной работе учебную основу потребуется расширить собственными требованиями, реализацией и проверками, явно указав использованные материалы.

Если вы учитесь на другой специальности или уже работаете инженером, путь для вас тоже открыт. Начните с первых глав и собственной задачи. Здесь важнее готовность проверить своё объяснение запуском программы, чем название диплома.

## Как читать и практиковаться

Перед запуском попробуйте угадать результат. Запишите ответ одной строкой. Затем запустите программу и сравните. Если ответы разошлись, у вас появилось точное место для разбора.

После примера измените одно условие. Уберите книгу из каталога. Поставьте нулевое количество. Передайте неизвестный идентификатор. Умение пользоваться программой начинается с понимания того, где она должна отказать.

Затем выполните обязательное задание без готового решения. Если застряли, прочитайте первую подсказку. Решение стоит открывать после собственной попытки: оно нужно для сравнения подходов и разбора ошибок.

Вопросы «Скажите своими словами» тоже часть работы. Можно ответить вслух, в заметках или объяснить товарищу. Если для объяснения приходится перечитать абзац, вернитесь к небольшому примеру и проверьте его ещё раз.

## Первое упражнение: обещание покупателю

Пока без кода. Рассмотрим три события:

1. Человек нажал кнопку «Купить».

2. В браузере открылась страница «Спасибо».

3. Сервер получил и проверил подтверждение оплаты.

После какого события можно считать оплату подтверждённой?

Не спешите выбирать ответ только по названию страницы. Браузер показывает то, что ему отдали, а переход по ссылке сам по себе ничего не говорит о движении денег.

**Ответ.** Основанием служит проверенное подтверждение платёжной системы, связанное с нужным заказом, суммой и валютой. Нажатие кнопки выражает намерение. Страница благодарности сообщает результат пользователю, но не доказывает его серверу.

Теперь продолжение: подтверждение пришло дважды. Должен ли магазин создать два заказа или выдать две независимые покупки? Нет. Повтор одного события должен сохранять уже достигнутый результат. Позже мы назовём это идемпотентностью и проверим тестом.

Пока достаточно заметить: одно короткое требование к поведению приводит нас к устройству данных, обработке ошибок и проверкам. Так будет построена вся книга.

## Как понять, что вы продвигаетесь

В конце главы должно измениться что-то наблюдаемое. Появилась карточка. Неверная форма получила объяснение. Заказ сохранился после перезапуска. Чужой файл остался недоступен. Программа завершилась и не потеряла принятую работу.

Если пример запускается только после необъяснённых исправлений, это недостаток материала, а не ваша обязанность угадывать. Сохраняйте команду, сообщение, версию Go и номер контрольной точки. Эти сведения позволяют воспроизвести проблему.

Наш следующий шаг маленький: подготовить рабочую папку и вывести название книги в терминал. У этого вывода ещё не будет покупателей. Но уже будет важное свойство — вы сможете запустить его сами и объяснить каждую строку.

# 1. Рабочее место, которое можно проверить

В конце этой главы у вас будет папка проекта и терминал, в котором работает команда go `version`. Пока не нужно выбирать базу данных или оплачивать сервер. Сначала убедимся, что можем сохранять файлы и запускать инструменты.

## Образ: свой рабочий стол

Представьте стол, на котором вы собираетесь написать письмо. На нём лежит папка, внутри — листы; у папки есть название. Если оставить письмо на соседнем столе, открытая перед вами папка его не получит. С программой происходит похожая путаница: вы исправили один `main.go`, а терминал запускает другой.

Наша опора — **стол, папка, сохранённый лист**. Рабочая папка объединяет файлы проекта. Текущая папка терминала определяет, где команда начинает работу. Сохранение переносит набранный в редакторе текст в файл, который прочитает инструмент Go. Редактор может показывать несколько проектов, поэтому взгляд на вкладку ещё не доказывает, что терминал открыт там же.

У образа есть граница: на компьютере команды могут принимать полный путь к чужой папке. Пока мы намеренно работаем из папки проекта. Нарисуйте прямоугольник с названием `bookshop`, а внутри — `go.mod`. Перед запуском показывайте пальцем: «Команду выполняю здесь». Это маленькая привычка, которая избавляет от долгих поисков не той ошибки.

## Сначала целое

Наша работа будет устроена так:

| Где | Что вы делаете | Что получаете |
|---|---|---|
| Редактор | Пишете и сохраняете файл `main.go` | Текст программы на диске |
| Терминал | Вводите `go run .` | Go собирает и запускает программу |
| Терминал | Читаете ответ | Проверяете, совпал ли результат с ожиданием |

Сохранённый файл и открытая вкладка редактора — не одно и то же. Если вы изменили текст, но не сохранили его, команда может запустить предыдущую версию. Эту причину стоит проверять прежде, чем переписывать программу.

Терминал — программа, принимающая команды. Оболочка внутри него разбирает эти команды. В macOS вы обычно встретите zsh, в Linux — bash, в Windows будем использовать PowerShell. Там, где команды различаются, книга покажет оба варианта.

## Установите Go

Откройте официальную страницу установки. Выберите свою операционную систему и процессор. Для Windows используется установщик MSI, для macOS — PKG; на Linux следуйте инструкции для архива вашей архитектуры. Не вставляйте в терминал команды удаления старой установки, если не понимаете, какой каталог они затронут.

Материалы этого рабочего выпуска проверяются на Go 1.27.1. Ваша версия указана в выводе команды, а не в названии скачанного файла. После установки закройте терминал и откройте новый: так он получит обновлённые настройки поиска программ.

Введите только команду, без символов приглашения вроде $ или >:

```
go version
```

Форма ответа:

```
go version go1.27.1 darwin/arm64
```

Последняя часть зависит от компьютера. `darwin` означает macOS, `arm64` — архитектуру процессора. На Windows или Linux она будет другой. Здесь важно, что вы получили ответ программы Go, а не сообщение «команда не найдена».

Если Go установлен раньше, не удаляйте его наугад. Сначала посмотрите версию и сверьтесь с инструкцией обновления. Работающий чужой проект может зависеть от своего набора инструментов.

## Подготовьте папку

Выберите место для учебных проектов, к которому у вас есть доступ. Не создавайте новый магазин внутри другого проекта. Откройте терминал в выбранной папке и выполните:

```
mkdir bookshop
cd bookshop
```

Эти две команды подходят и для PowerShell, и для bash/zsh. Первая создаёт папку, вторая делает её текущей: следующие команды будут работать относительно неё.

Если `bookshop` уже существует, сначала посмотрите, что внутри. Для чистого начала можно выбрать другое имя, например `bookshop-study`; тогда используйте его и в команде `cd`.

Проверьте, где вы находитесь. В macOS/Linux:

```
pwd
ls
```

В PowerShell:

```
Get-Location
Get-ChildItem
```

Первая команда показывает путь, вторая — содержимое. Не надо добиваться точного совпадения пути с книгой: у каждого своя домашняя папка. Важно увидеть выбранную вами папку `bookshop`.

Откройте её в редакторе через меню открытия папки. Подойдёт редактор обычного текста, который сохраняет UTF-8. Если используете VS Code, открывайте всю папку, а не случайный файл из соседнего проекта. Для первых глав дополнительные расширения необязательны.

## Дайте проекту имя

В терминале внутри `bookshop` выполните один раз:

```
go mod init example.com/bookshop
```

Появится `go.mod`. Модуль — группа пакетов Go с общим описанием версии и зависимостей. Пока достаточно понимать: этот файл обозначает корень нашего проекта. Запись `example.com/bookshop` служит именем учебного модуля; для этого упражнения не нужно покупать домен или создавать сайт по такому адресу.

Код книги связан с репозиторием dake-edu/gopher-shop. Контрольные точки находятся в `book/examples/`; их имена модулей отражают путь в репозитории. Для собственного проекта мы используем учебное имя `example.com/bookshop`, чтобы отличать его от готовых примеров.

Не редактируйте `go.mod` ради буквального совпадения с изображением в чужом уроке: версия в нём зависит от инструмента, который его создал. В комплекте книги лежат самостоятельные контрольные точки со своими готовыми `go.mod`; для них повторный `go mod init` не нужен.

## Если не получилось

| Симптом | Что проверить |
|---|---|
| `go` не найден | Установка завершена? Открыт новый терминал? Найден ли каталог Go через PATH? |
| `go.mod already exists` | Модуль уже создан; повторная инициализация не нужна |
| Отказ в создании папки | Выберите доступную папку пользователя |
| Редактор показывает другие файлы | Сравните открытую папку с выводом `pwd` или `Get-Location` |

PATH — список папок, где оболочка ищет исполняемые программы. Он похож на список адресов доставки, но саму программу не устанавливает. Если адрес записан верно, а файла по нему нет, поиск всё равно не сработает.

## Опорные пункты: команда состоит из частей

В `go mod init example.com/bookshop` пробелы отделяют имя программы `go`, подкоманду `mod`, действие `init` и имя модуля. Слеш `/` здесь является частью имени модуля, а не командой перехода в папку. В `go run .` последняя точка — отдельный аргумент «текущий пакет».

Символы приглашения терминала (`$`, `>`, иногда `%`) перед командой не набирают. Они принадлежат оболочке. Круглых или фигурных скобок в этих командах нет: это команды инструменту, а не исходный код Go.


Опора: сохранённый файл, команда Go и наблюдаемый результат

**Восстановите по памяти.** Закройте пример и назовите три места: где редактируем, что запускаем, где проверяем результат. Если путаются путь и имя модуля, вернитесь к выводу текущей папки и откройте `go.mod`.

## Разминка

**Предскажите.** Если создать файл в папке A, а команду запустить из папки B, обязана ли команда увидеть этот файл?

**Заполните.** Какая команда меняет текущую папку на `bookshop`?

**Почините.** Редактор сохранил файл в `bookshop-old`, а терминал открыт в `bookshop`. Как проверить расхождение, не переустанавливая Go?

## Задание

**Обязательное.** Создайте рабочую папку и модуль. Сохраните в `environment.txt` вывод `go version` и путь к папке. Проверьте в редакторе, что рядом существует `go.mod`.

**На своих данных.** Назовите папку по-своему и повторите проверку пути. Имя папки и имя модуля не обязаны совпадать.

**По желанию.** Найдите путь к установленной команде Go: `command -v go` в bash/zsh или `Get-Command go` в PowerShell.

## Ответы и самопроверка

Команда работает в своей текущей папке и не обязана искать файл в соседней. Для перехода используется `cd bookshop`. Чтобы найти расхождение, сравните путь терминала с местом сохранения файла в редакторе.

Вы готовы двигаться дальше, если можете объяснить, где лежит проект, показать `go.mod` и получить версию Go. Следующая глава даст этой папке первую программу.

# 2. Книга появляется в терминале

У нас есть рабочая папка и файл `go.mod`. Теперь напишем программу, которая выводит название книги и три формата её будущего издания. Результат маленький, но проверяемый: две строки в нужном порядке.

Контрольная точка: `examples/02-first-program`. Если начинаете с неё, откройте эту папку и сразу выполняйте команды запуска. Если пишете сами, продолжайте в своей `bookshop`.

## Образ: записка с двумя поручениями

Положите перед собой записку: «Покажи название книги. Затем покажи форматы». Если читать её сверху вниз, порядок понятен. Если поменять поручения местами, сначала появятся форматы. Именно такой небольшой порядок действий мы сейчас зададим программе.

На нашей опорной записке будет три строки: **пакет — нужный инструмент — поручения**. `package main` связывает файл с исполняемым пакетом; `import "fmt"` делает доступным инструмент вывода; внутри `main` стоят два вызова `fmt.Println`. Фигурные скобки можно мысленно обвести рамкой вокруг поручений.

Только не переносите образ дальше нужного: компьютер не понимает просьбы «покажи красиво», пока мы не запишем конкретное действие. И в больших программах порядок может быть сложнее простого чтения сверху вниз. Здесь мы проверяем ровно два последовательных вызова. После запуска спрячьте исходник и восстановите свою записку: что принадлежит пакету, что подключено, что выполняется?

## Сразу целиком

Создайте файл `main.go` рядом с `go.mod` и сохраните в нём:

```
package main

import "fmt"

func main() {
    fmt.Println("Go: от первой строки до книжного магазина")
    fmt.Println("Форматы: PDF, EPUB, HTML")
}
```

Перед запуском запишите предполагаемый вывод. Будут ли видны кавычки? Появится ли пустая строка между названием и форматами?

В терминале в этой же папке:

```
go run .
```

Результат:

```
Go: от первой строки до книжного магазина
Форматы: PDF, EPUB, HTML
```

Точка означает текущий пакет в текущей папке. Команда собирает программу и запускает её. Здесь программа заканчивает работу после двух вызовов печати — это ожидаемое поведение, а не ошибка.

## Цвет помогает найти знакомое

В книге исходник подсвечен как в редакторе. Та же запись сохраняет свои цвета в предложении: например, `import "fmt"`. Посмотрите сначала на строку программы, затем на её объяснение ниже — вам не придётся заново искать, какая часть относится к языку, а какая содержит данные.

| Пример | Цвет и подсказка |
|---|---|
| `package` | Оранжевый: объявляем, к какому пакету относится файл |
| `import` | Розовый: подключаем другой пакет |
| `func` | Синий: объявляем функцию или метод |
| `var`, `const`, `type`, `if`, `for`, `return` | Фиолетовый: другие ключевые слова; точное действие определяем по самому слову |
| `priceMinor`, `fmt`, `Println` | Тёмный сине-голубой: имена; их роль определяем по месту в программе |
| `"Книга"` | Зелёный: строковые данные внутри кавычек |
| `42` | Коричневый: число, записанное прямо в исходнике |
| `// название книги` | Серый: комментарий для человека |
| `{`, `}` | Малиново-фиолетовый: границы блока, а позднее — также записи составного типа или значения |
| `(`, `)`, `[`, `]` | Тёмно-фиолетовый: круглые и квадратные скобки; назначение зависит от места |
| `.`, `,`, `:=`, `+`, `-` | Серый: остальные разделители и операторы |

В первой программе можно прочитать цветовую опору так: оранжевый — принадлежность пакету, розовый — подключение, синий — объявление функции, малиново-фиолетовые скобки — границы её тела. Эти цвета повторяются в поясняющих предложениях.

Пока не нужно учить новые слова из таблицы: к каждому мы вернёмся по делу. Зелёная строка `"Книга"` хранит данные. А сине-голубой `priceMinor` — имя, по которому программа обращается к значению. Цвет помогает быстро заметить это различие.

Сине-голубой не означает «это обязательно переменная»: имена пакетов и функций в нашей палитре тоже сине-голубые. Предопределённые имена, например `int64`, `true` и `nil`, тоже сине-голубые: они не являются ключевыми словами Go. Что обозначает каждое из них, разберём при первом использовании. Подсветка разбирает запись, но не доказывает, что программа правильная.

В вашей IDE — редакторе со средствами разработки — цвета могут отличаться: их выбирает тема оформления. У языка Go нет обязательного цвета для переменной или строки. В читалке с собственными настройками и на чёрно-белом экране краски тоже могут измениться или

исчезнуть. Кавычки, слова, подписи и наши объяснения сохраняют смысл без цвета.

**Попробуйте глазами.** Найдите в первой программе слово `import`, затем строку `"fmt"`. Назовите их назначение без слов «розовое» и «зелёное». Если получается, цвет уже работает как подсказка, а не как единственный способ понять запись.

## Разбор по строкам

`package main` сообщает, что файл относится к пакету `main`. Из такого пакета можно собрать исполняемую программу. Одного объявления пакета недостаточно: программе нужна функция `main`.

`import "fmt"` подключает пакет стандартной библиотеки для форматирования и вывода. Нам не пришлось отдельно скачивать библиотеку: она поставляется вместе с Go.

`func main()` объявляет функцию с именем `main`. Скобки после имени относятся к параметрам; у нашей функции их нет. Фигурные скобки ограничивают тело функции — команды, которые она выполняет. Подробно параметры разберём в главе 5. Перед `main` Go выполняет инициализацию программы, поэтому запоминать правило «main всегда первая строка исполнения» не нужно.

`fmt.Println` означает: взять функцию `Println` из пакета `fmt` и вызвать её. Текст в двойных кавычках — строковое значение. Кавычки отделяют текст программы от её данных, поэтому сами в вывод не попадают. `Println` добавляет перевод строки после вывода.

Внутри `main` команды выполняются последовательно. Поменяете вызовы местами — поменяется и порядок строк. Go пока не знает ни о книге как товаре, ни о форматах как файлах. Мы передали функции печати обычный текст.

## Оформление без споров

Введите:

```
gofmt -w main.go
```

`gofmt` приводит код к стандартному оформлению; `-w` записывает результат в файл. Это может изменить отступы. Форматирование не проверяет, правильную ли книгу вы назвали, и не заменяет запуск программы.

Наберите лишние пробелы перед одним вызовом, сохраните и повторите `gofmt`. Затем снова запустите программу. Оформление изменится, результат останется прежним.

## Запуск и сборка

Когда нужен отдельный исполняемый файл, используется `go build`. Для macOS/Linux:

```
go build -o bookshop .
./bookshop
```

Для Windows PowerShell:

```
go build -o bookshop.exe .
.\bookshop.exe
```

Первая команда создаёт программу, вторая её запускает. Если после сборки изменить `main.go`, уже собранный файл сам не изменится: его нужно собрать ещё раз.

Этот пример не требует установки Go для запуска готового файла на совместимой системе. Но файл для macOS нельзя просто запускать как программу Windows: операционная система и архитектура сборки имеют значение. Позже мы также будем отдельно учитывать конфигурацию, данные и внешние сервисы приложения.

## Сломайте безопасно

Удалите закрывающую кавычку у названия книги. Сохраните и запустите `go run .`. Сборка завершится ошибкой. Прочитайте имя файла и номер строки; конкретное сообщение может различаться между версиями инструмента.

Верните кавычку и повторите запуск. Мы исправили причину в исходнике. Переустановка Go здесь ничего бы не дала.

## Читаем запись по символам

| Запись | Что означает именно здесь |
|---|---|
| `package main` | Файл принадлежит пакету с именем `main` |
| `import "fmt"` | Подключаем пакет по пути, записанному строкой |
| `func` | Начинается объявление функции |
| `main` | Имя функции входа в нашу программу |
| `()` после имени в объявлении | Список параметров пуст |
| `{` и `}` | Начало и конец тела функции |
| `fmt.Println` | Точка связывает имя пакета с именем доступной из него функции |
| `("текст")` при вызове | Внутри скобок передаётся аргумент — строка |
| " | Граница строкового литерала; сама кавычка не выводится |

Точка в `fmt.Println` и точка в `go run .` выполняют разные роли: первая относится к записи Go, вторая — к аргументам команды терминала. Пробелы и переносы между словами помогают разделить запись, а внутри кавычек являются частью текста.

В Go есть точки с запятой, но в обычном исходнике их чаще не пишут: по правилам языка они автоматически вставляются после определённых токенов в конце строки. Поэтому пока ставьте открывающую `{` функции на той же строке, что и `func main()`. Перенос этой скобки на следующую строку может изменить разбор программы; `gofmt` не исправит любой синтаксический дефект вместо вас.

## Программа: принадлежность, инструменты, действия


Скобки ограничивают список параметров и тело функции

Опора первой программы: пакет, подключение и тело main

**Восстановите по памяти.** Нарисуйте три блока, затем напишите программу заново без копирования. Если не помните скобку, найдите её пару и объясните, что она ограничивает.

## Разминка

**Предскажите.** Что изменится, если два вызова `Println` поменять местами?

**Заполните.** Какое имя пакета должно стоять перед точкой в вызове `____.Println("Книга")`?

**Почините.** Сообщение говорит `undefined: Fmt`. Найдите разницу между написанным именем и `fmt`.

## Задание

**Обязательное.** Добавьте третью строку `Статус: готовится`. Первые две строки должны остаться прежними. Сохраните файл, отформатируйте и запустите.

**На своих данных.** Замените название на название собственной будущей книги. Не меняйте список форматов: наш итоговый продукт должен поддерживать PDF, EPUB и HTML.

**По желанию.** Соберите программу, затем поменяйте название в исходнике. Сравните запуск готового файла и `go run .`. Объясните различие до повторной сборки.

## Скажите своими словами

Почему сохранение файла важно перед запуском? Что делает `import`? Почему `go build` не показывает название книги в терминале?

## Ответы

Порядок вызовов задаёт порядок строк. Нужен пакет `fmt`, с маленькой буквы: регистр символов различается. Дополнительная строка — ещё один вызов `fmt.Println("Статус: готовится")` внутри `main`, после двух существующих.

Сборка читает сохранённые файлы. `import` делает доступным указанный пакет. `go build` создаёт исполняемый файл и сам его не запускает.

В магазине появилась первая публичная информация — название и форматы. Пока она записана прямо в вызовах печати. В следующей главе отделим данные от действий над ними.

# 3. Название, цена и состояние книги

Мы уже умеем выводить две строки. Но цена будет участвовать в расчётах, а состояние публикации — определять доступность товара. Поэтому данные должны иметь подходящие типы.

Контрольная точка: `examples/03-values`. В своей рабочей папке замените `main.go` полным примером ниже. Цена учебная: это не объявленная стоимость будущей книги.

## Образ: ценник с подписанными местами

Возьмите воображаемый ценник. Сверху оставьте место для названия, ниже — для цены, сбоку — для отметки «показывать покупателю». Не нужно каждый раз переписывать весь ценник, чтобы поменять сумму. Достаточно найти нужное место по подписи.

Имя переменной помогает найти значение в программе. Тип задаёт, какие значения и действия допустимы. Строка названия и целочисленная цена решают разные задачи: мы не станем складывать название с процентом скидки. Изменение переменной меняет её текущее значение; `const` объявляет константу, которой новое значение присвоить нельзя.

На бумаге можно зачеркнуть всё что угодно. Правила языка строже: компилятор проверяет допустимость операций. Поэтому образ ценника помогает связать имя с назначением, а точный ответ о типе мы получаем из Go. Нарисуйте три подписанных места и, не глядя в код, впишите в каждое пример значения. Где можно поставить `false`, а где оно нарушит вашу задумку?

## Сразу целиком

```
package main

import "fmt"

func main() {
    const currency = "KZT"
    const minorPerUnit = 100
    var priceMinor int64 = 249900
    title := "Go: от первой строки до книжного магазина"
    published := false

    fmt.Println(title)
    fmt.Printf("Цена: %d.%02d %s\n", priceMinor/minorPerUnit, priceMinor%minorPerUnit, currency)
    fmt.Println("Опубликована:", published)
    published = true
    fmt.Println("Опубликована:", published)
}
```

Запустите `go run .` после того, как попробуете предсказать две последние строки.

```
Go: от первой строки до книжного магазина
Цена: 2499.00 KZT
Опубликована: false
Опубликована: true
```

Одна переменная участвовала в выводе дважды, и результат различается. Между вызовами мы поменяли её значение.

## Значение и его имя

`title := "..."` создаёт переменную `title` и помещает в неё строку. Go определяет тип по правой части. Короткое объявление `:=` применяется внутри функций.

`published := false` создаёт переменную типа `bool`. У логического типа два значения: `true` и `false`. Это удобно для вопроса с двумя ответами: опубликована книга или нет.

`published = true` меняет значение существующей переменной. Здесь стоит `=`, а не `:=`. Для повторного присваивания одному уже объявленному имени новое объявление не требуется.

`var priceMinor int64 = 249900` — более подробная форма: мы явно выбрали тип `int64`. Это знаковое целое фиксированной разрядности. Тип `int`, который часто используют для счётчиков и индексов, зависит от архитектуры. Для представления сумм в модели магазина мы выберем фиксированный целый тип.

Если объявить переменную без начального значения, Go задаст нулевое: для целого числа — `0`, для строки — пустую строку, для `bool` — `false`. Это правило языка, но не бизнес-решение. Не стоит считать нулевую цену автоматически разрешённой, пока магазин не определил правила бесплатных книг.

## Константа не меняется

`const currency = "KZT"` задаёт константу. Ей нельзя присвоить другое значение во время работы программы. В этом примере валюта одна, поэтому она неизменна.

Это не означает, что любой магазин обязан иметь одну валюту. Когда появится несколько валют, код валюты станет частью данных, а правила пересчёта и округления придётся определить отдельно.

`minorPerUnit` задаёт число минимальных денежных единиц в основной для нашего учебного представления KZT: сто. Имя помогает увидеть смысл числа в выражении. Число `100` без имени могло бы означать и процент, и ограничение длины, и количество товаров.

## Почему цена целая

Сумму `2499.00 KZT` мы храним как `249900` минимальных единиц. Такой способ позволяет выполнять расчёты целыми числами и явно решать, что делать с дробной минимальной единицей.

Тип `float64` полезен для приближённых вычислений, измерений и многих научных задач. Но многие десятичные дроби не представимы в двоичной форме точно. Для денежных правил нам важна воспроизводимая единица учёта, а не приближение, которое случайно выглядит правильно после печати.

Целое число тоже имеет предел. Сам по себе переход на `int64` не защищает от переполнения при умножении. В главе 5 мы ограничим входную цену до расчёта и проверим границы тестами. Сумму и валюту позже объединим в модель, чтобы не перепутать единицы.

## Как получилась точка и два нуля

В выражении `priceMinor / minorPerUnit` целочисленное деление даёт целую часть. В выражении с `%` получаем остаток. Для `249900` это `2499` и `0`.

`fmt.Printf` печатает по шаблону. В нашем шаблоне:

| Запись | Значение |
|---|---|
| `%d` | Целое десятичное число |
| `%02d` | Целое число шириной не менее двух позиций, с ведущим нулём |
| `%s` | Строка |
| `\n` | Перевод строки |

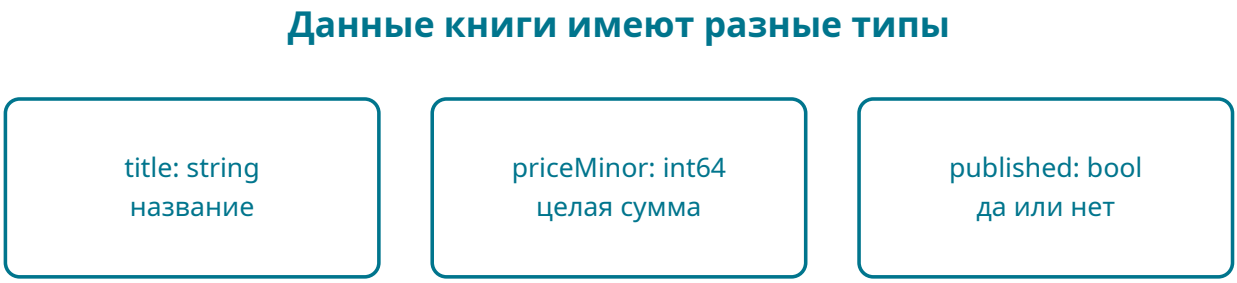
Поэтому остаток `0` становится `00`, а остаток `5` стал бы `05`. В отличие от `Println`, `Printf` не добавляет перевод строки автоматически: он записан в шаблоне.

Этот вывод рассчитан на неотрицательную цену. Он не является готовым форматировщиком отрицательных сумм или всех валют. Мы отделим такие правила, когда появится необходимость.

## Читаем новую запись по символам

| Запись | Как прочитать |
|---|---|
| `:=` | Объявить переменную и дать начальное значение; двоеточие вместе с равно — один оператор |
| `=` | Присвоить новое значение уже существующей переменной |
| `const` | Объявить константу |
| `var` | Объявить переменную |
| `int64`, `string`, `bool` | Имена типов, то есть допустимых видов значений |
| `249900` | Целочисленный литерал без кавычек |
| `false`, `true` | Логические значения без кавычек |
| `,` в `Printf(...)` | Разделитель аргументов одного вызова |
| `/`, `%` между числами | Деление и остаток от целочисленного деления |
| `%d` внутри кавычек | Часть формата `Printf`, а не оператор остатка |

Одни и те же видимые знаки читаются по месту. `%` снаружи строки участвует в арифметике Go; `%02d` внутри строки получает смысл, когда `Printf` читает шаблон. Обратный слеш в `\n` начинает управляющую последовательность строкового литерала, обозначающую перевод строки; это не два печатаемых символа.




Опора данных: название, целая сумма и состояние публикации

**Проверка понимания.** Покажите в примере присваивание, которое изменяет данные, и место, которое только задаёт вид вывода. Если `%` пока означает для вас одно и то же в обоих местах, перечитайте разбор цены.

## Разминка

**Предскажите.** Как будет выглядеть цена при `priceMinor = 105`?

**Заполните.** Для изменения опубликованной книги на черновик используйте `published ___ false`.

**Почините.** Кто-то написал `priceMinor = "249900"`. Почему это не то же самое, что число `249900`?

## Задание

**Обязательное.** Установите цену `350050`. Программа должна напечатать `Цена: 3500.50 KZT`. Вторая строка состояния должна оставаться `Опубликована: true`.

**На своих данных.** Возьмите положительную цену, у которой последние две цифры — `05`, и предскажите вывод перед запуском.

**По желанию.** Попробуйте присвоить константе `currency` другое значение. Прочитайте ошибку и отмените изменение.

## Скажите своими словами

Чем отличается объявление от присваивания? Почему сумма без валюты неполна? Почему целый тип не освобождает нас от проверки диапазона?

## Ответы

`105` минимальных единиц даст `1.05 KZT`. Для присваивания нужен `=`. Значение в кавычках имеет строковый тип, а `priceMinor` объявлена целой: это несовместимые типы для прямого присваивания.

В обязательном задании меняется только начальное значение цены. Не исправляйте шаблон вывода под конкретный ответ: он должен работать и для предыдущей цены.

У нашей книги появились данные трёх видов. Следующий шаг — принимать решение по этим данным, а не всегда печатать одно и то же.

# 4. Какие издания можно показать

Книга может быть черновиком, а одно из старых изданий — снятым с продажи. Напишем небольшую проверку, которая обходит три номера изданий и сообщает, какие доступны.

Контрольная точка: `examples/04-conditions`. Это упражнение над состоянием каталога. Оно не удаляет купленные книги из библиотеки покупателя: правила доступа к уже приобретённому изданию будут отдельными.

## Образ: дверь и три карточки

На столе лежат три карточки изданий. Перед витриной — дверь с условием: «Показывать только опубликованное». Сначала проверяем, открыта ли дверь для нашей книги. Затем берём карточки по одной. Второе издание снято с продажи — откладываем его и берём следующее.

В программе `if` выбирает действие по условию, а `for` повторяет шаги. `continue` в нашем обходе означает: закончить работу с текущей карточкой и перейти к следующему шагу цикла. `break` означает другое: завершить этот цикл целиком. Представьте, что после второй карточки вы либо тянетесь за третьей, либо убираете всю стопку. Разницу теперь можно увидеть до запуска.

Дверь не является отдельным объектом программы: это рисунок выбора. Само условие должно дать логическое значение. Нарисуйте дверь и три маленьких прямоугольника. Зачеркните второй, но оставьте третий на витрине. Какой переход нужен для такого результата? Проверьте свой ответ на нашем цикле.

## Сразу целиком

```
package main

import "fmt"

func main() {
    published := true
    for edition := 1; edition <= 3; edition++ {
        if !published {
            fmt.Println("Книга готовится")
            break
        }
        if edition == 2 {
            fmt.Println("Издание", edition, "снято с продажи")
            continue
        }
        fmt.Println("Доступно издание", edition)
    }
}
```

Сначала отметьте на бумаге, какие строки сработают для номера 2. Затем запустите `go run .`.

```
Доступно издание 1
Издание 2 снято с продажи
Доступно издание 3
```

Число 2 не потерялось: у него другой результат. Мы явно сообщили, что издание снято с продажи.

## Условие — вопрос с логическим ответом

`if edition == 2` проверяет равенство. Если условие истинно, выполняется блок в фигурных скобках. Один знак `=` присваивает значение; два знака `==` сравнивают значения. Перепутать их легко, поэтому читайте выражение словами: «номер издания равен двум?».

`!published` означает «не опубликована». Знак `!` меняет логическое значение на противоположное. Если книга ещё не опубликована, нет смысла показывать доступные издания этого учебного каталога.

Условия могут использовать `<`, `<=`, `>`, `>=` и `!=`. У равенства и сравнения тоже есть ограничения типов: не все значения языка сравниваются любым оператором. Сейчас мы работаем с целыми числами и логическим значением.

## Цикл: начало, проверка, шаг

В заголовке `for edition := 1; edition <= 3; edition++` три части.

Первая создаёт счётчик со значением 1 перед обходом. Вторая проверяется перед каждым выполнением тела: можно ли продолжать? Третья увеличивает счётчик на единицу после очередного завершённого прохода.

Получается последовательность: 1, 2, 3. Когда счётчик станет равен 4, условие `4 <= 3` окажется ложным, и цикл закончится. Переменная `edition` объявлена в заголовке цикла и недоступна после него. Это пример области видимости: имя существует не во всех местах файла.

`edition++` — отдельная команда увеличения, а не выражение, которое надо вкладывать в печать. Здесь она управляет переходом к следующему изданию.

## Два разных способа прекратить работу

`continue` завершает текущий проход цикла. В нашем случае при номере 2 строка «Доступно издание» пропускается. Для такого `for` затем выполняется шаг `edition++`, проверяется условие и начинается следующий проход.

`break` выходит из цикла целиком. Если изменить `published` на `false`, программа один раз напечатает `Книга готовится` и прекратит обход.

Представьте проверку полки. `continue` — перейти к следующей книге на полке. `break` — прекратить проверку этой полки. Но в программе границу определяет именно цикл, а не любое место, которое кажется нам «следующей задачей».

## Проверим границы

Измените `edition <= 3` на `edition < 3`. Третье издание исчезнет из вывода. Это не сбой инструмента: условие больше не разрешает проход с номером 3.

Верните `<=`. Затем установите начальное значение 4. Тело не выполнится ни разу, потому что условие проверяется до первого прохода.

Наконец, верните исходник и попробуйте убрать `edition++`, сохранив обе точки с запятой. Счётчик перестанет меняться, и при опубликованной книге программа будет повторять сообщение. Остановите её `Ctrl+C` и восстановите шаг. Не оставляйте такой вариант работающим: пользы от повторного вывода нет.

## Читаем условие и цикл по символам

| Запись | Роль |
|---|---|
| `for` | Начало цикла |
| `;` в заголовке `for` | Отделяет начало, условие и шаг |
| `<=` | Меньше или равно; два знака образуют один оператор |
| `++` | Увеличить переменную на единицу; отдельная команда |
| `if` | Выполнить блок, если условие истинно |
| `!` | Логическое отрицание |
| `==` | Проверка равенства; это не присваивание |
| `break` | Закончить этот цикл |
| `continue` | Закончить текущий проход и перейти к следующему |

Фигурные скобки вложенных блоков закрываются в обратном порядке: сначала внутренний `if`, потом внешний `for`, потом `main`. Отступы помогают это видеть; если потерялись, временно подпишите закрывающие скобки в заметках. Круглые скобки вокруг условия `if` Go не требует.

### Маршрут цикла for


После шага снова проверяем условие; false завершает цикл

Опора цикла: начало, проверка, тело и шаг

**Пройдите маршрут пальцем.** Для номеров 1, 2 и 3 укажите, какие условия сработают и где заканчивается проход. Сначала без запуска, затем сравните с выводом.

## Разминка

**Предскажите.** Сколько строк появится при `published := false`?

**Заполните.** Какой оператор оставляет номера до трёх включительно: `<` или `<=`?

**Почините.** У снятого издания удалили `continue`, и теперь оно одновременно «снято» и «доступно». Объясните, почему выполняются обе строки.

## Задание

**Обязательное.** Обойдите номера от 1 до 4 включительно. Снятым должно быть издание 3. Ожидаемый вывод:

```
Доступно издание 1
Доступно издание 2
Издание 3 снято с продажи
Доступно издание 4
```

**На своих данных.** Выберите другой снятый номер внутри диапазона. Перед запуском перечислите ожидаемые строки.

**По желанию.** Снимите два издания с помощью двух отдельных условий. Затем найдите в документации логический оператор `||` и объедините проверку. Сравните читаемость.

## Ответы

Для черновика будет одна строка. Включённая правая граница записывается через `<=`. Без `continue` программа после сообщения о снятии идёт дальше и печатает обычную строку этого же прохода.

В обязательном задании верхняя граница меняется на 4, а проверка снятого номера — на `edition == 3`. Начальное значение остаётся 1.

Мы научились повторять действие и выбирать его по данным. Теперь вынесем самостоятельное правило — расчёт скидки — в функцию и закрепим ожидаемое поведение тестами.

# 5. Правило цены и первый тест

Цена из главы 3 пока только выводилась на экран. Добавим скидку. Мы хотим один раз записать правило расчёта и вызывать его из разных мест, а не копировать арифметику по всему магазину.

Контрольная точка: `examples/05-functions`. Для этой главы нужны условия, целочисленное деление и понимание переменной. Структуры и коллекции пока не требуются.

## Образ: карточка расчёта

Представьте карточку кассира с двумя пустыми местами: «цена» и «скидка». Ниже записаны шаги расчёта, а внизу оставлено место для результата. Карточку можно применить к разным книгам; переписывать правило каждый раз не требуется.

У нашей функции два входных значения — параметры. Из вызова в неё поступают аргументы: конкретная цена и конкретный процент. Функция возвращает новую цену и признак того, что исходные данные допустимы. Если процент больше ста, нельзя молча заполнить нижнюю строку случайным числом и назвать расчёт удачным.

Опора: **два входа — проверка — два выхода**. Это описание именно нашей функции. Вообще функция Go может иметь другое число параметров и результатов, а некоторые функции меняют внешнее состояние. Не всякая функция похожа на спокойный расчёт по карточке. Сейчас нам удобно начать с правила, которое легко проверить отдельно. Закройте код и пройдите по карточке с ценой `100` и скидкой `10`: что окажется на каждом выходе?

## Сначала договоримся о результате

Наше учебное правило принимает цену в минимальных денежных единицах и целый процент скидки. Цена допустима от `0` до `100000000`, процент — от `0` до `100`, обе границы включены. Ограничение цены выбрано для упражнения и безопасного расчёта, а не как универсальный предел торговли.

Мы сначала считаем размер скидки. Если получается дробная минимальная единица, отбрасываем дробную часть скидки. Например, 10% от `105` — это `10.5`; скидка будет `10`, а покупатель заплатит `95`. Это конкретное правило округления в пользу продавца. Другой магазин может выбрать другое, но обязан описать и проверить его.

При недопустимых аргументах функция возвращает признак отказа. Возвращаемое число в этом случае нельзя использовать как цену.

## Сразу целиком

Файл `main.go`:

```go
package main

import "fmt"

// priceAfterDiscount принимает цену в минимальных денежных единицах.
// Допустимая цена: от 0 до 100000000 включительно; скидка: от 0 до 100.
// Дробную минимальную единицу в скидке отбрасываем в пользу продавца.
func priceAfterDiscount(priceMinor int64, percent int64) (int64, bool) {
    if priceMinor < 0 || priceMinor > 100000000 {
        return 0, false
    }
    if percent < 0 || percent > 100 {
        return 0, false
    }
    discountMinor := priceMinor * percent / 100
    return priceMinor - discountMinor, true
}

func main() {
    priceMinor, ok := priceAfterDiscount(249900, 10)
    if !ok {
        fmt.Println("Неверная цена или скидка")
        return
    }
    fmt.Println("К оплате:", priceMinor, "минимальных единиц KZT")
}
```

Запустите `go run .`:

```
К оплате: 224910 минимальных единиц KZT
```

Две величины на выходе помогают различить бесплатную книгу и ошибку. Для допустимой нулевой цены получим `(0, true)`, для недопустимой — `(0, false)`.

## Параметры и аргументы

В объявлении функции `priceMinor` и `percent` — параметры: имена для входных значений. В вызове `priceAfterDiscount(249900, 10)` числа — аргументы, переданные этим параметрам.

Функция не печатает ответ. Она возвращает его вызывающему коду. Благодаря этому одно правило можно будет использовать в терминале, HTTP-обработчике и тесте без захвата вывода с экрана.

`(int64, bool)` после списка параметров описывает два возвращаемых значения. `return 0, false` немедленно завершает вызов. Если проверки прошли, вычисляется скидка и возвращаются цена и `true`.

В строке `priceMinor, ok := ...` оба результата получают имена. `ok` — привычное имя для признака успеха. Сначала проверяем его и только потом используем цену. Позже, когда потребуется объяснить разные причины отказа, заменим логический признак на значение `error`.

Область видимости параметра ограничена функцией. Имя `priceMinor` в `main` — другая переменная, хотя написано одинаково. Передача целого числа копирует его значение; функция не изменяет переменную вызывающего кода.

## Почему проверки стоят до умножения

Операторы `||` соединяют условия значением «или». Цена не подходит, если она меньше нуля **или** превышает верхнюю границу. Процент проверяется отдельно.

Максимальное произведение в нашем договоре — `100000000 * 100`, то есть `10000000000`. Оно помещается в `int64`. Произвольную большую цену мы не умножаем: функция откажет раньше.

Переставить проверку после вычисления опасно. Если арифметика уже переполнилась, проверка искажённого результата не восстановит исходный смысл. Ограничения полезны только тогда, когда реально охватывают путь вычисления.

## Проверка, которая остаётся

Создайте рядом `main_test.go`. Минимальный первый тест выглядит так:

```
package main

import "testing"

func TestPriceAfterDiscount(t *testing.T) {
    got, ok := priceAfterDiscount(249900, 10)
    if !ok || got != 224910 {
        t.Fatalf("получили (%d, %t), хотели (224910, true)", got, ok)
    }
}
```

Запустите:

```
go test ./...
```

В успешном отчёте будет строка `ok` с именем модуля. Время и отметка кеша могут различаться. Для принудительного повторного выполнения используется `go test -count=1 ./...`.

Имя файла заканчивается на `_test.go`: инструмент Go включает такие файлы при тестировании. `TestPriceAfterDiscount` начинается с `Test`, а параметр `t *testing.T` даёт тесту способы сообщить о провале. Пока используйте эту сигнатуру как договор инструмента; указатель и методы разберём отдельно. Звёздочка здесь не участвует в формуле скидки.

`got` означает полученный результат. Условие проверяет и число, и признак успеха. `t.Fatalf` помечает тест неуспешным, печатает сообщение и прекращает этот тест. `%d` выводит число, `%t` — логическое значение. В сообщении указано и полученное, и ожидаемое: это помогает разбирать ошибку.

## Убедитесь, что тест умеет падать

В функции замените `priceMinor - discountMinor` на `priceMinor + discountMinor`. Это правдоподобная ошибка знака: скидка увеличит цену. Запустите тест и найдите сообщение о несовпадении.

Верните минус и снова выполните тест. Один успешный запуск полезен только вместе с пониманием того, какую ошибку проверка действительно замечает.

Не вычисляйте ожидаемый результат в тесте той же формулой, которую проверяете. Иначе опечатка может переехать в обе половины, и сравнение подтвердит их общую ошибку.

## Края важнее одного удачного числа

| Цена | Скидка | Результат | Успех |
|---|---|---|---|
| 249900 | 0 | 249900 | true |
| 249900 | 100 | 0 | true |
| 105 | 10 | 95 | true |
| 0 | 10 | 0 | true |
| -1 | 10 | не использовать | false |
| 100 | 101 | не использовать | false |

В полной контрольной точке есть также проверки отрицательной скидки, предельной цены и цены на единицу выше предела. Тесты не доказывают правильность для всех мыслимых программ, но сохраняют важные примеры нашего договора.

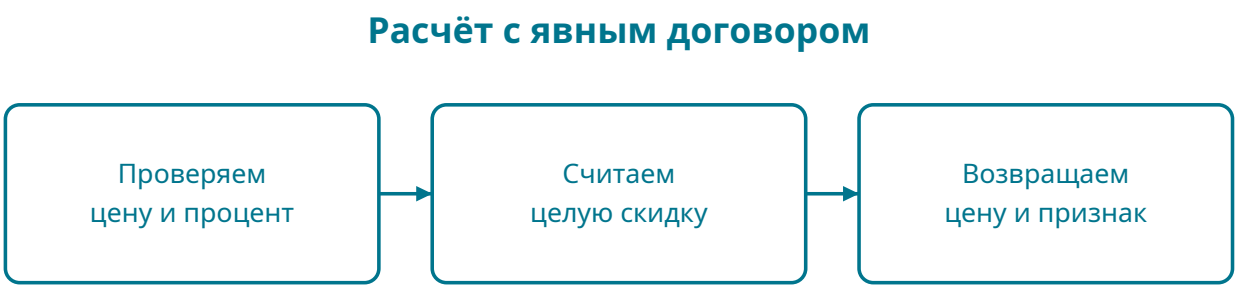
## Читаем функцию и тест по символам

| Запись | Как прочитать |
|---|---|
| `//` | Однострочный комментарий: текст до конца строки не исполняется |
| `(priceMinor int64, percent int64)` | Два параметра; после каждого имени указан его тип |
| `(int64, bool)` после параметров | Два возвращаемых значения, в этом порядке |
| `return` | Завершить функцию и передать результаты вызывающему коду |
| `\|\|` | Логическое «или»: достаточно истинности одного условия |
| `*`, `-` в формуле | Умножение и вычитание |
| `!=` в тесте | Не равно |
| `_` вместо имени результата | Пустой идентификатор: этот результат сознательно не используем |
| `t *testing.T` | Параметр `t` содержит указатель на объект тестирования типа `T` из пакета `testing` |
| `t.Fatalf` | Вызов метода объекта тестирования через точку |

Звёздочка между числами означает умножение. Звёздочка перед типом `testing.T` означает тип указателя: значение, через которое тест обращается к своему объекту отчёта. Это разные грамматические места. Объект создаёт тестовый инструмент; руками выделять его сейчас не нужно. В главе 8 вы сами создадите значение и проверите, как адрес помогает сохранить изменение.

В `if !ok || got != 224910` сначала читаем «успеха нет» и «число не совпало», затем соединяем через «или». Логическое `||` вычисляет правую часть только если левая ложна. При вычислении арифметики умножение и деление имеют приоритет перед вычитанием; операции одного такого приоритета группируются слева направо. Скобки позволяют явно изменить группировку.

`%t` в сообщении выводит логическое значение, `\n` внутри строки задаёт перевод строки. Комментарий описывает договор, но компилятор не доказывает, что код ему соответствует: для этого мы пишем проверки.




Опора функции: проверяем аргументы, считаем скидку, возвращаем результат

**Проверка понимания.** Назовите два разных значения `*` в этой главе и покажите место, где ошибка прекращает вычисление до умножения.

## Разминка

**Предскажите.** Что вернёт `priceAfterDiscount(105, 100)`?

**Заполните.** После получения `price, ok` можно ли печатать цену до проверки `ok`?

**Почините.** Тест проверяет только, что цена равна нулю. Почему он не отличит бесплатную книгу от отказа?

## Задание

**Обязательное.** Напишите тест `TestSmallDiscount`: цена `105`, скидка `1`; ожидаются `104` и `true`. Сначала вычислите результат на бумаге, затем запустите тест. Не меняйте функцию, чтобы подогнать её под ошибочное ожидание.

**На своих данных.** Выберите цену и процент, при которых размер скидки получается дробным. Объясните итог по нашему правилу округления.

**По желанию.** Измените правило: округляйте размер скидки вверх до целой минимальной единицы. Сначала сформулируйте новые ожидания для `105` и `10%`, затем измените отдельную копию функции и тесты. Основную контрольную точку сохраните с прежним договором.

## Ответы

При стопроцентной скидке на допустимую цену получим `(0, true)`. До проверки `ok` число не следует считать ценой. В тесте бесплатной книги надо проверить и ноль, и признак успеха.

Для обязательного задания скидка равна `105 * 1 / 100`, то есть `1`, а итог — `104`. По форме тест похож на первый: меняются аргументы, ожидаемое число и имя.

Теперь в проекте есть самостоятельное правило и автоматическая проверка. В следующей главе займёмся названием книги: выясним, почему длина строки в байтах не равна числу видимых букв.

# 6. Сколько букв в названии книги

Название магазина должно одинаково хорошо работать с русским, казахским и английским текстом. Если проверять его длину неподходящим способом, короткое название может неожиданно не пройти ограничение, а обрезанный текст — перестать читаться.

В этой главе мы научимся измерять строку и введём правило названия: убрать пробельные символы по краям, проверить кодировку и ограничить длину. Контрольная точка: `examples/06-strings`.

## Образ: мозаика и её кусочки

Посмотрите на короткое название `ӘGo`. Глаз легко различает три буквы. Теперь представьте, что каждую букву собирают из маленьких кусочков, а размер коробочки измеряют числом кусочков, а не числом букв. В записи UTF-8 латинским `G` и `o` нужен один байт каждой, а `Ә` — два.

Поэтому `len("ӘGo")` даст `4`: он считает байты строки. Обход `range` декодирует руны — кодовые точки Unicode. Для этого названия их три, а начальные байтовые позиции — `0`, `2`, `3`. Нарисуйте четыре клеточки и объедините первые две одной дугой. Так легче вспомнить, почему следующий символ начинается с позиции два.

Мозаика — только опора различия единиц счёта. Руна тоже не всегда равна одной видимой букве: букву с отдельным комбинируемым знаком можно представить несколькими кодовыми точками. Поэтому не запоминайте «руна — это буква» без оговорки. Каждый раз уточняйте, что требуется посчитать: байты, кодовые точки или воспринимаемые человеком символы.

## Сначала наблюдение

В контрольной точке два рабочих файла: `main.go` запускает опыт, а `title.go` содержит правило. Оба относятся к одному пакету и лежат рядом с `go.mod`. Команда `go run .` соберёт их вместе.

Файл `main.go`:

```
package main

import (
    "fmt"
    "unicode/utf8"
)

func main() {
    title, ok := normalizeTitle("  Шаңырақ Go  ")
    if !ok {
        fmt.Println("Название не подходит")
        return
    }
    fmt.Printf("Название: %q\n", title)
    fmt.Println("Байтов:", len(title))
    fmt.Println("Рун:", utf8.RuneCountInString(title))
    for offset, letter := range "ӘGo" {
        fmt.Printf("Байт %d: %c\n", offset, letter)
    }
}
```

Пока не запускайте: попробуйте посчитать длину `Шаңырақ Go`. В слове семь букв, затем пробел и ещё две буквы. Почему у программы будет два разных числа?

Файл `title.go`:

```
package main

import (
    "strings"
    "unicode"
    "unicode/utf8"
)

// normalizeTitle возвращает однострочное название без краевых пробелов.
// Лимит измеряется в рунах, а не в графемах или байтах.
func normalizeTitle(title string) (string, bool) {
    if !utf8.ValidString(title) {
        return "", false
    }
    title = strings.TrimSpace(title)
    if title == "" || utf8.RuneCountInString(title) > 80 {
        return "", false
    }
    for _, letter := range title {
        if unicode.IsControl(letter) {
            return "", false
        }
    }
    return title, true
}
```

Теперь выполните `go run .`:

```
Название: "Шаңырақ Go"
Байтов: 17
Рун: 10
Байт 0: Ә
Байт 2: G
Байт 3: o
```

И `17`, и `10` верны: они отвечают на разные вопросы.

## Байт, руна и то, что видит читатель

Байт — единица из восьми бит. В Go `byte` является другим именем типа `uint8`, то есть беззнакового целого от 0 до 255. Строка хранит последовательность байтов. Её длина `len(title)` измеряется именно в байтах.

UTF-8 — способ представить кодовые точки Unicode байтами. Для символов ASCII, например `G`, нужен один байт. Для букв в нашем казахском примере — по два. Поэтому семь букв дают четырнадцать байтов, а пробел и `Go` добавляют ещё три.

Руна в Go — другое имя типа `int32`; при работе с Unicode ею представляют кодовую точку. Функция `utf8.RuneCountInString` считает такие кодовые точки. Не путайте это с обещанием посчитать видимые знаки.

Один воспринимаемый знак может состоять из нескольких кодовых точек: например, буква и отдельно записанное ударение. Такие последовательности называют графемными кластерами. Ограничение в восемьдесят рун — понятное правило нашей учебной модели, но оно не равнозначно ограничению в восемьдесят видимых знаков.

## Что возвращает range

В цикле `for offset, letter := range "ӘGo"` первое значение — смещение в байтах, второе — руна. Смещения получились `0`, `2`, `3`. После `Ә` мы перешли сразу к байту с номером `2`, потому что эта буква занимает два байта.

Если вместо обхода написать `title[0]`, мы получим первый байт. Это может быть только часть многобайтовой последовательности. Для нашей строки такой байт нельзя считать первой буквой.

Пустая строка тоже имеет длину — ноль. Но элемента с индексом `0` у неё нет. Проверка длины до обращения по индексу нужна независимо от языка текста.

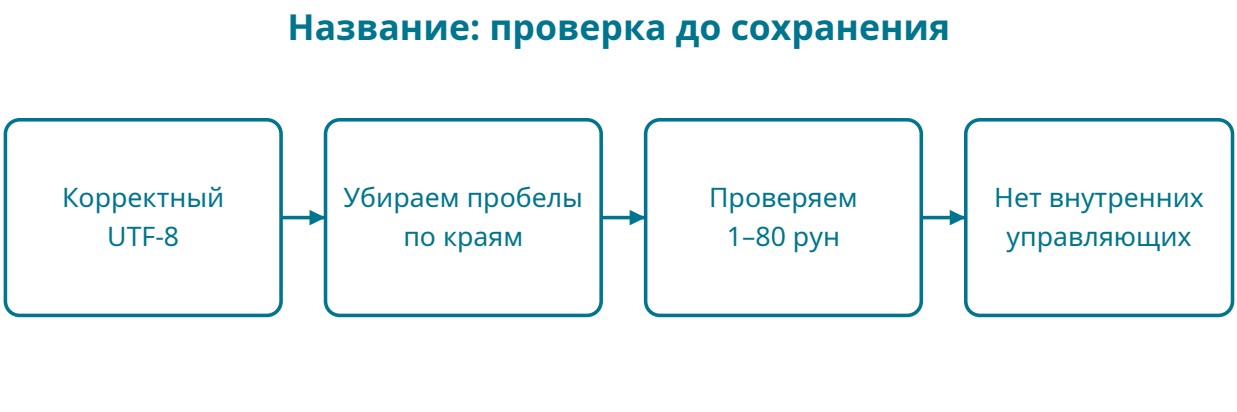
## Зачем проверять UTF-8

Исходные файлы Go записываются в UTF-8, но строка, полученная во время работы, может содержать произвольные байты. Например, файл мог прийти в другой кодировке или быть повреждён.

`utf8.ValidString` отвечает, является ли последовательность корректным UTF-8. В нашем правиле неверная последовательность приводит к отказу. Мы не пытаемся угадать исходную кодировку и не выдаём заменённые символы за первоначальное название.

Это отдельная проверка от подсчёта рун. Сам по себе успешный вызов счётчика не подтверждает, что исходные байты были корректным текстом.

## Четыре шага проверки




Карта проверки названия: UTF-8, пробелы, длина и управляющие символы

Сначала проверяем UTF-8. Затем `strings.TrimSpace` убирает пробельные символы с обоих краёв. Пробел между словами остаётся на месте.

После удаления краевых пробелов название не должно быть пустым, а его длина в рунах — превышать восемьдесят. В конце обходим оставшийся текст и отказываем при управляющих символах, например переводе строки внутри названия.

Обратите внимание на порядок. Строка из пробелов и переводов строк превратится в пустую и получит отказ. Название `Go` с переводом строки только в конце станет `Go`: краевые пробелы мы согласились удалять. А `Go`, перевод строки, `магазин` останется многострочным и будет отклонено.

Эта функция не выполняет полную Unicode-нормализацию и не защищает от всех визуально похожих символов. Её имя означает подготовку названия по перечисленным правилам. Когда понадобится поиск, мы отдельно обсудим сравнение и нормализацию Unicode, чтобы не менять авторское написание без объяснения.

## Строка не меняется внутри себя

Функция `TrimSpace` возвращает строку. Мы сохраняем её в переменной `title` внутри `normalizeTitle`. Это новое присваивание локальному имени, а не изменение отдельных байтов переданного значения.

Поэтому вызывающий код получает подготовленное название через `return`. Если проигнорировать возвращённую строку, исходная переменная у вызывающего кода не станет автоматически обрезанной.

## Проверяем границу, а не впечатление

В `title_test.go` есть проверка строки из восьмидесяти букв `Ә`. Она занимает сто шестьдесят байтов, но должна пройти наш лимит в рунах. Следующая проверка добавляет ещё одну такую букву и ожидает отказ.

Запустите `go test ./...`. Затем замените в функции подсчёт рун на `len(title)` и повторите тесты. Граница для казахского текста сломается. Верните исходную проверку и снова запустите тесты.

Это полезнее проверки только короткого английского слова: для него число байтов и рун совпадает, и неправильная реализация могла бы выглядеть правильной.

## Читаем новую запись по символам

`import (...)` объединяет несколько импортов в группу; скобки здесь не вызывают функцию. В `for offset, letter := range text` запятая разделяет два имени, а `range` задаёт обход строки. Когда первое значение не нужно, вместо `offset` ставят `_`.

В выводе `%q` показывает строку в кавычках с экранированием при необходимости, `%c` выводит символ по значению руны. Путь `"unicode/utf8"` внутри импорта содержит слеш: это часть пути пакета, а не арифметическое деление.

В тестах `\t` означает табуляцию, `\n` — перевод строки, `\xff` задаёт один байт по шестнадцатеричному коду. Последняя запись позволяет намеренно построить неверный UTF-8. Запись `\u0301` задаёт кодовую точку Unicode; язык сам кодирует её в UTF-8 внутри строки. Такие последовательности полезны для опытов, где обычный вид текста скрыл бы различие.

**Проверка понимания.** Не называя просто «длину», произнесите полное условие нашего ограничения: сколько чего мы считаем и что делаем с пробелами по краям.

## Разминка

**Предскажите.** Сколько байтов и рун у строки `Go`? А у `Ә`?

**Заполните.** Индекс в `for offset, letter := range title` измеряется в байтах или в рунах?

**Почините.** Ограничение названия обещает восемьдесят рун, но функция проверяет `len(title) <= 80`. Какой пример покажет ошибку?

## Задание

**Обязательное.** Допишите тест `TestInternalSpaces`: `normalizeTitle(" Go дүкені ")` должна вернуть `"Go дүкені"` и `true`. Между словами два пробела; наша функция их не объединяет.

**На своих данных.** Возьмите название на знакомом вам языке, предскажите число рун и сравните с результатом. Если ожидание разошлось с выводом, сначала уточните, не включает ли видимый знак несколько кодовых точек.

**По желанию.** Сравните строки `"é"` и `"e\u0301"`: напечатайте их длины в байтах, длины в рунах и результат `==`. Не исправляйте сравнение удалением ударения: это меняет данные.

## Ответы

У `Go` два байта и две руны; у `Ә` — два байта и одна руна. `range` возвращает байтовое смещение. Восемьдесят букв `Ә` обнаружат ошибку проверки через `len`: рун восемьдесят, а байтов сто шестьдесят.

В обязательном тесте сравните оба результата: подготовленный текст и признак успеха. Для дополнительного опыта два варианта `é` занимают соответственно два и три байта, содержат одну и две руны и не равны как строки. Они могут выглядеть одинаково, но последовательности байтов различны.

У магазина появилось правило названия. В следующей главе соберём несколько названий в упорядоченный каталог.

# 7. Каталог: список, массив и словарь

До сих пор программа работала с одной книгой. Теперь их будет несколько, и порядок на витрине станет частью результата. Для этого нам понадобятся три вида данных: фиксированный набор форматов, список идентификаторов и словарь названий.

Контрольная точка: `examples/07-catalog`. Книга о SQL — учебная карточка для второго элемента каталога, а не обещание готового второго издания в магазине shanraq.org.

## Образ: ряд карточек и окно

Разложите три карточки в ряд. Теперь прикройте ряд листом бумаги с прямоугольным окном: видны только две карточки. Передвинуть границу окна и переписать карточку — разные действия. Если два окна открывают одну карточку, исправление на ней будет видно через оба.

Такой образ помогает разобраться со срезом: он описывает участок базового массива. Копия среза сама по себе не создаёт независимые копии элементов. А вот копирование массива копирует его элементы. Для независимого набора элементов среза мы явно выделим новое хранилище и скопируем туда данные.

У окна есть важная граница сходства. append может использовать оставшееся место старого массива, а может выделить новый; это зависит от вместимости. Поэтому после добавления нельзя судить об общем хранилище только по похожим названиям переменных. Нарисуйте два окна над одним рядом, затем отдельный ряд для копии. В проверяемом примере измените одну карточку и предскажите, через какое окно изменение станет видно.

А `map` представьте отдельным набором карточек «ключ — название». Этот набор помогает найти название по ключу, но не обещает порядок витрины. Для порядка у нас остаётся собственный список ключей.

## Сразу целиком

Файл `main.go`:

```
package main

import "fmt"

func main() {
    formats := [3]string{"PDF", "EPUB", "HTML"}
    fmt.Println("Форматы:", formats)

    ids := []string{"go-shop"}
    ids = append(ids, "sql-notes")
    titles := map[string]string{
        "go-shop":   "Go: от первой строки до книжного магазина",
        "sql-notes": "Заметки о SQL",
    }
    ordered, ok := catalogTitles(ids, titles)
    if !ok {
        fmt.Println("Каталог неполон")
        return
    }
    for index, title := range ordered {
        fmt.Println(index+1, title)
    }
    _, found := titles["missing"]
    fmt.Println("Неизвестная книга найдена:", found)
}
```

Файл `catalog.go`:

```
package main

import "strings"

// catalogTitles сохраняет порядок ids и отказывает при неполных данных.
func catalogTitles(ids []string, titles map[string]string) ([]string, bool) {
    ordered := make([]string, 0, len(ids))
    for _, id := range ids {
        title, ok := titles[id]
        if !ok || strings.TrimSpace(title) == "" {
            return nil, false
        }
        ordered = append(ordered, title)
    }
    return ordered, true
}
```

Запустите `go run .`:

```
Форматы: [PDF EPUB HTML]
1 Go: от первой строки до книжного магазина
2 Заметки о SQL
Неизвестная книга найдена: false
```

Первая книга в списке — наша книга. Словарь помогает получить название по идентификатору, а список задаёт последовательность показа.

## Массив: длина входит в тип

`[3]string` означает массив из трёх строк. Число стоит между квадратными скобками и является частью типа: `[2]string` и `[3]string` — разные типы.

Мы сразу записали три значения: `PDF`, `EPUB`, `HTML`. К первому обращаются по индексу `0`, к последнему — по `2`. Индекса `3` у этого массива нет.

Массив выбран для демонстрации фиксированного набора. Это не утверждение, что вся архитектура магазина навсегда обязана хранить форматы в массиве. Когда у разных книг будет разный набор файлов, понадобится другая модель.

При присваивании массива другому массиву копируются элементы. Для массива строк можно поменять элемент копии, не заменяя строку в исходном массиве. Но если сами элементы содержат ссылки на общие данные, копирование внешнего массива не делает эти данные независимыми автоматически.

## Срез: видимая часть массива

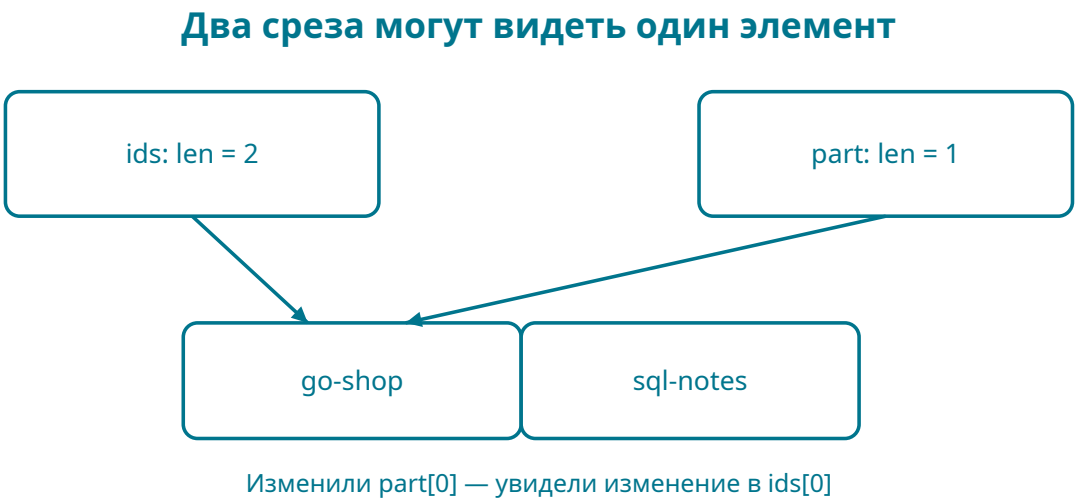
`[]string` означает срез строк. Числа между скобками нет. Начальный срез `ids` содержит один идентификатор. Вызов `append(ids, "sql-notes")` возвращает срез с добавленным элементом, и мы сохраняем этот результат обратно в `ids`.

Почему возвращённое значение важно? В срезе есть информация о начале данных, длине и доступной ёмкости. Добавление меняет длину, а иногда требует другого массива. Если не сохранить результат `append`, у вас не будет правильного описания расширенного среза.

`len` отвечает, сколько элементов сейчас видно. `cap` — сколько элементов доступно от начала этого среза до конца доступной ему области массива. Для среза, начинающегося в середине массива, это не обязательно размер всего исходного массива.

`make([]string, 0, len(ids))` создаёт срез с нулевой длиной и заранее достаточной ёмкостью для нашего результата. Длина ноль означает, что обращаться к `ordered[0]` пока нельзя. Сначала нужно добавить элемент.

## Общее хранилище: важная граница образа




Два среза могут смотреть на один массив; изменение элемента видно обоим

Срез удобно представлять окном в массив. Но два окна могут показывать одни и те же ячейки. Выражение `part := ids[:1]` не обещает независимую копию первого элемента.

Здесь левая граница пропущена и равна нулю, правая не включается: виден только элемент с индексом `0`. Если присвоить `part[0]` новое значение, оно будет видно через `ids[0]`.

Для независимого списка строк можно создать срез нужной длины через `make` и вызвать встроенную функцию `copy`. Она копирует столько элементов, сколько помещается в оба среза, и возвращает число скопированных элементов. Если создать приёмник длиной ноль, ничего не скопируется, даже при большой ёмкости.

Копирование списка строк достаточно для независимой замены его строковых элементов. Для среза структур с вложенными map или другими срезами вопрос независимости придётся рассматривать глубже.

Не запоминайте конкретный коэффициент роста `cap` после `append`: наш код не должен зависеть от стратегии выделения памяти. Проверять нужно наличие элементов и требуемую независимость данных.

## Словарь: ключ вместо позиции

В `map[string]string` первый тип относится к ключу, второй — к значению. Ключ `go-shop` связывается с названием книги.

Операция `title, ok := titles[id]` возвращает значение и признак присутствия ключа. Для неизвестного ключа строка будет пустой, а `ok` — ложным. Если ключ существует и хранит пустую строку, `ok` будет истинным. Поэтому функция проверяет два разных условия: ключ найден и его название не состоит из пробелов.

Нулевое значение map — `nil`. Из неё можно читать, получая отсутствие ключа. Записывать в nil map нельзя: нужен созданный словарь, например через `make(map[string]string)` или литерал, как в нашем примере. Это связано с начальным значением, а не со словом `var`: запись `var titles = make(map[string]string)` создаёт рабочий словарь.

При присваивании map другой переменной данные словаря не копируются в независимое хранилище. Изменение элемента через одно имя будет видно через другое. Подробнее проверим это вместе со структурами.

## Почему витрина не обходит map

Порядок обхода map не определён. Даже если несколько запусков случайно дали удобную последовательность, полагаться на неё нельзя.

Поэтому мы обходим `ids`. Для каждого идентификатора получаем соответствующее название. Если данных не хватает, возвращаем `nil, false` и не показываем частично собранный результат как полноценный каталог.

Два отдельных контейнера требуют согласованности. Можно добавить идентификатор и забыть название. В этой главе мы намеренно делаем такой риск видимым; следующая объединит поля книги в структуру и сократит число мест, которые нужно менять вместе.

Словарь полезен для поиска по ключу, но образ «мгновенного гардероба» не означает нулевого времени или гарантированной одинаковой стоимости любой операции.

Производительность измеряют на задаче и данных, а не выводят из метафоры.

## Проверяем поведение

`catalog_test.go` проверяет заданный порядок, отсутствующий ключ, пустое название и пустой каталог. Запустите `go test ./...`.

Пустой каталог считается допустимым: список результатов пуст, признак успеха истинен. Каталог с обещанным идентификатором, для которого не найдено название, — ошибка. Эти случаи отличаются смыслом, хотя оба могут не дать ни одной строки на витрине.

## Читаем коллекции по символам

| Запись | Как прочитать |
|---|---|
| `[3]string` | Массив ровно из трёх строк |
| `[]string` | Срез строк |
| `{"PDF", "EPUB", "HTML"}` после типа | Значения составного литерала |
| `[0]` после переменной | Обращение к элементу по индексу |
| `[:1]` после среза | Диапазон от начала до индекса 1, не включая его |
| `map[string]string` | Словарь: строки в качестве ключей и значений |
| `"go-shop": "Название"` | Двоеточие разделяет ключ и значение записи словаря |
| `make`, `append`, `copy`, `len`, `cap` | Встроенные функции языка; отдельный импорт не нужен |
| `nil` | Нулевое значение для среза, map и ряда других типов, но не для строки или целого числа |

В многострочном составном литерале после последнего элемента тоже стоит запятая. Она нужна для такой записи с переносом перед закрывающей скобкой. Не переносите правило на список аргументов в одну строку бездумно: здесь мы объясняем конкретный многострочный пример.

`range ordered` даёт индекс и значение элемента среза. Выражение `index+1` только меняет печатаемый номер: индекс самого среза по-прежнему начинается с нуля. Формат `%v` в сообщении теста показывает значение в обычном представлении.

**Восстановите по памяти.** Нарисуйте словарь как карточки «ключ — название» и отдельно список ключей для витрины. Проведите путь от первого элемента списка к его названию. Если хочется взять «первую запись map», вернитесь к правилу порядка обхода.

## Разминка

**Предскажите.** Сколько элементов видит срез после `make([]string, 0, 10)`?

**Заполните.** Как сохранить результат добавления: `ids = ___(ids, "new-book")`?

**Почините.** `copy` вернул ноль, хотя у приёмника ёмкость десять. Какое второе свойство приёмника нужно проверить?

## Задание

**Обязательное.** Добавьте третью карточку с идентификатором `go-tests` и названием `Тесты на Go`. Она должна появиться третьей. Затем удалите только её название из map: программа должна вывести `Каталог неполон`, а не первые две карточки как будто всё получилось.

**На своих данных.** Поменяйте порядок идентификаторов. Убедитесь, что меняется порядок витрины, а пары в словаре переписывать не нужно.

**По желанию.** Создайте срез из двух строк, получите `part := original[:1]`, поменяйте `part[0]`. Повторите с отдельным приёмником и `copy`. Сравните результаты.

## Ответы

У среза длина ноль: ёмкость не даёт права обратиться к ещё не существующему элементу. Для добавления используется `append`. `copy` учитывает длину приёмника, поэтому её тоже надо проверить.

В обязательной задаче изменяются два места: добавление идентификатора и запись в словаре. Это и есть неудобство, которое устранит структура книги.

После этой главы у нас есть работающий каталог с явным порядком и отказом при неполных данных. Следующая глава объединит идентификатор, название, цену и состояние публикации в одно значение.

# 8. Книга как значение: структуры, методы и указатели

Название, цена и признак публикации относятся к одной книге. Если хранить их в отдельных переменных или параллельных словарях, легко перепутать, что к чему относится. Объединим данные и сохраним уже написанные правила.

Контрольная точка: `examples/08-books`. В ней `pricing.go` содержит расчёт скидки из главы 5, а `title.go` — проверку названия из главы 6. Их поведение осталось прежним и проверяется теми же тестами.

## Образ: карточка книги и записанный адрес

На карточке книги стоят название, цена и отметка о публикации. Сделайте её копию и исправьте название на копии. Первая карточка сохранит прежний текст. Для нашей структуры Book с её нынешними полями это хорошее начало понимания копирования значения.

Теперь вместо карточки перепишите на другой лист её адрес. У вас два листа с адресом, но сама карточка одна. Если оба человека пришли по этому адресу и один поставил отметку о публикации, второй увидит эту же отметку. Так удобнее представить копирование указателя: копируется значение адреса, а новая книга от этого не появляется.

Опора: **копия карточки — другая карточка; копия адреса — путь к прежней карточке**. Вызов метода с получателем-указателем позволяет изменить исходную книгу. Но если внутри структуры появятся срезы или `map`, одной картинки двух независимых карточек уже мало: их копии могут обращаться к общим внутренним данным. Ниже мы отдельно проверим эту границу. Попробуйте нарисовать два адресных листка и только одну книгу, а затем объяснить, что именно копирует присваивание указателя.

## Сначала работающий результат

Новый файл book.go:

```go
package main

type Book struct {
    ID         string
    Title      string
    PriceMinor int64
    Currency   string
    Published  bool
}

// PriceAfterDiscount вычисляет цену, не изменяя книгу.
func (b Book) PriceAfterDiscount(percent int64) (int64, bool) {
    return priceAfterDiscount(b.PriceMinor, percent)
}

// Publish меняет состояние книги только после успешной проверки.
func (b *Book) Publish() bool {
    if b == nil || b.ID == "" || b.Currency != "KZT" {
        return false
    }
    title, ok := normalizeTitle(b.Title)
    if !ok {
        return false
    }
    _, ok = priceAfterDiscount(b.PriceMinor, 0)
    if !ok {
        return false
    }
    b.Title = title
    b.Published = true
    return true
}
```

Файл `main.go`:

```go
package main

import "fmt"

func main() {
    book := Book{
        ID:         "go-shop",
        Title:      "  Go: от первой строки до книжного магазина  ",
        PriceMinor: 249900,
        Currency:   "KZT",
    }
    fmt.Println("До публикации:", book.Published)
    if !book.Publish() {
        fmt.Println("Книга не готова к публикации")
        return
    }
    fmt.Println("После публикации:", book.Published)
    price, ok := book.PriceAfterDiscount(10)
    if !ok {
        fmt.Println("Скидка не подходит")
        return
    }
    fmt.Println(book.Title)
    fmt.Println("К оплате:", price, "минимальных единиц", book.Currency)
}
```

Добавьте из предыдущих контрольных точек `title.go` и функцию `priceAfterDiscount` в файл `pricing.go`. В готовой папке они уже есть. Не копируйте второй `main.go`: внутри пакета нам нужна одна функция `main`.

Запуск: `go run .`. Результат:

```
До публикации: false
После публикации: true
Go: от первой строки до книжного магазина
К оплате: 224910 минимальных единиц KZT
```

У книги изменилось состояние публикации, название потеряло краевые пробелы, а вычисленная цена использует прежнее правило скидки. Мы не написали магазин заново: соединили уже проверенные части.

## Структура связывает поля

`type Book struct` объявляет тип `Book` с перечисленными полями. У каждого поля есть имя и тип. В `main` мы создаём значение литералом `Book{...}` и явно подписываем поля.

Поле `Published` не указано в литерале, поэтому получает нулевое значение `false`. По нашему правилу новая книга начинается как черновик. Это уже осмысленное применение нулевого значения: правило программы согласовано с правилом языка.

`book.Title` обращается к полю конкретного значения. Две разные переменные типа `Book` могут содержать разные названия и цены. Сам тип описывает форму данных, а не единственную книгу на весь магазин.

Имена полей с заглавной буквы экспортируются из пакета. В этой главе весь код ещё внутри `main`; границы пакетов разберём позже. Экспортированное поле не обеспечивает запрета на некорректное присваивание: вызывающий код может записать туда отрицательную цену. Поэтому проверка перед публикацией остаётся необходимой.

## Метод — функция с получателем

У `PriceAfterDiscount` перед именем стоит `(b Book)`. Это получатель: значение книги, для которого вызывается метод. Вызов `book.PriceAfterDiscount(10)` передаёт в метод книгу и процент скидки.

Метод с получателем-значением получает копию `Book`. Для нашей структуры копируются строки, целое число и логическое значение. Сам метод просто передаёт цену знакомой функции и ничего не меняет.

Это не новое правило арифметики, а удобный способ связать уже существующее действие с данными книги. Нет необходимости превращать в методы все функции проекта: если связь с конкретным типом не помогает чтению, обычная функция может быть яснее.

## Указатель: куда записать изменение

`Publish` объявлен с получателем `(b *Book)`. Тип `*Book` означает указатель на значение `Book`. Такой указатель позволяет обратиться к исходной книге и изменить её поля.

В Go аргументы передаются по значению. Это верно и для указателя: копируется значение указателя, а обе копии указывают на ту же книгу. Поэтому формулировка «со звёздочкой копирования вообще нет» неверна.

`&book` получает адрес переменной. `*pointer` позволяет обратиться к значению по указателю. Для обычного доступа к полю достаточно `pointer.Title`: Go разрешает такую краткую запись без явного `(*pointer).Title`.

В `book.Publish()` компилятор может взять адрес переменной `book` автоматически. Это работает для адресуемого значения. Не делайте из этого правило «любой результат можно сразу менять методом»: временное значение, возвращённое некоторым выражением, может не быть адресуемым. Сначала сохраните его в переменную, если собираетесь изменять.

## Нулевой указатель

У указателя есть нулевое значение `nil`: адрес книги не задан. В нашем методе сначала стоит проверка `b == nil`, поэтому такой вызов вернёт отказ, а не попытается читать поле отсутствующей книги.

Это решение именно нашего метода. В Go вызов метода на nil-указателе не становится автоматически безопасным: всё зависит от того, что метод делает. `PriceAfterDiscount` имеет получателя-значение, и отсутствие книги не является для него допустимым аргументом.

## Изменение после всех проверок

Сначала `Publish` проверяет указатель, идентификатор и валюту. Затем получает подготовленное название в локальную переменную. Потом проверяет цену через существующую функцию скидки с нулевым процентом.

Только после всех успешных проверок два поля получают новые значения. Если цена не подходит, даже удаление краевых пробелов не сохранится в книге. Тест `TestInvalidBookUnchanged` сравнивает её с состоянием до вызова.

Это маленький пример важного правила: отказ не должен оставлять полувыполненное изменение, если договор операции обещает целостность. Для базы данных позже потребуется транзакция; сейчас достаточно правильного порядка присваиваний в одной функции.

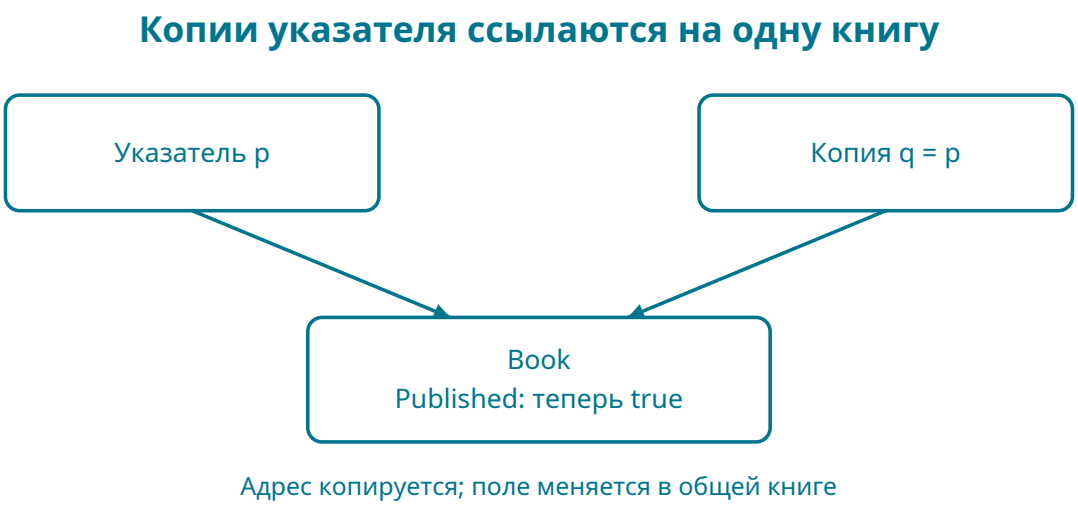
## Ловушка цикла: меняем копию

Если написать `for _, book := range books`, переменная `book` получает значение очередного элемента. Для среза структур `Book` это копия структуры. Вызов `book.Publish()` изменит её, а исходный элемент среза останется черновиком.

Для изменения элементов нужен обход по индексам: `for i := range books`, затем `books[i].Publish()`. Элемент среза адресуем, поэтому метод меняет нужную книгу.

В `book_test.go` проверяются оба варианта подряд. Сначала тест подтверждает, что изменение копии не затронуло оригинал, затем — что изменение элемента среза сохранилось. Здесь не требуется угадывать по внешнему виду кода: можно воспроизвести оба поведения.

## А если поле — map или срез?




Копирование значения указателя сохраняет адрес общей книги

Наша `Book` пока состоит из простых полей. Если добавить поле map или срез, копия внешней структуры может продолжать обращаться к общим внутренним данным. Изменение элемента такого контейнера отличается от замены поля целиком.

Именно здесь была ошибка в одном из исходных курсов: там утверждалось, что метод с получателем-значением не может изменить содержимое map внутри структуры. Проверяемая демонстрация находится в `research/reproductions/map-value`: запись элемента через копию структуры видна оригиналу, замена поля — нет.

Выбор получателя не стоит обосновывать лозунгом «указатели всегда быстрее». Сначала определите семантику: должно ли изменение сохраняться, есть ли общее состояние и разрешено ли копировать сам тип. Измерения производительности — отдельная работа.

## Читаем тип и методы по символам

`type` вводит объявление типа. `struct { ... }` перечисляет его поля. В `Book{ID: "go-shop"}` фигурные скобки создают составное значение, а двоеточие связывает имя поля с его значением. Внутри объявления `struct` поля записаны иначе — имя и тип без двоеточия.

Точка в `book.Title` выбирает поле; точка в `book.Publish()` выбирает метод, который затем вызывается круглыми скобками. В `(b Book)` перед именем метода объявлен получатель. В `(b *Book)` звёздочка входит в тип получателя.

`&book` берёт адрес. В выражении `*pointer` звёздочка обращается к значению по адресу, а в типе `*Book` обозначает указатель на книгу. В `%+v` внутри сообщения теста плюс просит `fmt` показать имена полей структуры: это часть формата, не сложение.

Сравнение `book != before` в нашем тесте возможно потому, что все поля текущей `Book` сравнимы. Если добавить срез или map, структура уже не будет сравниваться таким оператором. Нельзя бездумно переносить тест после изменения формы данных.

**Проверка понимания.** Прочитайте одну строку создания книги и одну строку вызова метода, вслух называя роль каждого вида скобок. Затем объясните, какой именно объект изменит получатель-указатель.

## Разминка

**Предскажите.** Если `copy := book`, затем `copy.Title = "Другая книга"`, изменится ли `book.Title` для нашей текущей структуры?

**Заполните.** Какой знак получает адрес переменной: `&` или `*`?

**Почините.** Цикл вызвал `Publish` для каждой переменной `book`, но элементы `books` остались черновиками. Какой вариант обхода нужен?

## Задание

**Обязательное.** Напишите тест для бесплатной книги: непустые `ID` и `Title`, валюта `KZT`, цена `0`. Метод `Publish` должен вернуть `true` и установить `Published`. Это согласовано с допустимым нулём из главы 5.

**На своих данных.** Создайте две книги в срезе. Опубликуйте их обходом по индексам и выведите названия опубликованных. Сделайте одну цену недопустимой и убедитесь, что именно эта книга остаётся черновиком.

**По желанию.** Добавьте метод `Unpublish`, который возвращает книгу в черновик. Определите, что он делает при повторном вызове, и запишите это в тесте. Не связывайте снятие с витрины с удалением уже купленных файлов: это разные решения магазина.

## Ответы

Изменение строкового поля копии не заменяет поле оригинала. Адрес получается через `&`. Для изменения структуры в срезе используйте индекс и обращайтесь к `books[i]`.

Тест бесплатной книги должен проверять и результат метода, и состояние поля. Одного `true` недостаточно: ошибочная реализация могла бы согласиться с операцией, ничего не изменив.

Мы получили модель книги и соединили проверку названия с правилом цены. Следующий шаг — заменить неразличимый `false` объяснимой ошибкой и научиться безопасно работать с файлами.