

O'REILLY®

Head First

Изучаем Swift

Джон Мэннинг,
Пэрис Баттфилд-Эддисон



ПОДАРОК ДЛЯ МОЗГА



Head First

Swift

Wouldn't it be dreamy to quickly learn Swift? I could build apps, websites, command-line tools...anything!

Jon Manning &
Paris Buttfield-Addison



Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Head First

Изучаем Swift

Хорошо бы найти книгу,
которая позволила бы мне быстро
изучить Swift. Я могла бы строить
приложения, веб-сайты,
программы командной строки...
короче, всё!

Джон Мэннинг,
Пэрис Баттфилд-Эддисон



 ПИТЕР®

Санкт-Петербург · Москва · Минск

2023

ББК 32.973.2-018.1
УДК 004.43

Мэннинг Джон, Баттфилд-Эддисон Пэрис

M97 Head First. Изучаем Swift / Пер. с англ. Е. Матвеева. — СПб.: Питер, 2023. — 400 с.: ил. — (Серия «Head First O'Reilly»).

ISBN 978-5-4461-2254-7

Swift известен как язык разработки приложений, выбранный компанией Apple для iOS, iPadOS, macOS, watchOS и tvOS. Но этим его применение не ограничивается. Swift с открытым кодом набирает популярность как язык системного программирования и программирования на стороне сервера, его используют и в Linux, и в Windows. С чего же начать?

«Head First. Изучаем Swift» охватывает все актуальные темы — от сбора и управления данными до повторного использования кода, построения нестандартных типов данных и структурирования программ и пользовательских интерфейсов в SwiftUI. Изучив Swift, вы будете готовы строить любые приложения — от мобильных и веб-приложений до игр, фреймворков, средств командной строки и многого другого.

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

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1491922859 англ.
ISBN 978-5-4461-2254-7

Authorized Russian translation of the English edition of Head First Swift (ISBN 9781491922859) © 2021 Secret Lab
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.
© Перевод на русский язык ООО «Прогресс книга», 2022
© Издание на русском языке, оформление ООО «Прогресс книга», 2022
© Серия «Head First O'Reilly», 2021

Руководитель дивизиона *Ю. Сергиенко*
Литературный редактор *Н. Викторова*
Корректор *М. Лауконен*
Художественный редактор *В. Мостипан*
Верстка *Л. Соловьева*

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: 12.2022. Наименование: книжная продукция. Срок годности: не ограничен.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ,
г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Налоговая льгота — общероссийский классификатор
продукции ОК 034-2014, 58.11.12.000 — Книги печатные
профессиональные, технические и научные.

Подписано в печать 07.10.22. Формат 84×108/16.
Бумага офсетная. Усл. п. л. 42,000.
Тираж 1000. Заказ 0000.

Спасибо команде разработки Swift с открытым кодом, сотрудникам Apple и другим участникам конференции /dev/world и более широкого сообщества Swift.

Спасибо Swift — языку достаточно сложному (и интересному!), чтобы о нем стоило написать эту книгу.

Да, и еще спасибо нашим семьям. Пэрис хочет особо поблагодарить свою маму и жену, а Джон — маму, папу и свою чрезвычайно обширную семью за поддержку.

Об авторах

Пэрис Баттфилд-Эддисон



Джон Мэннинг

Пэрис Баттфилд-Эддисон и **Джон Мэннинг** — соучредители студии разработки Secret Lab в Хобарте (штат Тасмания, Австралия).

И Пэрис, и Джон имеют кандидатскую степень в области компьютерных наук, а за прошедшие годы они написали более 30 книг. Они вместе работали во влиятельном стартапе эпохи «Web 2.0» Meebo и входят в команду одной из самых давних конференций разработчиков Apple AUC /dev/world.

В Secret Lab Пэрис и Джон совместно работали над тысячами приложений и игр. Они наиболее известны своей приключенческой игрой Night in the Woods, получившей награды Independent Game Festival и BAFTA, а также популярным проектом с открытым кодом Spinner (<https://yarnspinner.dev>), лежащим в основе тысяч повествовательных видеоигр.

Пэрис и Джон живут и работают в Хобарте, они увлекаются фотографией, кулинарией и выступлениями на многочисленных конференциях. С Пэрисом можно связаться на сайте <https://paris.id.au>, с Джоном на сайте <https://desplesda.net>, а с Secret Lab на сайте <https://secretlab.games>.



Студия Secret Lab

Содержание (сводка)

	Введение	21
1	Знакомство со Swift. <i>Приложения, системы и не только!</i>	31
2	По имени Swift. <i>Swift на практике</i>	57
3	Коллекции и управление. <i>Зацикленные на данных</i>	81
4	Функции и перечисления. <i>Повторное использование кода</i>	121
5	Замыкания. <i>Необычные гибкие функции</i>	155
6	Структуры, свойства и методы. <i>Типы, определяемые пользователем, и не только</i>	181
7	Классы, акторы и наследование. <i>О пользе наследования</i>	211
8	Протоколы и расширения. <i>Протокольные церемонии</i>	235
9	Опциональные типы, распаковка, обобщение и другое. <i>Неизбежные опциональные типы</i>	267
10	Знакомство со SwiftUI. <i>Пользовательские интерфейсы</i>	293
11	Практическое применение SwiftUI. <i>Круги, таймеры, кнопки – выбирайте!</i>	337
12	Приложения, веб-программирование и все такое. <i>Собирая все вместе</i>	365

Содержание (настоящее)

Введение

Ваш мозг и Swift. Вы пытаетесь изучить что-то новое, а ваш мозг хочет оказать вам услугу и как можно быстрее забыть выученное. Он думает: «Лучше оставить место для чего-то поважнее, например, от каких диких животных стоит держаться подальше или почему на сноуборде не стоит кататься нагишом». Так как же заставить ваш мозг думать, что ваша жизнь зависит от знания Swift?

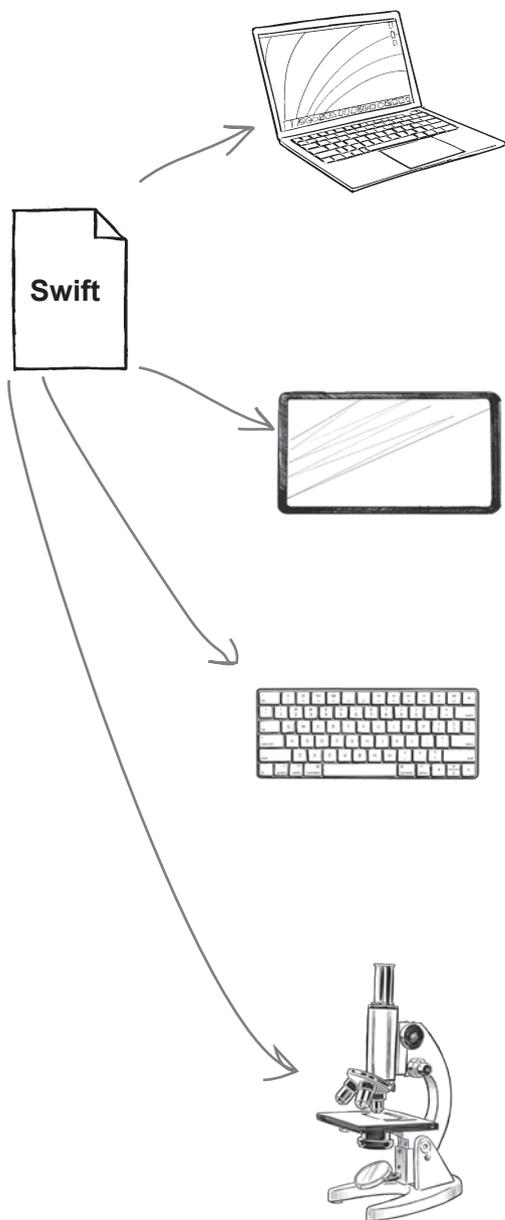
Для кого написана эта книга?	22
Мы знаем, о чем вы думаете	23
Метапознание: наука о мышлении	25
Вот что сделали мы	26
Что можете сделать вы	27
Примите к сведению	28

Знакомство со Swift

1

Приложения, системы и не только!

Swift — язык программирования, на который можно положиться. Вам не будет стыдно познакомить с ним вашу семью. Он безопасен, надежен, быстр, доступен и несложен. И хотя Swift получил наибольшую известность **как язык программирования для платформ Apple**, таких как iOS, macOS, watchOS и tvOS, проект с открытым кодом Swift также работает в Linux и Windows и постепенно набирает популярность как язык системного программирования, а также как серверный язык. На нем можно строить все что угодно, от мобильных приложений до игр, веб-приложений, фреймворков. Итак, за дело!



Swift — универсальный язык	32
Стремительная эволюция Swift	34
Стремление в будущее	35
Как вы будете писать код Swift	36
Путь, лежащий перед вами	38
Установка Playgrounds	39
Создание среды Playground	41
Использование среды Playground для написания кода Swift	42
Основные структурные элементы	44
Пример Swift	50
Поздравляем, вы сделали свои первые шаги в Swift!	55

По имени Swift

2 Swift на практике

Вы уже знаете азы Swift. Но пришло время изучить основные элементы языка более подробно. Вы узнали достаточно, чтобы вас воспринимали серьезно, пора употребить новые знания на практике. Мы применяем Playgrounds для написания кода, использования команд, выражений, переменных и констант — основных структурных элементов Swift. В этой главе мы заложим основу вашей будущей карьеры программиста Swift. Вы освоите систему типов Swift и изучите основы представления текста в строковом виде. Не будем терять времени — еще чуть-чуть, и вы начнете писать код Swift.

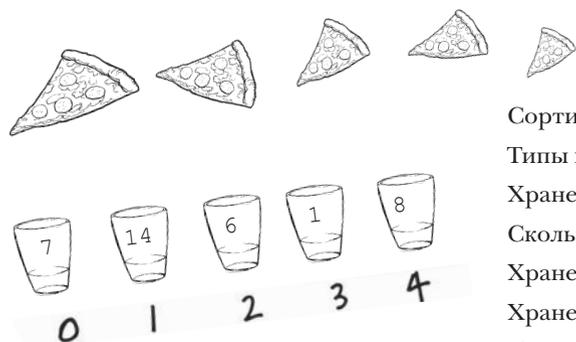


Из чего строятся программы	58
Базовые операторы	59
Математические вычисления	60
Выражайтесь яснее	61
Имена и типы	64
Не все данные являются числами	68
Определение строковых переменных	70
Строковая интерполяция	76

3 Коллекции и управление

Зацикленные на данных

Вы уже знаете о выражениях, операторах, переменных, константах и типах Swift. Пришло время собрать воедино все, что говорилось ранее, и на этой основе исследовать некоторые более сложные структуры данных и операторы Swift: **коллекции** и **управляющие команды**. В этой главе мы поговорим о сохранении коллекций данных в переменных и константах, о структурировании данных, обработке данных и работе с данными с использованием управляющих команд. Позднее в книге будут рассмотрены другие способы сбора и структурирования данных, а пока начнем с **массивов**, **множеств** и **словарей**.



Сортировка пиццы	82
Типы коллекций Swift	83
Хранение значений в массиве	84
Сколько элементов в массиве? И есть ли в нем элементы?	86
Хранение значений в множестве	87
Хранение значений в словаре	89
Кортежи	91
Хороший псевдоним пригодится каждому	92
Управляющие команды	94
Команды if	95
Команды switch	96
Построение команды switch	100
Операторы диапазонов	102
Более сложные команды switch	103
Многократное выполнение кода в циклах	104
Построение цикла for	105
Построение цикла while	108
Построение цикла repeat-while	109
Решение проблемы сортировки пиццы	110
Мы прошли большой путь!	112

4

Функции и перечисления

Повторное использование кода

Функции в языке Swift позволяют упаковать некоторое поведение или единицу работы в блок кода, который может вызываться из других частей вашей программы. Функции могут быть автономными, а могут определяться как часть класса, структуры или перечисления, где они обычно называются методами. При помощи функций можно разбить сложные задачи на меньшие части, более удобные и легкие в тестировании. Функции занимают центральное место в формировании структуры программ в Swift.



Функции Swift как средство повторного использования кода	123
Встроенные функции	124
Что можно узнать по встроенным функциям?	125
Когда могут пригодиться функции	128
Написание тела функции	129
Использование функций	130
Функции работают со значениями	132
С возвращением (из функций)	134
Переменное количество параметров	136
Что можно передать функции?	139
У каждой функции есть тип	140
Типы функций как типы параметров	144
Несколько возвращаемых типов	146
Функции не обязаны существовать автономно	148
Switch с перечислениями	151

5

Замыкания Необычные гибкие функции

Функции полезны, но иногда нужно больше гибкости. Swift позволяет использовать **функцию как тип** — так же, как вы используете целое число или строку. **Это означает, что вы можете создать функцию и присвоить ее переменной.** После того как функция будет присвоена переменной, ее можно вызывать через эту переменную или передавать другим функциям в параметре. Когда вы создаете и используете функцию подобным образом, это называется **замыканием**. Замыкания очень полезны, потому что они способны сохранять **ссылки** на константы и переменные из контекста, в котором они были определены. Это называется **замыканием по значению**, отсюда и название.

Мы закроем вопрос с замыканиями.

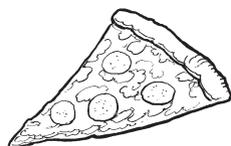
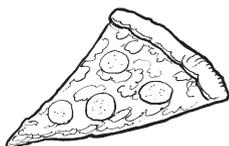


Знакомьтесь: простое замыкание	156
Замыкания с параметрами	158
Вычисление сводного значения	166
Свертки с использованием замыканий	167
Сохранение значений из внешней области видимости	170
Выходящие замыкания: нетривиальный пример	172
Автозамыкания обеспечивают гибкость	175
Сокращенные имена аргументов	178

6 Структуры, свойства и методы

Типы, определяемые пользователем, и не только

При работе с данными часто требуется определять собственные виды данных. Структуры, также нередко обозначаемые ключевым словом **struct**, позволяют создавать **типы данных, определяемые пользователем** (подобно тому, как `String` и `Int` являются типами данных), посредством **объединения других типов**. Использование структур для представления данных, с которыми работает ваш код Swift, позволяет отступить на шаг и подумать над взаимодействием данных, передаваемых в вашем коде. **Структуры могут содержать переменные и константы** (внутри структур они называются **свойствами**) и **функции** (называемые **методами**). Добавим немного порядка в ваш мир и займемся углубленным изучением структур.



Сделаем пиццу во всей красе...	183
Инициализатор ведет себя точно так же, как функция	189
Статические свойства повышают гибкость структур	192
Функции в структурах	196
Методы	196
Изменение свойств с использованием методов	197
Вычисляемые свойства	199
Get- и set-методы для вычисляемых свойств	203
Реализация set-метода	204
Строки Swift в действительности являются структурами	206
Для чего нужны отложенные свойства	207
Использование отложенных свойств	208

7

Классы, акторы и наследование

О пользе наследования

Структуры показали, насколько полезным может быть построение типов, определяемых пользователем. Но у Swift в запасе есть и другие средства, включая *классы*. Классы похожи на структуры: они позволяют создавать *новые типы данных, содержащие свойства и методы*. Кроме того что они являются *ссылочными типами*, то есть экземпляры конкретного класса совместно используют одну копию своих данных (в отличие от структур, которые являются типами-значениями, чьи данные копируются), *классы поддерживают наследование*. Наследование позволяет построить один класс на базе другого класса.

Структура с другим именем (и это имя — класс)	212
Наследование и классы	214
Переопределение методов	218
Финальные классы	222
Автоматический подсчет ссылок	226
Изменяемость	227

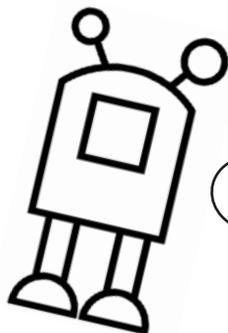
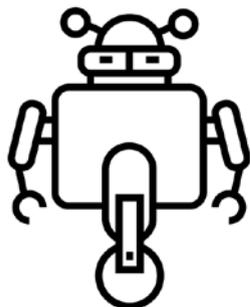


8 Протоколы и расширения

Протокольные церемонии

Вы все знаете о классах и наследовании, но у Swift еще есть немало средств структурирования программ. Знакомьтесь: протоколы и расширения. Протоколы в Swift позволяют определить шаблон с перечнем методов и свойств, необходимых для некоторой цели или некоторого блока функциональности. Протокол включается классом, структурой или перечислением, в которых содержится его фактическая реализация. Типы, которые предоставляют необходимую функциональность, называются поддерживающими этот протокол. Расширения позволяют легко добавлять новую функциональность в существующие типы.

Фабрика роботов	238
Наследование протоколов	243
Изменяющие методы	245
Протоколы как типы и коллекции	248
Вычисляемые свойства в расширениях	252
Расширение протокола	256
Полезные протоколы и вы	257
Поддержка протоколов Swift	260



Может, мы и не протокольные роботы, но мы узнаем хороший протокол, когда увидим его!

Опциональные типы, распаковка, обобщение и другое

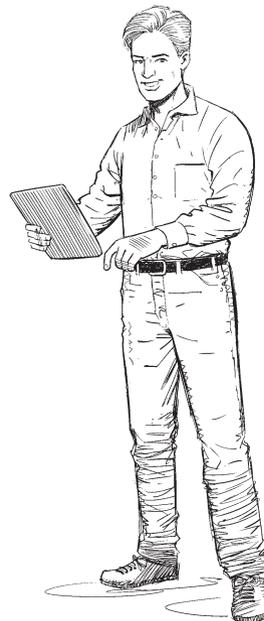
9

Неизбежные опциональные типы

Обработка несуществующих данных может быть весьма непростым делом. К счастью, в Swift для этого существует решение: опциональные типы. В Swift опциональный тип позволяет работать со значением или выполнить действия при отсутствии значения. Это одно из многих проявлений безопасности при проектировании Swift. Ранее вы уже встречались с опциональными типами в коде, а теперь мы изучим их более подробно. Опциональные типы улучшают безопасность Swift, потому что с ними снижается риск написания кода, который перестает работать при отсутствии данных, или возврата значения, которое в действительности значением не является.



Обработка отсутствующего значения	269
Для чего могут понадобиться опциональные типы	270
Опциональные типы и обработка отсутствующих данных	273
Распаковка опциональных типов	274
Распаковка опциональных типов с ключевым словом guard	277
Принудительная распаковка	278
Обобщения	288
Очередь с обобщениями	289
Новый тип Queue	290



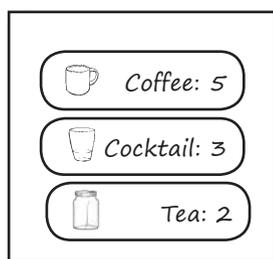
10

Знакомство со SwiftUI

Пользовательские интерфейсы

Пришло время применить на практике все приемы, возможности и компоненты **Swift**, о которых вы узнали в книге: мы займемся построением **пользовательских интерфейсов**. В этой главе мы сведем все воедино для построения первого настоящего пользовательского интерфейса. Он будет строиться на основе **SwiftUI**, **UI-фреймворка для платформ Apple**. Мы по-прежнему будем использовать Playgrounds (по крайней мере на первом этапе), но все, что здесь будет делаться, заложит фундамент для реальных приложений iOS. Приготовьтесь: в этой главе будет много кода и новых концепций. Вдохните поглубже и переверните страницу, чтобы с головой погрузиться в *SwiftUI*.

*У вас будет время выпить,
но только после того, как вы
освоите SwiftUI...*



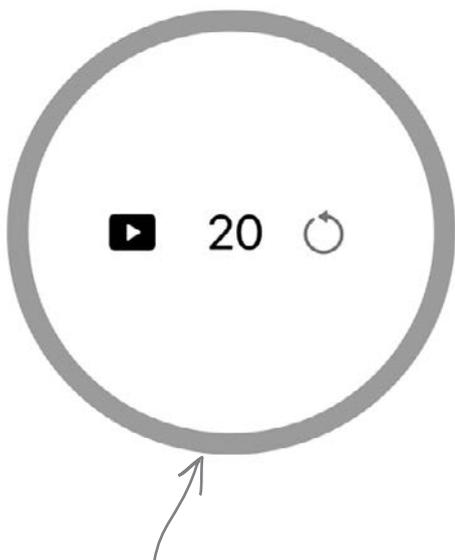
А что это вообще такое – UI-фреймворк?	294
Ваш первый пользовательский интерфейс SwiftUI UI	296
Строительные блоки пользовательских интерфейсов	300
Создаем список (и неоднократно возвращаемся к нему, чтобы довести до совершенства)	301
Пользовательские интерфейсы с состоянием	303
Как работают кнопки	304
Давайте посмотрим, насколько далеко мы зашли	307
ЗАДАЧА: Написать приложение на базе SwiftUI	311
Создание в Xcode нового проекта SwiftUI для iOS	312
Ваша среда Xcode должна выглядеть примерно так	314
Создание нового типа для хранения элемента списка задач	315
Однозначная идентификация каждого элемента списка задач	318
Создание пользовательского интерфейса приложения	319
Запустите приложение и посмотрите, что произойдет...	325
Реализация сохранения списка задач	326
Значит, это и есть UI-фреймворк?	330

11

Практическое применение SwiftUI

Круги, таймеры, кнопки — выбирайте!

SwiftUI вовсе не ограничивается кнопками и списками. В интерфейсах также можно использовать геометрические фигуры, анимации и многое другое! В этой главе будут рассмотрены некоторые **расширенные возможности построения пользовательских интерфейсов в SwiftUI** и их связывания с источниками данных, содержимое которых не генерируется пользователем (как в случае со списком задач). SwiftUI позволяет строить **пользовательские интерфейсы** с обработкой **событий**, поступающих из разных источников. Мы будем работать в Xcode, среде разработки от компании Apple, а основное внимание будет уделяться приложениям для iOS, но все, что вы узнаете в этой главе, в равной степени применимо к SwiftUI для iPadOS, macOS, watchOS и tvOS. Вперед, глубины SwiftUI ожидают вас!



Создайте свой собственный круг!

Что интересного можно сделать с UI-фреймворком?	338
Создание нового проекта SwiftUI для iOS	341
Пользовательский интерфейс и функциональность Executive Timer	343
Создание основных элементов приложения	344
Составляющие пользовательского интерфейса	345
Настройка пользовательского интерфейса приложения	346
Программирование составляющих пользовательского интерфейса	347
Объединение трех элементов	350
Завершающие штрихи	352
Запустите приложение и посмотрите, что произойдет...	353
Представления с вкладками	354
Создайте представления, которые вам нужны	354
Постройте представление TabView, содержащее ваши представления	354
Создание нового представления ContentView с вкладками	360
Создание вкладок и TabView	361
Запустите новую версию Executive Timer	363

12

Приложения, Веб-программирование и Все такое Собирая все вместе

Вы значительно продвинулись в изучении Swift. Вы освоили Playgrounds и Xcode. Было понятно, что когда-нибудь нам придется **попрощаться**, и сейчас этот момент настал. Расставаться нелегко, но мы знаем, что вы справитесь. В этой главе — последней, в которой мы будем вместе с вами (в этой книге), — мы еще раз **пройдемся по многим концепциям**, которые вы изучили, и совместно построим несколько приложений. Мы убедимся в том, что ваши навыки Swift закреплены, и дадим некоторые рекомендации относительно того, **что делать дальше**, — своего рода домашнее задание, если хотите. Это будет интересно, и мы расстанемся на высокой ноте.



Путешествие должно завершиться...	366
Построение заставки	371
Пошаговая сборка экрана заставки	372
Совместное использование состояния	378
На помощь приходит старый знакомый...	379
Построение приложения с несколькими представлениями, совместно использующими состояние	380
Построение приложения с двумя представлениями	381
ObservableObject	381
Первое представление	382
Второе представление	383
И снова первое представление	384
Первое представление (продолжение)	385
AsyncImage с наворотами	390
Varog: веб-фреймворк для Swift	394
Отправка данных средствами Varog	397

Как пользоваться этой книгой

Введение



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

Для кого написана эта книга?

Если на вопросы:

- 1 В вашем распоряжении есть устройство macOS или iPadOS, на котором работают последние общедоступные версии этих операционных систем?
- 2 Вы хотите изучить принципы программирования на примере языка Swift, чтобы потом продолжить свое путешествие в мире Swift?
- 3 Вы хотите в один прекрасный день заняться разработкой приложений для iPhone или любых других устройств в экосистеме Apple или изучить перспективный язык для написания веб-приложений?

вы отвечаете положительно, то эта книга для вас.

Кому эта книга не подойдет?

Если вы ответите «да» на любой из следующих вопросов...

- 1 Вы отличный разработчик с опытом программирования на macOS, iOS или Swift, которому нужен справочник?
- 2 Вы не хотите быть программистом и не хотите учиться программировать?
- 3 Вам не нравится пицца, еда, напитки или неуклюжие шутки?

...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 Swift *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.

В наши дни вы вряд ли попадете на обед к тигру. Но наш мозг постоянно остается настороже. Просто мы об этом не знаем.

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

Но как же мозг *знает*, что важно? Представьте, что вы отправились на прогулку, и вдруг прямо перед вами появляется тигр. Что происходит в вашей голове и в 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 ни в коем случае нельзя вывешивать фото с этой вечеринки на своей страничке в соцсетях.

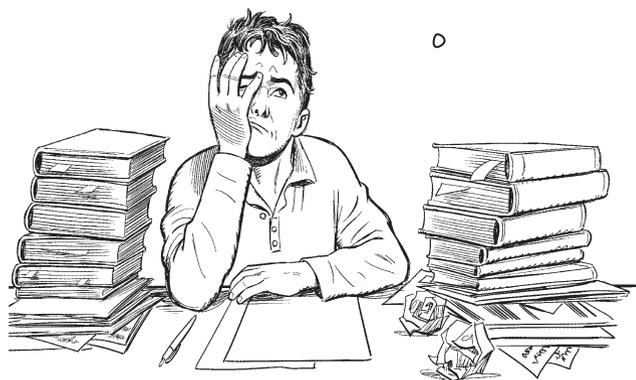
Нет простого способа сказать своему мозгу: «Послушай, мозг, я тебе, конечно, благодарен, но какой бы скучной ни была эта книга и пусть мой датчик эмоций сейчас на нуле, я *хочу* запомнить то, что здесь написано».

Ваш мозг считает, что **ЭТО** важно.



Ваш мозг полагает, что **ЭТО** можно не запоминать.

Замечательно. Еще 377 сухих скучных страниц.



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 нечто большее, чем просто прочесть текст. Мы знаем, как заставить ваш мозг работать.

Основные принципы серии «Head First»:

Наглядность. Графика запоминается гораздо 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.

Разговорный стиль изложения. Недавние исследования показали, что при разговорном стиле изложения материала (вместо формальных лекций) улучшение результатов на итоговом тестировании составляло до 40%. Рассказывайте историю, вместо того чтобы читать лекцию. Не относитесь к себе слишком серьезно. Что скорее 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.

Привлечение (и сохранение) внимания читателя. Ситуация, знакомая каждому: «Я очень хочу изучить это, но засыпаю на первой же 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 безразлично. 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 задумываетесь. 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 узнать, как программировать на Swift, и по возможности быстрее. Вы хотите запомнить прочитанное и применять новую информацию на практике. 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 на Swift так же важно, как и 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 интересуется, 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 стóит тысячи 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 включено более 80 *упражнений*, потому что мозг лучше запоминает, когда вы *работаете* самостоятельно. Мы постарались сделать их непростыми, но интересными — то, что предпочитает большинство читателей.

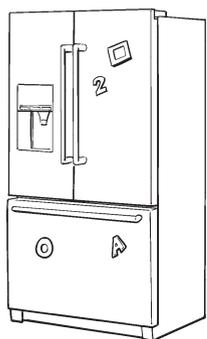
Мы совместили *несколько стилей обучения*, потому что одни читатели любят пошаговые описания, другие стремятся сначала представить «общую картину», а третьим хватает фрагмента кода. Независимо от ваших личных предпочтений, полезно видеть несколько вариантов представления одного и того же материала.

Мы постарались задействовать *оба полушария вашего мозга*: это повышает вероятность усвоения материала. Пока одно полушарие мозга работает, другое 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 качественнее 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 использованы антропоморфные образы. Ведь *вы* человек. И ваш мозг уделяет больше внимания *людям*, а не *вещам*.



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.

Вырежьте и прикрепите на холодильник.

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 всё. **Врезки – часть основного материала!** Не пропусайте 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 запомнить, произнесите вслух. А еще лучше – попробуйте объяснить кому-нибудь другому. Вы быстрее усвоите материал и, возможно, откроете что-то новое.

6 Пейте воду. И побольше.

Мозгу нужна влага, так он лучше работает. 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 начинает уставать. Если вы стали 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 почувствовать ничего.

9 Пишите программы!

Научиться программировать можно только одним способом: **писать код**. Именно этим вам предстоит заняться, читая книгу. Подобные навыки лучше всего закрепляются практикой. В каждой главе вы найдете упражнения. Не пропускайте их. Не бойтесь **подсмотреть** в решение задачи, если не знаете, что делать дальше! (Иногда можно застрянуть на элементарном.) Но все равно пытайтесь решать задачи самостоятельно. Пока ваш код не начнет работать, не стоит переходить к следующим страницам 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.

Мы начинаем с изложения основных концепций Swift и сведем все воедино только после того, как будет заложен надежный фундамент.

Невозможно написать приложение для iPhone, не зная, как работают переменные и константы (и многое другое). По этой причине мы начнем с азов, прежде чем переходить к основному материалу.

Мы не пытаемся рассказать всё во всех подробностях.

Swift — достаточно обширная область, и в мире найдется немало хороших книг (в том числе и написанных нами!), в которых Swift рассматривается на разных уровнях. Было бы неразумно обсуждать все аспекты Swift в одной книге. Мы рассматриваем то, что вам необходимо знать, чтобы уверенно чувствовать себя на начальной стадии.

Мы отобрали самые полезные темы.

Существует много способов построения пользовательских интерфейсов на базе Swift, от AppKit до UIKit и SwiftUI. В книге мы решили изложить азы работы со SwiftUI и не рассматривать остальные. Но поскольку в книге вы узнаете обо всех структурных элементах, вам будет гораздо проще изучать AppKit в будущем.

Упражнения обязательны к выполнению.

Упражнения и задачи являются частью основного содержания книги, а не дополнительным материалом. Некоторые помогают запомнить новую информацию, некоторые — лучше понять ее, а некоторые — научиться применять ее на практике. *Не пропускайте упражнения.* Необязательными являются только ребусы «У бассейна», но следует помнить, что они хорошо развивают логическое мышление.

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 ставят своей целью успешное запоминание, но это не справочник, а учебник, поэтому некоторые концепции излагаются в книге по несколько раз.

Мы постарались сделать примеры по возможности компактными.

Наши читатели говорят, что им не хочется просматривать 200-страничный листинг в поисках двух строк, которые нужно понять. Многие примеры в книге приводятся в минимально возможном контексте, чтобы часть, которую вы изучаете, была простой и понятной. Не ожидайте, что все примеры будут полностью защищенными от ошибок или хотя бы полными, — они написаны в учебных целях и не всегда полностью функциональны.

Мы разместили большой объем кода в интернете, чтобы вы могли копировать его в Playgrounds и Xcode. Код доступен по адресу <https://secretlab.com.au/books/head-first-swift>.

Упражнения «Мозговой штурм» не имеют ответов.

В некоторых из них правильного ответа вообще нет, в других вы должны сами решить, насколько правильны ваши ответы (это является частью процесса обучения). В некоторых упражнениях «Мозговой штурм» приводятся подсказки, которые помогут вам найти нужное направление.

Научные редакторы

Тим Нагент



Ник Сейрс



Ишмаэл Шабаз



Огромное спасибо всем, кто помог нам разобраться с технологическими сторонами этой книги. Они потратили много времени на проверку материала и его качества, а также на оповещения нас о том, что мы сделали что-то неразумное. Мы не всегда воспринимали ваши комментарии буквально, но они неизменно помогли нам улучшить книгу.

Нескольких человек мы хотим поблагодарить особо: это **Тим Нагент**, **Ник Сейрс** и **Ишмаэл Шабаз**.

Благодарности

Наш редактор:

Нам не удалось бы написать эту книгу без поддержки **Мишель Кронин**. Мы бы непременно написали нечто достойное ее и разместили здесь, если бы это вообще было возможно. За прошедшие годы мы написали много книг, но эта, пожалуй, была самой сложной, а работа над ней заняла больше всего времени. Нам доводилось работать с разными редакторами, пока мы не встретились с Мишель (кстати, все они были замечательными), но только с участием Мишель книга стала принимать правильную форму. Она постоянно помогала нам, с ней было интересно, и наше общение во время многих, многих встреч было просто фантастическим. Надеемся поработать с тобой над другими проектами. И еще раз: без тебя у нас ничего бы не вышло.



↑ Мишель Кронин

Команда издательства O'Reilly:

Огромное и искреннее спасибо **Кристоферу Фошеру** — выпускающему редактору этой книги, без которого нам не удалось бы так четко собрать книгу в единое целое. Сожалеем, что у нас все так плохо с InDesign.

Также спасибо **Кристен Браун**, которая помогла нам отшлифовать материал, и **Рэчел Хед** за потрясающе качественную редактуру (как обычно) и поддержку.

Благодарим **Зэна Маккуэйда**, который не только был невероятно интересным и веселым собеседником на всех наших встречах, но и терпел наши жалобы на InDesign, Apple и все между ними.



Рэчел Румелиотис

Еще хотим поблагодарить нашего друга и (одного из) редакторов O'Reilly: **Рэчел Румелиотис**. Нам не хватает наших встреч на конференциях каждые несколько месяцев, и мы надеемся, что все начнется снова, когда вот это всё *широкий жест руками* наконец-то закончится.

Спасибо **Брайану Макдональду** — одному из редакторов, работавших с нами в течение долгого времени, а также человеку, который уговорил нас заняться написанием книг, — Нилу Голдстейну.

Также нам ужасно жалко, что мы всегда используем австралийский диалект, приятель.

Спасибо всему коллективу O'Reilly Media в целом. Они лучшие... буквально. Другой такой команды не существует, и каждый новый человек из O'Reilly, с которым мы познакомились, был приятным в общении, высококлассным специалистом и в целом интереснейшей личностью.

1. Знакомство со Swift

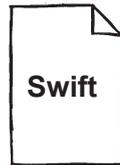
Приложения, системы и не только!

Есть множество разных мест, в которые можно добраться с помощью Swift. Но поездка всегда будет безопасной и быстрой, а решение — понятным и хорошо читаемым!

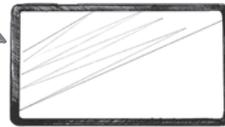


Swift — язык программирования, на который можно положиться. Вам не будет стыдно познакомить с ним вашу семью. Он безопасен, надежен, быстр, доступен и несложен. И хотя Swift получил наибольшую известность как язык программирования для платформ Apple, таких как iOS, macOS, watchOS и tvOS, проект с открытым кодом Swift также работает в Linux и Windows и постепенно набирает популярность как язык системного программирования, а также как серверный язык. На нем можно строить все что угодно, от мобильных приложений до игр, веб-приложений, фреймворков. Итак, за дело!

Swift — универсальный язык



На Swift можно создавать программы для **macOS**, **Linux** и **Windows**. В системе macOS можно строить полнофункциональные графические приложения при помощи UI-фреймворка, к которому мы вернемся позднее.



Swift для iPad, iPhone и iPod позволяет строить приложения **iOS** и **iPadOS**. Кроме того, существует возможность строить приложения tvOS для Apple TV и приложения watchOS для Apple Watch.



Swift также может использоваться для создания **веб-приложений**. Для веб-сайтов используются серверные части или статически генерируемые страницы, а для приложений — серверные API, что упрощает синхронизацию.



Swift также может использоваться в области **обработки данных** и **машинного обучения**, а также в области **системного программирования** для решения разнообразных задач: научных, моделирования и т. д.



Swift также распространяется с открытым кодом. Проект с открытым кодом Swift энергично развивается, эффективно управляется, а процесс привлечения новых участников отличается ясностью и доброжелательностью.

О том, как присоединиться к работе над проектом, будет рассказано ближе к концу книги.



Что бы вы ни хотели сделать, это можно сделать на Swift.

Эволюция Swift

Язык Swift постоянно развивается, в нем ежегодно появляются новые дополнения: от скромного старта в Swift 1 до дня сегодняшнего и в будущее.

В каждом **крупном** обновлении Swift добавлялись новые **языковые средства**, а некоторые элементы **синтаксиса** устаревали. С первых объявлений и выпуска в 2014 г. Swift стал одним из наиболее быстро растущих и самых популярных языков в истории программирования.

В последние годы Swift регулярно занимает место в **десятке самых популярных языков программирования**, а количество пользователей и проектов, использующих этот язык, продолжает расти. Навыки программирования на Swift также пользуются высоким спросом, так что не забудьте добавить Swift в свое резюме, когда завершите работу над книгой!

Swift создавался как язык, предназначенный исключительно для платформ Apple: iOS, macOS, tvOS, watchOS, но с момента его перехода на распространение с открытым кодом в 2015 г. он вырос, а его возможности расширились. Он прекрасно подходит для системного программирования, научных задач, веб-программирования и многих других областей.

Что бы вы ни собирались создавать, при выборе языка программирования Swift станет отличным кандидатом.

Их можно подсчитывать разными способами, причем некоторые способы логичнее других. Как бы то ни было, Swift занимает высокие места во всех списках!

Ключевые моменты

- **Средства языка** — такие вещи, как структуры, протоколы и представления SwiftUI.
- **Синтаксис** подразумевает конкретное размещение и использование таких элементов, как ! и ?, квадратные скобки и тому подобное.

В книге мы не будем рассматривать все без исключения грани Swift, но вы будете знать все необходимое для дальнейшего самостоятельного изучения!

Стремительная эволюция Swift

2014

Swift 1

Появление Swift было анонсировано (ко всеобщему удивлению и под фанфары) на большой конференции разработчиков Apple (WWDC) в июне 2014 года. В то время Swift был запатентованным языком, который мог использоваться только для платформ Apple, и он распространялся с закрытым кодом.



Первые выпуски Swift поддерживали только Xcode для macOS.

2015



Язык Swift был переведен на модель с открытым кодом в декабре 2015 г., в версии 2.2. С тех пор разработка Swift ведется у всех на виду по адресу <https://github.com/apple/swift>.

Swift 2

Версия Swift 2 была представлена на конференции WWDC в 2015 г. Многие элементы синтаксиса изменились на основании обратной связи с сообществом, а некоторые языковые средства эволюционировали. Swift продолжал набирать популярность и признание в сообществе.

2016

Swift 3

В версию Swift 3, выпущенную в сентябре 2016 г., был включен ряд изменений синтаксиса, также основанных на обратной связи от сообщества. Выход Swift 3 ознаменовал начало новой эпохи стабильности для Swift: было обещано, что в будущем синтаксис не будет изменяться так часто.



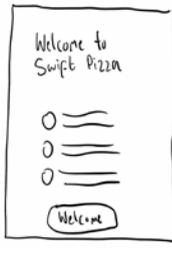
Версия Swift Playgrounds была выпущена для iPad одновременно с выпуском Swift 3.

Изображение молотка от Iconic из проекта Noun Project.

Стремление в будущее

Swift 4

Версия Swift 4 вышла в сентябре 2017 г. В ней появились новые возможности, но с сохранением более высокого уровня стабильности между версиями, чем прежде. Проект с открытым кодом Swift продолжал завоевывать новые позиции, а популярность языка продолжала расти.



SwiftUI, UI-фреймворк на базе Swift, был запущен в середине 2019 г.

Светлое будущее

Swift ждет светлое будущее. На горизонте появляется версия Swift 6; она сохранит синтаксис и функциональность Swift 5 и дополнит их рядом новых интересных возможностей. В Swift 5 появились новые средства (например, акторы), которые будут рассмотрены позднее в этой книге.

2017

2018



При проектировании в Swift были включены лучшие элементы других языков, включая Objective-C, Rust, Haskell, Ruby, Python, C# и т. д.

2019

Swift 5

Swift 5 — основная версия, которая рассматривается в книге, — была выпущена в марте 2019 г.; миру была представлена полностью стабильная версия Swift.

2020



Версия Swift Playgrounds была выпущена в виде приложения macOS в феврале 2020 г.

2021

2022

Как вы будете писать код Swift

↖ Да, именно вы!

Каждому языку программирования необходим инструмент для написания кода и запуска написанного кода. Чтобы сделать что-то полезное со Swift, вам неизбежно придется выполнить код Swift.

Как же это делается? В большей части книги мы будем использовать приложение **Playgrounds**, разработанное компанией Apple! →

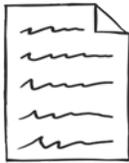


Playgrounds представляет собой нечто вроде (коллекции) больших традиционных текстовых файлов, но оказывается, вы также можете выполнять каждую написанную строку в компиляторе Swift и посмотреть результат.

Playgrounds будет использоваться в большой части кода Swift, написанного в книге.

↖ ...Но не во всем. Мы объясним чуть позднее.

Работа со Swift в Playgrounds проходит в три стадии:



Написание кода

1

Вы пишете свой код Swift в Playgrounds, который представляет собой **редактор для программистов** — такой же, какой может использоваться при работе с другими языками. Вы можете сохранить весь свой код Swift в одном файле или же разбить его на *несколько файлов*. Выбирайте по своему усмотрению.

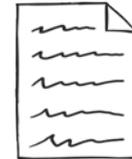
↖ О том, как это делается, будет рассказано позднее.



≡ Выполнение

2

Затем вы выполняете свой код Swift — либо *весь сразу*, либо *строку за строкой*, в Playgrounds. Вы видите ошибки, а также *вывод* вашего кода рядом с каждой строкой и в консоли под областью редактирования кода.



Настройка

3

Playgrounds формирует настолько быстрый цикл «написание кода/запуск», что вы можете легко внести изменения, доработать свой код и привести его в форму, необходимую для достижения ваших целей. Скорость разработки Swift — одна из самых сильных сторон языка. Пользуйтесь ею разумно!



↖ Использование Playgrounds — самый быстрый способ переключения между выполнением и модификацией кода, чтобы вы получили хорошее представление о работе программы.

Но я хочу изучить Swift, чтобы писать приложения и делиться своими идеями в App Store! Для чего мне изучать Playgrounds, когда я могу просто написать приложение в Xcode прямо сейчас?!

Изучать Swift в Playgrounds проще и быстрее

Работа в Playgrounds означает, что вам не придется беспокоиться обо всем шаблонном коде, который используется при построении приложений, а также о компиляции вашего кода в приложение, его тестировании на эмуляторе устройства iOS, тестировании на физическом устройстве iOS и отладке кода из-за незначительных различий между разными видами оборудования iOS и iPadOS.

Playgrounds позволяет *сосредоточиться на коде Swift*, не отвлекаясь на все остальные задачи, сопровождающие построение приложений (или написание серверной части веб-сайта, или решение задач обработки данных или любых других задач, которые могут решаться при помощи Swift). Именно поэтому мы начнем с Playgrounds — **эта среда избавляет вас от всего второстепенного.**

Такой подход ничем не хуже: вы пишете реальный, работоспособный код Swift и можете заниматься разработкой приложений (и не только) с использованием Swift.

Вы читаете эту книгу потому, что вы хотите писать приложения iOS (и это здорово, кстати!). Пожалуйста, будьте с нами и используйте Playgrounds. Это действительно лучший способ изучения Swift.



Xcode и Playgrounds

Возникает большой соблазн взяться прямо за Xcode, когда вы начинаете изучать Swift, особенно если у вас уже есть опыт программирования.

Если вы хотите изучить Swift, вместо того чтобы изучать Swift вместе со всеми непостижимыми странностями (очень мощной и очень сложной) среды Xcode, начните с Playgrounds. Это позволит вам сосредоточиться на Swift.

Путь, лежащий перед вами

Прежде чем писать приложения на Swift, необходимо сначала освоить процесс программирования на Swift.

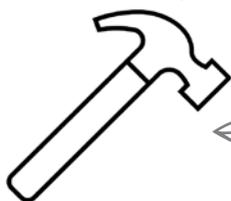
Playgrounds



Playgrounds — наша отправная точка. Эта интерактивная среда доступна для macOS и для iPadOS, и она позволяет очень быстро создавать работоспособный код при помощи компилятора Swift. Вы сможете быстро просмотреть результаты, опробовать новые идеи и разделить работу на логические страницы в одной среде Playground. Мы не станем это повторять: все, что делается в Playgrounds, делается очень быстро.

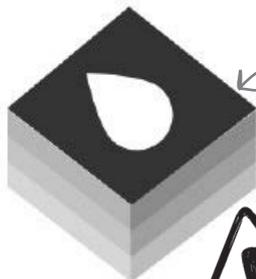
Приложения iOS, macOS, tvOS и watchOS

Когда вы основательно прокачаете свои навыки Swift, вы можете перевести их из Playgrounds в Xcode. В Xcode вы будете использовать Swift для построения приложений для всех платформ Apple.



Попутно вы изучите SwiftUI — UI-фреймворк на базе Swift от компании Apple. Мы начнем его использовать в Playgrounds, а немного позднее переместимся в Xcode.

Веб-сайты



В завершение мы в общих чертах покажем, как использовать Swift в области веб-программирования с Vapor и Publish — фреймворками для построения веб-сайтов и серверных частей на базе Swift.

Data Science



Возможно, у вас появится соблазн перейти сразу к Xcode.

Ничто не мешает вам установить Xcode, но пока мы не будем пользоваться этой средой. Мы еще вернемся к ней позднее в этой книге.

Установка Playgrounds

Загрузка и установка Playgrounds в macOS

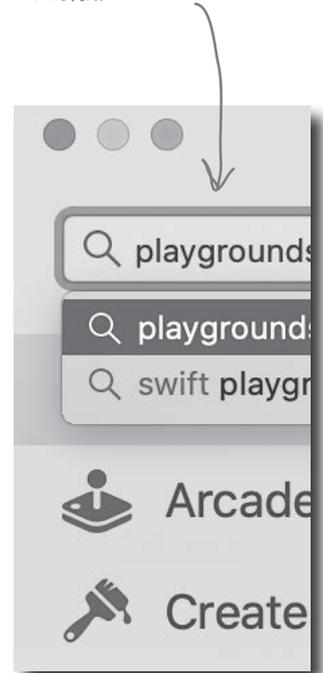
- 1 Откройте приложение App Store на вашем компьютере, найдите **поле поиска** и введите строку «playgrounds». Нажмите клавишу Return.
- 2 Подождите, когда App Store загрузит страницу с результатами поиска. Щелкните на строке **Swift Playgrounds** в списке (значок с оранжевой птичкой).
- 3 Щелкните на **кнопке установки** и подождите, когда приложение Playgrounds будет загружено и установлено на вашем компьютере.



Также обратите внимание на оранжевый логотип Swift с птичкой.

Кнопка установки находится здесь.

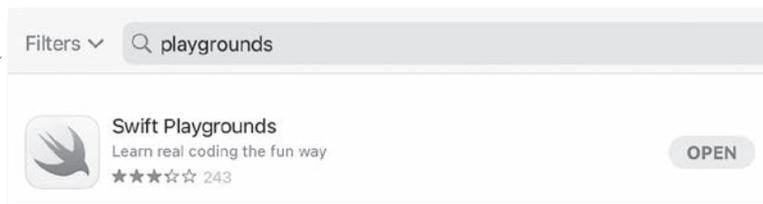
Проще всего воспользоваться средствами поиска.



Я очень мобилен и стараюсь использовать iPad всюду, где возможно. Я заметил, что у iPadOS в App Store тоже есть версия Playgrounds. Могу ли я использовать ее вместо версии для Mac?

Да, все, что мы делаем в macOS Playgrounds в этой книге, можно сделать в iPadOS Playgrounds.

Все, что делается в книге, можно делать в Playgrounds для macOS или iPadOS или смешивать эти две версии. Поиск Playgrounds в App Store для iPadOS приведет вас к iPadOS-версии Playgrounds. К счастью, iCloud синхронизирует ваш код между macOS и iPadOS, так что с технической точки зрения вы можете использовать обе версии!





Чтобы запустить Swift Playgrounds, перейдите в папку Applications и сделайте двойной щелчок на значке Swift Playgrounds.

Если вы предпочитаете использовать Launchpad или Spotlight для запуска приложений, эти способы работают и со Swift Playgrounds. В iPadOS найдите значок Playgrounds или запустите приложение через Spotlight.



Ключевые моменты

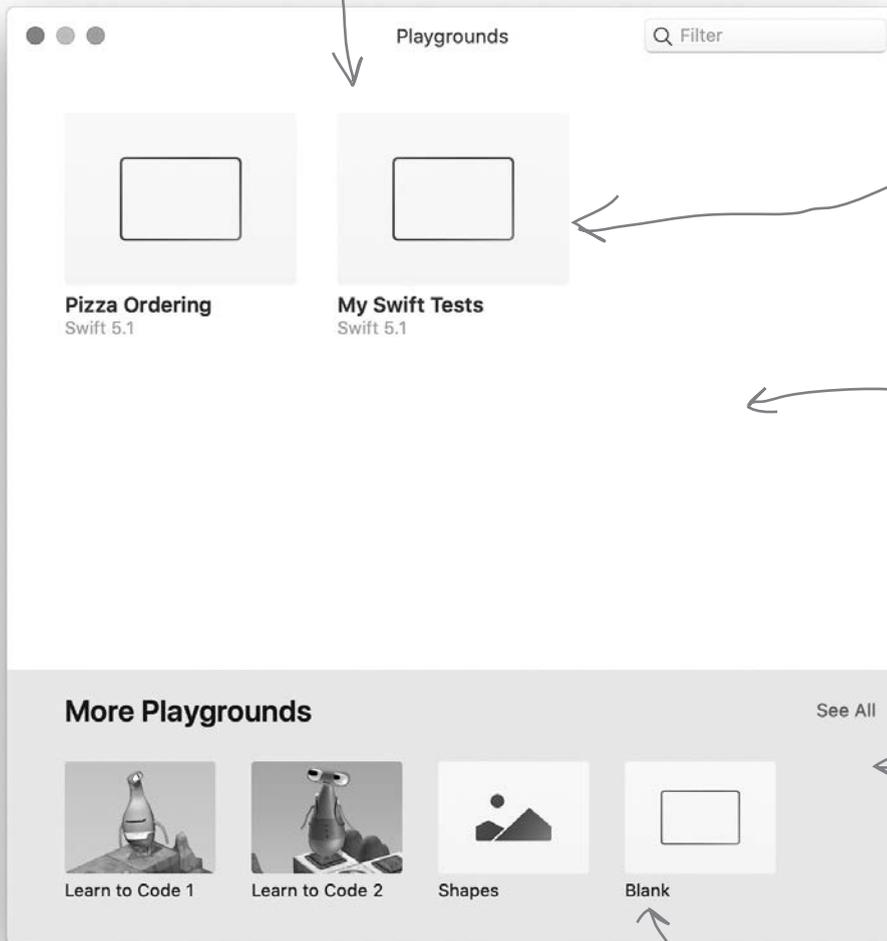
- Для написания кода Swift потребуется инструмент (или несколько разных инструментов), позволяющий писать код Swift, выполнять код Swift и изменять код Swift (прежде чем снова запустить его, снова изменить и т. д.)
- Один из таких инструментов — приложение Playgrounds компании Apple, доступное для iPadOS и macOS. Оно обладает минимальными возможностями, а основное внимание уделяется использованию языка программирования Swift без необходимости изучения как языка, так и инструментария.
- Среда Xcode компании Apple — еще один (более сложный) инструмент. Xcode — полностью интегрированная среда разработки, сходная с Visual Studio и другими аналогичными средами. Это очень большой, очень сложный инструмент, предназначенный для многих целей.
- Среда Xcode необходима, если вы хотите писать более сложные приложения для платформ Apple (например, для iOS). Мы будем использовать Xcode позднее в этой книге.
- Для изучения Swift лучше всего подходит Playgrounds. Вы даже сможете создавать собственные приложения!
- Весь код в книге, предназначенный для Playgrounds, будет работать в iPadOS и/или macOS. Выбор за вами.

Создание среды Playground

После запуска Swift Playgrounds вы сможете создать пустую среду Playground для работы или открыть существующую среду Playground, которая была создана ранее.

Чтобы открыть существующую среду Playground, щелкните на ней.

Если вы используете iCloud, ваши среды Playgrounds будут синхронизироваться между macOS и iPadOS автоматически.



Здесь будут отображаться все созданные вами среды Playgrounds.

Вы можете загрузить среды Playgrounds из сетевой библиотеки.

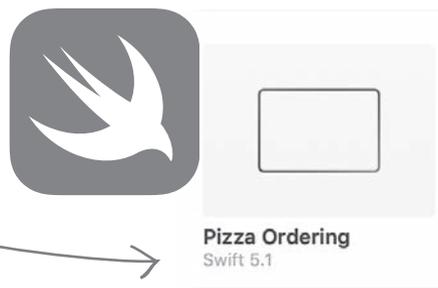
Чтобы создать новую пустую среду Playground, щелкните здесь.

Возможно, на вашем компьютере приложение Playgrounds будет выглядеть немного иначе, так как Apple часто обновляет его. Без паники! Основные элементы никуда не исчезнут.

Использование среды Playground для написания кода Swift

1 Создайте среду Playground

Создайте среду Playground в приложении Playgrounds.



2 Напишите код и запустите его

Напишите свой код (сколько угодно короткий или длинный), после чего запустите его при помощи этой кнопки.



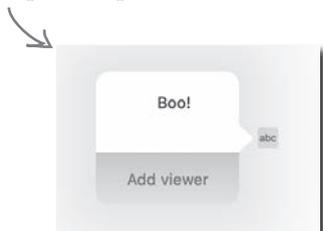
Кнопка Run может быть меньше, если вы используете окно Playgrounds меньших размеров.

Run My Code



3 Просмотрите выходные данные

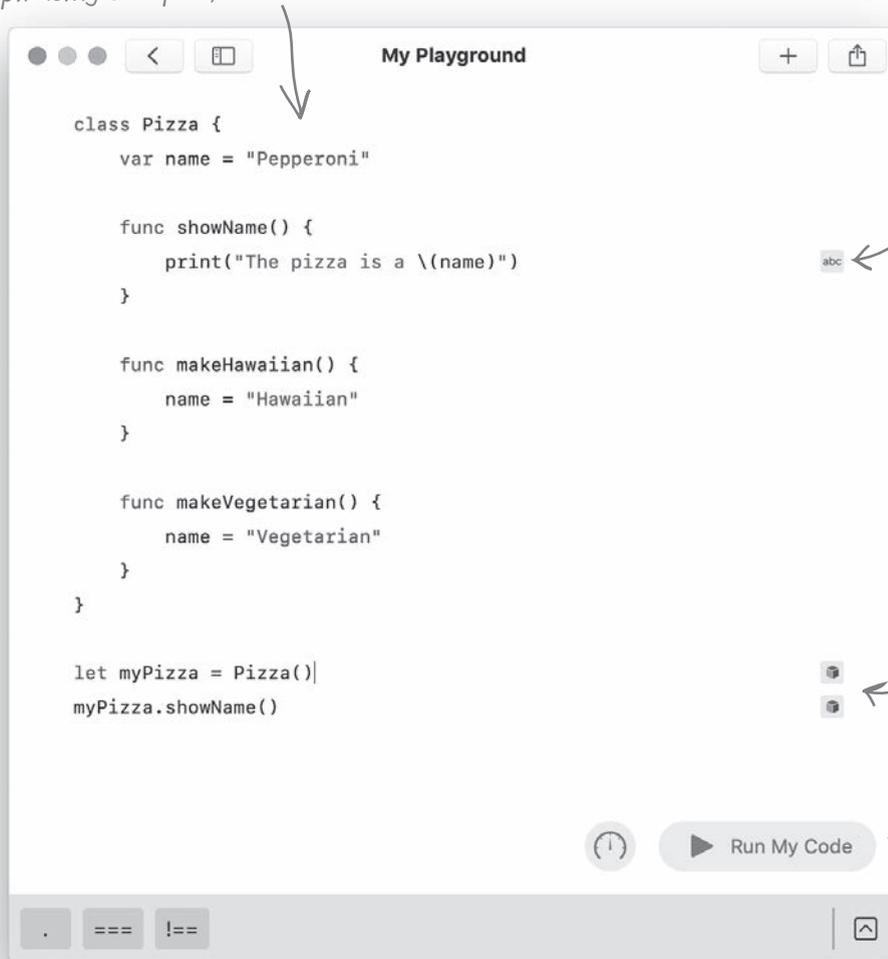
Когда выполнение вашего кода будет завершено, рядом со строками, которые выводят какой-то результат, будет отображаться индикатор. Щелкните на нем, чтобы добавить область просмотра.



```
print("Boo!")
```

Boo!

Среда Playground выделяет элементы вашего кода цветом, благодаря чему он проще читается.



Если фрагмент кода Swift вычисляет результат, рядом со строкой появляется небольшой значок.

Щелкните на значке результата, чтобы создать область просмотра, в которой отображается вывод или результат строки.

Эта кнопка используется для запуска кода.



Тест-драйв

Введите содержимое среды Playground на иллюстрации в новой среде Swift, которую вы сами создали. Запустите его.

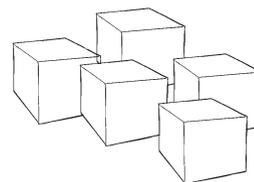
Что будет выведено? Используйте значки результата для добавления областей просмотра, в которых можно просмотреть результат.

Основные структурные элементы

Наше путешествие в страну Swift займет некоторое время, но что касается основ, мы введем вас в курс дела достаточно быстро. Сначала мы с головокружительной быстротой промчимся по азам Swift и дадим общее представление о многих областях языка. А после этого мы вернемся в начало и разберем каждый аспект Swift более подробно, главу за главой.

Существуют базовые структурные элементы, которые присутствуют практически в каждом языке программирования на планете (и наверняка, в языках программирования за ее пределами; в конце концов, программирование — штука универсальная).

К числу этих базовых структурных элементов относятся **операторы, переменные и константы, типы, коллекции, команды передачи управления, функции, протоколы, перечисления, структуры и классы.**

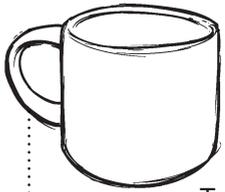


Кто и Что Делает ?

Мы еще не рассказывали, что делают те или иные базовые структурные элементы, но, возможно, вы и сами догадаетесь. Соедините каждый элемент языка с описанием того, что он делает. Решение приведено на с. 56 (мы уже провели одну линию, чтобы вам было проще взяться за дело).

Операторы	Именованные блоки кода.
Переменные и константы	Знаки или последовательности символов, используемые для проверки, изменения или объединения значений.
Типы	Механизм хранения данных по имени.
Коллекции	Заготовки для функциональности, которые можно использовать в программе.
Управляющие команды	Некоторые виды данных, которые можно хранить или представлять в программе.
Функции	Средство хранения групп значений.
Перечисления	Механизм группировки взаимосвязанных значений и функциональности.
Структуры	Механизм группировки взаимосвязанных значений и функциональности, которые могут наследоваться.
Протоколы	Средства для выполнения операций несколько раз или при определенных условиях.
Классы	Механизм группировки взаимосвязанных значений.

→ Ответы на с. 56.



Расслабьтесь

Никто не ожидает, что вы будете читать код Swift как заправский разработчик, припоминая историю и эволюцию Swift и выдавая кучу фактов о Swift, словно это вы изобрели этот язык.

Тем не менее вы потратили на книгу свои заработанные нелегким трудом деньги, поэтому мы считаем своим долгом приложить все усилия к тому, чтобы нагрузить вас знаниями о Swift. Возможно, сейчас вы поймете не все, но спустя какое-то время (причем не особо продолжительное) вы узнаете, как эти структурные элементы взаимодействуют друг с другом.

Помните код, который был приведен несколько страниц назад? Сейчас мы последовательно разберем его, чтобы вы поняли, как он работает.

```

class Pizza {
    var name = "Pepperoni"
    func showName() {
        print("The pizza is a \(name)")
    }
    func makeHawaiian() {
        name = "Hawaiian"
    }
    func makeVegetarian() {
        name = "Vegetarian"
    }
}

var myPizza = Pizza()
myPizza.showName()
    
```

Мы создаем класс с именем Pizza. В классе объединяются различные аспекты моделируемой сущности (то есть пиццы). Позднее мы создадим экземпляр класса Pizza и что-то сделаем с этими аспектами.

Создает переменную с именем name и присваивает ей значение «Pepperoni».

Создает функцию с именем showName, которая выводит name (то есть содержимое переменной name).

print приказывает Swift вывести данные для пользователя. Вскоре этот момент будет рассмотрен более подробно.

Создает функцию с именем makeHawaiian, которая присваивает переменной name строку «Hawaiian».

Делает то же самое, но для строки «Vegetarian».

Все, что предшествовало этой строке, было частью класса Pizza.

Создает переменную с именем myPizza, которая содержит экземпляр класса Pizza.

Вызывает функцию showName экземпляра класса Pizza, который мы создали ранее (myPizza).



Простите... Но я уже работала с другими языками программирования, и во многих из них код приходилось заключать в метод `main` или что-нибудь в этом роде. А как обстоят дела со Swift?

По крайней мере его не нужно специально определять. Код просто выполняется в Playgrounds.

Вам не понадобится метод `main` или что-либо подобное (например, точки с запятой).

В Swift вы также можете выполнять свой код так, как считаете нужным. В случае среды Playgrounds, с которой мы работаем в данный момент, выполнение следует от начала файла и продолжается сверху вниз.

Некоторые структурные элементы, о которых вы вскоре узнаете (например, классы), не будут выполняться так, что вы сможете немедленно увидеть их результаты, потому что они зависят от создания экземпляра в другой точке кода. Впрочем, в общем случае вам не понадобится метод `main` или какая-либо конкретная отправная точка.

Когда мы перейдем с Playgrounds на построение приложений на базе Swift, мы рассмотрим различные возможные начальные точки, которые могут существовать в приложении. А пока просто запомните этот момент.

Строки не нужно завершать точкой с запятой, а пробельные символы абсолютно ни на что не влияют.



Если вы перешли на Swift с другого языка, возможно, вы привыкли завершать строки кода точкой с запятой (;) или использовать пробельные символы, которые имеют особый смысл для логики вашего кода. В Swift эти правила не действуют.

Вы можете использовать символы ; для завершения команд, если уж вам их так не хватает по другим языкам, но мы так поступать не рекомендуем. Завершители команд расходятся со «стилем Swift», и они нехарактерны для программирования на Swift.

Возьмите в руку карандаш



Вы только делаете первые шаги в изучении Swift, но мы считаем, что вы достаточно умны и сможете обоснованно предположить, что происходит в коде Swift. Присмотритесь к каждой строке кода в приведенной ниже программе Swift и постарайтесь догадаться, что она делает. Запишите свои ответы на полях справа. Мы уже заполнили одну строку, чтобы вам было проще взяться за дело! Если вы захотите проверить свои ответы или окажетесь в тупике, версия со всеми объяснениями приведена на с. 49.

```
class Message {
    var message = "Message is: 'Hello from Swift!'"
    var timesDisplayed = 0

    func display() {
        print(message)
        timesDisplayed += 1
    }

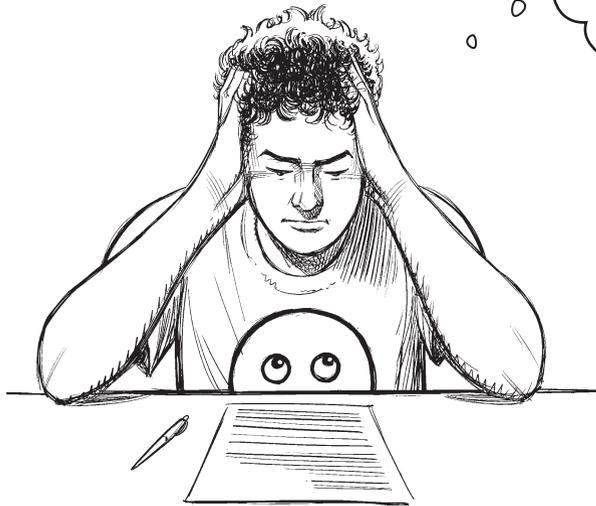
    func setMessage(to newMessage: String) {
        message = "Message is: '\(newMessage)'"
        timesDisplayed = 0
    }

    func reset() {
        timesDisplayed = 0
    }
}

let msg = Message()
msg.display()
msg.timesDisplayed
msg.display()
msg.timesDisplayed
msg.setMessage(to: "Swift is the future!")
msg.display()
msg.timesDisplayed
```

Объявляет переменную с именем `message` и присваивает ей строковое значение.

→ Ответы на с. 49.



Я слышал, что Swift — современный язык программирования. Что это значит? Как язык программирования может быть современным?

Swift стал итогом десятилетий развития и теоретических исследований языков программирования.

Swift называется современным, потому что в него был интегрирован опыт других языков программирования, а взаимодействие с пользователем, то есть с программистом, совершенствовалось в направлении удобочитаемости и простоты сопровождения кода. Программисту приходится вводить **меньший объем кода по сравнению с другими языками, а с языковыми средствами Swift код становится более **чистым**, вероятность ошибок в нем **снижается**. Swift — *безопасный* язык.**

Swift также поддерживает возможности, часто отсутствующие в других языках программирования, включая поддержку **международных языков, эмодзи, Юникод** и кодировку текста **UTF-8**. И еще вам не нужно уделять особое внимание управлению памятью — все происходит как по волшебству.

Что означает безопасность в этом контексте? Звучит хорошо, но что имеется в виду на самом деле?

Язык Swift безопасен, потому что он автоматически выполняет многие операции, для которых в других языках приходится писать специальный код (если они вообще возможны).

Есть много причин, по которым Swift часто называется безопасным языком. Например, переменные всегда инициализируются перед использованием, массивы автоматически проверяются на переполнение, а управление памятью, используемой программой, осуществляется автоматически.

В Swift также широко используются *типы-значения* — типы, при копировании которых происходит копирование данных (вместо копирования ссылок). А значит, вы можете быть уверены в том, что значение не будет случайно изменено из другой точки программы, пока вы работаете с ним.

Объекты Swift не могут быть равны `nil`, что помогает предотвратить ошибки на стадии выполнения. Впрочем, при необходимости вы все равно можете использовать `nil`, для чего понадобятся **опциональные типы**.

`nil` — признак отсутствия значения. Возможно, это обозначение (или `null`) уже встречалось вам в других языках. Одним из крупнейших источников ошибок в программировании, независимо от языка, становится попытка обращения к отсутствующему значению, то есть `nil`.

Возьмите в руку карандаш

С. 47

Решение

Не огорчайтесь, если что-то останется непонятным!

Все эти элементы очень подробно объясняются в книге, в основном на первых 40 страницах. Если Swift напоминает какой-нибудь язык, с которым вы работали в прошлом, часть материала покажется простой. Если нет — ничего страшного. Скоро все прояснится.

```

class Message {
    var message = "Message is: 'Hello from Swift!'"
    var timesDisplayed = 0

    func display() {
        print(message)
        timesDisplayed += 1
    }

    func setMessage(to newMessage: String) {
        message = "Message is: '\(newMessage)'"
        timesDisplayed = 0
    }

    func reset() {
        timesDisplayed = 0
    }
}

```

Создает класс (все, что заключено в фигурные скобки) с именем Message.

Объявляет переменную с именем message и присваивает ей строковое значение.

Объявляет переменную с именем timesDisplayed и присваивает ей целое число.

Объявляет функцию с именем display. Внутри функции выводится переменная message, а переменная timesDisplayed увеличивается на 1.

Объявляет функцию с именем setMessage, которая получает параметр String с именем newMessage. Внутри функции в переменной message сохраняется строка, переданная в newMessage, а переменной timesDisplayed присваивается 0.

Объявляет функцию с именем reset. Внутри функции переменной timesDisplayed присваивается значение 0.

Здесь заканчивается объявление класса Message.

```

let msg = Message()
msg.display()
msg.timesDisplayed
msg.display()
msg.timesDisplayed
msg.setMessage(to: "Swift is the future!")
msg.display()
msg.timesDisplayed

```

Создает экземпляр класса Message и сохраняет его в константе с именем msg.

В этих четырех строках вызывается функция display нашего экземпляра класса Message (msg), происходит обращение к переменной timesDisplayed, функция display вызывается снова, после чего не снова обращается к переменной timesDisplayed.

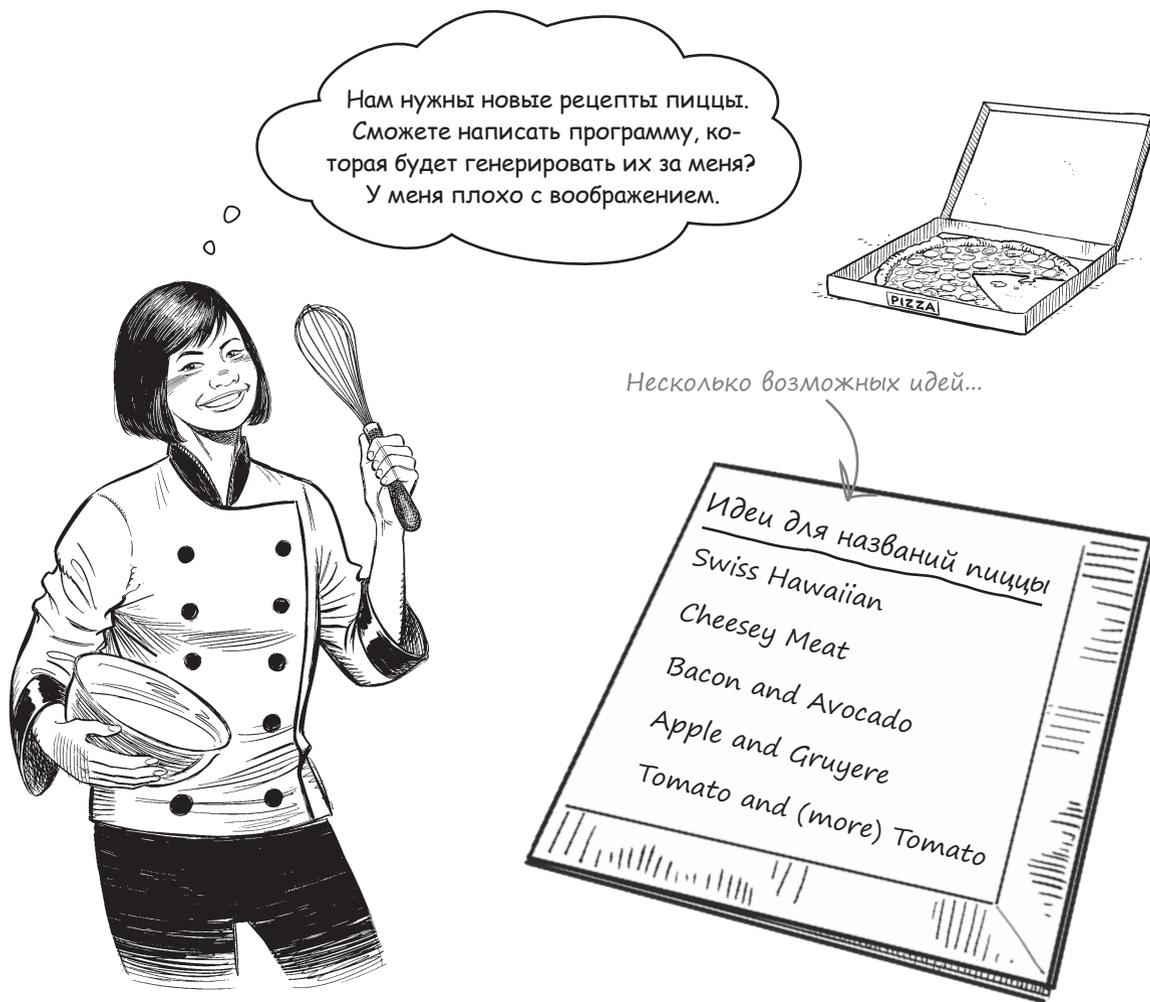
Вызывает функцию setMessage для нашего экземпляра класса Message (msg) и передает ей String в параметре.

Вызывает функцию display для нашего экземпляра класса Message (msg).

Это выражение обращается к переменной timesDisplayed для нашего экземпляра класса Message (msg).

Пример Swift

Со Swift столько мороки... Срочно нужна пицца! Но оказывается, местная пиццерия уже знает, что вы изучаете Swift, и предлагает применить ваши новые знания на практике.



Что нужно сделать

Мы создадим маленькую простую среду Swift Playgrounds, которая получает список всех возможных ингредиентов пиццы от шеф-повара и генерирует из этих ингредиентов случайную пиццу. Пока понятно?

Для этого необходимо решить три задачи:

1 Получить список ингредиентов

Нам понадобится список ингредиентов. Шеф-повар поможет вам с этим, но данные нужно где-то сохранить. Пожалуй, массив строк будет самым разумным решением.

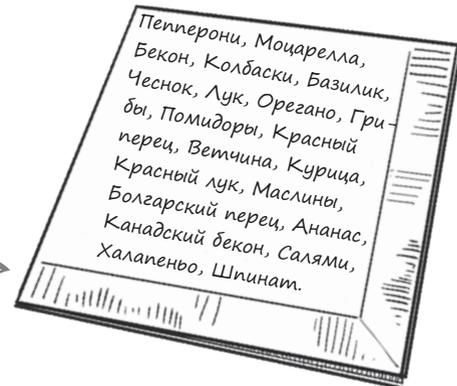
2 Выбрать случайный ингредиент

Также необходимо как-то выбрать из списка случайный ингредиент и сделать это четыре раза (потому что наша пицца состоит из четырех ингредиентов).

Если использовать массив для хранения списка ингредиентов, мы сможем вызвать функцию `randomElement` для получения случайного ингредиента.

3 Вывести случайно сгенерированную пиццу

Наконец, необходимо вывести случайно сгенерированную пиццу в формате «Ингредиент 1, ингредиент 2, ингредиент 3 и ингредиент 4».



Функция print

Вероятно, вы заметили в примерах кода строки, которые выглядят примерно так:

```
print("Some text to get printed!")
```

Это **функция print**, встроенная в Swift. Она приказывает Swift вывести указанные данные в текстовом виде.

Также иногда она используется в виде:

```
print("The pizza is a \(name)")
```

При таком использовании `print` включает в вывод значение переменной, имя которой заключено между `\` (и `)`. Удобно, верно? Мы вернемся к этому вопросу позднее, когда будем рассматривать **строковую интерполяцию**.

Готовый код



1 Построение списка ингредиентов

Первым шагом должно стать сохранение списка ингредиентов, полученного от шеф-повара, в удобной структуре данных.

Мы воспользуемся одним из типов коллекций, предоставляемых Swift, а именно **массивом** строк. Каждая строка, хранящаяся в массиве, представляет один из ингредиентов.

Массив ингредиентов создается примерно так:

```
let ingredients = [
    "Pepperoni", "Mozzarella",
    "Bacon", "Sausage",
    "Basil", "Garlic", "Onion", "Oregano",
    "Mushroom", "Tomato",
    "Red Pepper",
    "Ham", "Chicken",
    "Red Onion",
    "Black Olives",
    "Bell Pepper",
    "Pineapple",
    "Canadian Bacon", "Salami",
    "Jalapeño",
    "Spinach",
    "Italian Sausage", "Provolone",
    "Pesto", "Sun-Dried Tomato",
    "Feta",
    "Meatballs",
    "Prosciutto",
    "Cherry Tomato",
    "Pulled Pork", "Chorizo",
    "Anchovy", "Capers"
]
```

Начало списка определяется символом [.

Элемент 0

Этот массив строк (предназначенный для хранения ингредиентов) содержит 33 элемента. Это означает, что мы можем обращаться к элементам с 0 по 32, потому что первый элемент массива Swift считается элементом 0. Массивы будут использоваться в следующей главе.

Группировка некоторых элементов не имеет никакого особого смысла. Она нужна только для наглядности представления кода.

Конец списка определяется символом].

Элемент 32



Упражнение

Создайте новую среду Swift Playground и запрограммируйте список ингредиентов.

Используйте функцию `print` для вывода некоторых ингредиентов. Для обращения к отдельным ингредиентам можно использовать синтаксис вида `ingredients[7]`. Какой ингредиент вернет эта запись?

→ Ответ на с. 56.

2 Выбор четырех случайных ингредиентов

На втором шаге нужно выбрать случайный ингредиент (из списка, который мы только что сохранили в массиве) четыре раза подряд.

Тип коллекции `Array` в Swift поддерживает удобную функцию `randomElement`. Эта функция возвращает случайно выбранный элемент для массива, для которого она была вызвана.

Получение случайного элемента из массива `ingredients` выполняется так:

```
ingredients.randomElement()!
```

Знак `!` в конце необходим, потому что эта функция возвращает *опциональный тип*. Подробнее об этом будет рассказано позднее, но в двух словах дело обстоит так: здесь может быть значение (если в массиве хранятся ингредиенты), а может и не быть (потому что массив может быть пустым, и это вполне допустимо).

Для безопасности в Swift существует концепция опциональных типов, которые могут содержать значение, а могут и не содержать. Со знаком `!` значение игнорируется при его отсутствии, а если значение есть, то вы просто получаете его. *Когда вы поймете, как это работает, никогда так не делайте.* Эта тема более подробно рассматривается далее в этой главе.

Если существует какая-либо вероятность того, что этот массив изменится (в данном случае ее нет, потому что он объявлен как константа при помощи ключевого слова `let`), такое программирование станет небезопасным (потому что мы вызываем `randomElement()` и игнорируем вероятность того, что массив мог бы оказаться пустым).

Вспомогательное свойство `randomElement` может использоваться с нашим массивом (и другими типами коллекций Swift). У массивов есть и другие свойства и вспомогательные средства, включая `first` и `last`, возвращающие первый и последний элемент соответственно, и `append`, добавляющий новые данные в конец массива.



Упражнение

Откройте среду Swift Playgrounds со списком ингредиентов.

Снова используйте функцию `print` и выведите набор случайных ингредиентов с помощью функции `ingredients.randomElement()!`.

Попробуйте вывести последний элемент массива, используя свойство `last`. Какой это элемент?

Присоедините новый ингредиент в конец массива вызовом `ingredients.append("Banana")`, после чего снова выведите последний элемент массива. Убедитесь в том, что новый ингредиент находится именно там, где ему положено быть.

→ Ответ на с. 56.

3 Вывод случайной пиццы

Наконец, необходимо использовать массив ингредиентов и функцию `randomElement` для генерирования и вывода случайной пиццы.

Так как пицца должна состоять из четырех случайных ингредиентов, функцию нужно вызвать четыре раза, причем это можно сделать прямо внутри вызова функции `print`:

```
print("\(ingredients.randomElement()!), \(ingredients.randomElement()!),  
\(ingredients.randomElement()!), and \(ingredients.randomElement()!)")
```



Да, один случайный ингредиент может быть возвращен несколько раз.

Каждый отдельный вызов `randomElement()` понятия не имеет, как вы используете возвращенный им элемент и сколько раз вызывается функция — один раз или много. Так что теоретически можно сгенерировать пиццу, в которую не входит ничего, кроме ананаса.

↙ Red Pepper, Ham, Onion, and Pulled Pork
Pineapple, Mozzarella, Bacon, and Capers
Salami, Tomato, Prosciutto, and Meatballs
Pineapple, Pineapple, Pineapple, and Anchovy



Мозговой Штурм

В среде Swift Playgrounds прокомментируйте все, кроме определения массива `ingredients`.

Напишите код, необходимый для вывода случайно сгенерированной пиццы с четырьмя ингредиентами, как показано выше.

Проверьте, как работает программа. Все правильно?

Похоже, шеф-повару не нравится, что один ингредиент может повторяться несколько раз.

Удастся ли вам написать код так, чтобы этого никогда не происходило? Попробуйте.

Поздравляем, вы сделали свои первые шаги в Swift!

В этой главе вы прошли **большой** путь. Вы с ходу оказались в гуще событий, и вам было предложено разобраться в реальном коде Swift, пока вы осваивались с программированием Swift в среде Playgrounds. Также мы построили маленькую программу, которая помогает шеф-повару генерировать случайные рецепты пиццы. Очень полезно!

В следующей главе мы напишем более длинный код Swift. Настоящий, серьезный, реальный код, который делает что-то полезное (более или менее). Вы узнаете об основных структурных элементах Swift: **операторах, переменных и константах, типах, коллекциях, управляющих командах, функциях, перечислениях, структурах и классах.**

Но сначала необходимо ответить на несколько вопросов, которые у вас могли возникнуть, и привести еще ряд комментариев. И не забудьте выпить чашку кофе или хорошо выспаться.



У шеф-повара появилось множество новых идей относительно того, что еще можно запрограммировать на Swift, если вы выдержите высокий темп. Если так дело пойдет, вы можете рассчитывать на должность ведущего специалиста по Swift в пиццерии.

Часто Задаваемые Вопросы

В: Где все точки с запятой? В программах их всегда полным-полно!

О: Хотя многие другие языки программирования требуют, чтобы строки заканчивались точкой с запятой (;), Swift этого не требует. Впрочем, если с ними вам более комфортно, вы можете их использовать. Это разрешено, хотя и не обязательно!

В: Как мне выполнить код на iPhone или iPad? Я купил эту книгу, чтобы научиться писать приложения iOS и разбогатеть до неприличия.

О: Мы объясним все, что необходимо знать для использования ваших навыков Swift при построении приложений iOS далее в книге. Богатства не гарантируем, но обещаем, что к концу вы научитесь строить приложения iOS.

В: У меня есть опыт работы на Python. Пробельные символы имеют особый смысл в Swift?

О: Нет, в отличие от Python, пробельные символы в Swift особого смысла не имеют. Не беспокойтесь об этом!

В: Я хочу просто научиться строить приложения для iPhone. Мне обязательно возиться с Playgrounds? Я хочу заняться приложениями прямо сейчас!

О: Как мы говорили, вам необходимо научиться работать с кодом Swift, прежде чем начинать работать с приложениями. Вам как программисту будет намного проще, если вы начнете с изучения основ Swift, прежде чем переходить к созданию приложений.



Упражнение Решение

С. 52

Нумерация элементов массивов всегда начинается с 0. Это означает, что индекс 7 в массиве будет использоваться для обращения к 8-му элементу массива.

`ingredients[7]` вернет "Oregano".

Кто и что делает? Решение

С. 44



Упражнение Решение

С. 53

Последним элементом массива ингредиентов является строка "Capers".

После того как вы присоедините к массиву новый ингредиент ("Banana"), новым последним ингредиентом станет "Banana", потому что `append` присоединяет данные к концу массива.

2. По имени Swift

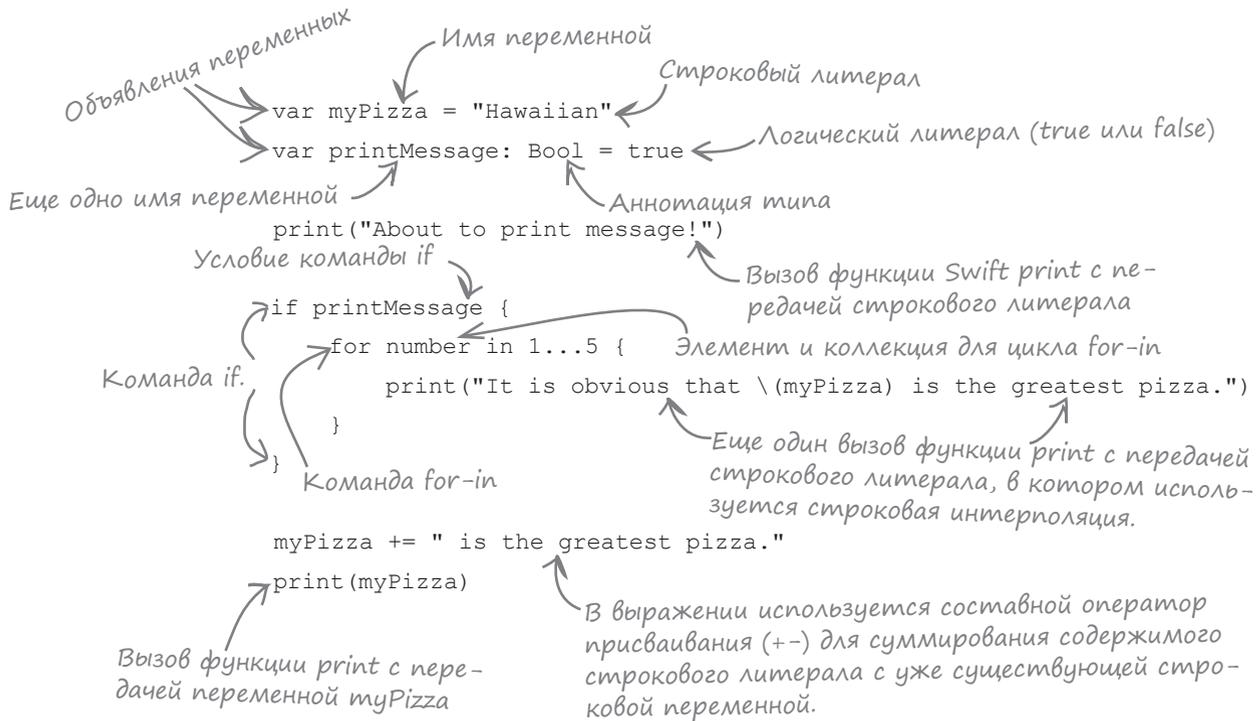
Swift на практике



Вы уже знаете азы Swift. Но пришло время изучить основные элементы языка более подробно. Вы узнали достаточно, чтобы вас воспринимали серьезно; пора употребить новые знания на практике. Мы применяем Playgrounds для написания кода, использования команд, выражений, переменных и констант — основных структурных элементов Swift. В этой главе мы заложим основу вашей будущей карьеры программиста Swift. Вы освоите систему типов Swift и изучите основы представления текста в строковом виде. Не будем терять времени — еще чуть-чуть, и вы начнете писать код Swift.

Из чего строятся программы

Каждая программа, которую вы пишете на Swift, *состоится* из разных элементов. В главах этой книги вы узнаете, что это за элементы и как объединить их, чтобы программа на языке Swift сделала именно то, что вам нужно.



Ключевые моменты

- Программы на языке Swift строятся из множества разных элементов.
- Наиболее фундаментальные структурные элементы — выражения, команды и объявления.
- Команда определяет действие, выполняемое вашей программой Swift.
- Выражение возвращает результат, то есть некоторое значение.
- Объявление вводит в программу Swift новое имя, например имя переменной.
- Переменные предназначены для хранения некоторого типа.
- Типы определяют, какие данные хранятся в некоторой области памяти.
- Литералы представляют значения, записанные непосредственно в программе (например, 5, true или "Hello").

Базовые операторы

Первый элемент, который мы рассмотрим, — **оператор**. Оператор представляет собой знак или последовательность символов, используемые для изменения, проверки или объединения значений, с которыми вы работаете в своей программе.

Операторы могут использоваться для выполнения математических вычислений, логических операций, присваивания значений и т. д. В Swift поддерживаются все операторы, встречающиеся в большинстве языков программирования, а также некоторые операторы, относительно уникальные для Swift... или по крайней мере используемые в Swift уникальным образом. Вы больше узнаете об операторах в процессе чтения книги.

Операторы бывают *унарными, бинарными и тернарными*.

← Это всего лишь означает, что операторы могут работать с одним значением (унарные операторы), двумя значениями (бинарные операторы) или тремя значениями (тернарные операторы).

«-» — унарный оператор. Он применяется к одному значению, в данном случае 1.

$11 + 2$

«+» — бинарный оператор. Он работает с двумя значениями (в данном случае 11 и 2).

$a ? b : c$

? и : — компоненты тернарного оператора. Он работает с тремя значениями (в данном случае a, b и c). Он будет рассмотрен позднее.



Алло?
Алло! Мне нужен оператор!

Оператор — знак или последовательность символов, предназначенные для изменения, проверки или объединения значений.



Расслабьтесь

Если вы не знаете все возможные операторы и их обозначения, не паникуйте.

По мере расширения ваших знаний и опыта использования Swift многие операторы будут казаться вам совершенно естественными (а многие уже кажутся, например математические операторы). Но со многими операторами этого не произойдет.

Для программиста абсолютно нормально искать справочную информацию об операторах, ключевых словах и синтаксисе, даже если он программировал на этом языке годами или десятилетиями. Нет ничего плохого в том, чтобы лишний раз освежить память.

Математические вычисления

В ходе программирования вы часто выполняете математические вычисления. Swift не подведет вас в том, что касается математики. Поддерживаются все классические операторы, включая операторы для сложения, вычитания, деления, умножения и вычисления остатка.

Каждая из этих строк, содержащих оператор, называется выражением.

Еще не забыли школьный курс математики? Вспомните порядок выполнения операций!

Складывает 100 и 50 бинарным оператором сложения +.
Вычитает 8 из 92 бинарным оператором вычитания -.
Делит 8 на 4 бинарным оператором деления /.
Умножает 9 на 10 бинарным оператором умножения *.
Оператор вычисления остатка. Возвращает остаток от деления одного числа на другое число.

100 + 50

92 - 8

8 / 4

9 * 10

42 % 2



Упражнение

Создайте новую среду Playground и займитесь вычислениями! Операторы можно комбинировать в одном выражении, как вас учили в школе.

Попробуйте умножить какое-нибудь число на 42, затем разделить результат на 3 и вычесть 4.

Попробуйте вычислить результат $4 + 5 * 5$.

Проверьте полученное число на четность оператором вычисления остатка (если остаток от деления на 2 равен нулю, то это четное число). Как вы думаете, какой результат вы получите?

Ответ на с. 65.

Выражайтесь яснее

Набор значений и операторов, который дает некоторый результат, называется выражением.

917 - 17 + 4

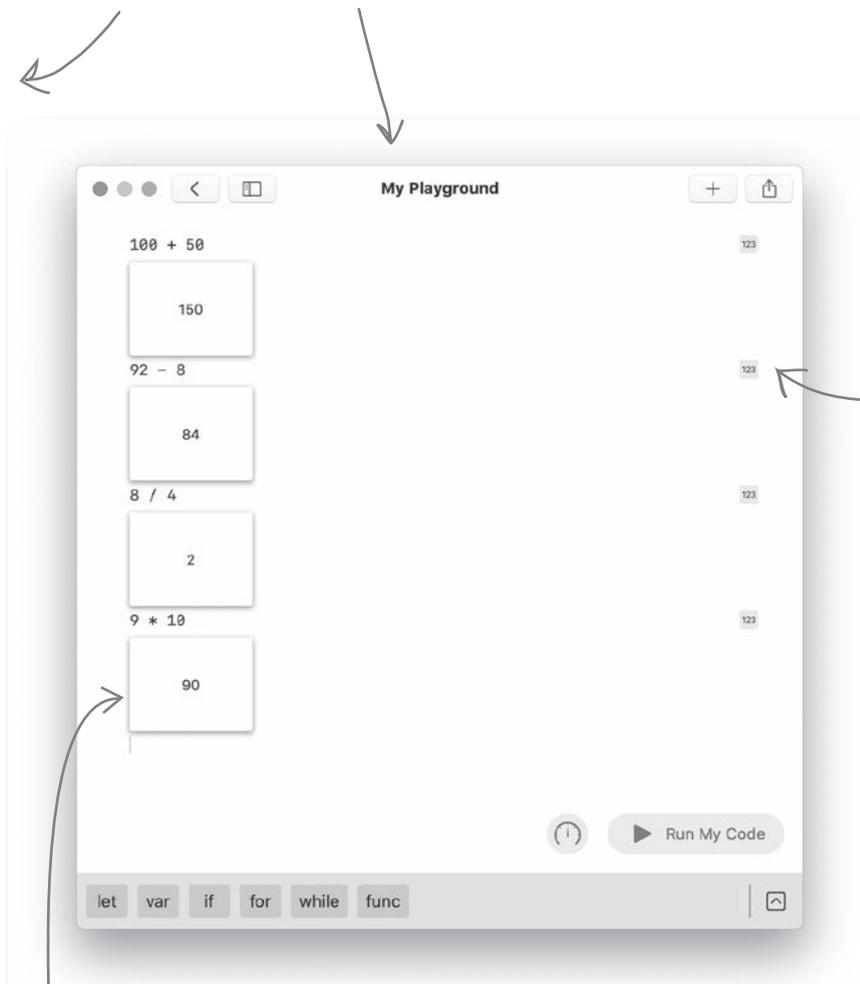
Это выражение, потому что это набор значений ((917, 17 и 4) и операторов (- и +), который генерирует результат (904).

На самом деле любой код, возвращающий значение, называется выражением. Даже если производимое значение полностью совпадает с выражением, это все равно выражение.

Выражайтесь яснее

Если вы введете эти **выражения** в среде Playground и выполните их, вы увидите их значение (то есть результат, полученный при *вычислении* выражения):

100 + 50
92 - 8
8 / 4
9 * 10



Так как у каждого из этих выражений существует результат, для них можно добавить области просмотра.

Области просмотра результата выводятся под каждым выражением, для которого они были добавлены.



Будьте осторожны!

Не используйте смешанный стиль включения пробелов...

...выберите один стиль и придерживайтесь его. Таким образом, ваши выражения и операторы могут иметь вид `8/4` или `8 / 4`, но только не `8 /4` или `8/ 4`.

Ключевые Моменты

- Программы Swift строятся из команд.
- Простая команда строится из выражений и объявлений.
- Выражение представляет собой код, при вычислении которого вы получаете результат.
- Команда не возвращает результат.
- Объявление вводит в программу новую переменную, константу, функцию, структуру, класс или перечисление.
- Операторы — знаки или последовательности символов, используемые для изменения, проверки или объединения значений в Swift.
- Некоторые часто используемые операторы выполняют математические вычисления.
- Другие операторы позволяют выполнять присваивание, сравнивать значения или проверять результат операций.

Кто и Что Делает ?

Подкрепите свои знания базовых операторов, используемых в выражениях Swift, и соедините каждое выражение в левой части с его результатом в правой части.

$7 + 3$

$9 - 1$

$10 / 5$

$5 * 4$

$9 \% 3$

-11

2

10

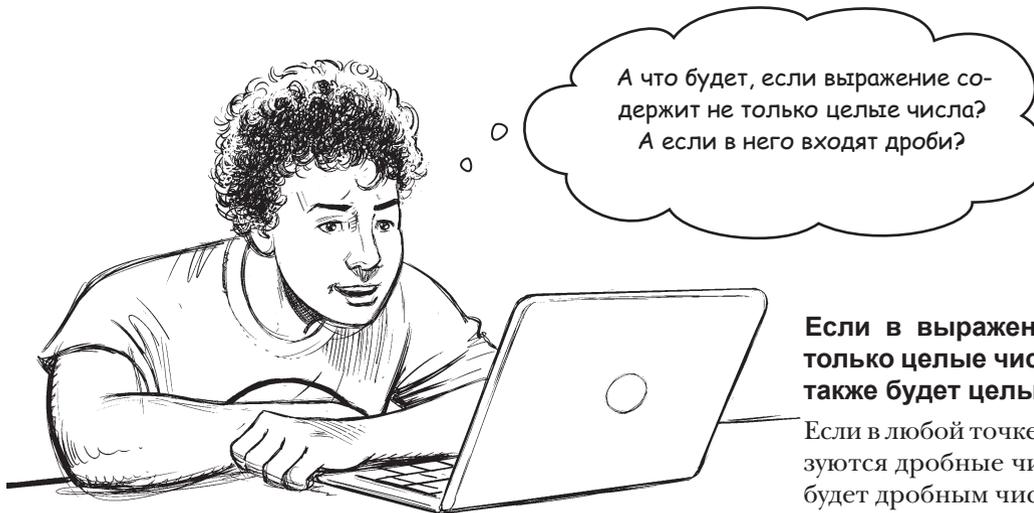
20

8

0

-11

→ Ответы на с. 65.



Если в выражении используются только целые числа, то и результат также будет целым числом.

Если в любой точке выражения используются дробные числа, то и результат будет дробным числом.

Предположим, вы хотите разделить 10 на 6. Казалось бы, при выполнении следующей команды результат должен быть равен приблизительно 1,6666666666666667:

```
10 / 6
```

Тем не менее результат будет равен 1, потому что результат может быть только целым числом, а дробь округляется до ближайшего целого значения.

С другой стороны, если вы прикажете Swift использовать дробные вычисления:

```
10.0 / 6.0
```

получится более точный ответ 1.6666666666666667, потому что Swift в этом случае работает исключительно с дробными числами.

В Swift также поддерживается еще один математический оператор: *остаток*. С его помощью можно узнать остаток от деления одного числа на другое.

Например, если вы хотите узнать, какой остаток будет получен от деления 9 на 2, выполните следующее выражение:

```
9 % 2
```

Результат равен 1. Дело в том, что 9 нацело не делится на 2; при делении мы получаем 4 с остатком 1.

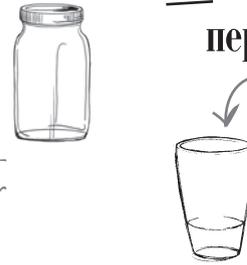
В Swift дробные числа также часто обозначаются своими типами double и float.

↑
Возможно, вы также слышали о других типах — строках и логических значениях.

Имена и типы

С данными, как и с домашними питомцами и сезонными напитками, намного удобнее работать по имени. **Константы** и **переменные** используются для хранения данных и обращения к ним по имени. Для присваивания им данных используется **оператор присваивания** =. **Переменная** представляет собой именованные данные, значение которых может изменяться, тогда как **константа** представляет собой данные, значение которых изменяться не может. Несколько примеров:

Ключевое слово **let** определяет константу, а ключевое слово **var** определяет переменную.



Знак = также является оператором. Он называется оператором присваивания.

```
let favNumber = 8.7
```

Эта команда объявляет константу с именем favNumber и присваивает ей дробное значение 8.7.

```
var coffeesConsumed = 17  
coffeesConsumed = 25
```

Эта команда объявляет переменную с именем coffeesConsumed и присваивает ей целое значение 17.

Переменную можно изменить позднее. Поэтому она и называется переменной. Понимаете?

Переменные и константы под увеличительным стеклом

Объявление переменной

```
var favNumber = 8.7
```

Необходимо показать, что это именно переменная, а не что-то другое, поэтому мы используем ключевое слово var.

Это оператор присваивания. Он присваивает значение, указанное в правой части, переменной в левой части.

Переменной необходимо назначить имя.

Переменная имеет тип double, потому что ей присвоено число с дробной частью.

Объявление константы

```
let goodNumber = 92
```

Это константа, потому что в объявлении использовано ключевое слово let.

Константе назначено имя.

Константа является целым числом, потому что ей присвоено целое число.



Упражнение
Решение

$10 \cdot 42 / 3 - 4$

136

$4 + 5 \cdot 5$

29

$10 \% 2$

0

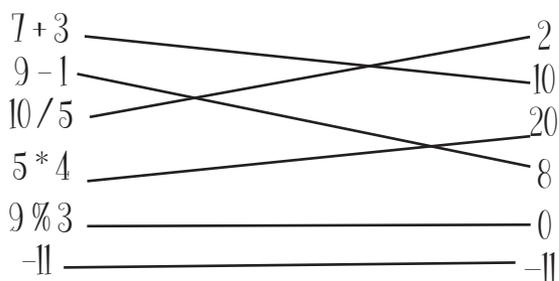
$11 \% 2$

1

С. 60

Кто и что делает?
Решение

С. 62





Константу невозможно изменить после того, как ей будет присвоено значение. Никогда.

Значение **переменной** можно изменить столько раз, сколько потребуется. Но после того как вы присвоите значение **константе**, изменить его уже не удастся. Давайте повнимательнее рассмотрим к тому, как объявляются переменные и константы:

```
let myNumber = 10
let myDouble = 10.5
var number = 55
number = 9.7
number = 10
let myConstant = 5
myConstant = 7
```

Определяет константу с именем `myNumber` и присваивает ей значение `10`. Swift понимает, что константа является целочисленной, потому что мы присвоили ей целое число при инициализации.

Выражение создает константу с именем `myDouble` и сохраняет в ней дробное число `10.5` (`double`).

Выражение объявляет переменную с именем `number`, которой присваивается целое число `55`. Переменная становится целочисленной.

Так как переменная `number` является целочисленной, ей невозможно присвоить `double` (хотя это и переменная). Попытка присваивания приведет к ошибке.

Впрочем, `number` можно присвоить другое целое число. Потому что это переменная.

Создает константу с именем `myConstant` и присваивает ей целое число `5`, поэтому константа становится целочисленной.

После присваивания исходного значения константу нельзя изменить. А значит, эта команда невозможна. Она приведет к ошибке.

Возьмите в руку карандаш



Создайте новую среду Swift Playground и напишите код Swift, который делает следующее:

- Создает переменную с именем `pizzaSlicesRemaining` и присваивает ей значение 8.
- Создает константу с именем `totalSlices` и присваивает ей 8.
- Делит `totalSlices` на `pizzaSlicesRemaining`.
- Обновляет `pizzaSlicesRemaining` значением 4.
- Снова делит `totalSlices` на `pizzaSlicesRemaining`.

→ Ответы на с. 11.



Упражнение

Создайте новую среду Playground и объявите переменную с типом `Integer`, присвойте ей некоторые данные (целое число, естественно). Затем в следующей строке попробуйте присвоить строку и посмотрите, что произойдет.

Какую ошибку выдает Swift?

→ Ответы на с. 11.



Расслабьтесь

Сколько всего нового...

Путешествие в страну Swift только начинается, а вам уже приходится запоминать множество элементов программ. Ваши знания будут расширяться, и со временем вы начнете себя чувствовать более уверенно.

Не все данные являются числами

Вы уже знаете, как создавать и назначать имена **константам** и **переменным**, когда необходимо сохранить данные в числовой форме (с дробной частью или без нее). Но не все значения являются числами! Пора представить одну из самых замечательных сторон мира Swift: систему типов.

Все это типы



Int

```
var age: Int
age = 35
```

Тип Int хранит целое число.

```
var name: String
name = "James Tiberius Kirk"
```



String

Тип String хранит последовательность символов.

```
var gravity: Double
gravity = 9.80665
```



Double

Тип Double хранит число с дробной частью.

```
let greatestFoodIsPizza: Bool
greatestFoodIsPizza = true
```



Bool

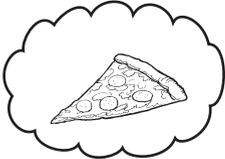
Тип Bool хранит true или false.

Задавать значение константы в момент ее создания не обязательно, но задать его можно только один раз.

Иногда вам нужно число, иногда строка, а иногда что-то совершенно иное. В Swift существует **система типов**, с помощью которой вы сообщаете Swift предполагаемый тип данных с использованием **аннотаций типов** или **автоматического определения типа**.

Некоторые типы вам уже знакомы (целые числа, дробные числа и строки), а для назначения типов наших переменных и констант использовалось *автоматическое определение* типов: вы поручаете Swift самостоятельно определить тип данных при присваивании значения.

Однако существует множество других типов; все они будут рассмотрены в этой книге. Важно понимать их, потому что после того, как тип переменной был задан, изменить его уже не удастся, и вы не сможете присвоить переменной данные неподходящего типа.



"Hawaiian Pizza"

Извини... ты не подходишь мне по типу. Вокруг много других переменных.



Int

Автоматическое определение типа означает, что Swift самостоятельно определяет, что собой представляют данные, и назначает тип за вас. Также тип можно явно задать при помощи аннотации типа.



Изменяемость

Хотя на уровне здравого смысла очевидно, что переменные существуют для того, чтобы их значения изменялись, а константы — нет, стоит рассмотреть происходящее на практике. Создайте среду Swift Playground и объявите следующую переменную с названием пиццерии:

```
var pizzaShopName = "Big Mike's Pizzeria"
```

Теперь допустим, что у пиццерии поменялся владелец и ее название решили изменить, допустим, на «Swift Pizza». Так как `pizzaShopName` — переменная, вы можете включить в программу команду присваивания, и никаких проблем не будет:

```
pizzaShopName = "Swift Pizza"
```

Но попробуйте заменить первую строку в среде Playground следующей строкой:

```
let pizzaShopName = "Big Mike's Pizzeria"
```

Вы увидите, что во второй строке, где вы пытаетесь изменить название, теперь обнаруживается ошибка (как и в первой строке, где вам предлагается преобразовать константу в переменную, чтобы сделать ее изменяемой).

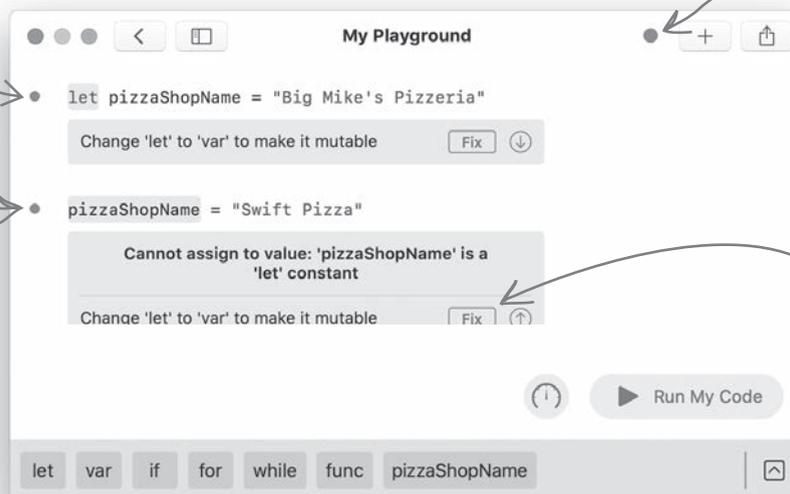
Это одно из простейших средств безопасности Swift. Если вы пытаетесь изменить то, что изменяться не должно, **Swift сообщит вам об этом.**

Красный индикатор выводится в каждой строке, содержащей ошибку или код, связанный с ошибкой.

Если в вашем коде обнаруживаются проблемы, вы увидите красный индикатор.

Эта строка содержит ошибку. В первой строке также появляется индикатор, потому что ее изменение приведет к исправлению ошибки.

Щелкните на нем, чтобы увидеть полный список всех ошибок в среде Playground.



Если щелкнуть на одной из кнопок Fix, константа в первой строке будет преобразована в переменную.

Определение строковых переменных

Как мы уже говорили, в процессе программирования часто выполняются математические вычисления. Но также часто приходится работать с текстом. В языках программирования текст обычно представляется *строкой*-типом, который называется String. Как было показано на предыдущей странице, строковые переменные могут создаваться так:

```
var greeting = "Hi folks, I'm a String! I'm very excited to be here."
```

Или так:

```
var message: String = "I'm also a String. I'm also excited to be here."
```

Когда вы включаете в свой код заранее определенное строковое значение (как мы только что сделали), это называется **строковым литералом**. Строковый литерал представляет собой последовательность символов, заключенную в двойные кавычки "".

Если вы хотите создать пустую переменную для хранения строки, это тоже возможно:

```
var positiveMessage = ""
```

В данном случае positiveMessage имеет тип String, потому что данные "" имеют тип String (какой бы короткой ни была строка), а механизм автоматического определения типов назначает переменной тип String.

А еще можно так:

```
var negativeMessage: String
```

negativeMessage — строка, потому что при объявлении указана аннотация типа String.

После этого строковым переменным можно присваивать значения:

```
positiveMessage = "Live long and prosper"
```

```
negativeMessage = "You bring dishonor to your house."
```



Типы неизбежно ограничивают гибкость.

Такой код работать не будет:

```
var pizzaTopping
    pizzaTopping = "Oregano"
```

Дело в том, что при объявлении переменной `pizzaTopping` мы не предоставили никаких данных, по которым Swift мог бы автоматически определить тип, и не указали аннотацию типа, которая бы явно сообщала Swift тип переменной.

Таким образом, хотя ничто не мешает вам объявить переменную, а затем присвоить ей данные (или изменить данные) в будущем, как это обычно делается (для чего в конце концов и нужны переменные), в данном случае переменная `pizzaTopping` объявлена без типа. А без данных, по которым можно было бы автоматически определить тип, это небезопасно, поэтому попытка не проходит.

Если вы хотите заранее объявить переменную, не предоставляя никаких данных, необходимо указать аннотацию типа как в следующем примере:

```
var pizzaTopping: String
```

Возьмите в руку карандаш

Решение

C.67

```
var pizzaSlicesRemaining = 8
```

8

```
let totalSlices = 8
```

8

```
totalSlices/pizzaSlicesRemaining
```

1

```
pizzaSlicesRemaining = 4
```

4

```
totalSlices/pizzaSlicesRemaining
```

2



Run My Code



Упражнение

Решение

C.67

```
var myInteger = 42
```

- myInteger = "forty two"

Cannot assign value of type 'String' to type 'Int'



Так как переменная myInteger создавалась с типом Integer, она не может содержать строку String.



String с автоматическим определением типа

Переменной назначается имя `pizzaShopName`.

```
var pizzaShopName = "Swift Pizza"
```

Это переменная, так что хранимое значение может изменяться.

Оператор присваивания. Значение в правой части присваивается переменной, указанной в левой части.

Значение — последовательность символов.



`pizzaShopName`

Swift знает, что я отношусь к типу `String`, потому что он проверил присвоенные данные (последовательность символов).

Когда вы поручаете Swift определить тип на основании присвоенных данных, это называется автоматическим определением типа.

String с аннотацией типа

Аннотация типа сообщает Swift, что наша переменная относится к типу `String`.

```
var pizzaName: String = "Swift Pizza"
```

После имени переменной, но перед аннотацией типа ставится двоеточие.

Swift знает, что я отношусь к типу `String`, потому что была предоставлена аннотация типа, которая объявляет меня с типом `String`.



`pizzaName`

Когда вы явно указываете тип, это называется аннотацией типа.

Ключевые моменты

- `String` (строка) хранит последовательности символов. Количество символов в строке может быть любым, включая ноль.
- `Double` хранит числа с дробной частью. Эти числа также могут быть отрицательными.
- `Int` хранит целые числа без дробной части. Эти числа могут быть отрицательными.
- `Bool` хранит `true` или `false` — и ничего более.

Если переменная или константа объявлена с аннотацией типа, но без данных, значит, данные будут присвоены позднее. Данные должны соответствовать аннотации типа.

Я понимаю аннотации типов и автоматическое определение типов. Но я не понимаю, как они работают с константами. Можно ли объявить константу, а потом присвоить ей значение? Какую роль здесь играет система типов?



Константам и переменным необходима аннотация типа или значение, которое присваивается при объявлении. Или и то и другое.

При объявлении переменной или константы можно **либо** предоставить аннотацию типа, **либо** немедленно присвоить значение.

Если вы присваиваете значение, то ничего больше не потребуется: система типов использует автоматическое определение, и на этом все хлопоты завершены.

Если вы не хотите присваивать значение при объявлении — это возможно, но тогда необходимо предоставить аннотацию типа, чтобы Swift знал, какой тип данных следует ожидать при присваивании данных.

Например, следующая команда допустима, потому что данные присваиваются константе во время объявления:

```
let bestPizzaInTheWorld = "Hawaiian"
```

Но если данные не присваиваются в момент объявления константы или переменной, произойдет ошибка:

```
let bestPizzaInTheWorld
bestPizzaInTheWorld = "Hawaiian"
```

Если вы выбираете этот способ, необходимо предоставить аннотацию типа:

```
let bestPizzaInTheWorld: String
bestPizzaInTheWorld = "Hawaiian"
```

Это константа (для объявления используется ключевое слово `let`), поэтому когда мы присваиваем ей строковый литерал «Hawaiian», она будет иметь тип `String` (благодаря автоматическому определению типов).

Так как мы предоставили аннотацию типа, такое объявление допустимо.

Позднее этой константе будет присвоена строка в соответствии с аннотацией типа. Так как это константа, после присваивания ей данных повторное присваивание будет невозможным.



Компания **типов** Swift, облаченных в маскарадные костюмы, развлекается игрой «Кто я?». Игрок дает подсказку, а остальные на основании сказанного им пытаются угадать, кого он изображает. Будем считать, что игроки всегда говорят правду о себе. Проведите линию от каждой подсказки к имени каждого участника. Мы уже провели одну линию, чтобы вам было проще взяться за дело.

Если упражнение покажется вам слишком сложным, не стесняйтесь заглянуть в ответы в конце главы!

Участники:

Я просто смотрю на вещи: либо true, либо false. Ничто другое меня не интересует.

String

Мне нравятся символы. И чтобы не разбрасываться, я в любой момент времени сосредоточен только на одном символе.

Bool

А я умею работать с множеством символов — сразу со всеми.

Int

Предпочитаю целые числа. Положительные и отрицательные. Дроби не для меня.

Double

Обожаю дробные части. Я большой любитель чисел, у которых они есть. А все остальное меня вообще не интересует.

Character

—————> Ответы на с. 80.

А если я захочу изменить строковую переменную после того, как я ее создал? Это возможно? Допустим, потребуется добавить к ней что-нибудь...



Изменить строку после того, как она была создана, несложно. Изменение работает (в основном) так же, как для чисел.

К строке можно добавить новый текст. Допустим, вы сохранили в переменной вдохновляющую речь:

```
var speech = "Our mission is to seek out new
              life and new civilizations."
```

А позднее вы решили добавить к ней еще немного текста:

```
speech +=
    " To boldly go where no one has gone before!"
```

Символы += образуют оператор. Он называется составным оператором присваивания. В данном случае он означает «взять speech, а затем сохранить в speech текущее содержимое speech и следующий текст».

Теперь переменная speech будет содержать следующий текст:

```
"Our mission is to seek out new life and new
civilizations. To boldly go where no one has gone
before!"
```



Оператор += присоединяет текст к строке. Вы не сможете использовать -= для удаления из строки.



Упражнение

Создайте новую среду Swift Playground и сделайте следующее:

- Создайте переменную типа String с именем favoriteQuote.
- Присвойте переменной свою любимую цитату.
- Добавьте к переменной favoriteQuote строку «by» и имя автора цитаты.
- Выведите содержимое переменной favoriteQuote.

→ Ответы на с. 79.

Строковая интерполяция

Кто-то решит, что термин «строковая интерполяция» происходит из научной фантастики, но это не так. Более того, это очень полезный механизм. **Строковая интерполяция** позволяет построить строку из разных значений — констант, переменных, строковых литералов и выражений; эти значения включаются в новый строковый литерал.

Допустим, у вас имеется число, представляющее скорость вымышленного космического корабля (из одного популярного научно-фантастического телесериала):

```
var warpSpeed = 9.9
```

Название корабля также хранится в строковой константе:

```
let shipName = "USS Enterprise"
```

И этот корабль летит к какой-то далекой планете, название которой тоже хранится в переменной:

```
var destination = "Ceti Alpha V"
```

При помощи строковой интерполяции можно построить совершенно новую строку, которая включает всю эту полезную информацию:

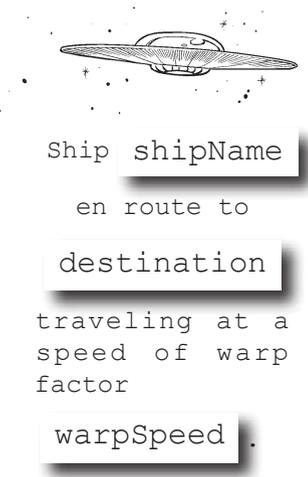
```
var status = "Ship \(shipName) en route to \(destination),  
traveling at a speed of warp factor \(warpSpeed)."
```

Если после этого вывести новую строковую переменную `status`, результат будет выглядеть так:

```
Ship USS Enterprise en route to Ceti Alpha V, traveling  
at a speed of warp factor 9.9.
```

На место заполнителя подставляется фактическое значение константы, переменной, строкового литерала или выражения.

Возможно, вы заметили, что мы используем строковую интерполяцию при вызове функции `print` для вывода значений переменных и констант как части выводимого результата.



Упражнение

Создайте новую среду Swift Playground и введите следующий код:

```
var name = "Head First Reader"  
var timeLearning = "3 days"  
var goal = "make an app for my kids"  
var platform = "iPad"
```

Примените строковую интерполяцию для вывода строки «Hello, I'm Head First Reader, and I've been learning Swift for 3 days. My goal is to make an app for my kids. I'm particularly interested in the iPad platform». Используйте переменные и строковую интерполяцию, чтобы дополнить выводимую строку другими подробностями по своему усмотрению.

→ Ответы на с. 79.



Почему вы все время говорите о «литералах»? Разве не все данные в программе являются литералом?

Литеральным значением называется значение, которое непосредственно записывается в вашем коде.

Например, если вы определите переменную с именем `peopleComingToEatPizza` и присвоите ей значение `8`, то непосредственно в исходном коде записывается значение `8`. Это и есть целочисленный литерал:

```
var peopleComingToEatPizza = 8
```

Если вы определите переменную для хранения оставшегося процента пиццы и присвоите ей значение `3.14159`, то вы тем самым присваиваете своей переменной литерал с плавающей точкой:

```
var pieRemaining = 3.14159
```

А если вы создадите описание пиццы в виде текста и сохраните его в переменной, то из-за автоматического определения типа вы тем самым создадите переменную с типом `String`, потому что ей присваивается строковый литерал:

```
var pizzaDescription =  
    "A delicious mix of  
    pineapple and ham."
```

Часть Задаваемые Вопросы

В: Мне казалось, что в Swift еще есть какие-то «протоколы». Когда мы будем изучать их? А как насчет классов?

О: В Swift действительно есть «протоколы», и мы обещаем, что вскоре ими займемся. Также очень скоро мы доберемся и до классов. В Swift есть несколько способов структурирования программ, и классы и протоколы открывают перед вами новые пути.

В: Зачем мне использовать константы? Ведь можно ограничиться только переменными на случай, если значение когда-нибудь потребуется изменить?

О: В Swift использованию констант уделяется гораздо больше внимания, чем в других языках программирования. Если вы будете использовать переменные толь-

ко для значений, которые должны изменяться, это будет сильно способствовать безопасности и стабильности программ. Кроме того, если вы сообщите компилятору Swift, что значения являются константами, компилятор сможет ускорить работу программ за счет применения некоторых оптимизаций.

В: Почему я не могу изменить тип значения после того, как оно было присвоено? В других языках это возможно.

О: Swift относится к категории языков с сильной типизацией. Он уделяет особое внимание работе системы типов и стимулирует вас к изучению этой системы. Вы не сможете изменить тип переменной после ее создания. Если возникнет такая необходимость, создайте новую переменную и преобразуйте значение к новому типу.

В: Переменные и константы — это тоже типы?

О: Нет, переменные и константы представляют собой именованные области памяти для хранения данных. Данные в переменной или константе обладают типом, поэтому с переменной или константой связывается тип, но сама по себе переменная или константа типом не является.

В: А если мне понадобится сохранить данные, для которых в Swift не найдется подходящего типа?

О: Хороший вопрос. Позднее в книге мы рассмотрим возможности создания собственных типов.

Ключевые моменты

- Операторы — знаки и последовательности символов, используемые для изменения, проверки или объединения значений, с которыми вы работаете в своих программах.
- Переменные и константы используются для хранения переменных и обращения к ним по имени.
- Переменные и константы относятся к некоторому типу (например, `Int` или `String`). Swift назначает тип на основании того, какие данные были сохранены в переменной или константе при создании, или на основании явно заданной аннотации типа.
- У переменных значение (но не тип!) может изменяться в любое время.
- У констант не может изменяться ни тип, ни значение.
- Программы Swift состоят из выражений, команд и объявлений.



Упражнение Решение

С. 75

```
var favoriteQuote: String
favoriteQuote = "I love it when a plan comes together!"
favoriteQuote += " by John 'Hannibal' Smith"
print(favoriteQuote)
```

I love it when a plan comes together! by John 'Hannibal' Smith



Упражнение Решение

С. 76

```
var name = "Head First Reader"
var timeLearning = "3 days"
var goal = "make an app for my kids"
var platform = "iPad"
print("Hello, I'm \(name), and I've been learning Swift for
\(\timeLearning). My goal is to \(goal). I'm particularly
interested in the \(platform) platform.")
```

Hello, I'm Head First Reader, and I've been learning Swift for 3 days. My goal is to make an app for my kids. I'm particularly interested in the iPad platform.



С. 74

Участники:

Я просто смотрю на вещи: либо true, либо false. Ничто другое меня не интересует.

String

Bool

Мне нравятся символы. И чтобы не разбрасываться, я в любой момент времени сосредоточен только на одном символе.

А я умею работать с множеством символов — сразу со всеми.

Int

Предпочитаю целые числа. Положительные и отрицательные. Дробы не для меня.

Double

Обожаю дробные части. Я большой любитель чисел, у которых они есть. А все остальное меня вообще не интересует.

Character

3. Коллекции и управление

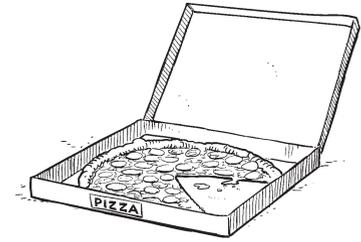
Зацикленные на данных

Мне нравится коллекционировать. Я хочу научиться коллекционировать и считать. Напомните еще раз, что надо сделать?



Вы уже знаете о выражениях, операторах, переменных, константах и типах **Swift**. Пришло время собрать воедино все, что говорилось ранее, и на этой основе исследовать некоторые более сложные структуры данных и операторы **Swift**: **коллекции** и **управляющие команды**. В этой главе мы поговорим о сохранении коллекций данных в переменных и константах, о структурировании данных, обработке данных и работе с данными с использованием управляющих команд. Позднее в книге будут рассмотрены другие способы сбора и структурирования данных, а пока начнем с **массивов**, **множеств** и **словарей**.

Сортировка пиццы



Зайдем в «Пиццерию *Swift*», самую популярную пиццерию в городе.

Шеф-повар, который почти всегда хочет что-то улучшить, желает отсортировать меню ресторана по алфавиту с использованием Swift (раз уж он присутствует в названии).

Но неожиданно возникли проблемы. Поможете ли вы шеф-повару **отсортировать** разные виды пиццы?

Я хочу упорядочить пиццу по алфавиту. Почему это так сложно? Я храню каждое название в строковой переменной.



```
var pizzaHawaiian = "Hawaiian"  
var pizzaCheese = "Cheese"  
var pizzaMargherita = "Margherita"  
var pizzaMeatlovers = "Meatlovers"  
var pizzaVegetarian = "Vegetarian"  
var pizzaProsciutto = "Prosciutto"  
var pizzaVegan = "Vegan"
```

Swift содержит набор специальных типов для хранения коллекций. Не стоит удивляться тому, что они называются типами коллекций.

Существуют три основных типа, которые могут использоваться для хранения коллекций в Swift: **массивы**, **множества** и **словари**. Они отличаются друг от друга, и умение выбрать подходящий тип для каждой задачи является одним из основных навыков программирования на Swift.

Если вам нужно сохранить список в коллекции, а затем отсортировать элементы списка по алфавиту, удобнее всего воспользоваться массивом — но мы к этому еще вернемся. Чтобы решить, как лучше помочь шеф-повару, необходимо понимать каждый тип коллекции и его возможности.

← Собственно, имя говорит само за себя

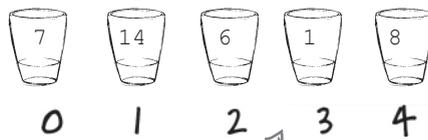
Типы коллекций Swift

Типы Swift, которые вы использовали до сих пор, позволяют хранить отдельные фрагменты данных произвольного типа (по большей части). Например, тип `String` позволяет хранить строки, `Int` — целые числа, `Bool` — логические значения, и т. д.

Типы коллекций Swift позволяют хранить **несколько** фрагментов данных.

Тип коллекции существенно упрощает хранение наборов значений.

Массивы



Индексы всегда представляют собой целые числа, начинающиеся с 0.

Массивы упорядочены, все элементы содержат значения одного типа и автоматически нумеруются индексами, начиная с нуля. А это означает, что вы можете обращаться к отдельным значениям из массива по индексам.

Этот код создает этот массив.

```
let numbers = [7, 14, 6, 1, 8]
```

Множества



Каждое значение только в одном элементе.

Множества не упорядочены, все элементы содержат значения одного типа. Индексов у множеств нет, поэтому для обращения к элементам используются либо средства множеств, либо перебор по всему множеству. Каждое значение может храниться в множестве только в одном экземпляре.

Оба фрагмента кода создают это множество.

```
var numbers = Set([1, 7, 9, 13, 3])
var numbers: Set = [1, 7, 9, 13, 3]
```

Словари



Ключи — все одного типа. Каждый ключ связан со значением.

Словари представляют собой неупорядоченные коллекции пар ключей и значений. Все ключи относятся к одному типу, и все значения относятся к одному типу. К значению можно обратиться по ключу, по аналогии с индексами в массивах. Типы ключей могут отличаться от типов значений.

Этот код создает этот словарь.

```
var scores = ["Tom": 5, "Bob": 10, "Tim": 9, "Mars": 14, "Tony": 3]
```

Хранение значений в массиве

Наше рассмотрение *типов коллекций* начнется с **массивов**. Массив представляет собой набор *однотипных* элементов, хранящихся в определенном порядке. Массив можно создать простым перечислением элементов, как следующий массив строк:

```
let catNames = ["Lucy", "Tom", "Billy", "Bruce", "Lady", "Doug", "Susan"]
```

Этот массив является константой.

Значения, хранящиеся в массиве, заключаются в квадратные скобки.

Система автоматического определения типов Swift заключает, что это массив строк, потому что все его элементы являются строками!

Массив — упорядоченная коллекция значений одного типа.

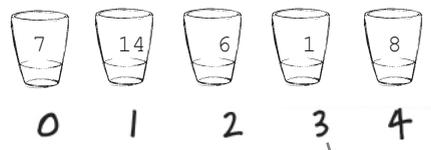
А еще можно создать массив с указанием аннотации типа как в следующем массиве целых чисел:

```
var numbers: [Int] = [7, 14, 6, 1, 8]
```

Аннотация типа заключается в квадратные скобки, чтобы объявить его как массив соответствующего типа.

Значения, хранящиеся в массиве, нумеруются индексами, начиная с нуля. Индексы массива из приведенного примера выглядят так:

К отдельным элементам массива можно обращаться по индексам.



```
print(numbers[0])
```

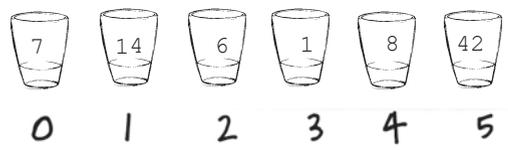
7

```
print(numbers[3])
```

1

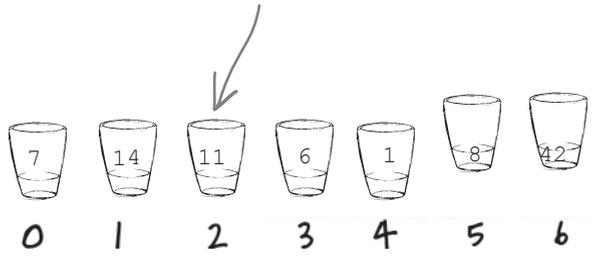
Данные можно добавлять в переменные-массивы несколькими разными способами, включая присоединение:

```
numbers.append(42)
```



И вставку:

```
numbers.insert(11, at: 2)
```



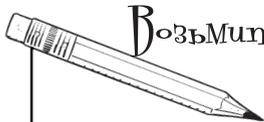
Также элементы можно удалять:

```
numbers.remove(at: 3)
```

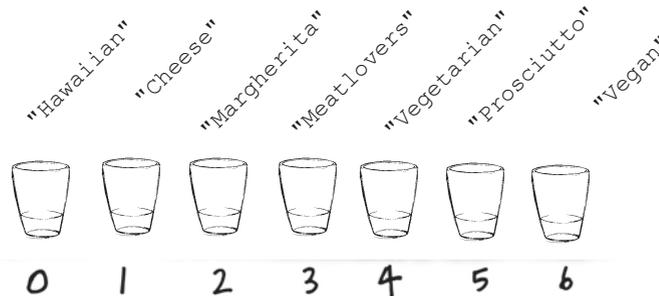
И изменять значения конкретных элементов:

```
numbers[2] = 307
```

Возьмите В руку карандаш



Возьмем следующий массив с набором видов пиццы:



Создайте новую среду Playground и напишите код, который делает следующее:

- Создает этот массив как переменную и сохраняет в ней названия видов пиццы.
- Выводит элемент массива, содержащий значение "Vegetarian".
- Присоединяет к массиву элемент "Pepperoni", не изменяя остальные элементы массива.
- Вставляет новое значение "Ham, Pineapple, and Pesto" в позицию с индексом 2, сдвигая все последующие элементы на одну позицию.
- Полностью удаляет значение "Cheese" из массива, оставляя остальные элементы без изменений.
- Меняет местами элементы "Prosciutto" и "Pumpkin and Feta", не изменяя другие элементы массива.
- Выводит весь массив.

После выполнения всех этапов массив должен выглядеть примерно так:



```
["Hawaiian", "Ham, Pineapple, and Pesto", "Margherita", "Meatlovers", "Vegetarian", "Pumpkin and Feta", "Vegan", "Pepperoni"]
```

→ Ответы на с. 88.

Сколько элементов в массиве? И есть ли в нем элементы?

До настоящего момента вы точно знали, сколько элементов в массиве (потому что вы их создали). Но что, если вы работаете с массивом, длина которого неизвестна заранее?

Представьте, что вам предложено что-то сделать с массивом, содержащим ингредиенты конкретной пиццы. Массив был создан где-то в другой точке программы, и вы не знаете, сколько в нем элементов.

Давайте заглянем за кулисы. Команда создания массива выглядит так:

```
var ingredients =
    ["Oregano", "Ham", "Tomato", "Olives", "Cheese"]
```

← Простой массив строк

Для каждого типа коллекции в Swift доступно свойство `count`, которое позволяет узнать количество элементов в коллекции:

```
print("There are \(ingredients.count)
      ingredients in this pizza.")
```

→ There are 5 ingredients in this pizza.

Также с массивами можно выполнять другие полезные операции, например проверить, пуст ли массив:

```
print(ingredients.isEmpty)
```

→ false

И еще существуют вспомогательные методы для получения наибольшего и наименьшего элемента массива:

```
print(ingredients.max())
print(ingredients.min())
```



Ключевые моменты

- В массивах хранятся элементы одного типа. Если вы создаете массив целых чисел, в нем могут храниться только целые числа.
- Индексы элементов массивов начинаются с нуля, а количество однотипных элементов в массиве может быть произвольным.
- При удалении элемента из массива происходит сдвиг элементов. Пустых мест в массиве не остается.
- В массиве можно удалять конкретные элементы, заданные индексами.
- Элементы могут присоединяться в конце массива.
- Элементы могут вставляться между двумя другими элементами (при этом происходит сдвиг элементов).
- Для массивов доступны многие полезные свойства (такие, как `count`) и вспомогательные методы (такие, как `max` и `min`).
- Массивы в Swift имеют много общего со списками в других языках. По сути, это действительно списки, но при этом они также являются массивами.

Хранение значений в множестве

Вторая разновидность коллекций, которую мы будем использовать, — **множество**. Множества похожи на массивы, они тоже должны содержать только один тип, но **множества не упорядочены, и каждое значение может встречаться в множестве не более одного раза**.

Множество создается так:

```
var evenNumbers = Set([2, 4, 6, 8])
```

Или так:

```
var oddNumbers: Set = [1,3,5,7]
```

Существуют и другие способы создания множеств, они будут рассмотрены позднее в книге.

Во множества можно вставлять элементы, как и в массивы:

```
oddNumbers.insert(3)
```

```
oddNumbers.insert(9)
```

Здесь не будет ни ошибки, ни сбоя, но ничего не произойдет. Дело в том, что значение 3 уже присутствует во множестве, а в множествах каждое значение встречается не более одного раза. Это не ошибка, так и задумано.

Эта операция добавит в множество значение 9.

Также из множества можно легко удалять элементы (даже если они не входят в множество, вы все равно можете запросить их удаление):

```
oddNumbers.remove(3)
```

Удаляет 3 из множества, если это значение присутствовало в множестве. Если значение отсутствует, ошибка не выдается.

Возьмите в руку карандаш

Попробуйте ответить на следующие вопросы о множествах. Вы можете выполнить код в среде Playground или же заглянуть в ответы на следующей странице.

```
var pizzas = Set(["Hawaiian", "Vegan", "Meatlovers", "Hawaiian"])
print(pizzas)
```

Что будет выведено? _____

```
let favPizzas = ["Hawaiian", "Meatlovers"]
```

Что содержит это множество? _____

```
let customerOrder = Set("Hawaiian", "Vegan", "Vegetarian", "Supreme")
```

Что содержит это множество? _____



Возьмите в руку карандаш

Решение

С. 85

```
var favoritePizzas = ["Hawaiian", "Cheese", "Margherita",
    "Meatlovers", "Vegetarian", "Prosciutto", "Vegan"]
print(favoritePizzas[4])
favoritePizzas.append("Pepperoni")
favoritePizzas.insert("Ham, Pineapple, and Pesto", at: 2)
favoritePizzas.remove(at: 1)
favoritePizzas[5] = "Pumpkin and Feta"
print(favoritePizzas)
```

Vegetarian
 ["Hawaiian", "Ham, Pineapple, and Pesto", "Margherita", "Meatlovers",
 "Vegetarian", "Pumpkin and Feta", "Vegan", "Pepperoni"]



Возьмите в руку карандаш

Решение

С. 87

В множестве каждое значение может содержаться только в одном экземпляре, так что «Hawaiian» встречается только один раз.

```
var pizzas = Set(["Hawaiian", "Vegan", "Meatlovers", "Hawaiian"])
print(pizzas)
```

Что будет выведено? _____

["Meatlovers", "Hawaiian", "Vegan"]

```
let favPizzas = ["Hawaiian", "Meatlovers"]
```

Что содержит это множество? _____

["Hawaiian", "Meatlovers"]

И это массив, а не множество. Сюрприз!

```
let customerOrder = Set("Hawaiian", "Vegan", "Vegetarian", "Supreme")
```

Что содержит это множество? **Ничего, потому что синтаксис неправильный. Список элементов должен быть заключен в квадратные скобки [].**

Хранение значений в словаре

Последний тип коллекции, который мы сейчас рассмотрим, — **словарь**. Как нетрудно догадаться по названию, словарь связывает одно значение с другим, как в обычных словарях слово связывается с определением. Как и множества, **словари в Swift не упорядочены**.

Элементы данных, хранящиеся в словарях, называются **ключами** и **значениями**.

Если вы хотите создать словарь с именами участников настольной игры и количеством набранных ими очков, это можно сделать так:

```
var scores = ["Paris": 5, "Marina": 10, "Tim": 9, "Jon": 14]
```

Имена участников игры.

Количество очков у каждого игрока.

Механизм автоматического определения типов Swift понимает, что значения являются целыми числами.

В данном случае механизм автоматического определения типов Swift понимает, что ключи являются строками.

А еще можно явно сообщить компилятору Swift типы ключей и значений:

```
var scores: [String: Int] = ["Paris": 5, "Marina": 10, "Tim": 9, "Jon": 14]
```

Аннотация типа для словаря указывает, что ключи являются строками (слева), а значения — целыми числами (справа).

Также можно создать пустой словарь, но задать типы заранее, потому что компилятор (очевидно) не может определить типы без заданных значений:

```
var scores: [String: Int] = [:]
```



Как мне прочесть данные из словаря? И еще важнее, как там что-нибудь сохранить?

Чтение значения из словаря может выглядеть немного странно по сравнению с тем, что мы делали до сих пор.

Чтобы прочесть значение из словаря, достаточно указать ключ:

```
print(scores["Paris"]!)
```

Восклицательный знак используется для прямого обращения к значению, связанному с ключом. Это называется *принудительной распаковкой* опционального типа. Вскоре мы вернемся к этому синтаксису, когда будем использовать опциональные типы. Впрочем, после того как вы изучите принудительную распаковку, вам никогда не придется ею пользоваться на практике.

Также можно добавлять новые значения или изменять существующие значения в словарях как в следующем примере:

```
scores.updateValue(17, forKey: "Bob")
```

Или так:

```
scores["Josh"] = 4
```



Упражнение

Пришло время применить ваши новообретенные навыки работы с коллекциями на практике. У вас есть пара коллекций, элементы которых вы хотите читать и изменять. Попробуйте выполнить приведенные ниже упражнения. Не огорчайтесь, если вам придется обращаться к ответам.

Следующее множество представляет набор запретных рецептов пиццы, которые никогда не должны заказываться:

```
var forbiddenPizzas: Set =
    ["Lemon and Pumpkin",
     "Hawaiian with a Fried Egg", "Schnitzel and Granola"]
forbiddenPizzas.insert("Chicken and Boston Beans")
forbiddenPizzas.remove("Lemon and Pumpkin")
```

Что содержит `forbiddenPizzas` после выполнения этого фрагмента?

А в словаре хранятся рецепты десертной пиццы и количество заказанных порций:

```
var dessertPizzaOrders = ["Rocky Road": 2, "Nutella": 3, "Caramel Swirl": 1]
```

Выведите количество заказов для Rocky Road и Caramel Swirl. Затем добавьте новый заказ на 17 порций Banana Split.

→ Ответы на с. 93.

Ключевые моменты

- Кроме обычных типов, в Swift также существуют типы коллекций, предназначенные для хранения групп значений.
- Массив — тип коллекции для хранения упорядоченной коллекции однотипных значений, к которым можно обращаться по целому индексу.
- Множество — тип коллекции для хранения неупорядоченной (несортированной) коллекции непо-

вторяющихся однотипных элементов. Все элементы уникальны, дубликатов быть не может.

- Словарь — тип коллекции для хранения неупорядоченной коллекции пар ключей и значений. Все ключи относятся к одному типу, все значения также относятся к одному типу. К значению можно обратиться по его ключу.



И это все? Не много у вас коллекций. Может, еще что-то найдется?

Да, три основных типа коллекций — массив, множество и словарь.

Но у Swift в запасе еще есть несколько трюков. Обобщенный тип коллекции, к которому мы перейдем немного позднее; он позволяет создавать собственные типы коллекций. Наконец, в вашем распоряжении тривиальные кортежи.

Кортежи позволяют хранить несколько значений в одной переменной или константе, но без затрат ресурсов на создание (допустим) массива.

Кортежи идеально подходят для создания специализированных коллекций значений, в которых каждый элемент имеет строго определенную позицию или имя.

Кортежи

Кортеж позволяет хранить сколько угодно значений — при условии, что их количество и тип известны заранее. Например, можно взять координаты x и y и сохранить их в одной переменной-кортеже:

```
var point = (x: 10, y: 15)
print(point.x)
```

Оба компонента кортежа являются целыми числами (из-за применения автоматического определения типов).

Выводит 10.

Компонентам кортежей можно назначить имена, но это не обязательно. Кортежи можно обновлять или изменять и без использования имен — при условии, что типы значений совпадают:

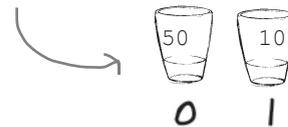
```
point = (50, 10)
```

Компонент x обновляется значением 50, а компонент y — значением 10.

Также можно обращаться к компонентам кортежей с указанием целых чисел, начиная с 0:

```
print(point.0)
```

Выводит 50.



Хороший псевдоним пригодится каждому

Когда вы работаете с разными типами данных в Swift, для того, чтобы ваш код лучше читался, можно воспользоваться **псевдонимами типов**, то есть назначением нового имени существующему типу.

Рассмотрим короткий пример: простой односторонний преобразователь температур шкалы Цельсия к шкале Фаренгейта, написанный без использования псевдонимов типов.

```
var temp: Int = 35
var result: Int
result = (temp*9/5)+32
print("\(temp) °C is \(result) °F")
```

Значение по шкале Цельсия хранится в целочисленной переменной с именем temp.

Результат, преобразованный к шкале Фаренгейта, сохраняется в целочисленной переменной с именем result.

Результат вычисляется преобразованием значения из шкалы Цельсия к шкале Фаренгейта, после чего сохраняется в переменной result.

Программа выводит результат с описанием выполненного преобразования.



Хотя перед вами очень маленький и простой фрагмент кода, его можно дополнительно упростить при помощи псевдонимов типов:

```
typealias Celsius = Int
typealias Fahrenheit = Int

var temp: Celsius = 35
var result: Fahrenheit

result = (temp*9/5)+32
print("\(temp) °C is \(result) °F")
```

Мы определяем псевдонимы типов для Int, имена которых соответствуют каждой из двух шкал, используемых в программе.

Теперь типы значений температур ясно показывают, к какой шкале они относятся.

Псевдонимы типов ничего не меняют в самом типе. Они только определяют новые имена для обозначения типа.



Будьте осторожны!

Псевдонимы типов упрощают чтение вашего кода. Но будьте осторожны!

Когда вы увлекаетесь и начинаете определять псевдонимы для всего подряд, ваш код быстро выходит из-под контроля — становится невозможно разобраться, с какими типами на самом деле работает ваш код. В основе типов Celsius и Fahrenheit лежат самые обычные целые числа. Ничто не мешает присвоить значение типа Celsius переменной типа Fahrenheit, потому что это просто целые числа.



Упражнение Решение

C.90

```
var forbiddenPizzas: Set = ["Lemon and Pumpkin", "Hawaiian
    with a Fried Egg", "Schnitzel and Granola"]
forbiddenPizzas.insert("Chicken and Boston Beans")
forbiddenPizzas.remove("Lemon and Pumpkin")
print(forbiddenPizzas)
```

```
var dessertPizzaOrders = ["Rocky Road": 2, "Nutella": 3,
    "Caramel Swirl": 1]
print(dessertPizzaOrders["Rocky Road"]!)
print(dessertPizzaOrders["Caramel Swirl"]!)
dessertPizzaOrders["Banana Split"] = 17
```

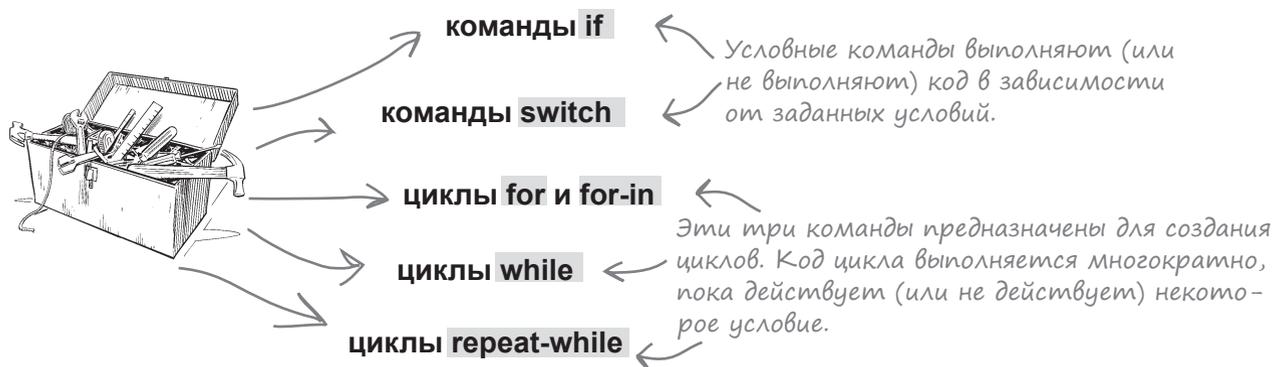
↓

```
["Chicken and Boston Beans", "Hawaiian with a Fried Egg",
    "Schnitzel and Granola"]
2
1
```

Управляющие команды

Когда вы программируете на Swift (и других языках программирования, естественно), часто требуется **делать что-то многократно** или **делать что-то при определенных условиях**. Эти две концепции называются общим термином *последовательность выполнения*.

В вашем инструментарии управления последовательностью выполнения много разных инструментов, все они работают по-разному и находят практическое применение.



Но как понять, когда использовать ту или иную команду? Они так похожи друг на друга...

Однозначных правил не существует.

Все эти управляющие команды могут использоваться в различных ситуациях. **Нет однозначных правил, которые бы указывали, когда что использовать, но обычно это относительно очевидно.**

Команды `if` лучше всего подходят для выполнения некоторого кода только в случае истинности некоторого условия.

Команды `switch` используются в похожем контексте, но при большом количестве возможных вариантов в зависимости от нескольких условий.

Циклы — `for`, `for-in`, `while` и `repeat-while` — позволяют многократно выполнять некоторый код в зависимости от некоторых условий.

Вы найдете применение для всех этих управляющих команд, когда напишете больше реального кода.



Команды if

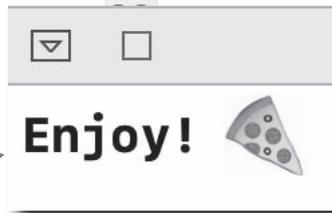
Команды `if` что-то делают при выполнении некоторого условия. В широком смысле команды `if` входят в группу **условных команд**.

Допустим, у вас есть переменная типа **Bool** с именем `userLovesPizza` и вы хотите использовать ее для определения того, нужно ли дать пользователю пиццу. Для этого можно воспользоваться командой `if`.

```
var userLovesPizza: Bool = true
```

```
if(userLovesPizza) {
    print("Enjoy! 🍕")
}
```

Выводится только в том случае, если пользователь любит пиццу (выражение дает результат `true`)



Команду `if` можно объединить с `else` и `else if` для расширения ее возможностей:

```
if(userLovesPizza) {
    print("Enjoy! 🍕")
} else {
    print("Sorry!")
}
```

Команда `else` выполняется, если условие `if` ложно!

Команда `else if` продемонстрирована на следующей странице.



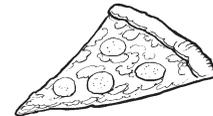
Условие в команде `if` должно быть логическим выражением. Это означает, что оно должно давать результат `true` или `false`. Если выражение дает результат `true`, то выполняется код, заключенный в фигурные скобки `{ }` в команде `if`.



Упражнение

Возьмем несколько логических переменных, каждая из которых сообщает, был ли заказан определенный тип пиццы (в таком случае его надо доставить):

```
var hawaiianPizzaOrdered = true
var veganPizzaOrdered = true
var pepperoniPizzaOrdered = false
```



В среде Swift Playground создайте несколько команд `if`, которые проверяют, нужно ли организовать доставку каждого из этих видов пиццы. Если они нуждаются в доставке, выведите сообщение для водителя, а затем присвойте переменной значение `false`, чтобы не доставлять лишние коробки с пиццей.

После этого попробуйте преобразовать этот код, чтобы в нем использовался словарь с заказами, и напишите команду `if` для проверки счетчика каждого вида пиццы в словаре перед выводом сообщения о доставке (которое должно включать значение счетчика).

→ Ответы на с. 98.

Команды switch

Часто при создании условных команд if приходится рассматривать множество вариантов. Вернемся к примеру с заказами пиццы; представьте, насколько *громоздким* станет код, если вы также захотите отправлять специальное сообщение для каждого типа пиццы. Посмотрим, как это можно сделать.



Проверим, содержит ли строковая переменная определенный тип пиццы, и выведем сообщение на основании этого типа. Для проверки будет использоваться оператор new и **оператор проверки равенства**. Оператор проверки равенства состоит из двух знаков равенства подряд: ==.

```
var pizzaOrdered = "Hawaiian"
```

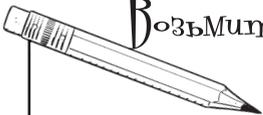
← Присваивается значение для заказанной пиццы.

Этот оператор проверяет, равны ли два значения. Если значения равны, оператор возвращает true.

```
if(pizzaOrdered == "Hawaiian") {
    print("Hawaiian is my favorite. Great choice!")
} else if(pizzaOrdered == "Four Cheese") {
    print("The only thing better than cheese is four cheeses.")
} else if(pizzaOrdered == "BBQ Chicken") {
    print("Chicken and BBQ sauce! What could be better?")
} else if(pizzaOrdered == "Margherita") {
    print("It's a classic for a reason!")
}
```

А затем многочисленные команды if и else if проверяют, какая пицца была заказана, чтобы мы могли вывести правильное сообщение.

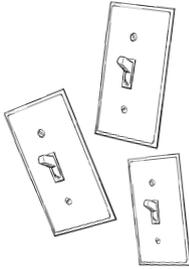
Возьмите в руку карандаш



Возьмите следующую команду if, которая выводит информацию о планетах Солнечной системы, и добавьте в нее планеты Нептун (Neptune), Марс (Mars) и Земля (Earth):

```
var planet = "Jupiter"
if planet == "Jupiter" {
    print("Jupiter is named after the Roman king of the gods.")
} else {
    print("All the planets are pretty cool.")
}
```

→ Ответы на с. 101.



Я-то понимаю, что элегантно, а что нет! Вот этот торт элегантный. А все эти команды if — нет! Неужели нет лучшего способа?

Если вам требуется выполнить разный код для разных значений переменной или константы, команда `switch` часто оказывается более эффективной.

Код с множеством команд `if` быстро становится нечитаемым и очень неэлегантным. Команда `switch` позволяет **выбирать** выполняемый фрагмент кода в зависимости от значений переменных или констант, по которым производится *выбор*.

Команда `switch` может состоять из нескольких секций, причем эти секции не ограничиваются проверкой равенства. Мы еще вернемся к этому позднее.

Возвращаясь к примеру с пиццерией, гигантскую серию команд `if` можно переписать в виде компактной, понятной команды `switch`:



```
var pizzaOrdered = "Hawaiian"
```

← Присваивается значение для заказанной пиццы.

```
switch (pizzaOrdered) {
  case "Hawaiian":
    print("Hawaiian is my favorite. Great choice!")
  case "Four Cheese":
    print("The only thing better than cheese is four cheeses.")
  case "BBQ Chicken":
    print("Chicken and BBQ sauce! What could be better?")
  case "Margherita":
    print("It's a classic for a reason!")
  default:
    break
}
```

Выбор в зависимости от переменной `pizzaOrdered`.

А затем для каждого случая...

← ...выводится подходящее сообщение.

← Секция `default` выполняется в том случае, если ни один вариант не подошел.

← Ключевое слово `break` немедленно завершает выполнение команды `switch`, и выполняется код, следующий за командой `switch`.

Упражнение
Решение

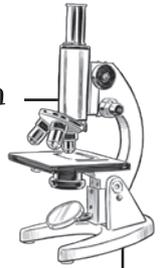
С. 95

```
var hawaiianPizzaOrdered = true
var veganPizzaOrdered = true
var pepperoniPizzaOrdered = false

if(hawaiianPizzaOrdered) {
    print("Please deliver a Hawaiian pizza.")
    hawaiianPizzaOrdered = false
}
if (veganPizzaOrdered) {
    print("Please deliver a Vegan pizza.")
    veganPizzaOrdered = false
}
if (pepperoniPizzaOrdered) {
    print("Please deliver a Pepperoni pizza.")
    pepperoniPizzaOrdered = false
}

var pizzaOrder = ["Hawaiian": 2, "Vegan": 1, "Pepperoni": 9]

if (pizzaOrder["Hawaiian"]! > 0) {
    print("Please deliver \(pizzaOrder["Hawaiian"]!)x Hawaiian pizza.")
}
if (pizzaOrder["Pepperoni"]! > 0) {
    print("Please deliver \(pizzaOrder["Pepperoni"]!)x Pepperoni pizza.")
}
if (pizzaOrder["Vegan"]! > 0) {
    print("Please deliver \(pizzaOrder["Vegan"]!)x Vegan pizza.")
}
```



Анатомия команды switch

1 Сначала идет ключевое слово switch

Команды switch начинаются с ключевого слова switch.

2 Затем указывается проверяемое значение

Значение может быть переменной или константой. Именно оно проверяется командой switch для каждого случая.

```
switch Значение
```

3 И возможные варианты

Каждое значение case сравнивается с проверяемым. Также можно использовать более сложные шаблоны поиска, к которым мы еще вернемся. Вы даже можете проверить несколько вариантов в одной секции.

```
case Значение
// Выполняемый код
```

```
case Другое значение :
```

```
// Выполняемый код
```

4 Код для каждого варианта

Код, который выполняется при совпадении значения для этой секции case.

```
default:
```

```
// Выполняемый код
```

5 И наконец, секция default с кодом

Секция default выполняется в том случае, если ни одно значение в секциях case не совпало. Секция default обязательна, если в команде switch перечислены не все возможные значения.

Упражнение

Напишите команду switch для проверки того, является ли число счастливым. Предусмотрите несколько вариантов вывода сообщений для разных счастливых чисел. Если проверяемое число не является счастливым или если была выведена информация о числе, выведите соответствующее сообщение.

→ Ответы на с. 101.



Построение команды switch

Если вы (по какой-то причине) захотели проверить, равно ли число 9, 42, 47 или 317, для этого можно воспользоваться командой switch.

Сначала идет ключевое слово switch

1

```
var number = 42  
switch number {
```

2

Затем указывается проверяемое значение

```
case 9:
```

```
    print("The number is 9! A mundane number.")
```

```
case 42:
```

```
    print("It's the meaning of life!")
```

```
case 47:
```

```
    print("The meaning of life, corrected for inflation.")
```

```
case 317:
```

```
    print("A rather large number.")
```

```
default:
```

```
    break
```

```
}
```

Код для каждого варианта

4

И возможные варианты

5

И наконец, секция default с кодом

Элегантнее... но ненамного! Чего мы, собственно, добились?

Команда switch создает меньше проблем с сопровождением, а код лучше читается.

Мы вернемся к командам switch позднее (когда будем рассматривать перечисления). Но в общем случае, если вы проверяете более двух-трех вариантов в большом блоке команд if-else, стоит рассмотреть возможность использования команды switch.





Возьмите в руку карандаш

Решение

С. 96

```
var planet = "Jupiter"
if(planet == "Jupiter") {
    print("Jupiter is named after the Roman king of the gods.")
} else if(planet == "Neptune") {
    print("Neptune is inhospitable to life as we know it.")
} else if(planet == "Mars") {
    print("Mars has a lot of Earth-made rovers on it.")
} else if(planet == "Earth") {
    print("Earth is infested with something called 'humans'.")
} else {
    print("All the planets are pretty cool.")
}
```



Упражнение

Решение

С. 99

```
var number = 6

switch(number) {
case 6:
    print("6 means easy and smooth, all the way!")
case 8:
    print("8 means sudden fortune!")
case 99:
    print("99 means eternal!")
default:
    print("I've told you everything I know about lucky numbers.")
}
```

Операторы диапазонов

Кроме уже известных вам математических операторов и оператора присваивания, в Swift поддерживаются другие полезные операторы, включая набор диапазонных операторов. **Диапазонные операторы предоставляют сокращенную запись для представления диапазонов значений.**

Каждое из следующих двух выражений представляет диапазон 1, 2, 3, 4:

1...4 ← Замкнутый диапазонный оператор
 1..**<**5 ← Полузамкнутый диапазонный оператор

Диапазонные операторы особенно удобны для использования в циклах и других управляющих командах.

Замкнутый диапазонный оператор определяет диапазон от значения в левой части оператора до значения в правой части оператора.

Полузамкнутый диапазонный оператор определяет диапазон от значения в левой части оператора до значения, непосредственно предшествующего значению в правой части.

Также имеется **односторонний диапазонный оператор**, который позволяет определить диапазон, продолжающийся до бесконечности в одном направлении. Например, следующие константы определяют диапазон, который начинается с числа 5 и продолжается до бесконечности, и диапазон, который начинается с отрицательной бесконечности и продолжается до 100 соответственно:

```
let myRange = 5... ← Односторонний диапазонный оператор
let myOtherRange = ...100
```

После этого можно проверить, содержит ли константа некоторое значение:

```
myRange.contains(1) ← Возвращает false, потому что 1 не содержится в диапазоне от 5 до бесконечности.
myRange.contains(70) ← Возвращает true, потому что 70 входит в диапазон.

myOtherRange.contains(50) ← Возвращает true, потому что 50 содержится в диапазоне от отрицательной бесконечности до 100.
myOtherRange.contains(-10) ← Возвращает true, потому что -10 тоже входит в диапазон.
```



Упражнение

Создайте новую среду Swift Playground и напишите диапазонные операторы для следующих диапазонов:

- от 72 до 96
- от -100 до 100
- от 9 до бесконечности
- от отрицательной бесконечности до 37 000

→ Ответы на с. 107.

Более сложные команды switch

Допустим, вы хотите написать для колледжа код, который выводит текстовое описание оценки студента в баллах. В колледже принята следующая система диапазоновых оценок: от 0 до 49 баллов, от 50 до 59 баллов, от 60 до 69 баллов, от 70 до 79 и от 80 до 99. Задачу можно решить набором команд `if`, но выглядит такое решение не очень хорошо.

Вместо этого стоит воспользоваться командой `switch`:

```
let studentScore = 88
var result = "TBD"

switch studentScore {
  case 0...49:
    result = "Fail"
  case 50...59:
    result = "Pass"
  case 60...69:
    result = "Credit"
  case 70...79:
    result = "Distinction"
  case 80...99:
    result = "High Distinction"
  case 100:
    result = "Perfect"
  default:
    result = "Unknown"
}
```

Программа получает на входе оценку студента в баллах.

Создается строка с временным значением для хранения результата.

Затем происходит выбор в зависимости от оценки студента.

В каждой секции `case` для проверки результата используется диапазонный оператор. Так как студент набрал 88 баллов, его оценка принадлежит диапазону от 80 до 99. Отличный результат!

Мы еще рассмотрим другие примеры нетривиальных операторов, но на самом деле диапазонный оператор — всего лишь еще один бинарный оператор.



Но на этом возможности команды `switch` не исчерпаны. Рассмотрим напоследок еще один трюк (хотя позднее мы еще вернемся к `switch` в этой книге).

Представьте, что вы хотите выбрать выполняемый код в зависимости от четности или нечетности числа. Это можно сделать так:

```
var num = 5
switch num {
  case _ where num % 2 == 0:
    print("This number is an Even number!")
  default:
    print("This number is an Odd number!")
}
```

Если этот синтаксис вам еще не знаком, не беспокойтесь — мы еще к нему вернемся!

При помощи оператора остатка можно определить, является число четным или нечетным.

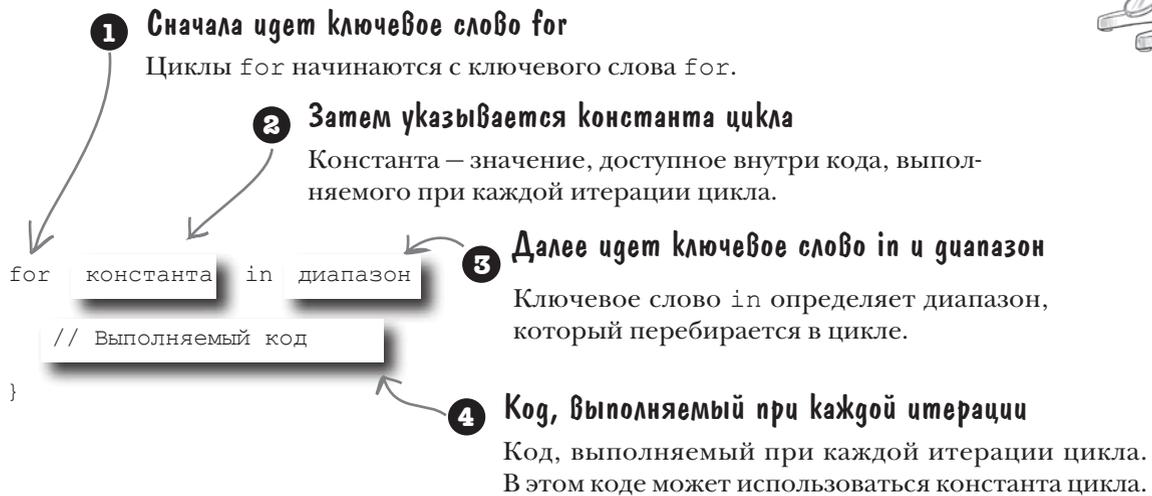
Множественное выполнение кода в циклах

В Swift поддерживаются две основные разновидности циклов. Цикл **for** (и **for-in**) позволяет выполнить код некоторое количество раз или перебрать все элементы коллекции, а цикл **while** (и **repeat-while**) выполняет код снова и снова, пока некоторое условие не будет равно true или false.

Анатомия Цикла for



Циклы for и for-in используются для выполнения кода определенное количество раз.

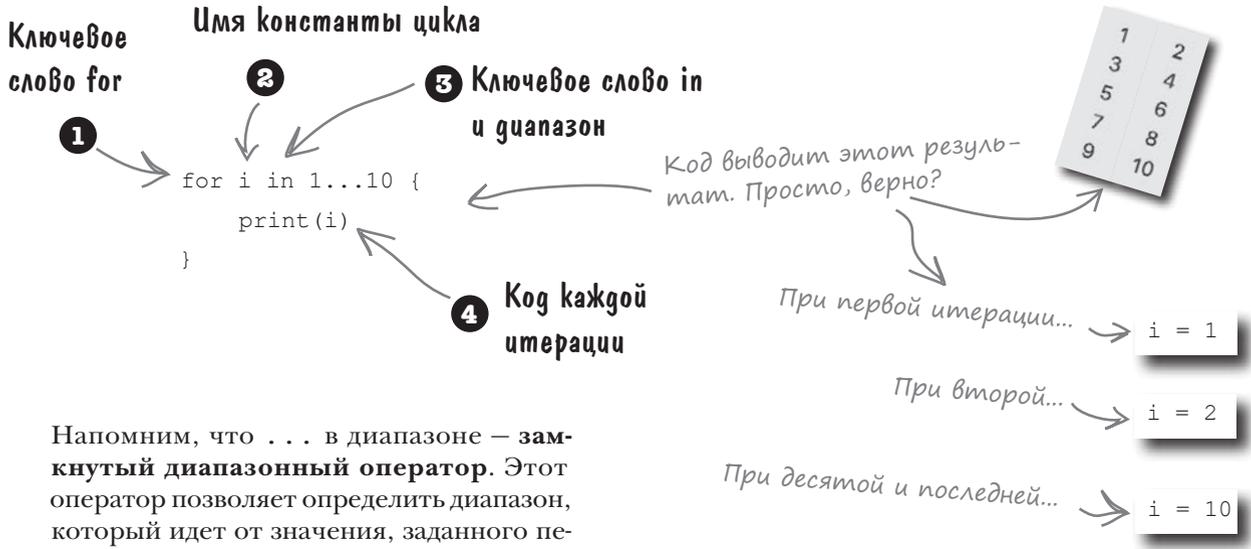


Ключевые моменты

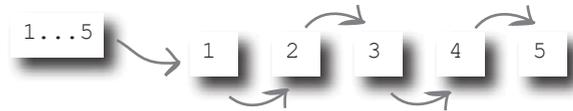
- Циклы **for** выполняют код многократно, обычно с перебором некоторого числового диапазона.
- Цикл **for-in** может перебрать содержимое последовательности — чаще всего элементы коллекции, но также и числа из заданного диапазона.
- Циклы **while** выполняют код, пока некоторое условие остается истинным.
- Циклы можно размещать внутри других циклов.

Построение цикла for

Напишем код для вывода чисел от 1 до 10 с использованием цикла `for`:



Напомним, что `...` в диапазоне — **замкнутый диапазонный оператор**. Этот оператор позволяет определить диапазон, который идет от значения, заданного перед оператором, до значения, заданного после оператора, включая оба значения. Таким образом, диапазон `1...5` содержит значения 1, 2, 3, 4, 5.



Упражнение

Объедините ваше знание математических операторов Swift со знанием синтаксиса циклов `for` в Swift и напишите цикл `for` для вывода нечетных чисел от 1 до 20. Некоторые советы приводятся ниже; возможно, для этого упражнения стоит создать новую среду Swift Playground:

- Вспомните оператор остатка, упоминавшийся ранее. Этот оператор вычисляет остаток от операции целочисленного деления.
- Замкнутый диапазонный оператор позволяет легко провести отсчет от 1 до 20.
- Используйте функцию `print` для вывода нечетных чисел. Проверку нечетности можно реализовать разными способами: применением промежуточной переменной, командой `if` или их комбинацией... Или еще каким-нибудь способом!

→ Ответ на с. 107.

Одну минуту... Я думала, что циклы for удобно использовать для перебора элементов массива. Как это работает?



Цикл for-in позволяет легко перебрать элементы массива.

От программирования пересыхает в горле. Представьте, что вы открыли кафе, а массив строк с именем drinks содержит информацию о том, какие напитки еще остаются в запасе.

Если вы хотите перебрать этот массив и сообщить посетителям, что они могут заказать, для этого можно воспользоваться циклом for-in.

Массив с напитками

Конечно, у нас есть лицензия на крепкие напитки...

```
let drinks = ["Coffee", "Tea", "Water", "Whisky"]
```

Перебор всех элементов в массиве drinks

```
for drink in drinks {
    print("\(drink) is still available!")
}
```

Делает то же самое, что и предыдущий фрагмент кода.

```
for d in drinks {
    print("\(n) is still available!")
}
```

Но откуда Swift знает, что напиток должен обозначаться словом drink?

Swift этого не знает.

Просто мы выбрали слово, которое поможет человеку понять, что здесь происходит.

Если ваш цикл предназначен только для перебора элементов некоторой структуры данных, также можно воспользоваться циклом forEach:

```
drinks.forEach {
    print("\( $0 ) is still available!")
}
```

Мы вызываем forEach для drinks. Мы можем использовать этот метод, потому что drinks является типом коллекции. Все типы коллекций поддерживают forEach.

Не беспокойтесь, если этот синтаксис кажется вам незнакомым; мы вернемся к нему позднее. Это всего лишь удобный способ обращения к текущему элементу.



Упражнение Решение

- 72 to 96
- -100 to 100
- 9 to infinity
- Negative infinity to 37,000

```
72...96
-100...100
9...
...37000
```

С. 102



Упражнение Решение

```
for i in 1...20 {
  if(i%2 != 0) {
    print(i)
  }
}
```

```
1
3
5
7
9
11
13
15
17
19
```

С. 105

Анатомия Цикла *while*



Цикл *while* используется в тех случаях, когда код должен выполняться снова и снова, пока некоторое условие не станет ложным.

1 Сначала идет ключевое слово *while*

Циклы *while* начинаются с ключевого слова *while*.

```
while условие {
    // Выполняемый код
}
```

2 Затем указывается некоторое условие

Цикл продолжает выполняться, пока условие дает результат *true*.

3 Далее следует код, выполняемый при каждой итерации

Код выполняется при каждой итерации цикла.

Построение цикла *while*

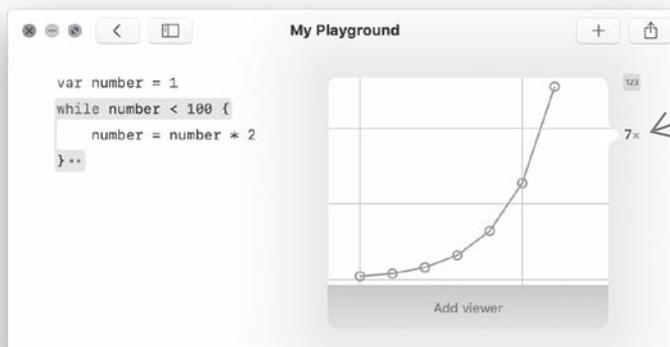
Напишем цикл *while*, который умножает число на 2 до тех пор, пока его значение остается меньше 100:

Цикл *while* особенно удобен тогда, когда вы не уверены, сколько итераций пройдет цикл до завершения.

Начинаем с числа 1.

```
var number = 1
while number < 100 {
    number = number * 2
}
```

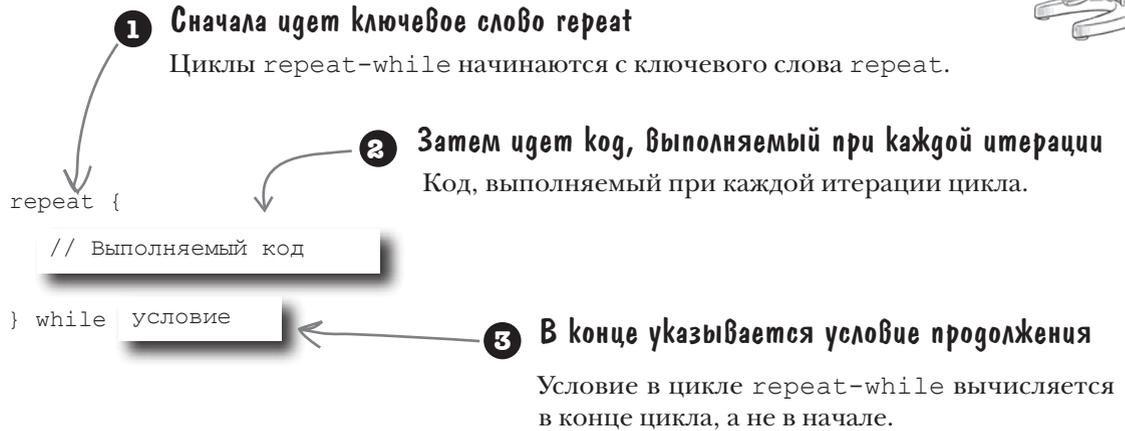
Кнопка вывода результата в Playground позволяет наглядно представить результаты многих операций,





Анатомия Цикла repeat-while

Цикл repeat-while очень похож на цикл while, не считая того, что условие вычисляется в конце цикла, а не в начале. Цикл всегда обрабатывает как минимум один раз.



Построение цикла repeat-while

Перепишем наш цикл while с использованием repeat-while:

```
var number = 1
repeat {
    number = number * 2
} while number < 100
```



Упражнение

Рассмотрим простую игру, которая начинается с уровня 1 и завершается на уровне 10. Мы уже написали часть цикла repeat-while для проведения игры. Заполните два пропуска.

```
var currentLevel = 1
var winningLevel = 10

repeat {
    print("We're at level \(currentLevel) of \(winningLevel)!")
    currentLevel = _____
} while (_____)
print("Game finished!"): _____
```

Ответ на с. 119.

Решение проблемы сортировки пиццы



Все это хорошо,
но вы так и не решили мою
проблему сортировки пиццы! Как это
сделать? У меня есть кое-какие идеи,
но программист здесь вы...

```
var pizzaHawaiian = "Hawaiian"  
var pizzaCheese = "Cheese"  
var pizzaMargherita = "Margherita"  
var pizzaMeatlovers = "Meatlovers"  
var pizzaVegetarian = "Vegetarian"  
var pizzaProsciutto = "Prosciutto"  
var pizzaVegan = "Vegan"
```



Упражнение

Возьмите список рецептов пиццы, хранящихся в независимо созданных строковых переменных, и предложите:

- Более эффективный способ представления списка рецептов пиццы с использованием одного из типов коллекций, о которых вы узнали ранее.
- Способ выполнения операций с коллекцией, который позволит отсортировать элементы в алфавитном порядке и вывести их.

→ Ответ на с. 119.

Часть Задаваемые Вопросы

В: Могу ли я определять собственные типы коллекций?

О: Мы займемся этим в главе 9, так как для этого потребуются некоторые расширенные средства Swift. А пока скажем: да, вы можете создавать собственные типы коллекций, которые не уступают трем стандартным (массивы, множества и словари).

В: А если мне нужно создать переменную, но я не знаю заранее, какой тип ей следует назначить?

О: Отличный вопрос! Возможно, вы заметили, что в нашем коде в выражениях, создающих переменные, иногда тип указывается, а иногда нет. Система типов Swift

может определять тип автоматически или выбирать его на основании предоставленной аннотации типа.

В: Я уже видел в других языках команды `switch`. Меня они не впечатлили, и я никогда ими не пользовался. Похоже, в Swift команды `switch` намного мощнее. Это так?

О: Да, вы правы. Команда `switch` в Swift обладает большими возможностями, чем в большинстве других языков. Команды `switch` чрезвычайно полезны, и в Swift они очень хорошо реализованы.

В: Как решить, когда использовать ту или иную управляющую команду?

О: К сожалению, это ситуация из ряда «увидел — узнал». Чем больше вы программируете, тем проще вам будет выбрать нужную команду.

В: Обязательно ли использовать синтаксис `for-in` для перебора элементов массива?

О: Нет, это можно сделать и вручную, как и в более старых языках программирования. Но зачем?

В: При использовании цикла `for-in` переменная должна называться `i`?

О: Нет, вы можете назвать ее как угодно.

Если вы захотите еще больше узнать о типах коллекций, посетите на GitHub проект Swift Collections. Он не является частью Swift, но его сопровождением занимается компания Apple, а сообщество разработчиков с открытым кодом создало ряд более полезных типов коллекций!

<https://github.com/apple/swift-collections>



Мозговой Штурм

Напишите собственные циклы `for-in`, `while` и `repeat-while`. Проявите фантазию и попробуйте написать бесконечный цикл, который никогда не завершится. Какое условие вы используете в своем бесконечном цикле? Подумайте, как предотвратить появление бесконечных циклов в вашем коде.

структурные элементы

выражения

константы

Так много всего

операторы

переменные

типы

За такое короткое время

Ого

Мы прошли большой путь!

коллекции

массивы

множества

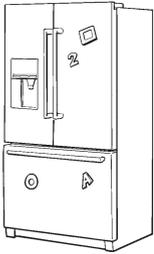
Невероятно: в своем путешествии в мир Swift вы прошли всего **три главы**, а уже сделано так много! Вы узнали о структурных элементах Swift, научились создавать среды Swift Playground и строить программы из операторов, выражений, переменных, констант, типов, коллекций и управляющих команд.

На нескольких ближайших страницах приводятся упражнения, которые проверят ваше понимание изложенных концепций. Мы также включили врезки ключевых моментов, которые напомнят вам важнейшие положения этой главы.

Повторение поможет закрепить то, что вы узнали о Swift. Не пропускайте эти страницы! А когда вы справитесь с упражнениями, сделайте перерыв и немного отдохните, чтобы двигаться дальше с новыми силами.



словари



Развлечения с магнитами

На холодильнике из магнитов была выложена программа Swift, но магниты перепутались. Удастся ли вам переставить фрагменты кода и получить работоспособную программу, которая выдает результат, приведенный на следующей странице? В коде используются концепции, подробно рассмотренные выше, а также те, которые в книге еще не упоминались.

```
var fido = Dog(name: "Fido", color: .brown, age: 7)
```

```
enum DogColor {
    case red
    case brown
    case black
}
```

```
listDogsInPack()
```

```
var pack: [Dog] = [fido, bruce]
```

```
var moose = Dog(name: "Moose", color: .red, age: 11)
```

```
addDogToPack(dog: moose)
```

```
var bruce = Dog(name: "Bruce", color: .black, age: 4)
```

```
class Dog {
    var name: String
    var color: DogColor
    var age: Int

    init(name: String, color: DogColor, age: Int) {
        self.name = name
        self.color = color
        self.age = age
    }
}
```

```
func listDogsInPack() {
    print("The pack is:")
    print("--")
    for dog in pack {
        print(dog.name)
    }
    print("--")
}
```

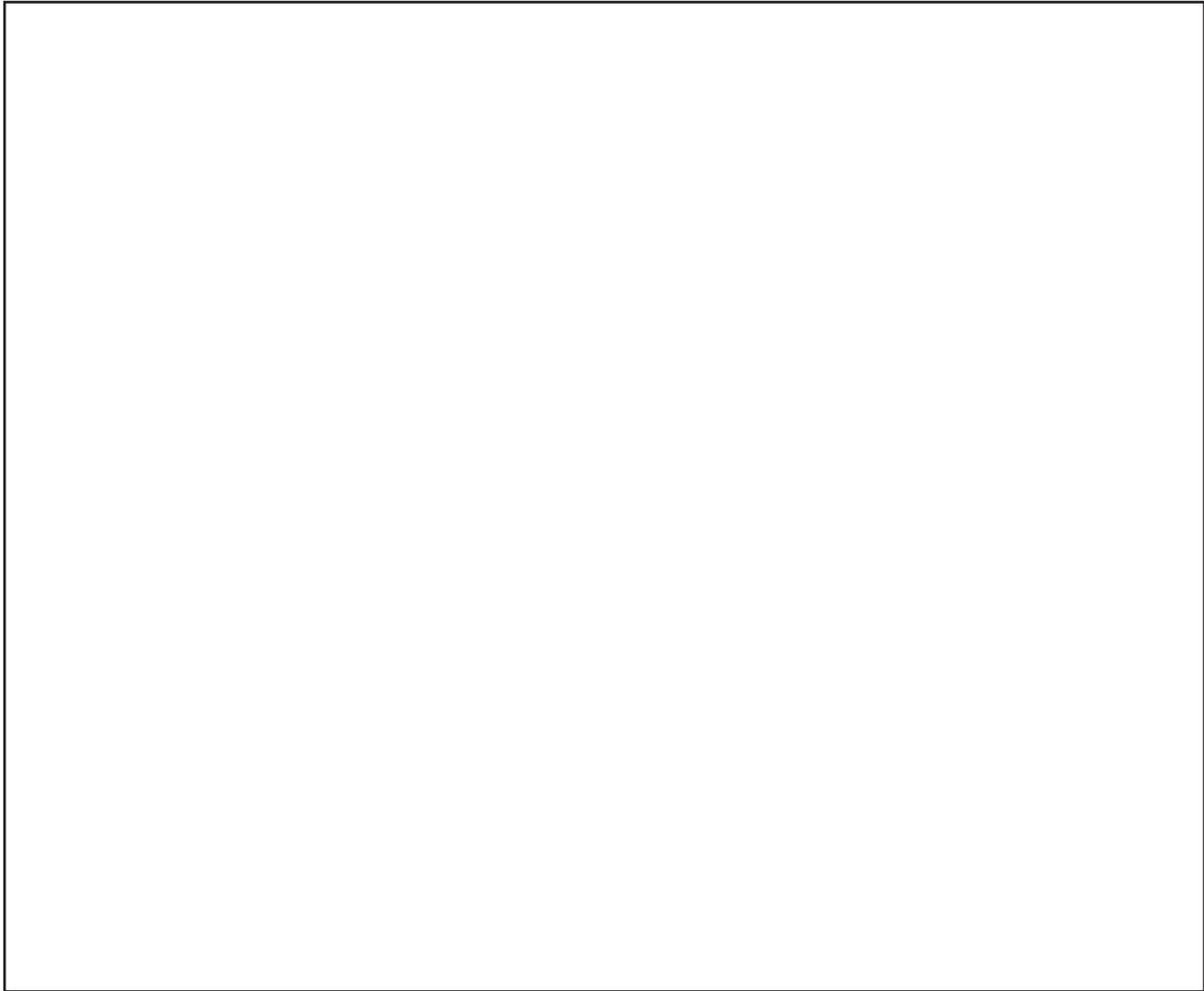
```
listDogsInPack()
```

```
func addDogToPack(dog: Dog) {
    pack.append(dog)
    print("\(dog.name) (aged \(dog.age)) has joined the pack.")
}
```

→ Ответ на с. 117.

Развлечения с магнитами, продолжение

Разместите здесь магниты с предыдущей страницы.



Вывод перепутанной программы. Удастся ли вам расставить фрагменты кода в правильном порядке? Каждый фрагмент должен быть использован в программе.

```
The pack is:
--
Fido
Bruce
--
Moose (aged 11) has joined the pack.
The pack is:
--
Fido
Bruce
Moose
--
```

У бассейна



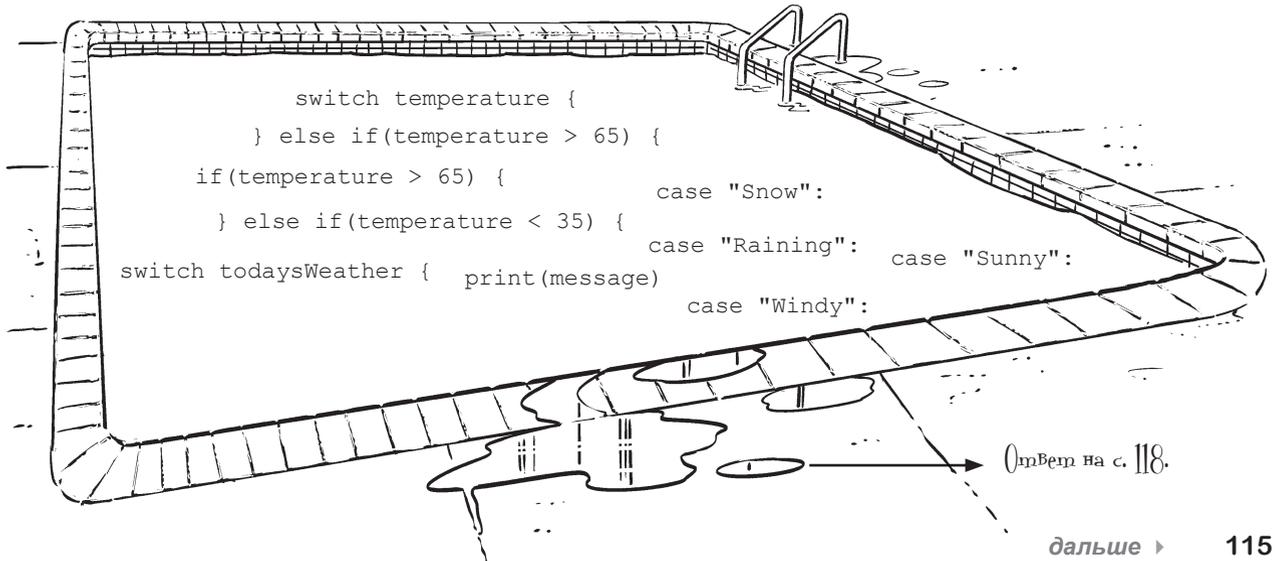
Выловите из бассейна строки кода и расставьте их в пустых строках среды Playground. Каждая строка может использоваться **только один раз**; использовать все строки не обязательно. Ваша задача — построить код, который будет генерировать приведенный ниже результат для исходных переменных:

```
var todaysWeather = "Windy"
var temperature = 35
```

Strap your hat on. It's windy! And it's not really cold or hot!

```
var message = "Today's Weather"
message = "It's a lovely sunny day!"
message = "Strap your hat on. It's windy!"
message = "Pack your umbrella!"
message = "Brrr! There's snow in the air!"
default:
  message = "It's a day, you know?"
message += " And it's not cold out there."
message += " And it's chilly out there."
} else {
  message += " And it's not cold or hot!"
}
```

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



Ответ на с. 118.

СТАТЬ КОМПИЛЯТОРОМ Swift



Каждый из фрагментов кода Swift на этой странице представляет законченную среду Playground. Вообразите себя на месте компилятора Swift и определите, будут ли выполняться каждый из этих фрагментов. Если они не будут компилироваться, то как вы их исправите?

A

```
let dogsAge = 10
let dogsName = "Trevor"
```

```
print("My dog's name is \(dogsName) and
they are \(dogsAge) years old.")
```

```
dogsAge = dogsAge + 1
```

B

```
var number = 10

for i in 1...number {
    print(number*92.7)
}
```

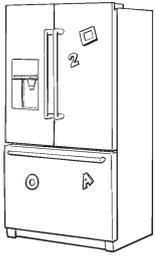
C

```
var bestNumbers: Set = [7, 42, 109, 53, 12, 17]

bestNumbers.remove(7)
bestNumbers.remove(109)
bestNumbers.remove(242)

bestNumbers.insert(907)
bestNumbers.insert(1002)
bestNumbers.insert(42)
```

→ Ответ на с. 118.



Развлечения с магнитами. Решение

С. 113

```

class Dog {
    var name: String
    var color: DogColor
    var age: Int

    init(name: String, color: DogColor, age: Int) {
        self.name = name
        self.color = color
        self.age = age
    }
}

enum DogColor {
    case red
    case brown
    case black
}

var fido = Dog(name: "Fido", color: .brown, age: 7)
var bruce = Dog(name: "Bruce", color: .black, age: 4)
var moose = Dog(name: "Moose", color: .red, age: 11)
var pack: [Dog] = [fido, bruce]

func addDogToPack(dog: Dog) {
    pack.append(dog)
    print("\ (dog.name) (aged \ (dog.age)) has joined the pack.")
}

func listDogsInPack() {
    print("The pack is:")
    print("--")
    for dog in pack {
        print(dog.name)
    }
    print("--")
}

listDogsInPack()
addDogToPack(dog: moose)
listDogsInPack()

```



У бассейна. Решение

С. 115

```

var todaysWeather = "Windy"
var temperature = 35
var message = "Today's Weather"

switch todaysWeather {
case "Sunny":
    message = "It's a lovely sunny day!"
case "Windy":
    message = "Strap your hat on. It's windy!"
case "Raining":
    message = "Pack your umbrella!"
case "Snow":
    message = "Brr! There's snow in the air!"
default:
    message = "Its a day, you know?"
}
if(temperature > 65) {
    message += " And it's not cold out there."
} else if(temperature < 35) {
    message += " And it's chilly out there."
} else {
    message += " And it's not cold or hot!"
}
print(message)

```

СТАТЬ КОМПИЛЯТОРОМ Swift. Решение

С. 116



- А** Произойдет ошибка, потому что мы пытаемся изменить константу (`dogsAge`). Чтобы это сработало, необходимо преобразовать константу `dogsAge` в переменную.
- Б** Произойдет ошибка при попытке умножения `Double` (92.7) на целое число (`number`). Кроме того, переменная-счетчик `i` вообще не используется внутри цикла.
- С** Работает нормально в приведенном виде.



Упражнение Решение

С. 109

```
We're at level 1 of 10!
We're at level 2 of 10!
We're at level 3 of 10!
We're at level 4 of 10!
We're at level 5 of 10!
We're at level 6 of 10!
We're at level 7 of 10!
We're at level 8 of 10!
We're at level 9 of 10!
We're at level 10 of 10!
Game finished!
```



Упражнение Решение

С. 110

```
var pizzas = ["Hawaiian", "Cheese", "Margherita", "Meatlovers", "Vegetarian",
"Prosciutto", "Vegan"]
pizzas = pizzas.sorted()
print(pizzas)
```

```
["Cheese", "Hawaiian", "Margherita",
"Meatlovers", "Prosciutto", "Vegan",
"Vegetarian"]
```


4. Функции и перечисления

Повторное использование кода

Все мы выполняем определенную функцию. Но очень важно выбрать для этой функции имя и ясно обозначить, что она возвращает.

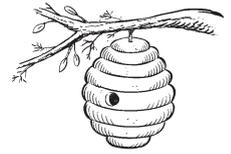


Функции в языке Swift позволяют упаковать некоторое поведение или единицу работы в блок кода, который может вызываться из других частей вашей программы. Функции могут быть автономными, а могут определяться как часть класса, структуры или перечисления, где они обычно называются методами. При помощи функций можно разбить сложные задачи на меньшие части, более удобные и легкие в тестировании. Функции занимают центральное место в формировании структуры программ в Swift.

Хорошо бы создавать автономные фрагменты кода, которые можно было бы использовать снова и снова для решения конкретных задач... А еще лучше, если бы им можно было присваивать имена. Как жаль, что это только мечты...



Функции Swift как средство повторного использования кода



Как и во многих языках программирования, в Swift поддерживаются *функции*. Они позволяют **упаковать код и повторно использовать его, если этот код должен быть выполнен несколько раз.**

На самом деле мы уже использовали функции: например, `print` — встроенная функция для вывода данных. В Swift также существует множество других встроенных функций, вскоре вам представится возможность использовать некоторые из них.

И это далеко не полный список!

Что могут делать функции?

Функции могут использоваться для самых разных целей, в том числе для определения блоков кода, которые:

- * Просто что-то делают, не получая никаких входных данных и не выдавая никаких результатов.
- * Получают столько входных значений конкретных типов, сколько требуется.
- * Возвращают столько выходных значений конкретных типов, сколько требуется.
- * Получают переменное количество входных значений.
- * Изменяют исходные значения переданных им переменных в дополнение к возвращению значений (или без него).

Функции позволяют упаковать код для решений конкретной задачи, присвоить этому коду имя и повторно использовать его, вызывая в нужный момент.



Будьте осторожны!

Опытные программисты иногда смешивают функции и методы.

Классы в книге еще не рассматривались, но когда речь пойдет о них, мы вернемся к методам. Вкратце: функции называются методами, когда они связываются с конкретным типом. Функцию можно создать внутри класса, структуры или перечисления, и такая функция (метод) будет инкапсулировать некоторый повторно используемый код, предназначенный для работы с экземпляром типа, в котором она определяется. Если вы ничего не поняли из сказанного, не огорчайтесь — скоро все будет понятно!

Встроенные функции

Прежде чем узнать, как реализовать собственные функции, выделим немного времени на некоторые встроенные функции Swift. **Эти функции встроены непосредственно в язык.**

Начнем с функций `min` и `max`. По именам можно предположить, что они возвращают наименьший и наибольший аргумент (соответственно) как свой результат.

Простой пример использования обеих функций, который можно протестировать в среде Playground:

```
print (min(9, -3, 12, 7))
print (max("zoo", "barn", "cinema"))
```

Поэкспериментируйте с примерами: что произойдет, если ввести «Zoo» вместо «zoo»?

Как нетрудно предположить, первый пример выведет "-3", потому что это наименьшее значение в списке аргументов.

Второй пример поначалу выглядит странно, но он выведет "zoo". Так как все аргументы являются строками, сравнение элементов в списке является строковым сравнением, а «zoo» в алфавитном порядке следует после «barn» и «cinema», поэтому считается, что эта строка больше двух других.



Мозговой Штурм

Представьте себя на месте компилятора Swift и предположите, что делает каждый из этих вызовов встроенных функций Swift. Будут ли они все работать? Если они работают, то какой результат они выдадут?

```
print(max(8, "Hello", -1))
```

```
print("Hello, world!")
```

```
print(min("Potato", "Tomato", "Cabbage"))
```

```
print(min(7*2, 3+4, 82-9, 76))
```

```
print(max(8, 9, 10, 11, 104, -104))
```

Что можно узнать по встроенным функциям?

По этим двум функциям можно сделать ряд важных выводов относительно функций в Swift:

- Функции получают входные значения (они называются *аргументами* функции) и возвращают результат.
- Некоторые функции поддерживают переменное количество аргументов. Например, `min` и `max` могут получать произвольное количество аргументов при условии, что их не менее двух и все они относятся к одному типу.
- Некоторые функции поддерживают более одного типа аргументов. Например, как `min`, так и `max` может работать с целыми числами, дробными числами и строками — при условии, что все аргументы имеют одинаковый тип. Более того, если Swift умеет сравнивать два экземпляра типа, то этот тип может использоваться с `min` и `max`.

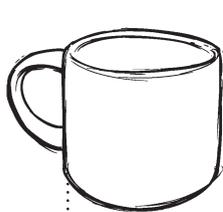
→ Эта возможность называется *вариадическими параметрами*, — вскоре мы вернемся к ней.

→ Чтобы понять, как работают «обобщенные» аргументы, необходимо кое-что знать о Swift. Мы все объясним, честное слово!

Вероятно, вы запомнили из предыдущих глав (или хотя бы сделали вывод из приведенного примера), что `print` тоже является встроенной функцией.

Она может получать переменное количество аргументов, и эти аргументы могут относиться к любому типу. Swift уже знает, как выводить простые типы (такие как числа и строки) и сложные типы (например, массивы и словари).

Для типов, определяемых пользователем (таких как классы или структуры), по умолчанию `print` выводит имя типа, но вы можете добавить в тип специальное свойство, которое описывает, как тип должен представляться в строковой форме. Если такое описание существует, то `print` выведет его вместо имени типа.

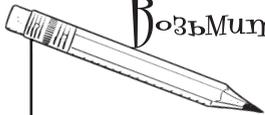


Расслабьтесь

Функции — дело хлопотное

При написании функции приходится учитывать много всего, что касается возвращаемых значений, параметров и т. д. Не обязательно делать каждую строку кода частью какой-нибудь функции, и не всегда очевидно, когда что-то должно быть функцией, пока вы не займетесь непосредственным построением своего продукта.

Не пытайтесь делать все функцией с того момента, как вы только начали программировать на Swift. Руководствуйтесь здравым смыслом, когда всплывают новые обстоятельства, и вскоре вы начнете понимать, когда вам могут пригодиться отдельные блоки повторного используемого кода. И не пытайтесь планировать в слишком отдаленной перспективе.



Возьмите в руку карандаш

Взгляните на следующий код и подумайте над тем, как он работает. Как вы считаете, он выглядит разумно? А когда вы прочитаете его (если потребуется, создайте среду Playground и выполните код), напишите небольшой анализ кода. Мы начали записывать некоторые свои мысли на следующей странице, но прежде чем заглядывать в них, сначала подумайте самостоятельно.

```
var pizzaOrdered = "Hawaiian"
var pizzaCount = 7

if (pizzaCount > 5) {
    print("Because more than 5 pizzas were ordered, a discount of 10% applies to the order.")
}

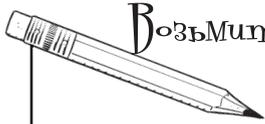
if (pizzaOrdered == "Prosciutto") {
    print("\(pizzaCount)x Prosciutto pizzas were ordered.")
}

if (pizzaOrdered == "Hawaiian") {
    print("\(pizzaCount)x Hawaiian pizzas were ordered.")
}

if (pizzaOrdered == "Vegan") {
    print("\(pizzaCount)x Vegan pizzas were ordered.")
}

if (pizzaOrdered == "BBQ Chicken") {
    print("\(pizzaCount)x BBQ Chicken pizzas were ordered.")
}

if (pizzaOrdered == "Meatlovers") {
    print("\(pizzaCount)x Meatlovers pizzas were ordered.")
}
```



Возьмите в руку карандаш

Решение

Код на предыдущей странице неплох... в той степени, в какой можно назвать неплохой вчерашнюю холодную пиццу, с которой отпали почти все добавки. Да, он работает, но написан он не очень разумно в нескольких отношениях.

На самом деле это плохо. Очень плохо.

Хороший код в действительности делает всего **две** ключевые вещи:

```
var pizzaOrdered = "Hawaiian"
var pizzaCount = 7
```

Мы проверяем, что переменная `pizzaCount` больше 5, и, если условие выполняется, выводим сообщение о том, что на заказ распространяется скидка. **1**

```
if (pizzaCount > 5) {
    print("Because more than 5 pizzas were ordered, a discount of 10%
    applies to the order.")
}
```

```
if (pizzaOrdered == "Prosciutto") {
    print("\(pizzaCount)x Prosciutto pizzas were ordered.")
}
```

```
if (pizzaOrdered == "Hawaiian") {
    print("\(pizzaCount)x Hawaiian pizzas were ordered.")
}
```

...<snip>

Мы проверяем все возможные рецепты пиццы, а затем выводим специальное сообщение об этом. Каждая из этих команд `if` делает фактически одно и то же. **2**

Снова и снова писать один и тот же код для каждого вида пиццы, не говоря уже о том, что в него будет трудно добавить новые виды пиццы или изменить логику при изменении требований (потому что изменения придется вносить отдельно для каждого вида пиццы). При таких изменениях очень легко допустить ошибку. Вы можете забыть обновить выводимое сообщение (например, логика будет проверять, сколько было заказано гавайской пиццы, а потом выводить сообщение о веганской пицце!).

Возникает масса потенциальных проблем.



Мозговой Штурм

Как бы вы исправили этот код? Если вы уже знакомы с функциями, рассмотрите возможные варианты разбиения кода на фрагменты. Здесь возможны разные варианты ответов, и все они будут более или менее верными в зависимости от общих целей вашего кода.

Когда могут пригодиться функции

Пару страниц назад был приведен неряшливый код Swift, который, по сути, делал одно и то же несколько раз подряд: он проверял, было ли заказано достаточно пиццы для применения скидки, после чего выводил количество и тип заказанной пиццы.

Напишем функцию `pizzaOrdered`.

Функция определяется ключевым словом `func`, за которым следует имя функции, ее параметры и возвращаемый тип функции.

Наша функция `pizzaOrdered` реализует процесс заказа с предыдущей страницы, но с существенно меньшим риском ошибок. У нее есть имя, которое вы уже знаете, а также два параметра: название пиццы в виде строки, а также объем заказа в виде целого числа. У нее нет возвращаемого значения (потому что функция не возвращает никакой результат, а только выводит сообщение).

Определение нашей новой функции выглядит так:

```
func pizzaOrdered(pizza: String, count: Int) {  
}
```

Определение функции начинается с ключевого слова `func`.

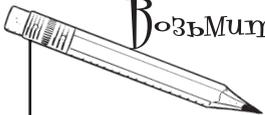
Функции присваивается имя, по которому она будет вызываться в программе.

Для работы функции необходимо знать две вещи: тип заказанной пиццы и количество заказанных пицц.

Далее следует тело функции. Тело содержит код, который выполняется при вызове функции.

Именованные типизированные значения, которые получает функция на входе, называются параметрами.

Возьмите в руку карандаш



Какой из следующих вариантов является действительным вызовом функции `pizzaOrdered`, определенной выше? Во всех случаях предполагается, что заказана гавайская пицца (Hawaiian), а объем заказа равен 7.

- A** `pizzaOrdered("Hawaiian", 7)`
- B** `pizzaOrdered(pizza: "Hawaiian", count: 7)`
- C** `pizzaOrdered(Hawaiian, 7)`
- D** `pizzaOrdered(pizza: Hawaiian, count: 7)`

Написание тела функции

Тело функции — то место, где происходит основное волшебство (в данном случае — повторно используемая логика). В нашем примере функция `pizzaOrdered` должна делать две вещи — те же, которые делал *неплохой* код из предыдущего примера:

```
func pizzaOrdered(pizza: String, count: Int) {
    if(count > 5) {
        print("Because more than 5 pizzas were ordered, a
            discount of 10% applies to the order.")
    }

    print("\(count)x \(pizza) pizzas were ordered.")
}
```

1 Скудка

Сначала необходимо проверить, что `count` (один из параметров функции) больше 5. Если он больше, выводится сообщение о скидке.

2 Описание заказа

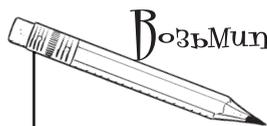
Затем выводится количество (`count`) и тип (`pizza`) заказанной пиццы.

Функции позволяют взять избыточный код (или код, который должен использоваться снова и снова с минимальными различиями) и сохранить его в месте, где вы сможете использовать его повторно.

Типизованные значения `count` и `pizza` существуют только внутри { тела } функции.



Вы можете вызывать свои функции (проще говоря, пользоваться ими) как по телефону.



Возьмите в руку карандаш

Решение

Правильный вызов функции `pizzaOrdered` — В. Это единственный ответ с двумя параметрами (`pizza` и `count`) и правильными аргументами ("Hawaiian" — строка, которая должна представляться в соответствующем виде, чего не происходит в ответах С и D).

О том, как вызвать функцию, рассказано на следующей странице.

Использование функций

Вернемся к путаному, неряшливому коду с заказом пиццы. Написанная нами функция позволяет заменить целую страницу кода из предыдущей версии следующей:

```
var pizzaOrdered = "Hawaiian"
var pizzaCount = 7
```

← Эта версия делает то же, что и предыдущая. Сначала мы просто определяем несколько переменных, которые будут использоваться для заказа.

Объявление функции и тело функции были приведены выше.

```
func pizzaOrdered(pizza: String, count: Int) {
    if(pizzaCount > 5) {
        print("Because more than 5 pizzas were ordered,
            a discount of 10% applies to the order.")
    }

    print("\(count)x \(pizza) pizzas were ordered.")
}
```

```
pizzaOrdered(pizza: pizzaOrdered, count: pizzaCount)
```

Здесь наша функция вызывается.

Мы передаем аргумент `pizzaOrdered` (строковая переменная, определенная выше) для параметра `pizza` (который должен получать строку).

↑ А здесь аргумент `pizzaCount` (целочисленная переменная, определенная выше) передается для параметра `count`, который должен получать целое число.

Также значения можно передать непосредственно в аргументах без создания или использования промежуточных переменных. Например, следующий код тоже допустим и делает абсолютно то же самое:

```
pizzaOrdered(pizza: "Hawaiian", count: 7)
```

Метки аргументов и имена параметров — не одно и то же. Иногда они совпадают — в функциях, которые определяются так:

```
func pizzaOrdered(pizza: String, count: Int) { }
```

← `pizza` и `count` — имена параметров и метки аргументов.

Но вы также можете независимо задавать метки аргументов и имена параметров:

```
func pizzaOrdered(thePizza pizza: String, theCount count: Int) { }
```

Наша функция будет вызываться так, как показано ниже. Обратите внимание, в теле функции по-прежнему будут использоваться те же имена (то есть имена параметров):

```
pizzaOrdered(thePizza: "Hawaiian", theCount: 7)
```

↑ `thePizza` и `theCount` — метки аргументов для параметров с именами `pizza` и `count`.

Я думала, что код Swift должен легко читаться, быть аккуратным и логичным. Странно, что `pizza` — имя параметра, а также часть имени функции!

Чтобы ваши функции стали более понятными, стоит тщательно выбрать их имена и метки аргументов для их параметров.

В нашей функции `pizzaOrdered`, так как `pizza` входит в имя функции, можно отказаться от назначения метки первому параметру. Swift часто подталкивает разработчика к тому, чтобы написанные им функции и код читались как обычные фразы на английском языке. Вспомните вызов нашей функции `pizzaOrdered`:

```
pizzaOrdered(pizza: "Vegetarian", count: 5)
```

Второе вхождение `pizza` избыточно для понимания того, что делает эта функция. Его можно убрать, для чего определение функции приводится к следующему виду:

```
func pizzaOrdered(_ pizza: String, count: Int) {
    if(pizzaCount > 5) {
        print("Because more than 5 pizzas were ordered,
            a discount of 10% applies to the order.")
    }

    print("\(count)x \(pizza) pizzas were ordered.")
}
```

Это означает, что функцию можно вызывать в следующем виде:

```
pizzaOrdered("Vegetarian", count: 5)
```

Такой вызов получается более логичным.

Если вместо метки аргумента используется символ подчеркивания `_`, это означает, что при вызове метку аргумента можно опустить. Символ подчеркивания может присутствовать не только в одном, но и в нескольких параметрах.



Будьте
осторожны!

Параметры функций являются константами.

Невозможно изменить значение параметра функции из тела функции. Например, если имеется параметр с именем `pizza` и для этого параметра функции передается строка `"Hawaiian"`, то имя `pizza`, существующее внутри функции, содержит константу со значением `"Hawaiian"`. Изменить ее не удастся.

Функции работают со значениями

Независимо от того, знакомы ли вы с другими языками программирования или просто пытаетесь разобраться, что происходит при вызове функции в Swift, важно осознать: **что бы вы ни передавали функции, это не изменит исходное значение**, если только вы явно не потребуете его изменить. Вы можете убедиться в этом:

1 Создайте функцию, которая выполняет простые вычисления

Допустим, функция умножает число на 42:

```
func multiplyBy42(_ number: Int) {
    number = number * 42
    print("The number multiplied by 42 is: \(number)")
}
```

2 Протестируйте свою функцию

При вызове этой функции для некоторого значения должен выводиться результат умножения этого значения на 42:

```
multiplyBy42(2)
```

Для этого вызова должно быть выведено сообщение "The number multiplied by 42 is: 84".

3 Создайте переменную

Создайте переменную для тестирования функции:

```
var myNumber = 3
```



myNumber

4 Вызовите функцию и передайте переменную как значение

Вызовите функцию `multiplyBy42` и передайте переменную `myNumber`:

```
multiplyBy42(myNumber)
```

Значение, хранящееся в `myNumber` (в данном случае 3), копируется в параметр функции. Переменная `myNumber` при этом не изменяется.

Функция должна вывести сообщение "The number multiplied by 42 is: 126".

5 Убедитесь в том, что переменная не изменилась

Добавьте вызов `print` и убедитесь в том, что исходная переменная `myNumber` не изменилась:

```
print("The value of myNumber is: \(myNumber)")
```



А если мне понадобится функция, которая изменяет значение переданной ей переменной?

Вы можете создать специальный параметр, который позволяет изменить исходное значение.

Если вам понадобится написать функцию, которая изменяет исходное значение, создайте **inout**-параметр.

Если вы хотите написать функцию, которая получает строковую переменную для своего единственного параметра и присваивает этой строковой переменной строку "Bob", это можно сделать так:

```
func makeBob(_ name: inout String) {
    name = "Bob"
}
```

После этого при вызове можно использовать любую строковую переменную, например такую:

```
var name = "Tim"
```

Передайте эту переменную функции makeBob:

```
makeBob(&name)
```

После вызова в переменной name будет храниться строка "Bob". Чтобы убедиться в этом, выведите переменную name:

```
print(name)
```

Символ &, помещенный перед именем переменной при передаче ее в функцию, указывает на то, что она может быть изменена функцией.

- Если параметр объявлен с ключевым словом **inout**, то для этого параметра нельзя передать функции константу или литеральное значение.

Ключевые моменты

- Поставьте перед типом **inout**-параметра ключевое слово **inout** в определении функции.
- Поставьте перед **inout**-параметром символ **&** в вызовах функции.
- В вызовах функции **inout**-аргументы должны быть переменными — они не могут быть выражениями или константами.



Мозговой Штурм

Перепишите функцию **multiplyBy42** так, чтобы вместо возвращения значения функция использовала **inout**-параметр.

С возвращением (из функций)

Функции, которые мы писали до настоящего момента, ничего не возвращали. Иначе говоря, они что-то делали, но не передавали никаких данных в точку, из которой они были вызваны. Тем не менее такая возможность существует. Функцию можно определить с возвращаемым типом. Тем самым вы сообщаете Swift, какое значение должно возвращаться функцией. Значение можно использовать или сохранить — на ваше усмотрение. Рассмотрим следующую функцию:

Синтаксис определяет возвращаемый тип (в данном случае это строка).

```
func welcome(name: String) -> String {
    let welcomeMessage = "Welcome to the Swift Pizza shop, \(name)!"
    return welcomeMessage
}
```

Эта строка возвращает текущее значение константы `welcomeMessage` (строка, на что указывает от определенных нами возвращаемый тип).

Но ведь Swift должен упрощать жизнь...

...поэтому создание сообщения и его возвращение можно объединить в одну строку:

```
func welcome(name: String) -> String {
    return "Welcome to the Swift Pizza shop, \(name)!"
}
```

А для очень простых функций можно поступить еще проще и воспользоваться возможностью неявного возвращения в языке Swift. Если все тело функции представляет собой одно выражение, то оно будет возвращено автоматически:

```
func welcome(name: String) -> String {
    "Welcome to the Swift Pizza shop, \(name)!"
}
```



Мозговой Штурм

Попробуйте дописать следующую функцию. Когда это будет сделано, измените функцию так, чтобы она возвращала `String` вместо прямого вывода сообщения.

```
func greet(name: String, favoriteNumber: Int, likesKaraoke: Bool) {
}
greet(name: "Paris", favoriteNumber: 6, likesKaraoke: true)
Hi, Paris! Your favorite number is 6, and you like karaoke.
```

Мне нравится предсказуемость. Можно ли написать функцию, у которой все параметры имеют значение по умолчанию, чтобы они изменялись только при крайней необходимости? Разумные значения по умолчанию — это хорошо.



Для каждого параметра функции можно определить значение по умолчанию.

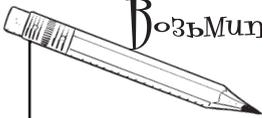
Когда вы определяете свою функцию, для каждого ее параметра можно предоставить значение по умолчанию. Например, если вы захотите определить значение по умолчанию в функции `welcome`, созданной на предыдущей странице, это будет выглядеть так:

Значение по умолчанию для `name` внутри функции

```
func welcome(name: String = "Customer") -> String {
    let welcomeMessage = "Welcome to the Swift Pizza shop, \(name)!"
    return welcomeMessage
}
```

Если в функции определено значение по умолчанию, то по желанию при вызове функции параметр можно не указывать.

Возьмите в руку карандаш



Что возвращает каждый из следующих вызовов обновленной функции `welcome`? Если какие-то вызовы недопустимы, то почему?

- `welcome(name: "Paris")`
- `welcome("Tim")`
- `welcome(Michele)`
- `welcome(name: String = "Mars")`

→ Ответ на с. 143.

Переменное количество параметров (Вариадические параметры)

В некоторых ситуациях удобно написать функцию, которая может получать переменное количество параметров. Например, можно написать функцию, которая получает список чисел и вычисляет его среднее значение. Такая функция будет куда более гибкой, если при каждом вызове ей можно передать столько чисел, сколько вам требуется в конкретном случае.

Иногда функция должна получать три числа, иногда только одно.

В Swift такая возможность существует — это так называемые **вариадические параметры**. Например, функцию для вычисления среднего значения по списку чисел можно реализовать так:

```
func average(_ numbers: Double...) -> Double {
    var total = 0.0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
```

Три точки (многоточие) означают, что параметр может представлять произвольное количество аргументов.

Внутри функции аргументы из вариадического параметра доступны в массиве с именем **numbers** (по имени параметра). Цикл **for** используется для перебора массива и вычисления накапливаемой суммы, и в конце сумма делится на количество элементов в массиве для вычисления среднего значения.

Вы уже видели код с циклами и массивами. Мы вернемся к этой теме в следующих главах.

Пример вызова функции:

```
let a = average(10, 21, 3.2, 16)
print (average(2, 4, 6))
```

→ a присваивается 16.625
→ Выводит 4.0



Будьте осторожны!

Вариадический параметр может содержать нуль элементов.

Вариадический параметр может быть «пустым» в точке вызова; вы должны проследить за тем, чтобы ваш код мог правильно обработать такую ситуацию. Попробуйте вызвать определенную ранее функцию average без аргументов и посмотрите, что при этом происходит. В дальнейших главах вы узнаете об опциональных типах и выдаче ошибок — двух разных вариантах обработки такой ситуации.

Ключевые моменты

- Вариадический параметр обозначается многоточием.
- Функция может иметь только один вариадический параметр, но он может находиться в любой позиции списка параметров.
- У всех параметров, определяемых после вариадического параметра, должны быть определены имена аргументов.
- При вызове функции с вариадическим параметром все параметры должны относиться к одному типу.



Беседа у камина

Актуальная тема: что в имени тебе моем?

Учитель Swift:

Наверное, сейчас ты уже осознал, что **имена функций должны быть уникальными?**

Вот только это не совсем так...

Потому что формально уникальной должна быть *сигнатура* функции, а не ее имя.

Сигнатура функции — это **совокупность ее имени, имен и типов ее аргументов и возвращаемого типа** (если он есть).

Конечно. Две функции с одинаковыми именами, но разными сигнатурами:

```
func addString(a: String, to b: String)
func addString(_ a: String, to b: String)
```

Нет! Имена аргументов первой функции — `a:` и `to:`, а имена аргументов второй функции — `_:` и `to:`.

Ученик

Да, Учитель, и это логично! Иначе как бы я или Swift смогли различать их?

Учитель! Но разве может быть иначе? Посмотрите, я даже только что попробовал. Если одно имя функции используется дважды, Swift жалуется на «недопустимое переобъявление».

Сигнатуры? А это еще что?

Учитель, а вы можете привести пример?

Я вижу, что они *слегка* отличаются, но разве имена аргументов не совпадают?

Точно. Я забыл, что имя аргумента совпадает с именем параметра только в том случае, если оно задано. Но вы упомянули, что различаться должны *сигнатуры*, а я еще не совсем понимаю, что это значит.

Учитель Swift:

Я рад, что ты не отвлекаешься. Как я говорил, сигнатура определяется совокупностью *имени функции, имен и типов аргументов и возвращаемого типа*. Таким образом, сигнатура первого примера выглядит так:

```
addString(a: String, to: String)
```

```
addString(_: String, to: String)
```

Хотя *имена функций* совпадают, их *сигнатуры* различаются.

Да — в этих примерах возвращаемого типа нет, поэтому формально возвращается **тип Void**.

Да. Так как возвращаемый тип отличается от двух других примеров, твоя новая функция имеет другую сигнатуру, и Swift не возражает.

Вовсе нет!

Верно, но функция с именем `addString`, которая получает два параметра `Float`, нарушает неписаные правила хорошего вкуса. Имя функции должно отражать ее назначение или особенности ее взаимодействия с аргументами. И ты должен хорошо запомнить этот урок!

Ученик:

Хорошо, а другая?

Ага, я начинаю понимать... кажется! Имена параметров не являются частью сигнатуры. Но вы упомянули, что возвращаемый тип *является* частью сигнатуры. Это так?

Выходит, я могу определить в том же файле другую функцию с другой сигнатурой — вот так?

```
addString(a: String, to: String) -> Bool
```

Начинаю понимать, Учитель! Получается, я могу написать другую функцию, которая выглядит так:

```
addString(a: Float, to: Float) -> Bool
```

Что? Вот теперь я совсем запутался, Учитель! Ведь сигнатуры различаются!

Да, Учитель! Против этого у меня нет ни одного аргумента.

Что можно передать функции?

Короткий ответ: функции можно передать практически все что угодно. Чуть более длинный ответ: функции можно передать все, что вам может понадобиться. **Передать можно любое значение Swift:** строку, логическое значение, целое число, массив строк... словом, все:

```
func doAThingWithA(string: String, anInt: Int, andABool: Bool) {
    print("The string says \"\(string)\",
          the integer is \(anInt),
          and the Boolean value is \(andABool)")
}
```

Всего лишь примеры типов, передаваемых функции в параметре...

```
doAThingWithA(string: "I am a string!", anInt: 7, andABool: true)
```

Но функциям могут передаваться не только значения. **Выражения тоже подходят:**

```
var name = "Bob"
```

```
doAThingWithA(string: "Hello, \(name)!", anInt: 20+22, andABool: true)
```

Результаты этих выражений будут переданы соответствующим параметрам.

В параметрах функций также можно передавать переменные и константы; собственно, именно это вы будете делать в большинстве случаев. Ниже приведены вызовы той же функции `doAThingWithA` с использованием переменных и констант вместо обычных значений:

```
var myString = "I love pizza."
```

```
var myInt = 42
```

```
let myBool = false
```

```
doAThingWithA(string: myString, anInt: myInt, andABool: myBool)
```

Это то же самое, что вызов `doAThingWithA(string: "I love pizza.", anInt: 42, andABool: false)` с литеральными значениями вместо переменных и констант.

У каждой функции есть тип

Определяя функцию, вы также определяете тип функции. Тип функции образуется из типов параметров функции, а также возвращаемого типа функции.

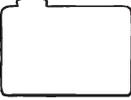
Примеры типов функций:

`(Int, Int) -> Int` ← Тип функции, которая получает два целочисленных параметра и возвращает целое число.
`(String, Int) -> String` ← Эта функция получает строковый параметр и целочисленный параметр и возвращает строку.
`(Int) -> Void` ← Функция получает один целочисленный параметр и не имеет возвращаемого значения.
`() -> String` ← Функция не имеет параметров и возвращает строку.


`(Int, Int) -> String`

Мой тип получает два целых числа и возвращает строку. А какой у тебя тип?

Мой тип просто возвращает целое число. И всё.


`() -> Int`

Ничего против тебя не имею, но мне нужна эта функция, которая возвращает строку!



Возьмите в руку карандаш



Взгляните на следующие функции. Будут ли они работать? Что они делают? Если они не работают, то что с ними не так?

```

A   func addNumbers(_ first: Int, _ second: Int) -> Int {
        return first + second
    }

B   func multiplyNumbers(_ first: Int, _ second: Int) -> Int {
        return first * second
    }

C   func sayHello() {
        print("Hello, friends!")
    }

D   func welcome(user: String) -> String {
        print("Welcome, \(user)!")
        return "User '\(user)' has been welcomed."
    }

E   func checkFor42(_ number: Int) -> Bool {
        if(number == 42) {
            return true
        } else {
            return false
        }
    }

```

→ Ответ на с. 143.



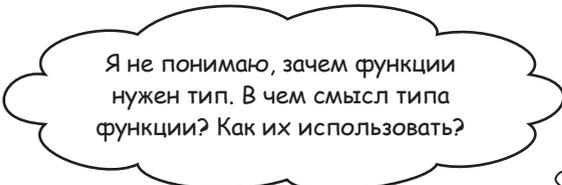
За сценой

Эта эпоха началась с выходом языка программирования С.

Когда-то в начале эпохи программирования разработчики слишком увлеклись использованием функций для всего, что только приходило им в голову. Это приводило к появлению модульных, но очень плохо читаемых программ.

Старайтесь выдерживать баланс. Если вы пришли в Swift с опытом другого языка программирования, вероятно, у вас уже имеется своя базовая линия относительно того, до какой степени нужно заходить в модульности ваших функций. Если ваш подход работает, используйте его в Swift!

Разбивать на функции абсолютно весь код не нужно, а вызовы функций Swift обходятся не слишком дорого. **Вы распознаете правильный баланс, когда увидите его.** И не забывайте ограничивать свои функции, чтобы они делали что-то одно.



Типы функций работают так же, как и любой другой тип.

Так как тип функции с точки зрения Swift ничем не отличается от других типов, его можно использовать как любой другой тип.

Особенно полезно иметь возможность определить переменную с типом функции:

```
var manipulateInteger: (Int, Int) -> Int
```

А затем создайте функцию, соответствующую этому типу функции:

```
func addNumbers(_ first: Int, _ second: Int) -> Int {
    return first + second
}
```

Присвоив функцию этой переменной:

```
manipulateInteger = addNumbers
```

вы сможете затем использовать присвоенную функцию через переменную, которой она была присвоена:

```
print("The result is: (manipulateInteger(10,90))")
```

← Суммирует 10 и 90, давая результат 100.

Так как переменная manipulateInteger может хранить любую функцию с тем же типом, ей можно присвоить другое значение:

```
func multNumbers(_ first: Int, _ second: Int) -> Int {
    return first * second
}
```

← Перемножает 2 и 5, давая результат 10.

```
manipulateInteger = multNumbers
```

```
print("The result is: (manipulateInteger(2,5))")
```

Раз этот тип не отличается от других, вы также можете поручить Swift автоматически определить тип функции при создании переменной:

```
var newMathFunction = multNumbers
```

← Система типов Swift автоматически определит тип переменной newMathFunction как (Int, Int) -> Int.

Возьмите в руку карандаш

Решение

C. 135

- `welcome(name: "Paris")`
 Будет работать и вернет сообщение.
- `welcome("Tim")`
 Работать не будет, потому что мы не указываем, что параметру присвоено имя `name`.
- `welcome(Michele)`
 Пытается передать переменную с именем `Michele`. Если это не строковая переменная или константа, работать не будет.
- `welcome(name: String = "Mars")`
 Работать не будет — вам не нужно указывать, что `name` является строкой (аннотация типа здесь не нужна и недопустима).

Возьмите в руку карандаш

Решение

C. 143

- ```

A func addNumbers(_ first: Int, _ second: Int) -> Int {
 return first + second
 }
B func multiplyNumbers(_ first: Int, _ second: Int) -> Int {
 return first * second
 }
C func sayHello() {
 print("Hello, friends!")
 }
D func welcome(user: String) -> String {
 print("Welcome, \(user)!")
 return "User '\(user)' has been welcomed."
 }
E func checkFor42(_ number: Int) -> Bool {
 if(number == 42) {
 return true
 } else {
 return false
 }
 }

```
- ← Складывает два переданных числа, работает нормально.
  - ← Перемножает два переданных числа, работает нормально.
  - ← Выводит приветственное сообщение, работает нормально.
  - ← Выводит приветственное сообщение и возвращает подтверждение. Работает!
  - ← Проверяет, что значение `number` равно 42, и возвращает соответствующее логическое значение. Работает нормально.

## Типы функций как типы параметров

Как вы уже поняли, типы функций с точки зрения Swift являются обычными типами. Это означает, что тип функции может использоваться как тип параметра при создании функции.

Взяв за основу пример, над которым мы работали ранее, рассмотрим следующую функцию:

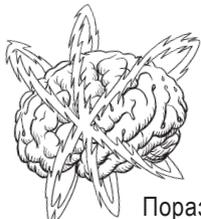
*Параметру присвоено имя, и он имеет тип (Int, Int) -> Int.*

```
func doMathPrintMath(_ manipulateInteger: (Int, Int) -> Int, _ a: Int, _ b: Int) {
 print("The result is: \(manipulateInteger(a, b))")
}
```

```
doMathPrintMath(addNumbers, 5, 10)
```

*Функция, переданная параметру manipulateInteger, используется для вывода результата, для чего она вызывается с двумя другими параметрами.*

*Вызывая эту новую функцию с функцией addNumber, созданной на предыдущей странице, для первого параметра, мы используем новую функцию для суммирования чисел и вывода результата.*



### Мозговой шторм

Поразмыслите над тем, как типы функций работают как типы параметров, и напишите собственные примеры.

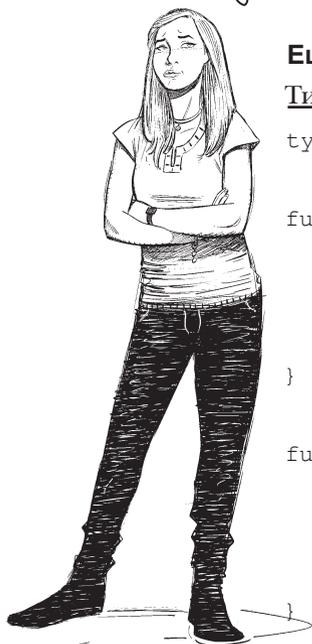
Рассмотрите следующую функцию:

```
func doMathPrintMath(_ manipulateInteger: (Int, Int) -> Int, _ a: Int, _ b: Int){
 print("The result is: \(manipulateInteger(a, b))")
}
```

Напишите три новые функции, которые могут передаваться как manipulateInteger. Ваши функции должны получать два целых числа и возвращать целое число.

Что делают ваши функции?

Означает ли это, что я могу вернуть функцию из другой функции?



**Еще раз: типы функций работают как любые другие типы.**

Типы функций являются **типами** во всех отношениях:

```
typealias Pizza = String

func makeHawaiianPizza() -> Pizza {
 print("One Hawaiian Pizza, coming up!")
 print("Hawaiian pizza is mostly cheese, ham, and pineapple.")
 return "Hawaiian Pizza"
}

func makeCheesePizza() -> Pizza {
 print("One Cheesey Pizza, coming up!")
 print("Cheesey pizza is just cheese, more cheese, and more cheese.")
 return "Cheese Pizza"
}

func makePlainPizza() -> Pizza {
 print("One Plain Pizza, coming up!")
 print("This pizza has no toppings! Not sure why you'd order it.")
 return "Plain Pizza"
}

func order(pizza: String) -> () -> Pizza {
 if (pizza == "Hawaiian") {
 return makeHawaiianPizza
 } else if (pizza == "Cheese") {
 return makeCheesePizza
 } else {
 return makePlainPizza
 }
}

var myPizza = order(pizza: "Hawaiian")
print(myPizza())
```

Простите... Я очень занятая функция, и я хочу вернуть сразу два значения при возвращении управления. Поможете?

## Несколько возвращаемых типов



### Отличная идея. Все для занятой функции.

Функция может вернуть кортеж с любым необходимым количеством значений. Например, если для пиццерии понадобится функция, которая получает имя и возвращает приветственное и прощальное сообщение в виде строк, это можно сделать так:

```
func greetingsFor(name: String) -> (hello: String, goodbye: String) {
 var hello = "Welcome to the pizza shop, \(name)!"
 var goodbye = "Thanks for visiting the pizza shop, \(name)!"
 return (hello, goodbye)
}
```

Присваивать имена возвращаемым значениям не нужно, так как имена были указаны ранее как часть возвращаемого типа.

После этого вы сможете вызвать функцию и обратиться к тексту приветственного сообщения следующим образом:

```
print(greetingsFor(name: "Bob").hello)
```

Чтобы обратиться к компоненту `hello` возвращаемого значения, мы используем точечную запись с вызовом функции.

Потрясающе! Спасибо. Теперь я смогу вернуть все, что захочу!



### Мозговой Штурм

Как вы думаете, как можно обратиться к части `goodbye` возвращаемого кортежа? Напишите код для ее вывода.

Когда это будет сделано, обновите функцию, чтобы она возвращала три значения: добавьте сообщение, которое просит посетителя подождать, пока готовится его заказ.



## Беседы у камина

**Актуальная тема: в порядке вещей**

### Учитель Swift:

Я вижу, у тебя есть вопрос...

Как же быстро ты все забываешь!

Да, мы уже говорили об этом в прошлом. Я было подумал, что ты полностью грокнул тот урок.

Не помню — я ведь учитель, а не ученик. Но вернемся к твоему вопросу: когда ты вызываешь функцию, компилятор сначала определяет, какая у нее должна быть *сигнатура*, а потом вызывает функцию с этой сигнатурой.

Да, но грокнул ли ты?

### Ученик:

Да, Учитель. Если я написал функцию, у которой каждому аргументу соответствует имя параметра, почему я не могу ее вызвать, передавая аргументы в любом порядке?

Забываю? Разве вы уже объясняли это, Учитель?

Что я *полностью грокнул тот урок*? Учитель! Что я вам говорил, почему *не стоит* пытаться подстраиваться под современный жаргон?

И при изменении порядка аргументов я **использую сигнатуру функции**, которая еще не определена! Кажется, я понял!

УЧИТЕЛЬ!



## Мозговой Штурм

Напишите функцию `billSplit` для разбиения обеденного счета на равные доли. Функция должна получать в параметрах общую стоимость обеда (`Double`), процент чаевых (`Int`) и количество обедающих (`Int`) и выводить сумму, которую оплачивает каждый участник.

## Функции не обязаны существовать автономно

Все примеры функций, которые рассматривались в этой главе, представляли так называемые **автономные**, или **глобальные**, функции. Они определяются в глобальной области видимости и могут вызываться из любой точки вашей программы.

Но в начале этой главы также говорилось о том, что функции могут определяться как часть класса. Функции, определяемые подобным образом, иногда называются **методами**. А использование функций (методов) как части класса является одним из характерных признаков объектно-ориентированного программирования. Swift также разрешает делать то, что не может быть сделано в других языках: определять функции как часть структур и перечислений.

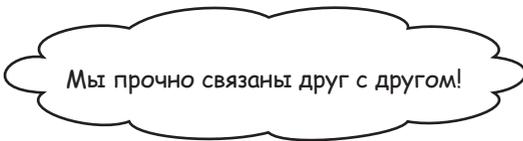
Все, что говорится в этой главе о функциях, также применимо к их определению в других контекстах. Вам не придется забывать все, что вы знали ранее; синтаксис, использование аргументов и имен параметров, вариативные параметры и inout-параметры — все это в равной степени относится к функциям, которые являются частью класса, структуры или перечисления.

### Вложенные функции

Функции также могут определяться внутри других функций. Этот способ определения помогает упростить излишне сложные функции, которые требуют повторяющегося выполнения кода или вычисления выражений, без необходимости объявлять функции в глобальной области видимости.

```
func sayHello() {
 func showMessage() {
 print("Hello, Swift Programmers!")
 }
 showMessage()
}
sayHello()
```

Эта функция вложена в другую функцию. Она существует только в области видимости sayHello.



### Мозговой Штурм

Напишите улучшенную версию функции `billSplit` для равномерного разбиения счета за обед. На этот раз функция должна иметь разумные значения по умолчанию для некоторых значений: 2 участника, 20% чаевых. При этом она, как и прежде, должна сообщать, сколько должен заплатить каждый участник.

Хммм... Я часто играю в настольные игры. Хорошо бы понять, как представить текущее состояние игрока в Swift.



### Перечисления позволяют создавать группы взаимосвязанных значений.

Представьте, что вы играете в сложную настольную игру, в которой игрок может находиться в разных состояниях. Пример:

```
var playerState: String
playerState = "dead"
playerState = "blockaded"
playerState = "winner"
```

Такой способ хранения данных оставляет желать лучшего: он создает сложности с управлением данными, а также недостаточно надежен и безопасен, если вы допустите опечатку при вводе строки, используемой для представления состояния. Короче говоря, он просто ужасен. На помощь приходят перечисления!

```
enum PlayerState {
 case dead
 case blockaded
 case winner
}
```

Теперь для описания состояния игрока можно воспользоваться перечислением вместо потенциально ненадежной строки:

```
func setPlayerState(state: PlayerState) {
 print("The player state is now \(state)")
}

setPlayerState(state: .dead)
```



## Определение перечисления

```
enum Thing {
}
```

## Связанные значения

```
enum Response {
 case success
 case failure(reason: String)
}
```

Это называется реализацией протокола — в данном случае протокола `CaseIterable`. Здесь протокол делает перечисление итерируемым. Далее протоколы будут рассмотрены более подробно.

## Итерируемое перечисление

```
enum Fruit: CaseIterable {
 case pineapple
 case dragonfruit
 case lemon
}

for fruit in Fruit.allCases {
 print(fruit)
}
```

## Инициализация по низкоуровневому значению

```
case Hat: Int {
 case top = 1
 case bowler
 case baseball
}

var hat = Hat(rawValue: 3)
```

## Добавление вариантов в перечисление

```
enum Fruit {
 case apple
 case banana
 case pear
}
```

## Неявное присваивание низкоуровневых значений

```
enum Planet: Int {
 case mercury
 case venus
 case earth
 case mars
 case jupiter
 case saturn
 case uranus
 case neptune
}
```

## Низкоуровневые значения

```
case Suit: String {
 case heart = "Red Hearts"
 case diamond = "Red Diamonds"
 case club = "Black Clubs"
 case spade = "Black Spades"
}

var card: Suit = .heart
print(card.rawValue)
```

## Switch с перечислениями

Команды `switch` и перечисления словно созданы друг для друга. В командах `switch` для выбора можно использовать связанные элементы перечисления, что очень удобно:

Используя перечисление `PlayerState` со связанными значениями...

...можно использовать элементы для выбора.

```
var playerOneState: PlayerState = .dead(cause: "crop failure")
switch (playerOneState) {
 case .dead(let cause):
 print("Player One died of \(cause).")
 case .blockaded(let byEnemy):
 print("Player One was blockaded by \(byEnemy).")
 case .winner(let score):
 print("Player One is the winner with \(score) points!")
}
```

Чтобы извлечь связанные значения при выборе, включите одно или несколько объявлений констант или переменных в команду `switch`.

Player One died of crop failure.



Будьте осторожны!

**Помните, что команды `switch` должны покрывать все возможные варианты.**

Если вы используете для выбора перечисление, то для каждого элемента перечисления должен быть предусмотрен вариант. Иначе говоря, в идеале команда `switch` содержит секцию `case` для каждого элемента в перечислении.

Также для обеспечения полноты команды `switch` можно предоставить секцию `default` и опустить все варианты, которые для вас неактуальны:

```
enum Options {
 case option1
 case option2
 case option3
}

var option: Options = .option1
switch(option) {
 case .option1:
 print("Option 1")
 default:
 print("Not Option 1")
}
```

Перечисление может иметь либо  
низкоуровневые, либо связанные значения,  
но не и то и другое одновременно.



## Упражнение

Создайте перечисление с названиями кинематографических жанров. Включите в него подборку жанров по своему выбору. После этого напишите команду `switch` с выбором в зависимости от переменной, в которой хранится жанр (например, с именем `favoriteGenre?`). Выведите короткий комментарий для каждого жанра.

Когда команда заработает, попробуйте создать новое перечисление, элементы которого представляют разные коктейли. Для каждого коктейля сохраните низкоуровневое строковое значение с описанием коктейля.

Когда это будет сделано, переберите элементы перечисления в цикле и выведите описание каждого коктейля.

→ Ответ на с. 154.

## Анатомия функции



### Определение функции

Ключевое слово `func` сообщает Swift, что вы собираетесь определить функцию.

Этой функции присвоено имя `welcomeCustomers`.

```
func welcomeCustomers {
}
```

Здесь размещается код, который выполняется при вызове функции (по имени). Он называется телом функции.

### Тело функции

```
func welcomeCustomers {
 print("Welcome to the pizza shop!")
}
```

### Функции могут иметь параметры и возвращаемые значения любых типов

Эта функция получает один параметр с именем `name`, который является строкой.

И она возвращает строку.

```
func sayHello(name: String) -> String {
 print("Hello, \(name)! Welcome to the pizza shop!")
 return "'\(name)' was welcomed."
}
```

### Функция с переменным количеством параметров и несколькими возвращаемыми типами

```
func biggestAndSmallest(numbers: Int...) -> (smallest: Int, biggest: Int) {
 var currentSmallest = numbers[0]
 var currentBiggest = numbers[0]
 for number in numbers[1..

```

Упражнение  
Решение

C. 152

```
enum Genres {
 case scifi
 case thriller
 case romcom
 case comedy
}

var favoriteGenre: Genres = .scifi

switch(favoriteGenre) {
case .comedy:
 print("Comedy is fine, as long as it's British.")
case .romcom:
 print("It had better star Hugh Grant.")
case .thriller:
 print("Only Die Hard counts.")
case .scifi:
 print("Star Trek is the best.")
}

enum Cocktails: String, CaseIterable {
 case oldfashioned = "Sugar, bitters, and whisky."
 case manhattan = "Bourbon, sweet vermouth, and bitters."
 case negroni = "Gin, red vermouth, and Campari."
 case mojito = "White rum, sugar, lime juice, soda water, and mint."
}

for drink in Cocktails.allCases {
 print(drink.rawValue)
}
```

## 5. Замыкания

# Необычные гибкие функции



Функции полезны, но иногда нужно больше гибкости. Swift позволяет использовать функцию как тип — так же, как вы используете целое число или строку. Это означает, что вы можете создать функцию и присвоить ее переменной. После того как функция будет присвоена переменной, ее можно вызывать через эту переменную или передавать эту функцию другим функциям в параметре. Когда вы создаете и используете функцию подобным образом, это называется замыканием. Замыкания очень полезны, потому что они способны сохранять ссылки на константы и переменные из контекста, в котором они были определены. Это называется замыканием по значению, отсюда и название.

## Знакомьтесь: простое замыкание

Перед вами замыкание:

```
let pizzaCooked = {
 print("Pizza is cooked.")
}
```

Начало функции. У этой функции нет имени.

Тело функции

Конец функции

Функция присваивается константе `pizzaCooked`.

`pizzaCooked()` вызывается точно так же, как обычная функция:

`pizzaCooked()`

Pizza is cooked.

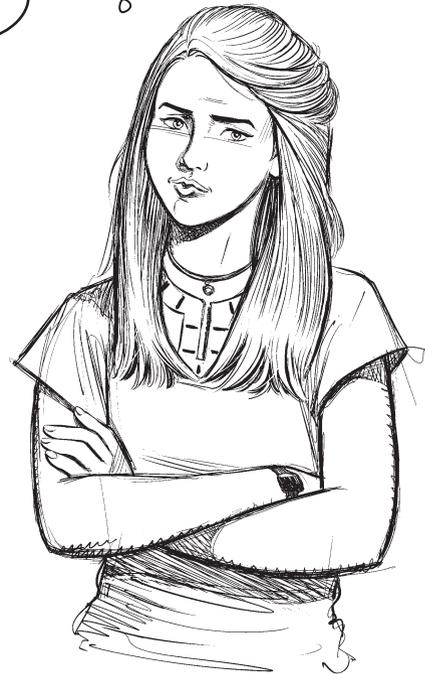


И какая от всего этого польза?

На первый взгляд может показаться, что замыкание — всего лишь еще один способ записи функций.

И это *отчасти* правильно, но замыкания позволяют делать много такого, на что функции не способны. Мы объясним, чем они так полезны, через несколько страниц.

Можно сказать, что замыкания — это функции, которые можно передавать в переменных. Впрочем, это только начало — дальше все становится еще интереснее!



Не понимаю. Это функция, а еще это переменная. Но почему бы просто не использовать функцию?!



### Безусловно, замыкания могут вызвать легкое замешательство.

Если вы не до конца понимаете замыкания, принципы их работы и чем они полезны — это вполне нормально, даже если программирование в целом вас не смущает. Без паники! К концу книги вы все поймете, а к концу этой главы будете чувствовать себя более уверенно.

**Замыкания очень полезны, когда вы хотите сохранить некоторую функциональность и использовать ее позднее — или использовать повторно.**

Некоторые ситуации, в которых бывает удобно сохранить функциональность:

- \* Вы хотите подождать некоторое время, а затем выполнить код в будущем.
- \* Имеется пользовательский интерфейс, и вы хотите дождаться завершения некоторой анимации или операции, прежде чем выполнять свой код.
- \* Вы выполняете некоторую сетевую операцию, время выполнения которой неизвестно заранее (например, загрузку данных), и вы хотите выполнить свой код после ее завершения.



## Тест-драйв

Рассмотрите несколько примеров замыканий. Одни из них допустимы, другие нет. Как вы думаете, чего не хватает недопустимым замыканиям? Самый простой способ опробовать их — выполнить фрагменты в среде Playgrounds, а потом вызвать. А когда вы будете знать, какие варианты правильные, а какие нет, попробуйте самостоятельно написать правильное замыкание и вызвать его.

```
var closureMessage = {
 print("Closures are pretty similar to functions!")
}

let bookTitle = {
 print("Head First Swift")
}

var greeting() = {
 print("Hello!")
}
```

## Замыкания с параметрами

Замыкания без параметров особой пользы не приносят. Конечно, интересно создать и присвоить безымянную функцию переменной или константе, но их потенциал в полной мере начинает проявляться тогда, когда вы начинаете понимать возможности параметризованных замыканий.

Небольшой недостаток этого плана заключается в том, что *замыкания получают параметры не так, как это делают обычные функции*. Представьте, что замыкание, которое мы присвоили ранее константе `pizzaCooked`, было бы обычной функцией.

Если вы захотите добавить параметр, который указывает конкретную разновидность пиццы, определение функции будет выглядеть примерно так:

```
func cooked(pizza: String, minutes: Int) {
 print("\(pizza) Pizza is cooked in \(minutes) minutes.")
}
```

Тогда вызов функции `cooked` выглядел бы так:

```
cooked(pizza: "Hawaiian", minutes: 7)
```

Если бы вы захотели записать ту же функциональность в виде замыкания, это могло бы выглядеть так:

```
let pizzaCooked = { (pizza: String, minutes: Int) in
 print("\(pizza) Pizza is cooked in \(minutes) minutes.")
}
```

А вызов замыкания выглядел бы так:

```
pizzaCooked("Hawaiian", 7) ← Вероятно, вы уже догадались,
 что будет выведено.
```



### За сценой

Почему же замыкания получают свои параметры в фигурных скобках и используют ключевое слово `in` для пометки конца списка параметров (и начала тела замыкания)? Представьте, что записали их более интуитивным (на первый взгляд) способом:

```
let pizzaCooked = (pizza: String, minutes: Int)
```

Может показаться, что мы пытаемся создать кортеж, и компилятор Swift придет в замешательство. Размещение параметров в фигурных скобках ясно показывает, что все замыкание представляет собой блок данных, хранящийся в переменной (или константе). Ключевое слово `in` четко показывает, где начинается тело, потому что второй пары фигурных скобок нет.



Когда я создаю функцию, она может возвращать значения. А с замыканиями это возможно?

**Замыкания, как и функции, могут возвращать значения.**

По своей функциональности замыкания в основном идентичны функциям, но синтаксис их создания несколько отличается. Чтобы создать замыкание, которое может возвращать значение, определите возвращаемый тип перед ключевым словом `in`:

```
let hello = { (name: String) -> String in
 return "Hello, \(name)!"
}
```

Для объявления возвращаемого типа используется такой же синтаксис, как для функций.

Пример использования этого замыкания с выводом возвращаемого значения:

```
let greeting = hello("Harry")
print(greeting)
```



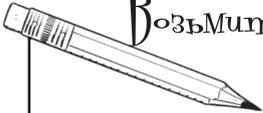
**Мозговой Штурм**

Напишите замыкание, которое получает в параметре одно целое число и возвращает это число, умноженное само на себя, включенное в строку.

Напишите код для использования этого замыкания и передайте значение 10 (замыкание должно вывести строку вида "The result is 100" или что-нибудь в этом роде).

Затем измените замыкание, чтобы оно получало в параметрах два целых числа. Протестируйте замыкание с числами 10 и 50 (должна быть возвращена строка вида "The result is 500").

## Возьмите в руку карандаш



Взгляните на следующие примеры замыканий. Как вы думаете, какой тип имеет каждое из них? Если у вас возникнут затруднения, вспомните типы функций, о которых говорилось в предыдущей главе.

**A**

```
let pizzaCooked = { (pizza: String, minutes: Int) -> String in
 return "\(pizza) Pizza is cooked in \(minutes) minutes."
}
```

**B**

```
let pizzaCooked = { (pizza: String, minutes: Int) -> String in
 var message = "\(pizza) Pizza is cooked in \(minutes) minutes."
 print(message)
 return(message)
}
```

**C**

```
let pizzaCooked = { (pizza: String) in
 print("\(pizza) Pizza is cooked in 10 minutes.")
}
```

**D**

```
let pizzaCooked = { (minutes: Int) in
 print("Hawaiian Pizza is cooked in \(minutes) minutes.")
}
```

—————> Ответ на с. 162.

## Ключевые моменты

- Замыкания — блоки кода, которые можно передавать и использовать в программах, как и переменные.
- Замыкание может иметь произвольное количество параметров, в том числе и ни одного.
- Каждое отдельное замыкание имеет тип, включающий типы параметров замыкания.
- Замыкания хорошо подходят для кода, который должен выполняться в определенный момент времени, например при завершении некоторой операции или если потребуется передать блок кода, который что-то делает, и использовать его в разных контекстах.
- Замыкания в целом работают как функции. В целом.

Конечно, я не программист,  
но нельзя ли внедрить замыкания  
в систему заказа пиццы, которую  
мы создали ранее?

### Замыкания могут использоваться в параметрах.

Ранее при изучении функций мы создали функцию `order`, которая получает название пиццы в формате `String` и возвращает функцию с типом `() -> Pizza` (`Pizza` – псевдоним типа для `String`).

Наблюдательный шеф-повар прав: замыкания можно использовать для создания сходных блоков функциональности. Прежде всего функция `order` могла бы выглядеть так:

```
func order(pizza: () -> Void) {
 print("# Ready to order pizza! #")
 pizza()
 print("# Order for pizza placed! #")
}
```

Для каждого вида пиццы определяется замыкание. Например, для гавайской пиццы оно может выглядеть так:

```
let hawaiianPizza = {
 print("One Hawaiian Pizza, coming up!")
 print("Hawaiian pizza is cheese, ham, & pineapple.")
}
```

Тогда для заказа гавайской пиццы можно будет воспользоваться командой:

```
order(pizza: hawaiianPizza)
```

Как вы думаете, что выведет  
эта команда?



## Мозговой Штурм

Замыкание для гавайской пиццы записано выше. А как будут выглядеть замыкания для других рецептов, например пиццы с сыром (`Cheesey`) и базовой пиццы (`Plain`)? Обратитесь к разделу «Типы функций как типы параметров» на с. 142 предыдущей главы, ознакомьтесь с рецептами и напишите замыкания. Чтобы протестировать замыкания, передайте их функции `order`.



Одну минуту. А я могу использовать замыкание с параметрами как параметр?



**Замыкания с параметрами сами могут использоваться как параметры.**

Разбираться в том, что происходит при использовании замыкания как параметра, уже немного хлопотно. Но когда вам нужно использовать замыкание, получающее параметры, как параметр для чего-то еще, голова начинает идти кругом.

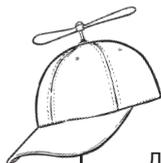
Не беспокойтесь. Разобраться в происходящем местами непросто, но у вас непременно все получится.

В замыканиях нет ничего волшебного. Возможно, термин звучит устрашающе, но в действительности это всего лишь функция. Вы уже знаете, как передать функцию в параметре; то же самое можно сделать с замыканиями. (Ведь это обычные функции!)

Возьмите в руку карандаш  
Решение

C. 160

- A (String, Int) -> String
- B (String, Int) -> String
- C (String) -> Void
- D (Int) -> Void



## Серьезное программирование

Давайте немного поэкспериментируем с замыканиями и параметрами. Местами разобраться в происходящем бывает непросто, но эта концепция регулярно встречается в мире Swift.

Рассмотрим следующую функцию:

*Присвоено имя performWithPiano.*

```
func performWithPiano(song: (String) -> String) {
 var performance = song("Piano")
 print(performance)
}
```

*В параметре получает замыкание, к которому обращается по имени song.*

*Замыкание получает String в параметре, а также возвращает String.*

Если у вас имеется замыкание, удовлетворяющее требованию о получении String и возврате String, допустим, с именем neverGonnaGiveYouUp, — функцию performWithPiano можно вызвать так:

```
performWithPiano(song: neverGonnaGiveYouUp)
```

'Never Gonna Give You Up' on Piano

Как может выглядеть замыкание neverGonnaGiveYouUp, чтобы оно удовлетворяло требованиям, а функция performWithPiano в итоге вывела правильный результат?

Мы начали писать его за вас. Попробуйте завершить его и протестируйте замыкание в среде Playgrounds.

```
var neverGonnaGiveYouUp = {
```

Когда замыкание заработает, попробуйте создать другое замыкание для другой песни.

На самом деле замыкания идеально подходят для некоторых задач, как вы узнаете ближе к концу книги...

Ладно... Так мы сможем запрограммировать систему заказа пиццы, в которой пиццы можно изменять... с использованием замыканий?

**Безусловно, сможем! Эта задача идеально подходит для замыканий.**

Ранее мы реализовали функцию с именем `order`, которая получала замыкание в параметре. Функцию можно изменить, чтобы она получала замыкание, которое имеет собственные параметры.

Функция `order` может выглядеть так:

Функция получает имя клиента в виде `String`.

Тип параметра замыкания указывается в круглых скобках.

```
func order(customer: String, pizza: (Int) -> String) {
 // Заказ пиццы
}
```

А еще она получает тип `Int` — замыкание типа, который получает `Int` и возвращает `String`.



Функция `order` сама решает, что она будет делать с замыканием. Эта функция может получать в параметре `pizza` любое замыкание — при условии, что оно получает один целочисленный параметр и возвращает строку. В нашем представлении целочисленный параметр представляет объем заказа конкретной пиццы, а возвращаемая строка содержит подтверждающее сообщение с информацией об объеме заказа и типе заказанной пиццы, но в принципе может содержать что угодно.

Так как замыкание представляет собой удобный и аккуратный блок функциональности, мощь кроется в возможности простой замены этого блока: функции `order` не нужно думать о том, как работает замыкание, представляющее конкретную пиццу, и если в будущем шеф-повар изобретет действительно сложную пиццу (они это делают постоянно), мы сможем написать новое замыкание для этого рецепта.

Все, что потребуется от замыкания, — чтобы оно получало целое число и возвращало строку.

Это `(Int) -> String`, если вы вдруг забыли...



## Анатомия замыкания

### Выражение замыкания

Выражение замыкания – способ записи замыканий в простом и прямолинейном синтаксисе:

Параметры могут быть объявлены с `input` (но не могут иметь значение по умолчанию), также может использоваться вариативный параметр при условии, что ему назначено имя.

Начало тела замыкания обозначается ключевым словом `in`.

```
{ (parameters) -> return type in
 команды
}
```

Внутри замыкания размещаются команды — их может быть сколько угодно.

### Завершающие замыкания

Если замыкание является последним или единственным параметром функции (или другого замыкания), то для передачи замыкания можно использовать синтаксис *завершающего замыкания*.

Допустим, у вас имеется функция, которая получает замыкание в своем последнем или единственном параметре:

```
func saySomething(thing: () -> Void)
{
 thing()
}
```

В таком случае для замыкания можно воспользоваться синтаксисом завершающего замыкания:

```
saySomething {
 print("Hello!")
}
```

Синтаксис называется «завершающим замыканием», потому что оно завершает функцию. Оно записывается после круглых скобок вызова функции, хотя и остается аргументом функции.

Без синтаксиса завершающего замыкания вам пришлось бы использовать запись, которая выглядит приблизительно так:

```
saySomething(thing: {
 print("Hello!")
})
```

По сути, это просто удобный, компактный синтаксис!

## Вычисление сводного значения

### Задача

Требуется написать код, который получает массив целых чисел и вычисляет на основании массива одно целое число. Мы хотим иметь возможность изменить способ вычисления сводного значения.



### Решение

Можно написать специальную функцию, которая получает массив целых чисел и возвращает целое число, и написать отдельную функцию для каждой выполняемой операции: одну для перемножения чисел с целью вычисления итогового числа, одну для суммирования, и т. д.

А можно воспользоваться замыканиями.

#### 1 Создание функции для работы с массивом

В первом параметре передается массив целых чисел.

```
func operateOn(_ array: [Int], operation: (Int, Int) -> Int) -> Int
{
 // Здесь размещается код
}
```

Во втором параметре передается замыкание (или функция) типа, который получает два целых числа и возвращает целое число.

А сама функция возвращает целое число.

#### 2 Реализация массива

```
func operateOn(_ array: [Int], using operation: (Int, Int) -> Int) -> Int {
 var cur = array[0] ← Сохраняет первый элемент переданного массива.

 for item in array[1...] {
 cur = operation(cur, item) ← Перебирает массив со следующего (1) элемента и обновляет текущее число, передавая его и текущий элемент перебора массива переданному замыканию operation.
 }

 return cur ← Возвращает текущее число.
}
```

## Свертки с использованием замыканий

### 3 Создание массива для тестирования

```
let numbers = [7, 14, 6, 1, 8]
```

### 4 Тестирование функции

Константа для хранения результата

```
let test = operateOn(numbers, operation: {(total: Int, next: Int) in
 return total * next
})
```

Результат функции `operateOn` присваивается нашей константе.

Массив `numbers` передается для первого параметра.

Для второго параметра передается замыкание в форме выражения замыкания.

Замыкание получает два параметра (целые числа `total` и `next`).

Возвращает значение `total`, умноженное на следующее целое число.

### 5 Проверка результата

```
print(test)
```



За сценой

Операторы Swift на самом деле являются функциями! Это означает, что если вы хотите выполнить свертку массива, используя нашу функцию `operateOn` с суммированием всех элементов, то для этого можно просто передать встроенный оператор Swift `+`:

```
let sumResult = operateOn(numbers, operation: +)
```

Также опробуйте операторы `*`, `-` и `!`

Выходит, замыкание — это просто... переменная, которая может содержать код?



### В десятку.

Вы уже неплохо знаете, как хранить целые числа, числа с плавающей точкой, строки, логические значения, словари и массивы. Замыкание — всего лишь еще один тип данных.

Замыкание представляет собой переменную (или константу), в которой хранится некий код. И если в следующей команде нет ничего необычного:

```
var myNumber = 100
```

то ничего необычного нет и в этой команде:

```
let myClosure = {
 print("Hello!")
}
```

Это всего лишь замыкание.



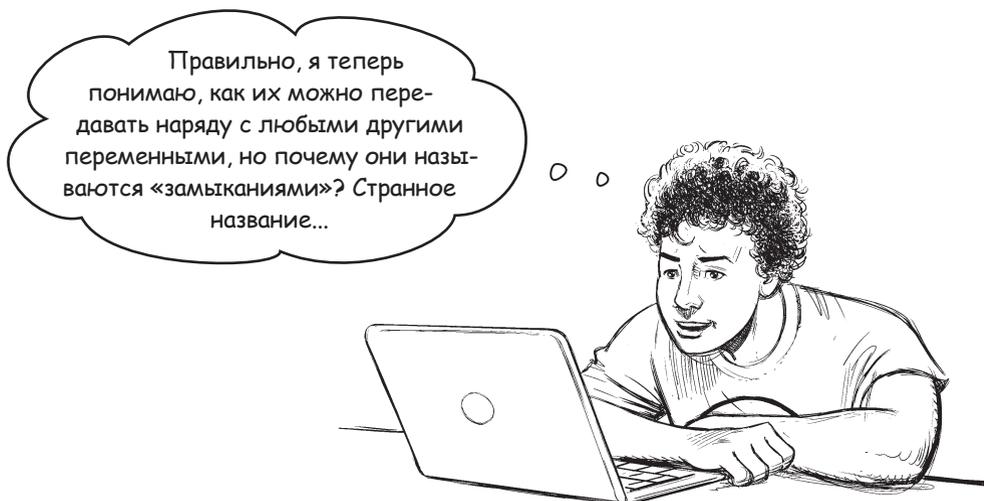
## Мозговой Штурм

Вместо того чтобы передавать функции `operateOn` выражение замыкания, создайте замыкание, присвоенное переменной, и передайте его.

Ваше замыкание должно выполнять свертку суммированием элементов массива (то есть оно должно взять все числа, содержащиеся в массиве, и свести их к одному числу — в данном случае сумме всех элементов).

Протестируйте свое замыкание (предполагается, что оно называется `sumClosure`), выполнив его следующей командой:

```
let sumTest = operateOn(numbers, operation: sumClosure)
```



### Название странное, здесь не поспоришь.

Замыкания так называются, потому что они сохраняют переменные за пределами своей области видимости (или замыкаются по ним).

Если вы создадите следующую целочисленную переменную:

```
var count = 5
```

а затем сразу же после этого создадите замыкание, которое выглядит так:

```
let incrementer = {
 count +=1
}
```

у этого замыкания не возникнет проблем с увеличением `count`, потому что и `count`, и `incrementer` (замыкание) были созданы в одной области видимости.

Изменения `count` выполняются одинаково как внутри замыкания, так и вне его. Замыкание `incrementer` сохранило значение `count` (замыкается по нему).



### Мозговой Штурм

Как вы думаете, чему будет равна переменная `count`, если вы введете этот код в среде Playgrounds, а затем вызовете `incrementer` десять раз подряд?

## Сохранение значений из внешней области видимости

= невероятно полезно

Вследствие такого поведения замыканий мы можем решать полезные задачи как в следующей функции подсчета:

Функция с именем `counter` не получает параметров.

```
func counter() -> () -> Int {
 var count = 0
 let incrementer: () -> Int = {
 count += 1
 return count
 }
 return incrementer
}
```

Возвращает замыкание. Возвращенное замыкание не имеет параметров и возвращает целое число.

Возвращенное замыкание увеличивает внутреннюю переменную `count` при каждом вызове.

И при каждом вызове функции `overall` вы получаете новый счетчик.

Имея такую функцию, можно использовать ее следующим образом:

```
let myCounter = counter()
myCounter() → 1
myCounter() → 2
myCounter() → 3
```

Таким образом, вы можете независимо работать с любым количеством счетчиков!



### Мозговой Штурм

Создайте другой счетчик:

```
let secondCounter = counter()
```

Несколько раз увеличьте исходный счетчик:

```
myCounter()
```

```
myCounter()
```

```
myCounter()
```

Что произойдет, если теперь увеличить второй счетчик?

А вы уверены, что замыкания надежны? Я слышала, что они могут уйти из функции! И как их потом ловить?



**Замыкание может выйти за пределы функции, которой оно было передано.**

Если замыкание существует дольше функции, которой оно вызывается, такое замыкание называется *выходящим замыканием*. Оно вышло за пределы функции, смысл понятен?

Проще говоря: если замыкание вызывается после возврата из функции, которой оно было передано, то оно называется *выходящим замыканием*.

Чтобы пометить замыкание как выходящее, поставьте ключевое слово `@escaping` перед типом параметра. Оно сообщает Swift, что замыкание может выйти за границы функции.

Выходящие замыкания чаще всего появляются при сохранении в переменной, определенной вне области видимости функции.

Даже если вы сохраните замыкание в переменной, определенной за пределами функции, параметр все равно необходимо снабдить ключевым словом `@escaping`.

### Что появилось сначала, функция или замыкание?

Иногда бывает трудно определить, в каком порядке что-либо происходит, особенно при участии замыканий (и еще труднее — с выходящими замыканиями).

Пора прибегнуть к помощи нашего лучшего друга — хитроумного примера! ✨

В этом хитроумном примере участвует массив замыканий и функция, которая в своем единственном параметре получает выходящее замыкание типа `() -> Void`.

При вызове функция выводит сообщение о своем вызове, присоединяет переданное замыкание к массиву замыканий, выполняет замыкание и возвращает управление.

Тогда при вызове функции передается замыкание. в данном случае это будет завершающее замыкание, которое выводит сообщение о вызове замыкания.

Совсем запутались? Давайте разберемся в происходящем более подробно.

## Выходящие замыкания: нетривиальный пример

### 1 Массив замыканий

`var closures: [() -> ()] = []` ← Замыкания могут храниться в массиве.

Чтобы пометить замыкание как выходящее, используется ключевое слово @escaping.

### 2 Функция

```
func callEscaping(closure: @escaping () -> Void) {
 print("callEscaping() function called!")
 closures.append(closure)
 closure()
 return
}
```

Замыкание, переданное функции, сохраняется в массиве, определенном за границами функции.

Из-за этого замыкание должно выйти из функции.

### 2 Вызов функции

```
callEscaping {
 print("closure called")
}
```

← Передается функции `callEscaping` с использованием синтаксиса завершающего замыкания.

Замыкание не делает ничего, а только сообщает о том, что оно было вызвано.

## Возьмите в руку карандаш

Допустим, вы ввели приведенную выше программу Swift с выходящим замыканием и всем прочим в среде Playground. Как вы думаете, какой результат она выведет?

**A**

```
callEscaping() function was called!
closure called!
```

**B**

```
closure called!
callEscaping() function was called!
```

Как вы думаете, почему результат будет именно таким?

Подсказка: в замыканиях нет ничего сверхъестественного, и если вы терпеливо и аккуратно отследите путь выполнения программы, вы получите правильный ответ. На следующей странице мы обсудим, что здесь происходит.

Когда мои коллеги готовят, они иногда забывают, что делать дальше. Надеюсь, замыкания мне с этим помогут...



**Замыкания прекрасно подходят для обработчиков завершения.**

Обработчиком завершения называется код (обычно функция или замыкание), который вызывается после того, как другой код (обычно тоже функция или замыкание) завершит выполнение.

Итак, если у вас имеется функция `cookPizza` и вы хотите, чтобы приготовленная пицца была передана на доставку клиенту, это можно сделать примерно так:

```
func cookPizza(completion: () -> ()) {
 print("The pizza is cooking!")
 print("The pizza is cooked!")
 completion()
 print("Pizza cooked & everything is done.")
}
```

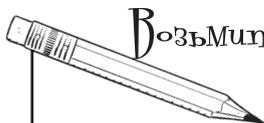
Затем определяется замыкание, которое служит обработчиком завершения (пицца подается на стол клиента):

```
var servePizza = {
 print("Delivered pizza to the customer!")
}
```

После этого вы сможете элементарно изменить действие, которое должно выполняться после приготовления пиццы (чтобы, допустим, пицца была упакована для доставки вместо подачи на стол):

```
cookPizza(completion: servePizza)
```

← Также можно воспользоваться синтаксисом завершающего замыкания, чтобы сделать что-то совершенно новое!



**Возьмите В руку карандаш**

**Решение**

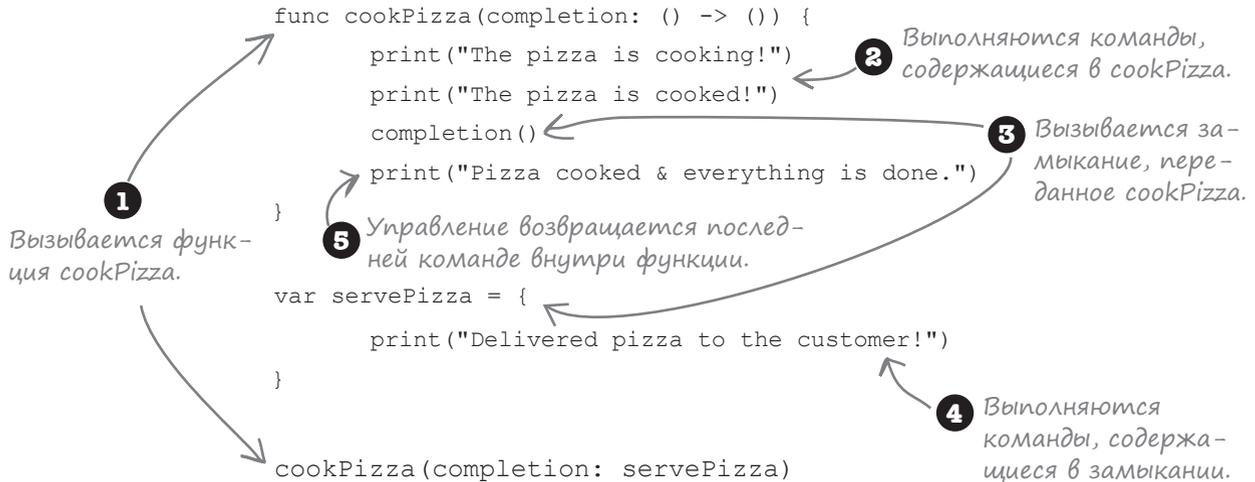
Правильный ответ — А. Сначала свое сообщение выводит функция, а потом замыкание. На первый взгляд это кажется тривиальным, но во внутренней реализации в Swift происходит много всего, что обеспечивает достижение правильного результата.

Так как массив замыканий `closures` был создан за пределами функции `callEscaping`, замыкание внутри `callEscaping` должно выйти за пределы функции, чтобы быть сохраненным в массиве.



За сценой

Во внутренней реализации последовательность выполнения обработчика завершения выглядит примерно так:



Будьте осторожны!

**Замыкания являются ссылочными типами.**

*Важно подчеркнуть, что замыкания являются ссылочными типами.*

*Таким образом, если вы создаете замыкание:*

```

var myClosure = {
 print("Amazing!")
}

```

*и присваиваете его другой переменной:*

```

var coolClosure = myClosure

```

*то coolClosure и myClosure будут ссылаться на одно и то же замыкание.*

## Автозамыкания обеспечивают гибкость

### Ситуация

Имеется функция, используемая для сервировки пиццы. Функция получает два параметра. Bool определяет, нужно ли упаковать пиццу в коробку, а String содержит название пиццы:

```
func servePizza(box: Bool, pizza: String) {
 if box {
 print("Boxing the pizza '\(pizza)'")
 } else {
 print("We're all done.")
 }
}
```

Схема довольно тривиальна, а функция вызывается просто:

```
servePizza(box: true, pizza: "Hawaiian")
```

Также имеется другая функция, представляющая произвольную пиццу, которая возвращает String:

```
func nextPizza() -> String {
 return "Hawaiian"
}
```

Эта функция может использоваться при вызове servePizza:

```
servePizza(box: true, pizza: nextPizza())
```

Но что произойдет, если пицца не упаковывается в коробку, а ваш вызов выглядит так?

```
servePizza(box: false, pizza: nextPizza())
```

### Проблема

Проблема в том, что, хотя вызов функции nextPizza не нужен, как в этом примере, функция все равно будет вызвана. Это приводит к неэффективному расходованию ресурсов. Проблему можно решить при помощи замыканий. Если обновить определение функции:

```
func servePizza(box: Bool, pizza: () -> String)
```

то этот вызов servePizza() компилироваться вообще не будет, потому что параметр pizza может быть только замыканием, возвращающим строку (а вызов nextPizza() вернет String):

```
servePizza(), pizza, nextPizza(), String ← Не компилируется!
```

...и вполне очевидно, что мы не сможем просто передать String.

```
String
```

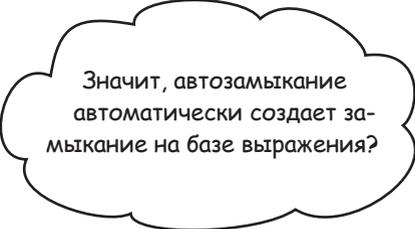
## Решение

Проблема решается при помощи автозамыкания! Добавив атрибут `@autoclosure` в параметр, можно полностью избежать возникновения этой проблемы. Обновим определение функции:

```
func servePizza(box: Bool, pizza: @autoclosure () -> String)
```

Теперь передавать можно все что угодно — при условии, что результатом будет String:

```
servePizza(box: true, pizza: nextPizza())
servePizza(box: false, pizza: "Vegetarian")
servePizza(box: true, pizza: "Meaty Meat Surprise")
```



Значит, автозамыкание автоматически создает замыкание на базе выражения?

**Вот именно. Автозамыкание преобразует выражение в замыкание.**

Код, написанный вами для выражения, не является замыканием, но он превращается в замыкание. Это обеспечивает гибкость при использовании выражений.



Будьте осторожны!

**Вряд ли вам придется регулярно пользоваться автозамыканиями.**

Автозамыкания — чрезвычайно мощный механизм. Они эффективно работают при написании кода, который будет использоваться другими людьми, но при злоупотреблениях они усложнят чтение вашего кода. Используйте автозамыкания осмотрительно!

Хорошо. Вы меня убедили. Что еще можно делать при помощи замыканий?



**Замыкания играют важную роль в сетевом программировании, пользовательских интерфейсах и реализации сложного поведения, требующего временных привязок.**

Пользовательские интерфейсы будут рассмотрены в ближайших главах, а пока важно разобраться с принципами использования замыканий.

Последнее, что можно сделать с замыканиями (по крайней мере последнее, что мы сделаем в этой главе), — применить их для сортировки коллекций.

Если у вас имеется массив чисел:

```
var numbers = [1, 5, 2, 3, 7, 4, 6, 9, 8]
```

...вы можете воспользоваться замыканием для того, чтобы отсортировать его!

Типы коллекций Swift содержат метод `sort (by:)`, которому передается замыкание. Так как операторы Swift в действительности являются функциями, при желании методу можно передать `<` или `>`:

```
numbers.sort (by: <)
numbers.sort (by: >)
```

Впрочем, `sort (by:)` также можно передать собственное замыкание, определенное на месте:

```
numbers.sort (by: { a, b in
 return a < b
})
```

Делают одно и то же.



### Мозговой Штурм

Создайте новое замыкание для сортировки массива `numbers`. Определите его в следующем виде:

Разместите здесь свою логику сортировки. Она должна возвращать логическое значение.

```
let sortClosure = { (a: Int, b: Int) -> Bool in
}
```

Рассмотрите разные варианты поведения, сортируя массив `numbers` array (или другие числовые массивы) с собственным замыканием сортировки. Замыкание можно передавать `sort (by:)` в следующем виде (вместо использования выражения замыкания, как делалось ранее):

```
numbers.sort (by: sortClosure)
```

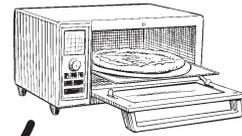
## Сокращенные имена аргументов

Когда вы работаете со встроенной записью замыканий, можно опустить имена аргументов и ссылаться на значения по обозначениям \$0, \$1, \$2, \$3 и т. д. Таким образом, для нашего примера с сортировкой массива с предыдущей страницы можно передать собственное заключение во встроенной записи с сокращенными обозначениями аргументов:

```
numbers.sort (by: { $0 < $1 })
```

Также можно опустить  
ключевое слово *in*.

### ГОТОВЫЙ КОД



## Кто и Что Делает ?

Соедините каждый термин, относящийся к замыканиям, с его описанием. Все эти термины вам уже знакомы; если у вас возникнут затруднения, обратитесь к решению или просто вернитесь на несколько страниц назад.

|                              |                                                                                                                                  |
|------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| Автозамыкание                | Автономный блок кода, который может передаваться в параметрах.                                                                   |
| Обработчик завершения        | Передача функции в аргументе после круглых скобок вызова функции, если замыкание является последним или единственным параметром. |
| Тип функции                  | Код (обычно замыкание), который вызывается после завершения чего-либо (обычно функции).                                          |
| Выражение замыкания          | Константа или переменная, сохраненная замыканием из окружающего контекста.                                                       |
| Сохраненное значение         | Возможность ссылаться на первый, второй и т. д. аргументы во встроенных замыканиях по обозначениям вида \$0, \$1 и т. д.         |
| Выходящее замыкание          | Определение типа для функции или замыкания с типами параметров и возвращаемым типом.                                             |
| Завершающее замыкание        | Простой синтаксис для встроенной записи замыканий.                                                                               |
| Сокращенные имена аргументов | Замыкание, которое вызывается (или продолжает существовать) после возвращения из функции.                                        |
| Цепный возврат               | Замыкание, автоматически создаваемое на базе выражения.                                                                          |
| Замыкание                    | Отсутствие ключевого слова <code>return</code> в замыканиях, состоящих из одного выражения.                                      |
| Замыкание без выхода         | Замыкание, которое не вызывается за пределами той функции, в которой оно используется.                                           |

→ Ответ на с. 180.

## СТАТЬ КОМПИЛЯТОРОМ Swift



Каждый фрагмент кода Swift на этой странице отражает некоторый аспект замыканий. Представьте себя на месте компилятора Swift и определите, какие из этих фрагментов допустимы.

A

```
var createPizza(for name: String) = {
 print("This pizza is for: \(name)")
}
createPizza(for: "Bob")
```

C

```
let cookPizza = {
 print("The pizza is cooking!")
}
cookPizza()
```

E

```
eatPizza = {
 print("Now eating!")
}
```

G

```
eatPizza: String = {
 print("Now eating!")
}
```

I

```
let garlicBread() = {
 print("Making garlic bread!")
}
upgrade()
```

B

```
var deliverPizza() {
 print("The pizza is delivered!")
}
```

D

```
let boxPizza = {
 print("The pizza is in the box!")
}
boxPizza()
```

F

```
let pizzaHawaiian {
 print("Delicious pineapple on it!")
}
pizzaHawaiian()
```

H

```
var slicePizza = {
 print("Slicing into 8 pieces.")
}
slicePizza()
```

→ Ответ на с. 180.

# Кто и что делает? Решение

С. 178

|                              |                                                                                                                                  |
|------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| Автозамыкание                | Автономный блок кода, который может передаваться в параметрах.                                                                   |
| Обработчик завершения        | Передача функции в аргументе после круглых скобок вызова функции, если замыкание является последним или единственным параметром. |
| Тип функции                  | Код (обычно замыкание), который вызывается после завершения чего-либо (обычно функции).                                          |
| Выражение замыкания          | Константа или переменная, сохраненная замыканием из окружающего контекста.                                                       |
| Сохраненное значение         | Возможность ссылаться на первый, второй и т. д. аргументы во встроенных замыканиях по обозначениям вида \$0, \$1 и т. д.         |
| Выходное замыкание           | Определение типа для функции или замыкания с типами параметров и возвращаемым типом.                                             |
| Завершающее замыкание        | Простой синтаксис для встроенной записи замыканий.                                                                               |
| Сокращенные имена аргументов | Замыкание, которое вызывается (или продолжает существовать) после возвращения из функции.                                        |
| Цепный возврат               | Замыкание, автоматически создаваемое на базе выражения.                                                                          |
| Замыкание                    | Отсутствие ключевого слова return в замыканиях, состоящих из одного выражения.                                                   |
| Замыкание без выхода         | Замыкание, которое не вызывается за пределами той функции, в которой оно используется.                                           |

# СТАНЬ компилятором Swift. Решение

С. 179



А не работает, В не работает, С работает,  
 D работает, E не работает, F не работает,  
 G не работает, H работает, I не работает.

## 6. Структуры, свойства и Методы

# Типы, определяемые пользователем, и не только

Без типов, разработанных специально для нас, далеко не уйдешь! Никто не будет воспринимать вас серьезно, если у вас не будет чего-то своего — и только своего.



При работе с данными часто требуется определять собственные виды данных. Структуры, также нередко обозначаемые ключевым словом Swift **struct**, позволяют создавать **типы данных, определяемые пользователем** (подобно тому, как `String` и `Int` являются типами данных), посредством **объединения других типов**. Использование структур для представления данных, с которыми работает ваш код Swift, позволяет отступить на шаг и подумать над взаимодействием данных, передаваемых в вашем коде. **Структуры могут содержать переменные и константы** (внутри структур они называются **свойствами**) и **функции** (называемые **методами**). Добавим немного порядка в ваш мир и займемся углубленным изучением структур.



Мы выпекаем все больше пиццы!  
Нельзя ли создать какой-нибудь удобный тип Pizza, который определяет пиццу, и использовать его вместо String?

Как и String, Int и т. д.

**Вы можете создавать в Swift собственные типы данных.**

В Swift для этого чаще всего используются *структуры*. Вас этот факт может удивить, если вы привыкли использовать другой язык, в котором люди не мыслят жизни без классов.

Структуры предоставляют феноменальные возможности по включению *переменных, констант и функций* в выбранный вами *тип*!

Тип Pizza, определяемый как структура:

```
struct Pizza {
 var name: String
}
```

↑ Переменные, находящиеся внутри структур, называются свойствами.

После того как вы определили структуру Pizza, вы можете создать *экземпляр* Pizza:

```
var myPizza = Pizza(name: "Hawaiian")
```

а потом обратиться к его *свойству* name (String) и вывести его:

```
print(myPizza.name)
```

Так как свойство name является переменной, с ним можно выполнять те же операции, что и с любыми другими переменными:

↪ Свойство name теперь содержит значение «Meatlovers».

```
myPizza.name = "Meatlovers"
print(myPizza.name)
```

↪ На самом деле вы не изменяете свойство, а копируете всю структуру с созданием нового значения. Но в итоге создается впечатление, что структура изменяется.

## Сделаем пиццу во всей красе...

Шеф-повар на предыдущей странице отметил очень важный момент: будет очень полезно иметь **тип** для пиццы, чтобы нам не пришлось, как и прежде, использовать строки для ее представления.

**Структура представляет коллекцию свойств и методов.** По стечению обстоятельств пиццу также можно представить в виде коллекции свойств и методов.

Какие свойства и методы необходимы для представления пиццы? Попробуем рассматривать пиццу как концепцию, а не как конкретную пиццу, присоединенную к чьему-то заказу (таким образом, размер пиццы в данный момент роли не играет).

### Структура Pizza



```
struct Pizza {
 var name: String
 var ingredients: [String] = []
}
```

У пиццы есть название — например, «Hawaiian».

Пицца состоит из нескольких ингредиентов.

Переменные `name` и `ingredients` называются хранимыми свойствами, потому что они являются свойствами сущности, которая представляется структурой. Также существует другая разновидность свойств — так называемые вычисляемые свойства, которые выполняют некоторый код для получения своего значения (вместо того чтобы просто прочитать хранимое значение).

### Возьмите в руку карандаш

Сможете ли вы представить, как создать экземпляр новой структуры Pizza? На предыдущей странице мы создали структуру Pizza, единственным свойством которой было название пиццы, но новая версия также получает массив строк с ингредиентами.

Как должна выглядеть команда для инициализации нового экземпляра структуры Pizza, представляющей гавайскую пиццу (с ингредиентами Cheese, Pineapple, Ham и Pizza Sauce)?

А как насчет вегетарианской специальной пиццы (Cheese, Avocado, Sundried Tomato, Basil)?

Или пиццы с курицей барбекю (Chicken, Cheese, BBQ Sauce, Pineapple)?

→ Ответ на с. 185.



Погодите-ка... А откуда Swift знает, как инициализировать Pizza? Передача значений свойств структуры мне очень напоминает вызов функции...

Инициализатор позволяет создавать новые экземпляры чего-либо.

К структурам Swift прилагается синтезированный поэлементный инициализатор.

**Поэлементный инициализатор** — особая разновидность функций, существующих внутри структур. Он называется инициализатором, потому что используется для инициализации нового экземпляра этой структуры.

Поэлементный инициализатор, который используется для создания экземпляра структуры, в действительности *синтезирует* за вас. Он получает значения всех свойств структуры, которые также становятся метками параметров.

Таким образом, если у вас имеется структура Pizza с тремя свойствами:

```
struct Pizza {
 var name: String
 var ingredients: [String]
 var dessertPizza: Bool
```

Термин «синтезированный» в данном случае означает только то, что он создается за вас автоматически.

Инициализатор *требует* передачи трех параметров — name, ingredients и dessertPizza:

```
var rockyRoadPizza = Pizza(name: "Rocky Road",
 ingredients: ["Marshmallows",
 "Peanuts", "Chocolate",
 "Sugar"],
 dessertPizza: true)
```

Но если вы предоставите **значение по умолчанию** для каких-либо свойств, синтезированный инициализатор также знает о нем, поэтому вы можете решать, нужно ли указывать его значение:

```
struct Pizza {
 var name: String
 var ingredients: [String]
 var dessertPizza: Bool = false
}
```

Мы предоставили значение по умолчанию false для логического параметра dessertPizza (десертные пиццы встречаются относительно редко, поэтому в типичном случае это сокращает объем текста, вводимого при создании пиццы).

А значит, новый экземпляр можно создавать с передачей параметров или без них:

```
var cheesePizza = Pizza(name: "Cheese", ingredients: ["Cheese"])
var candyPizza = Pizza(name: "Candy",
 ingredients: ["Gummie Bears", "Nerds"],
 dessertPizza: true)
```

## Ключевые моменты

- Структуры — конструкции, позволяющие создавать собственные типы в Swift.
- В структуры можно включать переменные и константы других типов. Если переменная или константа находится внутри структуры, она называется свойством (потому что это свойство фрагмента данных, который представляется структурой).
- К структуре прилагается автоматически синтезированный поэлементный инициализатор, который позволяет создать экземпляр структуры, получая значения всех свойств структуры.
- Свойство может иметь значение по умолчанию, указанное в определении структуры. Такие свойства не обязательно передавать автоматически синтезированному инициализатору.

## Возьмите в руку карандаш

### Решение

С. 183

Чтобы создать экземпляр гавайской пиццы (с ингредиентами Cheese, Pineapple, Ham и Pizza Sauce) с использованием нашей структуры, необходимо вызвать автоматически сгенерированный инициализатор и передать значения соответствующих типов для всех именованных свойств. При создании структуры Pizza необходимо передать строку с названием и массив строк с ингредиентами.

```
var hawaiianPizza =
 Pizza(name: "Hawaiian",
 ingredients: ["Cheese", "Pineapple", "Ham", "Pizza Sauce"])
```

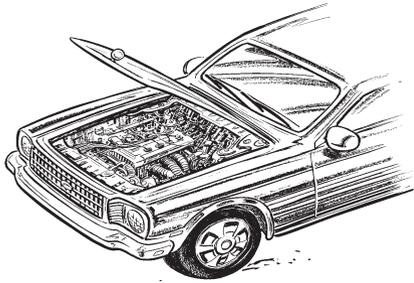
А вот как создается экземпляр вегетарианской специальной пиццы:

```
var vegetarianSpecialPizza =
 Pizza(name: "Vegetarian Special",
 ingredients: ["Cheese", "Avocado", "Sundried Tomato", "Basil"])
```

И наконец, экземпляр пиццы с курицей барбекю:

```
var bbqChickenPizza =
 Pizza(name: "BBQ Chicken",
 ingredients: ["Chicken", "Cheese", "BBQ Sauce", "Pineapple"])
```

## Пог канотом...



В Swift типы делятся на типы-значения и ссылочные типы.



Структуры являются **типами-значениями**. Это означает, что если вы присваиваете структуру другой переменной или константе, вы **создаете копию текущего значения**, а *не* копируете ссылку на оригинал.

Допустим, вы берете десертную пиццу с предыдущей страницы:

```
var rockyRoadPizza =
 Pizza(name: "Rocky Road",
 ingredients: ["Marshmallows", "Peanuts",
 "Chocolate", "Sugar"],
 dessertPizza: true)
```

И *присваиваете* ее новой переменной:

```
var anotherRockyRoadPizza = rockyRoadPizza
```

Затем эта переменная изменяется:

```
anotherRockyRoadPizza.name = "Fury Road"
```

В результате вы получите *два разных экземпляра Pizza*: в одном свойство name содержит строку Fury Road, а в другом — строку Rocky Road.

В этом можно убедиться при помощи команды print:

```
print(rockyRoadPizza.name)
print(anotherRockyRoadPizza.name)
```

Структуры являются типами-значениями. К этой категории относятся типы, экземпляры которых копируются при присваивании.



Будьте осторожны!

У типов-значений каждый экземпляр содержит уникальную копию хранящихся в нем данных.

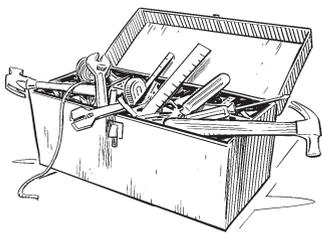
Тип-значение копирует свои данные при присваивании, инициализации или передаче аргументов. Большинство встроенных типов данных Swift относится к категории типов-значений (в большинстве случаев это побочный эффект их внутренней реализации в виде структур): Array, String, Dictionary, Int и многие другие типы реализуются в виде структур, и все они являются типами-значениями.

Если тип относится к категории ссылочных типов, это означает, что существует один (неявно) общий экземпляр данных и несколько переменных могут ссылаться на один экземпляр данных. Вы еще не сталкивались со ссылочными типами, но вскоре познакомитесь с ними. Структуры всегда являются типами-значениями.

А может, и не только мечты!

Если бы только я могла создать собственный инициализатор. Было бы очень удобно настроить типы данных структуры так, как мне нужно... Как жаль, что это только мечты.





Самоделные инициализаторы

Используется такой же синтаксис, как при создании функции. Так как созданному нами инициализатору не обязательно передавать какие-либо параметры, мы их не передаем.

**На самом деле создать собственный инициализатор совсем несложно.**

Синтезированный инициализатор можно заменить собственным. Это очень похоже на создание функции, но эта функция не имеет имени и начинается с ключевого слова `init` вместо ключевого слова `func`.

Таким образом, чтобы для всех пицц в момент создания по умолчанию выбиралась разновидность `Cheese`, это можно сделать так:

```

struct Pizza {
 var name: String
 var ingredients: [String]
 var dessertPizza: Bool

 init() {
 name = "Cheese"
 ingredients = ["Cheese"]
 dessertPizza = false
 }
}

```

Ваши инициализаторы должны позаботиться о том, чтобы к концу инициализатора каждому свойству структуры было присвоено значение.

Тогда новый экземпляр `Pizza` можно создать без передачи каких-либо значений:

```

var pizza = Pizza()

```

Из-за инициализатора (который предоставляет значение для каждого свойства) любой созданный экземпляр `Pizza` не получает (и не может получить) параметры, и всегда будет создаваться сырная пицца (`Cheese`).



Будьте осторожны!

**Если вы создаете собственный инициализатор, то синтезированный инициализатор становится недоступным.**

Инициализатор, который вы создаете, заменяет синтезированный инициализатор. Это означает, что вы уже не сможете создать экземпляр `Pizza` с указанием названия, списка ингредиентов и признака десертной пиццы. Все пиццы создаются с типом `Cheese` без какого-либо выбора.

Конечно, так как свойства `Pizza` являются переменными, их значения можно обновить позднее:

```

pizza.name = "Margherita"
pizza.ingredients.append("Tomato")

```

## Инициализатор ведет себя точно так же, как функция

В инициализаторе можно делать большую часть того, что можно делать в функциях. Например, можно получать параметры и использовать их для задания значений свойств.

Этот инициализатор повторяет функциональность синтезированного инициализатора, который мы использовали ранее:

```
struct Pizza {
 var name: String
 var ingredients: [String]
 var dessertPizza: Bool

 init(name: String, ingredients: [String], dessertPizza: Bool) {
 self.name = name
 self.ingredients = ingredients
 self.dessertPizza = dessertPizza
 }
}
```



### Мозговой Штурм

Шеф-повар подумывает о том, чтобы расширить ассортимент пиццерии и включить в него чесночный хлеб. Она следит за вашими успехами в программировании на языке Swift, а также мечтает реализовать структуру для представления нового продукта:

```
struct GarlicBread {
 var strength: Int
 var vegan: Bool
}
```

Пользуясь своими знаниями инициализаторов и структур, напишите собственный инициализатор для структуры `GarlicBread`, который присваивает логическому свойству `vegan` значение `false` (шеф-повар еще не готова выпекать веганский чесночный хлеб, но планирует делать это в будущем) и получает интенсивность аромата в виде значения `strength`.

Шеф-повар также хотела бы, чтобы инициализатор выводил сообщение вида «New Garlic Bread of strength x created!» каждый раз, когда инициализируется экземпляр `GarlicBread`.



Но, конечно, я могу  
создать более одного инициализатора,  
верно? Я хочу иметь возможность создать  
экземпляр Pizza любым удобным для меня  
способом.

**Инициализаторов может быть сколько угодно.**

Вы можете добавить новые инициализаторы, которые делают то, что вам нужно. Обновите структуру Pizza следующей версией, чтобы получить исходную функциональность синтезированного инициализатора, а также функциональность первого инициализатора, намеренно написанного нами:

```
struct Pizza {
 var name: String
 var ingredients: [String]
 var dessertPizza: Bool

 init() {
 name = "Cheese"
 ingredients = ["Cheese"]
 dessertPizza = false
 }

 init(name: String, ingredients: [String], dessertPizza: Bool) {
 self.name = name
 self.ingredients = ingredients
 self.dessertPizza = dessertPizza
 }
}
```

**Единственное,  
что требуется  
от инициализатора —  
чтобы до его завершения  
каждому свойству было  
присвоено значение.**



## Мозговой Штурм

Вернемся к структуре `GarlicBread`, с которой мы работали ранее:

```
struct GarlicBread {
 var strength: Int
 var vegan: Bool
}
```

Шеф-повар хочет обновить структуру и включить в нее целое число, представляющее остроту чесночного хлеба. При этом она желает иметь возможность инициализировать экземпляры `GarlicBread` любым из следующих способов:

- ✦ Без параметров (интенсивность 1, острота 0, невеганский)
- ✦ Только с параметром интенсивности (острота 0, невеганский)
- ✦ С передачей параметров интенсивности и остроты (невеганский)
- ✦ С передачей параметра интенсивности, параметра остроты и статуса веганского рецепта

Напишите все инициализаторы `GarlicBread`, необходимые для поддержки этих способов инициализации.

## Ключевые Моменты

- Для создания собственных типов в Swift обычно используются структуры.
- Структура позволяет объединять переменные, константы и функции в ваш собственный тип. Переменные и константы в структуре называются свойствами, а функции в структуре называются методами.
- Структура автоматически синтезирует за вас инициализатор, называемый поэлементным инициализатором. Он позволяет задать значения всех свойств структуры при создании экземпляра.
- Структуры в Swift являются типами-значениями. Это означает, что, присваивая экземпляр такого типа другой переменной, вы создаете его новую копию, в которой хранятся те же данные.
- Вместо того чтобы полагаться на синтезированный поэлементный инициализатор, вы можете создать сколько угодно собственных инициализаторов — при условии, что к концу каждого из ваших инициализаторов каждому свойству структуры будет присвоено значение.
- После того как вы напишете собственный инициализатор для структуры, синтезированный инициализатор становится недоступен.



## Статические свойства повышают гибкость структур

При создании структуры свойство можно объявить **статическим** при помощи ключевого слова `static`. Тем самым вы сообщаете Swift, что **значение** этого свойства **совместно используется всеми** экземплярами этой конкретной структуры.

Например, если вы хотите добавить счетчик инициализируемых экземпляров типа `Pizza`, это можно сделать так:

```
struct Pizza {
 var name: String
 var ingredients: [String]
 var dessertPizza: Bool
 static var count = 0
```

```
 init() {
 name = "Cheese"
 ingredients = ["Cheese"]
 dessertPizza = false
 Pizza.count += 1
 }
```

```
 init(name: String, ingredients: [String], dessertPizza: Bool) {
 self.name = name
 self.ingredients = ingredients
 self.dessertPizza = dessertPizza
 Pizza.count += 1
 }
}
```

```
var pizza1 = Pizza()
var pizza2 = Pizza()
print(Pizza.count)
```

Статическое свойство объявляется ключевым словом `static`. Свойство содержит количество инициализированных экземпляров `Pizza`.

В инициализаторе статическое свойство увеличивается. Чтобы получить доступ к свойству `count`, следует сначала указать имя структуры.

Чтобы счетчик точно представлял количество экземпляров, счетчик должен увеличиваться в обоих возможных инициализаторах.

Свойство `count` принадлежит самой структуре, а не ее отдельным экземплярам, поэтому мы обращаемся к нему по имени структуры `Pizza`.



## Мозговой Штурм

Какие из следующих примеров статических свойств правильные? Если они неправильные, то почему? Попробуйте разобраться в этом самостоятельно.

- ```
struct Car {
    static let maxSpeed = 150
    var color: String
}

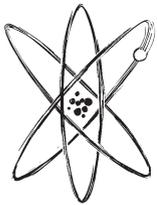
```
- ```
struct Job {
 var title: String
 var location: String
 static salary = 60000
}

```
- ```
struct Spaceship {
    static let ships = [Spaceship]()
    init() {
        Spaceship.ships.append(self)
    }
    static func testEngines() {
        for _ in ships {
            print("Testing engine!")
        }
    }
}

```
- ```
struct Cactus {
 static var cactuses = 0
 var type: String
 init(cactusType: String) {
 type = cactusType
 cactuses += 1
 }
}

```

## Управление доступом и структуры



Когда вы создаете структуру, по умолчанию ничто не мешает вам напрямую обращаться к любым свойствам, содержащимся в структуре.

Если структура `Pizza` выглядит так:

```
struct Pizza {
 var name: String
 var chefsNotes: String
}
```

После того как вы создали экземпляр `Pizza`:

```
var hawaiian = Pizza(name: "Hawaiian",
 chefsNotes: "A tasty pizza, but pineapple
 isn't for everyone!")
```

ничто не помешает вам обратиться к свойству `chefsNotes` экземпляра структуры и сделать с ним все что угодно:

```
print(hawaiian.chefsNotes)
```

Механизм управления доступом предоставляет возможность ограничить доступ к свойству или методу внутри ваших структур.

Если применить ключевое слово **private** к свойству (или методу), к нему нельзя будет обратиться за пределами структуры.

Таким образом, если обновить определение `chefsNotes`:

```
private var chefsNotes: String
```

...все должно работать, не считая того, что к свойству `chefsNotes` невозможно будет обратиться за пределами структуры. Однако добавление ключевого слова `private` означает, что `chefsNotes` не будет частью автоматически синтезированного инициализатора, поэтому вам придется создать собственный инициализатор.

'Pizza' initializer is inaccessible due to 'private' protection level



## Мозговой Штурм

Воспользуйтесь своими знаниями управления доступом, структур и создания собственных инициализаторов и завершите приведенный ниже фрагмент кода.

Из-за ограничения доступа к свойству `chefsNotes` необходимо добавить инициализатор, который позволяет определить заметки шеф-повара при создании экземпляра структуры `Pizza`.

```
struct Pizza {
 var name: String
 private var chefsNotes: String

 init(_____) {
 _____ = _____
 _____ = _____
 }
}
```

После того как инициализатор будет определен, создайте несколько новых экземпляров `Pizza` и протестируйте возможность (или невозможность) обратиться к свойству `chefsNotes`.

```
print(hawaiian.chefsNotes)
```

'chefsNotes' is inaccessible due to 'private' protection level

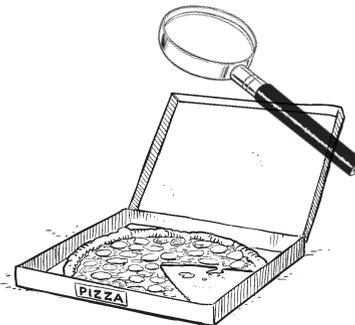
## Ключевые Моменты

- Структуры являются типами-значениями. Это говорит о том, что значение структуры копируется при присваивании, инициализации или передаче аргумента.
- Большинство встроенных типов Swift (`String`, `Array` и т. д.) также являются типами-значениями.
- Любой тип, который создается как структура, также является типом-значением.
- Вы можете создать собственные инициализаторы для типов, создаваемых как структуры, при помощи ключевого слова `init`.
- Инициализаторов может быть сколько угодно. Единственное требование — чтобы к моменту завершения инициализатора каждому свойству структуры было присвоено значение.
- Если вы решили создать собственный инициализатор, то синтезированный инициализатор становится недоступным.
- Чтобы пометить свойство как статическое, используйте ключевое слово `static`. Значение такого свойства совместно используется всеми экземплярами структуры.
- Свойство также можно пометить ключевым словом `private`. Это означает, что к нему невозможно будет обратиться за пределами указанной структуры.

## Функции в структурах Методы

Функции – очень полезная штука; впрочем, вы это уже знаете. Но еще полезнее размещать функции внутри структур. Функция, находящаяся внутри структуры, называется **методом**.

Методы объявляются тем же ключевым словом **func**, что и функции, и в остальном от них ничем не отличаются. Просто *когда функция находится внутри структуры, она называется методом*.



Функции представляют собой отдельные блоки кода. Функции, находящиеся внутри чего-либо, называются методами того, в чем они находятся.

### Возьмите в руку карандаш

Вспомните все, что вы знаете о функциях, и примените эти знания к методам структуры Pizza.

Возьмите простую структуру Pizza и включите в нее метод с именем `getPrice`, который не получает параметров и возвращает общую стоимость пиццы в виде целого числа.

```
struct Pizza {
 var name: String
 var ingredients: [String]
```

---

---

---

---

---

---

---

---

---

---

```
}
```



Не люблю запоминать лишние цифры... Каждый ингредиент моей пиццы стоит \$2. Удобно, правда?

Когда метод будет реализован, создайте несколько экземпляров Pizza и протестируйте его с этими экземплярами:

```
var hawaiian = Pizza(name: "Hawaiian", ingredients: ["Ham", "Cheese", "Pineapple"])
var meat = Pizza(name: "Meaty Goodness",
 ingredients: ["Pepperoni", "Chicken", "Ham", "Tomato", "Pulled Pork"])
var cheese = Pizza(name: "Cheese", ingredients: ["Cheese"])
```

→ Ответ на с. 198.

## Изменение свойств с использованием методов

В этом проявляется стремление Swift к безопасности.



Если вы захотите создать метод, изменяющий свойство внутри структуры, вам придется проделать немного дополнительной работы.

Возьмем следующую структуру Pizza:

```
struct Pizza {
 var name: String
 var ingredients: [String]
}
```

Если вы хотите написать метод для обновления свойства name, казалось бы, для этого достаточно просто включить его прямо в структуру:

```
func setName(newName: String) {
 self.name = newName
}
```

Но если вы попытаетесь это сделать, окажется, что метод не работает. Вы получите сообщение об ошибке.

Mark method 'mutating' to make 'self' mutable

В сообщении точно сказано, что необходимо сделать: пометить метод как изменяющий (**mutating**), чтобы разрешить изменение данных в области видимости self.

Обновите код метода:

```
mutating func setName(newName: String) {
 self.name = newName
}
```

Изменяющий метод не может вызывать неизменяющий метод. Впрочем, он может вызывать другие изменяющие методы.

И опробуйте новую версию метода setName на практике:

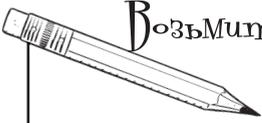
```
var hawaiian = Pizza(name: "Hawaiian", ingredients: ["Pineapple", "Ham", "Cheese"])
hawaiian.setName(newName: "Pineapple Abomination")
```



Будьте осторожны!

Одного использования ключевого слова **mutating** достаточно, чтобы Swift не позволил вам использовать этот метод со структурами-константами.

Swift верит вам на слово, что это изменяющий метод. Таким образом, даже если метод вообще не обращается к свойствам, вы не сможете вызвать его для структуры, которая была создана как константа.



Возьмите в руку карандаш

Решение

С. 196

```
struct Pizza {
 var name: String
 var ingredients: [String]

 func getPrice() -> Int {
 return ingredients.count * 2
 }
}

var hawaiian = Pizza(name: "Hawaiian", ingredients: ["Ham", "Cheese",
 "Pineapple"])
var meat = Pizza(name: "Meaty Goodness", ingredients: ["Pepperoni",
 "Chicken", "Ham", "Tomato", "Pulled Pork"])
var cheese = Pizza(name: "Cheese", ingredients: ["Cheese"])

var hawaiianPrice = hawaiian.getPrice()
var meatPrice = meat.getPrice()
var cheesePrice = cheese.getPrice()

print("The hawaiian costs \(hawaiianPrice), the meat costs \(meatPrice),
and the cheese costs \(cheesePrice)")
```

## Вычисляемые свойства

Все свойства, создававшиеся нами ранее, были **хранимыми свойствами**, потому что их значения хранились в структуре.

Другую разновидность свойств составляют **вычисляемые свойства**: *вычисляемые свойства могут выполнять код для определения своего значения.*



Снова скажу: я не программист, но, похоже, мы можем воспользоваться этими вычисляемыми свойствами для проверки того, не содержит ли пицца лактозу?

**Шеф-повар права. При помощи вычисляемого свойства можно проверить, содержит ли пицца лактозу.**

Представьте, что у вас имеется простая структура `Pizza` (сейчас она должна уже казаться хорошо знакомой) и вы добавили поле `Bool`, которое показывает, содержит ли пицца сыр или нет:

```
struct Pizza {
 var name: String
 var ingredients: [String]
 var lactoseFree: Bool
}
```

Такое свойство позволяет легко проверить, содержит ли пицца лактозу или нет:

```
var hawaiian = Pizza(name: "Hawaiian",
 ingredients: ["Pineapple", "Ham", "Cheese"],
 lactoseFree: false)
```

Тем не менее при таком решении разработчик, создающий экземпляры `Pizza`, должен помнить о необходимости правильно устанавливать флаг `lactoseFree`. Но ведь суть программирования совсем не в этом! Особенно программирования Swift.

На помощь приходят **вычисляемые свойства**.

## Анатомия Вычисляемого свойства



Вычисляемые свойства — это свойства, которые вычисляют свое значение при обращении к ним.

```
struct FavNumber {
 var number: Int
 var isMeaningOfLife: Bool {
 if number == 42 {
 return true
 } else {
 return false
 }
 }
}
```

1 **Объявление свойства**

У вычисляемых свойств есть имя и тип, как и у хранимых свойств.

2 **Код, вычисляющий значение**

Код должен задавать значение свойства. Для этого он должен вернуть требуемое значение. Похоже, здесь будет уместно воспользоваться замыканием?

Тогда вперед. Сделайте так, чтобы проверка на отсутствие лактозы работала.



## Упражнение

Измените свойство `lactoseFree` структуры `Pizza` так, чтобы оно автоматически определяло, содержит ли пицца лактозу (`Cheese` в списке ингредиентов), при помощи вычисляемого свойства.

Протестируйте только что измененную структуру `Pizza` на следующих экземплярах `Pizza` и командах `print`:

```
var hawaiian =
 Pizza(name: "Hawaiian",
 ingredients: ["Pineapple", "Ham", "Cheese"])
print(hawaiian.lactoseFree)

var vegan =
 Pizza(name: "Vegan",
 ingredients: ["Artichoke", "Red Pepper",
 "Tomato", "Basil"])
print(vegan.lactoseFree)
```

→ Ответ на с. 205.



Иногда бывает полезно выполнить некий код до или после изменения свойства.

В Swift существует волшебная штука, которая позволяет это сделать: наблюдатели свойств.

Представьте, что в структуру Pizza также включено поле количества quantity:

```
struct Pizza {
 var name: String
 var ingredients: [String]
 var quantity: Int
}
```

Было бы полезно, если бы при каждом изменении quantity выводилось сообщение, которое указывает шеф-повару, сколько экземпляров пиццы этого типа осталось.

Возможное решение 1 ← Честно говоря, это скорее обманка, чем возможное решение.

Можно написать метод, который используется для задания свойства quantity, и внутри этого метода вывести текущее значение quantity. В принципе, такое решение работает, но оно противоречит стремлению Swift к удобству работы со свойствами.

Возможное решение 2

Можно воспользоваться наблюдателем свойств. Наблюдатели свойств позволяют определить код, который выполняется до или после изменения свойства. Чисто и в стиле Swift.

Потенциальное решение 2 в данном случае побеждает.

## Создание наблюдателя свойств

Наблюдатели свойств определяются с использованием ключевых слов didSet или willSet. Ключевое слово didSet позволяет определить код, который выполняется после изменения свойства, а willSet — код, который выполняется до изменения свойства.

Если обновить определение свойства quantity и включить в него didSet, вы будете получать сообщение при каждом изменении quantity:

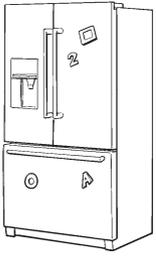
```
var quantity: Int {
 didSet {
 print("The pizza \(name) has \(quantity) pizzas left.")
 }
}
```

Если теперь создать экземпляр Pizza и изменить значение quantity, будет выведено сообщение:

```
var hawaiian = Pizza(name: "Hawaiian",
 ingredients: ["Pineapple", "Ham", "Cheese"], quantity: 10)
hawaiian.quantity -= 1
hawaiian.quantity -= 1
```

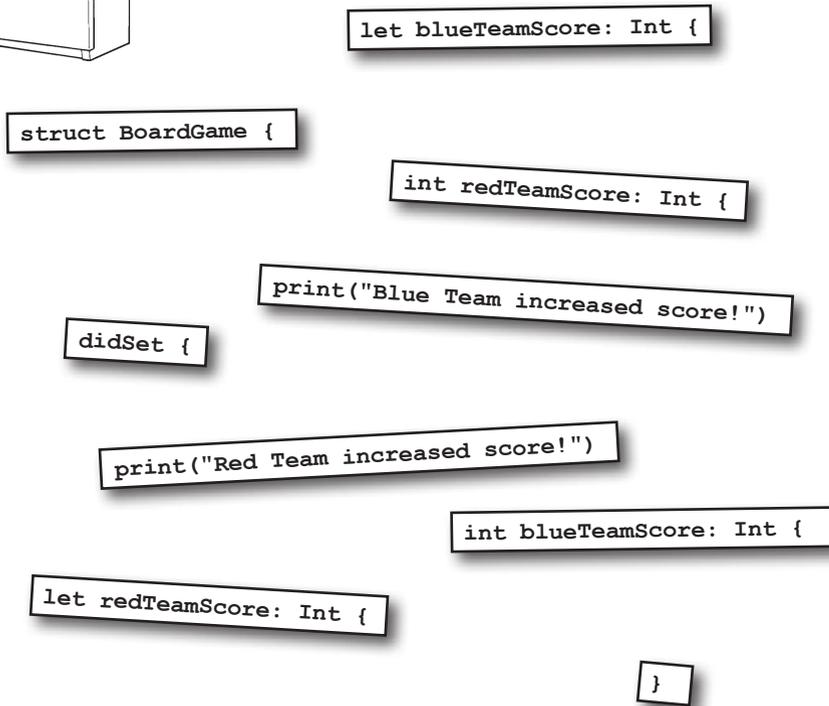
The pizza Hawaiian has 9 pizzas left.  
The pizza Hawaiian has 8 pizzas left.

Наблюдатель свойств не вызывается при создании экземпляра структуры, а только при его изменении после создания.



## Развлечения с магнитами

Попробуйте собрать полезный код из магнитов. Будьте внимательны; здесь есть несколько лишних магнитов, а некоторые магниты могут использоваться многократно.



Наблюдатель свойств всегда реагирует на изменение значения свойства. Это означает, что наблюдатели свойств будут срабатывать даже в том случае, если вновь присвоенное значение полностью совпадает с ранее присвоенным (текущим) значением.

→ Ответ на с. 210.



## Расслабьтесь

**Скорее всего, вам никогда не придется использовать willSet. Ключевое слово didSet применяется намного чаще.**

Выбор ключевого слова зависит от того, хотите ли вы что-то сделать после того, как свойство уже было изменено (didSet), или до этого (willSet). Как правило, в программах вам приходится реагировать на изменения, а не выполнять действия перед этим изменением (каким бы неизбежным оно ни было).

Например, если вы обновляете некий пользовательский интерфейс или что-то сохраняете, то с большей вероятностью это стоит делать после того, как изменение уже вступило в силу.

Ключевое слово willSet может пригодиться в ситуации, когда вам для чего-либо нужно знать значение свойства до и после изменения: willSet сохраняет предшествующее состояние, чтобы вы могли потом с ним что-то сделать. В частности, такая возможность будет полезной при анимации пользовательского интерфейса.

## Get- и set-методы для вычисляемых свойств

Но у вычисляемых свойств в запасе есть еще один эффектный трюк: **get- и set-методы**. Давайте ненадолго отвлечемся от пиццы и рассмотрим структуру для представления температуры.

```
struct Temperature {
 var celsius: Float = 0.0
 var fahrenheit: Float {
 return ((celsius * 1.8) + 32.0)
 }
}
```

Хранимое свойство, представляющее значение температуры по шкале Цельсия

Вычисляемое свойство (такое же, как было создано выше) для вычисления температуры по шкале Фаренгейта на основании значения по шкале Цельсия

Вариант ее возможного использования:

```
var temp = Temperature(celsius: 40)
print(temp.fahrenheit)
```

Вычисляемое свойство `fahrenheit` также можно выразить в следующем виде с использованием ключевого слова **get**:

```
struct Temperature {
 var celsius: Float = 0.0
 var fahrenheit: Float {
 get {
 return ((celsius * 1.8) + 32.0)
 }
 }
}
```

Чтобы вычислить температуру по Фаренгейту, следует умножить температуру по Цельсию на 1,8 и прибавить 32.

**Ключевое слово `get`** (и синтаксис вычисляемых свойств по умолчанию) упрощает чтение вычисляемого свойства. За ним следует код для вычисления значения этого свойства.

### Ключевые моменты

- Get-методы определяются с ключевым словом `get`. Get-методы позволяют получить значение свойства из другого источника.
- Set-методы используют ключевое слово `set`. Set-методы позволяют записать значение свойства в другой приемник.
- Также в синтаксис `get`-метода можно включить код для вычисления значения свойства.

## Реализация set-метода

Также для любого вычисляемого свойства можно определить set-метод, используя ключевое слово set:

```
struct Temperature {
 var celsius: Float = 0.0
 var fahrenheit: Float {
 get {
 return ((celsius * 1.8) + 32.0)
 }
 set {
 self.celsius = ((newValue - 32) / 1.8)
 }
 }
}
```

**Ключевое слово set  
обеспечивает возмож-  
ность записи для вы-  
числяемых свойств.**

Можно указать локальное имя в круглых скобках после ключевого слова set или использовать имя по умолчанию: newValue.

После этого его можно использовать следующим образом:

```
var temp = Temperature(celsius: 40)
print(temp.fahrenheit)
temp.fahrenheit = 55
print(temp.celsius)
```

Да, Java, это про тебя.



Будьте  
осторожны!

**В других языках программирования get-методы и set-методы используются на каждом шагу. В Swift дело обстоит иначе.**

*В Swift get- и set-методы отличаются от одноименных конструкций в других языках. В некоторых языках при создании переменной экземпляра (свойства в терминологии Swift) часто приходится создавать специальные методы для обращения к этим переменным экземплярам и их изменения. Именно эти методы называются get-методами и set-методами в большинстве других языков. Swift выполнит эту работу за вас.*



## Упражнение Решение

С. 203

Измените свойство `lactoseFree` структуры `Pizza`, чтобы оно автоматически определяло, содержит пицца лактозу или нет, с использованием вычисляемого свойства:

```
struct Pizza {
 var name: String
 var ingredients: [String]
 var lactoseFree: Bool {
 if ingredients.contains("Cheese") {
 return false
 } else {
 return true
 }
 }
}
```

В главе 3 упоминался метод `contains`, поддерживаемый для массивов. Конечно, мы предполагаем, что сыр – единственный возможный ингредиент, содержащий лактозу...

Протестируйте измененную структуру `Pizza` на следующих экземплярах `Pizza` и командах `print`:

```
var hawaiian =
 Pizza(name: "Hawaiian",
 ingredients: ["Pineapple", "Ham", "Cheese"])
print(hawaiian.lactoseFree)

var vegan =
 Pizza(name: "Vegan",
 ingredients: ["Artichoke", "Red Pepper",
 "Tomato", "Basil"])
print(vegan.lactoseFree)
```



За кулисами



## Строки Swift в действительности являются структурами

Ранее мы уже упоминали об этом, но важно подчеркнуть, что строки Swift в действительности реализованы как структуры. Во внутренней реализации *структуры предельно упрощены*. Это означает, что их легко создать, легко уничтожить и забыть о них, не беспокоясь о быстродействии.

Строки Swift содержат много полезных методов, которые выполняют полезные операции, включая count, uppercase и isEmpty.

Так как тип String реализован в виде структуры, вся функциональность этих многочисленных **методов экземпляров** может быть инкапсулирована в реализации String. Такое решение компактно и удобно.

```
352 public struct String {
353 public // @SPI(FOUNDATION)
```

## Кто и что делает?

Соедините результат со строкой, которая его создала:

```
var myString = "Pineapple belongs on pizza"
```

|                             |                                |
|-----------------------------|--------------------------------|
| PINEAPPLE BELONGS ON PIZZA  | myString.hasPrefix("p")        |
| true                        | myString.uppercased()          |
| 26                          | myString += "!"                |
| false                       | myString.contains("Pineapple") |
| Pineapple belongs on pizza! | myString.count                 |

→ Ответ на с. 209.

## Для чего нужны отложенные свойства

Иногда лучше действовать не спеша. Отложенные свойства — это свойства, которые создаются только тогда, когда в них возникнет необходимость, а не во время инициализации типа, определяемого структурой, которой они принадлежат.

Рассмотрим структуру `Pizza` с вычисляемым свойством, которое представляет продолжительность приготовления пиццы:

Здесь этот метод просто возвращает 100, но представьте, что для вычисления времени приготовления необходимо проделать много сложной работы, которая требует значительных затрат ресурсов.

```
struct Pizza {
 var cookingDuration = getCookingTime()
}

func getCookingTime() -> Int {
 print("getCookingTime() was called!")
 return 100
}

var hawaiian = Pizza()

print(hawaiian.cookingDuration)
```

Разберемся, что происходит в этом коде:

- ❶ Определяется структура `Pizza`.
- ❷ Определяется метод `getCookingTime()`, который выводит сообщение и возвращает целое число.
- ❸ Создается экземпляр `Pizza` с именем `hawaiian`. Это приводит к вызову метода `getCookingTime()`, потому что он необходим для сохранения значения в свойстве `cookingDuration`, обязательном для нового экземпляра структуры `Pizza`.
- ❹ Выводится значение `cookingDuration` экземпляра `hawaiian` структуры `Pizza`.

↑  
Несколько неудобно, что значение `cookingDuration` должно вычисляться при обращении, а не при создании экземпляра структуры. Логичнее, если `cookingDuration` будет использоваться, когда клиент уже на пути в ресторан.

## Использование отложенных свойств

Если вы хотите, чтобы вычисление `cookingDuration` начиналось, допустим, когда клиент подъезжает к пиццерии (чтобы заказ не подгорел), или если процесс вычисления `cookingDuration` требует значительных затрат ресурсов, вы предпочтете отложить его до того момента, когда он станет действительно необходим.

Для этого можно добавить к свойству ключевое слово **lazy**:

```
struct Pizza {
 lazy var cookingDuration = getCookingTime()
}

func getCookingTime() -> Int {
 print("getCookingTime() was called!")
 return 100
}

var hawaiian = Pizza()

print(hawaiian.cookingDuration)
```

Разберем, что происходит в этой версии кода:

- 1 Как и прежде, определяется структура `Pizza`.
- 2 Как и прежде, определяется метод `getCookingTime()`, который выводит сообщение и возвращает целое число.
- 3 Создается экземпляр `Pizza` с именем `hawaiian`. Свойство `cookingDuration` не вычисляется, потому что оно помечено ключевым словом **lazy**.
- 4 Выводится значение `cookingDuration` экземпляра `hawaiian` структуры `Pizza`. Запрос на вывод инициирует вычисление `cookingDuration`, что, в свою очередь, приводит к вызову метода `getCookingTime()`.

Ключевое слово `lazy` означает, что свойство создается и вычисляется только при обращении.



### Мозговой Штурм

Выполните приведенный выше код с ключевым словом **lazy** в свойстве `cookingDuration` и без него.

Когда выполняется команда `print` внутри `getCookingTime()`

Как на это влияет использование ключевого слова `lazy`?

Понаблюдайте за тем, как ключевое слово `lazy` влияет на создание свойства `cookingDuration` (а следовательно, на вызов метода `getCookingTime()`).

## Часто Задаваемые Вопросы

**В:** Для чего нужен механизм управления доступом, когда я сам пишу свой код? Почему я не могу просто ограничить доступ на основании того, к чему я сам обращаюсь? При желании я могу отключить управление доступом, тогда зачем он вообще нужен?

**О:** Первый и наиболее очевидный ответ — написанный вами код может использоваться другими людьми. Впрочем, это объяснение слишком упрощенное. В реальности управление доступом позволяет вам определять то, как, например, может использоваться некоторое значение. Если с чем-то нужно обращаться особенно осторожно, управление доступом позаботится о том, чтобы вы проявили необходимую осторожность. Имея дело с собственным кодом, вы не всегда находитесь настороже и хотя бы помните, о чем вы думали при написании этого кода.

**В:** Почему отложенные свойства называются отложенными?

**О:** Это связано со временем вычисления свойства — оно откладывается до момента, когда возникнет необходимость в значении. «Отложенный» здесь фактически означает «по требованию».

**В:** Во внутреннем представлении Swift все реализовано в виде структур?

**О:** Не все, но очень многое. Структуры Swift намного мощнее, чем в других языках, и в Swift они широко применяются.

**В:** А как насчет классов?

**О:** Мы доберемся и до них, но почти все, для чего в других языках вам пришлось бы использовать класс, в Swift можно сделать при помощи структур.

**В:** Выходит, get- и set-методы — всего лишь термины Swift для определения кода, который может выполнять чтение/запись вычисляемых свойств? На них не распространяются, скажем, требования к get- и set-методам из языка Java?

**О:** Все верно. Get- и set-методы в Java содержат код для обращения к переменной внутри чего-либо или записи значения переменной внутри чего-либо. Swift автоматически обеспечивает эту возможность, так что get- и set-методы в Swift имеют несколько другой смысл.

**В:** А почему у вас пицца с ананасами?

**О:** Так ведь ананасы — это здорово, разве нет?

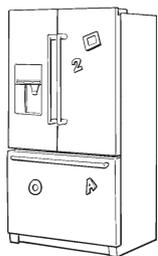
## Кто и что делает? Решение

С. 206

Соедините результат со строкой, которая его создала:

```
var myString = "Pineapple belongs on pizza"
```

|                             |   |                                |
|-----------------------------|---|--------------------------------|
| PINEAPPLE BELONGS ON PIZZA  | → | myString.hasPrefix("p")        |
| true                        | → | myString.uppercased()          |
| 26                          | → | myString += "!"                |
| false                       | → | myString.contains("Pineapple") |
| Pineapple belongs on pizza! | → | myString.count                 |



## Развлечения с магнитами. Решение

© 202

```
struct BoardGame {
 int redTeamScore: Int {
 didSet {
 print("Red Team increased score!")
 }
 }
 int blueTeamScore: Int {
 didSet {
 print("Blue Team increased score!")
 }
 }
}
```

## 7. Классы, акторы и наследование



# О пользе наследования

Иногда наследование доставляет много хлопот. Всегда приходится возиться, чтобы что-то подправить и заставить его работать.



Структуры показали, насколько полезным может быть построение типов, определяемых пользователем. Но у Swift в запасе есть и другие средства, включая *классы*. Классы похожи на структуры: они позволяют создавать *новые типы данных, содержащие свойства и методы*. Кроме того что они являются *ссылочными типами*, то есть экземпляры конкретного класса совместно используют одну копию своих данных (в отличие от структур, которые являются типами-значениями, чьи данные копируются), *классы поддерживают наследование*. Наследование позволяет построить один класс на базе другого класса.

## Структура с другим именем (и это имя — класс)

Класс очень похож на структуру, но есть несколько важных отличий:

*Это означает, что вам придется писать собственный инициализатор при создании класса.*

\* Для класса **поэлементный инициализатор не создается, а для структуры создается.** *Структуры не могут наследовать.*

\* Класс **может наследовать от другого класса. При этом он получает доступ ко всем его свойствам и методам.** *Класс является ссылочным типом. Структура является типом-значением.*

*Наследование возможно только от класса. Кроме того, не поддерживается множественное наследование (для разработчиков с опытом работы на других языках).*

\* **Копирование класса просто создает ссылку на единый набор общих данных. При копировании структуры создается отдельный дубликат данных.**

\* У класса **имеется деинициализатор, который вызывается при его уничтожении. У структуры деинициализатора нет.** *У структуры не может быть деинициализатора, да он ей и не нужен.*

\* **Свойство-переменную можно изменить в классе-константе, но не в структуре-константе.**

`class Pizza {` *← Это класс, представляющий пиццу. В нем хранится название (строка) и список ингредиентов (список строк).*

```
var name: String
var ingredients: [String]

init(name: String, ingredients: [String]) {
 self.name = name
 self.ingredients = ingredients
}
}
```

*У него есть инициализатор, потому что классы требуют, чтобы вы представили инициализатор.*

`struct Pizza {` *← Это структура. Она тоже представляет пиццу, и в ней хранится название (строка) и список ингредиентов (список строк).*

```
var name: String
var ingredients: [String] = []
}
```

Значит, я всегда  
должна писать иници-  
ализаторы для своих  
классов?

Если ваш класс содержит свойства, вы должны создать инициализатор. Значит, да... если у него есть свойства.

При написании класса *если вы вообще включаете в него какие-либо свойства, то вы должны предоставить инициализатор.*

Допустим, вы написали класс для представления типов растений:

```
class Plant {
 var name: String
 var latinName: String
 var type: String

 init(name: String, latinName: String, type: String) {
 self.name = name
 self.latinName = latinName
 }
}
```

Определение инициализатора



Тогда экземпляры этого класса могут создаваться так:

```
let bay = Plant(name: "Bay laurel",
 latinName: "Laurus nobilis",
 type: "Shrub")
```

Выглядит точно так же,  
как для структур.

**При создании инициализатора действуют те же правила, что и для структур. Необходимо лишь иметь инициализатор.**

## Наследование и классы

Иногда бывает нужно создать похожие, но слегка различающиеся сущности — или объекты, обладающие общими свойствами и методами, но между которыми есть различия. **Наследование** — механизм классов Swift, который позволяет создать класс *на основе существующего класса* и *добавить* к нему собственную функциональность.

Наш класс `Plant` позволяет создавать разные виды растений:

```
class Plant {
 var name: String
 var latinName: String
 var type: String

 init(name: String, latinName: String, type: String) {
 self.name = name
 self.latinName = latinName
 self.type = type
 }
}
```

Но класс `Plant` также можно *специализировать*, создав новый класс с именем `Succulent`. Наш класс `Succulent` наследует от `Plant` все свойства, а также его инициализатор, так что включать дополнительный код не нужно:

*Succulent* — дочерний (производный) класс.

```
class Succulent: Plant {
}
```

*Plant* — родительский класс.

*Код здесь не нужен. Все наследуется от родительского класса.*

```
let americanAloe = Succulent(name: "American aloe",
 latinName: "Agave americana",
 type: "Succulent")
```

### Ключевые моменты

- Класс может наследовать от другого класса. Этот механизм называется наследованием (или субклассированием).
- Класс, от которого он наследует, называется родительским классом (или суперклассом).
- Класс, который создается посредством наследования, называется дочерним (или производным) классом.
- Все свойства и методы, включая инициализаторы, наследуются дочерним классом.
- Дочерний класс не может обращаться к приватным переменным и методам.

Наверное, желание определить инициализатор в дочернем классе — это не пустые мечты, а вполне естественная идея. Но как это сделать?

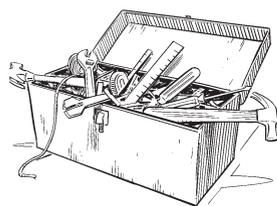


На самом деле создать собственный инициализатор несложно.

Вы можете предоставить Succulent собственный инициализатор, однако делать это следует только при необходимости.

Если в дочернем классе появляются новые элементы, которые должны инициализироваться, то ему понадобится собственный инициализатор. Также определение собственного инициализатора может стать необходимым по другой причине: если вы должны задать значения свойств родительского класса с учетом того факта, что экземпляр относится к типу Succulent.

Для Succulent можно создать новый инициализатор, которому требуются только значения `name` и `latinName`, а затем воспользоваться этой информацией (и уже известным типом) для вызова `super.init()`, который вызывает инициализатор родительского класса:



Собственные инициализаторы

(снова)

```
class Succulent: Plant {
 init(name: String, latinName: String) {
 super.init(name: name, latinName: latinName, type: "Succulent")
 }
}
```

Значение `type` задается и передается инициализатору родительского класса.

Инициализатор родительского класса всегда должен вызываться из дочерних классов.

После этого можно создать новый экземпляр Succulent без необходимости указывать `type`:

```
let americanAloe = Succulent(name: "American aloe", latinName: "Agave americana")
```



Будьте осторожны!

### Классы Swift не поставляются с поэлементным инициализатором.

Вы всегда должны предоставлять собственный инициализатор при создании класса, в отличие от структур. Классы в Swift не имеют поэлементных инициализаторов из-за механизма наследования: если вы построили класс, который наследует от другого класса, а затем в этот класс будут добавлены новые свойства, код дочернего класса перестанет работать.

Здесь действует простое правило: если вы создаете класс, вы всегда должны написать инициализатор самостоятельно. Таким образом, вы всегда помните и отвечаете за обновление инициализатора при любых изменениях в свойствах класса.

У этого правила существует исключение: при субклассировании инициализатор может оказаться излишним, потому что инициализатор родительского класса делает все необходимое.



## Мозговой Штурм

Потренируйтесь в создании новых экземпляров Succulent. Создайте следующие экземпляры дочернего класса Succulent:

- < Elephant's foot (Beaucarnea recurvata)
- < Calico hearts (Adromischus maculatus)
- < Queen victoria (Agave victoria regina)

Когда это будет сделано, создайте новый дочерний класс для представления дерева (Tree). Свойство `type` у дерева всегда содержит строку "Tree", а к свойству `name` всегда присоединяется суффикс "tree":

```
class Tree: Plant {

}
```

После того как вы создадите дочерний класс Tree, создайте несколько экземпляров Tree. Создайте следующие экземпляры дочернего класса Tree:

- < European larch (Larix decidua)
- < Red pine (Pinus resinosa)
- < Northern beech (Fagus sylvatica)

## Замена Переопределение методов

Инициализатор — не единственный метод, который можно заменить при создании subclasses: вы также можете заменить любые методы родительского класса. Такая замена называется переопределением.

Вот наш класс `Plant`:

```
class Plant {
 var name: String
 var latinName: String
 var type: String

 init(name: String, latinName: String, type: String) {
 self.name = name
 self.latinName = latinName
 self.type = type
 }

 func printInfo() {
 print("I'm a plant!")
 }
}
```



Класс `Plant` содержит метод `printInfo`, который выводит информацию о том, что он вызывается для экземпляра `Plant`.

А вот дочерний класс `Succulent`, переопределяющий метод родителя:

```
class Succulent: Plant {
 init(name: String, latinName: String) {
 super.init(name: name, latinName: latinName, type: "Succulent")
 }

 override func printInfo() {
 print("I'm a Succulent!")
 }
}
```

Если потребуется, здесь также можно вызвать родительскую реализацию `printInfo()` при помощи `super.printInfo()`.

Мы переопределяем метод `printInfo` родительского класса. Тем самым мы изменяем реализацию этого конкретного метода, когда он вызывается из дочернего класса.



## Расслабьтесь

Чтобы переопределить метод, необходимо использовать ключевое слово `override`.

Вы не рискуете случайно переопределить метод, создав другой метод с таким же именем.

Если вы случайно включите в дочерний класс метод с таким же именем, как у метода родительского класса, не указав при этом ключевое слово `override`, Swift выдаст сообщение об ошибке:

```
func printInfo() {
 // ...
}
// Overriding declaration requires an 'override' keyword
```

Вы также получите ошибку при попытке переопределить метод, не существующий в родительском классе:

```
override func printDetails() {
 // ...
}
// Method does not override any method from its superclass
```

## Возьмите в руку карандаш



Следующий класс представляет инопланетянина. Он содержит метод, при вызове которого инопланетянин пьет:

```
class Alien {
 func drink() {
 print("Drinking some alien wine!")
 }
}
```

Создайте новый дочерний класс, представляющий клингона (разновидность инопланетянина). Переопределите метод `drink`, чтобы клингон делал что-то специфическое для его расы (например, восклицал «рагх!» и пил бладвейн).

→ Ответы на с. 223.



## Мозговой Штурм

Если потребуется, вы также можете переопределять свойства, унаследованные от родительского класса. Добавьте в класс `Plant` новое вычисляемое свойство `description` для хранения описания:

```
var description: String {
 return "This is a \(name) (\(latinName)) \(type)."
```

Затем обновите дочерний класс `Succulent` и включите в него свойство `age`:

```
var age: Int
```

Теперь ваш ход:

Обновите инициализатор в классе `Succulent`, чтобы он получал значение `age` в параметре. Затем используйте переданное значение для обновления `self.age` (только что добавленного свойства `age`).

А когда это будет сделано, переопределите свойство `description` в классе `Succulent`. Используйте тот же синтаксис, который использовался ранее для методов. Возвращаться должно новое описание `description`, включающее свойство `age` класса `Succulent`.

Следует помнить, что вы также можете обращаться к свойствам суперкласса. Для этого используется синтаксис вида:

```
super.name
```

А когда это будет сделано, вы сможете создать экземпляр `Succulent`:

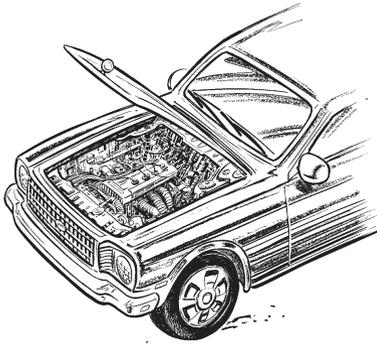
```
let americanAloe =
 Succulent(name: "American aloe",
 latinName: "Agave americana",
 age: 5)
```

Если теперь вывести свойство `description`, в него будет включено свойство `age`:

```
print(americanAloe.description)
```

This is a American aloe (Agave americana)  
Succulent. It is 5 years old.

Под капотом...



**Классы являются  
ссылочными типами.  
Экземпляры ссылочных  
типов всегда относятся  
к одному объекту  
в памяти.**

В Swift типы делятся на типы-значения и ссылочные типы.



Классы являются ссылочными типами. Это означает, что, присваивая их другой переменной или константе, вы создаете **новую ссылку на другой объект в памяти вместо** создания копии.

Для примера возьмем экземпляр Succulent с предыдущей страницы:

```
let americanAloe =
 Succulent(name: "American aloe",
 latinName: "Agave americana")
```

Присвойте ссылке новой переменной:

```
var anotherAloe = americanAloe
```

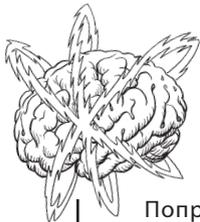
Затем измените эту переменную:

```
anotherAloe.name = "Sentry plant"
```

Вы изменили *исходный и единственный экземпляр Succulent*, у которого свойству name теперь присвоено значение "Sentry plant".

В этом можно убедиться при помощи команды print:

```
print(americanAloe.name)
print(anotherAloe.name)
```



Мозговой  
Шторм

Попробуйте реализовать Succulent в виде структуры. Когда это будет сделано, создайте экземпляр новой структурной реализации Succulent и присвойте его новой переменной, затем измените эту переменную. Что произойдет с исходным экземпляром? В этом проявляются различия между типами-значениями и ссылочными типами.



Будьте  
осторожны!

Различия между типами-значениями и ссылочными типами нередко приводят к путанице.

Копирующее поведение типов-значений (например, структур) — одна из главных причин, по которым Swift рекомендует использовать структуры для большинства типов, определяемых пользователем. Каждая область вашей программы располагает собственной копией данных, без риска их изменения или нарушения целостности.

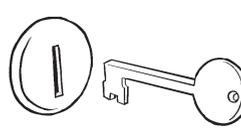
## Финальные классы

Иногда бывает полезно запретить наследование от некоторого класса (то есть объявить его финальным). Это можно сделать при помощи ключевого слова **final**. Финальные классы *не могут* иметь дочерних классов, поэтому их методы не могут переопределяться. Финальные классы должны использоваться в том виде, в каком они были написаны.

```
final class Garden {
 var plants: [Plant] = []

 init(plants: [Plant]) {
 self.plants = plants
 }

 func listPlants() {
 for plant in plants {
 print(plant.name)
 }
 }
}
```



Финальный класс фиксируется в своем исходном состоянии и не может использоваться для subclassирования.



Если вы попытаетесь унаследовать от финального класса, Swift выдает сообщение об ошибке.

```
• class RooftopGarden: Garden {
 Inheritance from a final class 'Garden'
}
```



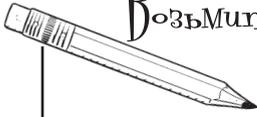
Будьте осторожны!

**Ходят упорные слухи, что финальные классы лучше работают в Swift.**

Когда-то это было правдой: если класс помечен как финальный, компилятор Swift знает, что он никогда не будет изменяться, и может применять некоторые оптимизации, которые были бы невозможны в иных ситуациях.

Иногда это действительно так. Но если только в вашем коде не приходится учитывать каждую миллисекунду, этот фактор перестал действовать, начиная со Swift 5. Финальные классы просто не могут subclassироваться.

Возьмите в руку карандаш



Решение

(. 219)

```
class Alien {
 func drink() {
 print("Drinking some alien wine!")
 }
}

class Klingon: Alien {
 override func drink() {
 print("Drinking blood wine! Ragh!")
 }
}

let martok = Klingon()
martok.drink()
```



Будьте  
осторожны!

**В Swift еще существуют так называемые акторы, которые имеют много общего с классами.**

*Акторы немного выходят за рамки этой книги, но они, по сути, работают как классы, не считая того, что они могут безопасно использоваться при параллельном выполнении. Что это означает? Swift всеми силами старается гарантировать, что данные внутри актора не могут быть изменены в то время, когда что-то другое пытается сделать то же самое. Эта проблема довольно типична для параллельного выполнения. Ниже приведен актор, определяющий человека с максимальным возрастом `maximumAge`:*

```
actor Human {
 var maximumAge = 107

 func printAge() {
 print("Max age is currently \(maximumAge)")
 }
}
```

*Акторы очень похожи на классы и структуры, они также могут содержать свойства и методы. И они, как и классы, являются ссылочными типами, так что у них больше общего с классами, нежели со структурами. Вскоре мы еще вернемся к этой теме.*

Ранее вы упоминали  
о деинициализаторах.  
А это что такое?

### Верно. У классов могут быть деинициализаторы.

Деинициализатор содержит код, который выполняется при уничтожении экземпляра класса.

Деинициализатор создается ключевым словом **deinit** и в остальном работает так же, как и инициализатор (не получая параметров):

```
class Plant {
 var name: String
 var latinName: String
 var type: String

 init(name: String, latinName: String, type: String) {
 self.name = name
 self.latinName = latinName
 self.type = type
 }

 deinit {
 print("The plant '\(name)' has been deinitialized.")
 }
}
```



Чтобы убедиться в том, что деинициализатор выполняется, можно воспользоваться ключевым словом `_` и создать экземпляр `Plant` (который будет немедленно уничтожен, потому что так работает синтаксис `_`):

```
var _ = Plant(name: "Bay laurel",
 latinName: "Laurus nobilis", type: "Evergreen Tree")
```

The plant 'Bay laurel' has been deinitialized.

Почему же у структур нет деинициализаторов?

**Классы намного сложнее структур. Это короткий ответ.**

Развернутый ответ: классы обладают более сложным поведением копирования, и в программе могут существовать несколько копий класса, которые все указывают на один экземпляр. Это означает, что будет очень трудно определить, когда экземпляр класса уничтожается (он уничтожается, когда пропадает последняя ссылающаяся на него переменная).

Деинициализатор сообщает Swift, когда уничтожается экземпляр класса. Для структур это определяется легко: структура исчезает тогда, когда перестает существовать ее владелец.

Структурам деинициализаторы не нужны, потому что каждая структура содержит собственную копию данных, и при уничтожении структуры ничего особенного происходить не должно.

Так мы вернемся к акторам?



**Конечно, давайте вернемся к акторам.**

Вспомните актер `Human` со свойством `maximumAge`, приведенный пару страниц назад. Это был актер, но мы не делали с ним ничего, что бы относилось к специфике акторов.

Представьте, что ему потребовалось обновить значение `maximumAge` на основании данных другого актора `Human`, который прожил дольше максимума. Вот как это безопасно делается для акторов:

```
func updateMaximumAge(from other: Human) async {
 maximumAge = await other.maximumAge
}
```

*Вероятно, пока вы не пользуетесь акторами. Тем не менее будет полезно знать об их существовании.*

Ключевые слова `async` и `await` фактически сообщают Swift, что вы хотите отправить быстрое сообщение другим экземплярам `Human` и предложить им сообщить значение `maximumAge` при первой возможности. Это может произойти немедленно, а может произойти по прошествии заметного времени. При таком подходе с ними можно безопасно работать.



3 за кулисами

# Автоматический подсчет ссылок → АПС

Во внутренней реализации Swift используется процесс **автоматического подсчета ссылок**, или **АПС**. Механизм АПС отслеживает каждый экземпляр создаваемого класса. Именно так Swift узнает, когда следует вызвать деинициализатор.

Каждый раз, когда создается очередная копия экземпляра класса, *АПС увеличивает счетчик ссылок этого экземпляра на единицу.* Каждый раз, когда уничтожается очередная копия экземпляра класса, *АПС уменьшает счетчик ссылок этого экземпляра на единицу.*

Когда счетчик ссылок достигает нуля, АПС точно знает, что ни одной ссылки на этот класс не осталось, и Swift может вызвать деинициализатор, уничтожающий объект.

```
class Thing {
 let name: String
 init(name: String) {
 self.name = name
 print("Thing '\(name)' is now initialized.")
 }

 deinit {
 print("Thing '\(name)' is now deinitialized.")
 }
}
```

**1** Класс Thing содержит свойство name, а также инициализатор и деинициализатор, которые выводят сообщение о своем вызове.

**2** Три опциональные переменные могут содержать Thing или nil.

```
var object1: Thing?
var object2: Thing?
var object3: Thing?
```

**3** Создаем экземпляр Thing и присваиваем его object1. В программе появляется сильная ссылка из object1 на новый экземпляр Thing.

```
object1 = Thing(name: "A Thing")
object2 = object1
object3 = object1
object1 = nil
object2 = nil
object3 = nil
```

**4** На этот момент существуют три сильные ссылки на наш экземпляр Thing (в object1, object2 и object3).

Счетчик ссылок для экземпляра Thing равен 1.

Счетчик ссылок равен 3.

**5** Теперь две сильные ссылки уничтожаются (потому что двум опциональным переменным присваивается nil). Экземпляр Thing остается в памяти, потому что все еще существует сильная ссылка на него из object3.

**6** Когда последней переменной присваивается nil, ни одной сильной ссылки не остается, и экземпляр Thing удаляется из памяти.

Счетчик ссылок равен 1.

Счетчик ссылок равен 0.

Thing 'A Thing' is now deinitialized.



## Изменяемость

При работе с классами важно помнить об *изменяемости*.

Свойство структуры, объявленной как константа, изменяться не может.

Однако **класс-константа со свойством позволяет изменить свойство в любой момент:**

```
class Plant {
 var name: String

 init(name: String) {
 self.name = name
 }
}
```

Чтобы предотвратить это, необходимо объявить свойство как константу ключевым словом *let*.

```
let myPlant = Plant(name: "Vine")
print(myPlant.name)
myPlant.name = "Strawberry"
print(myPlant.name)
```

Экземпляр класса *Plant*, хранимый как константа

Vine

Strawberry

И в отличие от структур, классы не требуют включения ключевого слова *mutating* для методов, изменяющих свойства.

## Ключевые моменты

- Классы-переменные допускают изменение свойств-переменных.
- Классы-константы допускают изменение свойств-переменных.
- Структуры-переменные допускают изменение свойств-переменных.
- Структуры-константы не допускают изменения свойств-переменных.
- Акторы похожи на классы, но они проектировались для параллельного выполнения и безопасности.
- При помощи ключевых слов *async* и *await* акторы могут безопасно запрашивать данные от других данных в условиях параллельного выполнения.
- Создавая актор, вы можете делать с его свойствами все что угодно, но если вам понадобится взаимодействовать с другим типом, необходимо использовать *async* и *await* для обеспечения безопасности.
- Это связано с тем, что акторы гарантируют безопасность в условиях параллельного выполнения.



## Операторы проверки тождественности

Как вы уже знаете, классы в Swift являются ссылочными типами. Это означает, что **несколько переменных или констант могут содержать ссылки на один конкретный экземпляр** класса.

По этой причине будет полезно иметь возможность:

- 1 Проверить, относятся ли две ссылки к одному экземпляру, с помощью оператора `===` . Этот оператор проверяет два экземпляра на тождественность, а не на равенство. Равенство проверяется оператором `==` .
- 2 Проверить, относятся ли две ссылки к разным экземплярам, с помощью оператора `!==` . Этот оператор проверяет, что два экземпляра не тождественны, а не что они не равны. Неравенство проверяется оператором `!=` .

```
var plantOne = Plant(name: "Bay tree")
var plantTwo = Plant(name: "Lemon tree")
var plantThree = plantOne
```

```
1 if (plantOne === plantThree) {
 print("plantOne and plantThree ARE the same instance of Plant")
}
```

```
2 if (plantOne !== plantTwo) {
 print("plantOne and plantTwo ARE NOT the same instance of Plant")
}
```



Будьте осторожны!

**Операторы проверки тождественности и равенства сильно различаются, несмотря на то что в них используются похожие символы.**

Тождественность проверяется с помощью оператора `===` (и `!==`), а равенство — с помощью оператора `==` (и `!=`). Тождественность означает, что две переменные относятся к одному экземпляру класса. Равенство означает, что два экземпляра содержат равные значения.

Часто  
Задаваемые  
Вопросы

**В:** Ранее я программировал на другом языке. В этом языке при создании классов (и структур) мне приходилось создавать два файла: для интерфейса (этот файл определял, какие свойства и методы содержит класс) и для реализации (файл определял реализацию этих свойств и методов). Кажется, в Swift это делать не нужно. Почему?

**О:** Когда вы создаете структуру или класс в Swift, вы делаете это в одном файле. Swift автоматически создает внешний интерфейс, который используется другим кодом. В общем, короткий ответ выглядит так: в Swift создавать два файла не нужно.

**В:** Экземпляр класса — это объект? Кажется, в Swift объекты упоминаются нечасто.

**О:** Экземпляр класса можно называть объектом, но это не соответствует стилю Swift. Так как структуры и классы в Swift

очень похожи, в Swift правильнее говорить об экземплярах, а не объектах.

**В:** Что могут делать классы, чего не могут делать структуры?

**О:** Наследование возможно только для классов. Структуры могут поддерживать протоколы, но не могут наследовать (вскоре вы узнаете о протоколах). К классам можно применять приведение типов, которое позволяет проверить тип экземпляра класса во время выполнения, а со структурами это невозможно. Классы также могут иметь деинициализаторы, которых нет у структур.

**В:** Ключевое слово `final` делает что-то еще, кроме запрета наследования от класса?

**О:** Нет, это все.

**В:** Если две переменные ссылаются на один экземпляр класса, они действительно ссылаются на одни и те же данные?

**О:** Да. Если две переменные содержат ссылки на один экземпляр класса, они действительно указывают на одну область памяти. Это означает, что изменение одной переменной приведет к изменению другой.

**В:** Все выглядит так, словно я могу изменять свойства-переменные в классах, несмотря на то что экземпляр класса, которому они принадлежат, является константой. Так и должно быть?

**О:** Да, все правильно. В том, что касается свойств-переменных и экземпляров-констант, поведение классов отличается от поведения структур.

★ КТО И ЧТО ДЕЛАЕТ? ★

Проверьте свое знание ключевых слов, используемых с классами и наследованием. Соедините каждое ключевое слово в левом столбце с его назначением в правом столбце.

|                       |                                                                             |
|-----------------------|-----------------------------------------------------------------------------|
| <code>super</code>    | Я могу запретить наследование от класса.                                    |
| <code>init</code>     | Я объявляю определяемый класс.                                              |
| <code>final</code>    | Меня вызывают при удалении экземпляра класса из памяти.                     |
| <code>class</code>    | Я могу заменить метод или свойство в родительском классе новой реализацией. |
| <code>override</code> | Меня вызывают для создания нового экземпляра класса.                        |
| <code>deinit</code>   | Я использую для ссылок на элементы родительского класса.                    |

—————▶ Ответ на с. 233.



## Беседы у камина

**Актуальная тема: что использовать — структуру или класс?**

### Ученик:

Итак, Учитель, что мне использовать — структуру или класс? Мне трудно понять, чем они отличаются, а еще труднее понять, что использовать. Вы бы не могли поделиться своей мудростью?

Учитель, и это все? Совет полезный, но не настолько полезный, как вам кажется.

Учитель, пожалуйста. Расскажите об этом чуть подробнее.

Очень убедительно, Учитель. Спасибо. Еще что-нибудь, что мне следует знать?

### Учитель:

Используй структуры. До свидания.

Используй структуры, но знай о классах... чтобы понимать, что лучше использовать структуры. До свидания.

Ладно, будь по-твоему. Используй структуры по умолчанию. Структуры лучше подходят для представления распространенных видов данных, а структуры Swift имеют много общего с классами в других языках программирования. Вычисляемые и хранимые свойства, методы — в других языках все это относится к области классов. Но только не в Swift.

Да, не суй пальцы в розетку. А если серьезно, со структурами гораздо проще анализировать состояние программы без учета состояния всей программы — структуры как типы-значения проще устроены. Впрочем, старайся пореже изменять структуры, чтобы не израсходовать всю память. Не забывай: каждое изменение приводит к копированию!



*Будьте осторожны!*

### **Классы не всегда оказываются лучшим решением в Swift.**

*Хотя в других языках программирования программы часто строятся с применением классов и наследования, возможности структур Swift настолько широки, что вам почти не придется использовать классы вместо них.*

*Кроме того, как вы вскоре узнаете, в Swift существует еще одна интересная возможность, которая отчасти снижает полезность наследования: протоколы.*

# СТАТЬ КОМПИЛЯТОРОМ Swift



Каждый фрагмент кода Swift на этой странице представляет законченную среду Playground. Представьте себя на месте компилятора Swift и определите, будет ли выполняться каждый из этих фрагментов или нет. Если фрагмент выполняется, то что он сделает?

**A**

```
class Airplane {
 func takeOff() {
 print("Plane taking off! Zoom!")
 }
}
```

```
class Airbus380: Airplane { }
```

```
let nancyBirdWalton = Airbus380()
nancyBirdWalton.takeOff()
```

**C**

```
class Appliance { }
```

```
class Toaster: Appliance {
 func toastBread() {
 print("Bread now toasting!")
 }
}
```

```
let talkieToaster = Toaster()
talkieToaster.toastBread()
```

**B**

```
class Sitcom {
 func playThemeSong() {
 print("<generic sitcom theme>")
 }
}
```

```
class Frasier: Sitcom {
 override func playThemeSong() {
 print("I hear the blues are calling...")
 }
}
```

```
let show = Frasier()
show.playThemeSong()
```

→ Ответ на с. 233.

упражнение swift



Выловите из бассейна строки кода и разместите их в пустых строках в среде Playground. Каждая строка может использоваться **только один** раз; использовать все строки необязательно. Ваша **задача**: построить код, который сгенерирует приведенный ниже результат.

```

class Dog {

 var age: Int
 var breed: String
 init(name: String, age: Int, breed: String) {
 self.name = name
 self.age = age

 }

 print("\(name) the \(breed) barks loudly!")
}

class Greyhound _____ {
 _____ {

 }
 _____ bark() {
 print("\(name) the greyhound doesn't care to bark.")
 }
}

var baggins =
 Dog(name: "Bilbo Baggins", age: 12, breed: "Poodle")
var trevor = Greyhound(name: "Trevor", age: 10)

trevor.bark()

```

Bilbo Baggins the Poodle barks loudly!  
Trevor is a greyhound, and doesn't care to bark.

↑  
Вывод

**Внимание: каждый предмет из бассейна может использоваться только один раз!**

```

 _____ : Dog
 override func _____
 let name: String
 super.init(name: name, age: age, breed: "Greyhound")
 _____ func bark() baggins.bark()
 init(name: String, age: Int) _____
 self.breed = breed

```

→ Ответ на с. 234.



## СТАТЬ КОМПИЛЯТОРОМ Swift. Решение

С. 231

Все фрагменты что-то выведут:

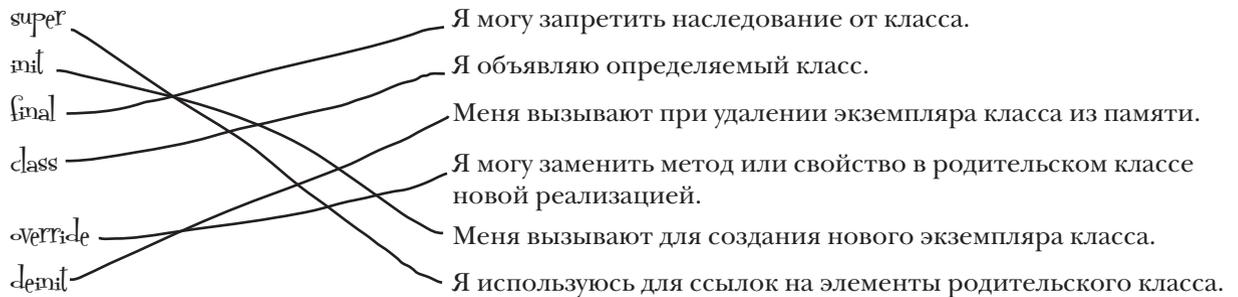
A выводит "Plane taking off! Zoom!"

B выводит "I hear the blues are calling..."

C выводит "Bread now toasting!"

## \* КТО И ЧТО ДЕЛАЕТ? РЕШЕНИЕ

С. 229





## У бассейна. Решение

C. 232

```

class Dog {
 let name: String
 var age: Int
 var breed: String
 init(name: String, age: Int, breed: String) {
 self.name = name
 self.age = age
 self.breed = breed
 }
 func bark() {
 print("\(name) the \(breed) barks loudly!")
 }
}

class Greyhound: Dog {
 init(name: String, age: Int) {
 super.init(name: name, age: age, breed: "Greyhound")
 }
 override func bark() {
 print("\(name) the greyhound doesn't care to bark.")
 }
}

var baggins =
 Dog(name: "Bilbo Baggins", age: 12, breed: "Poodle")
var trevor = Greyhound(name: "Trevor", age: 10)
baggins.bark()
trevor.bark()

```

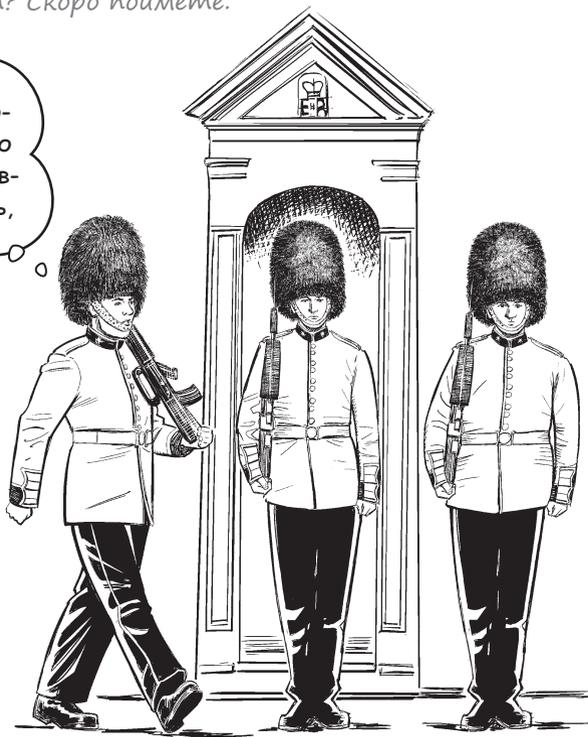
## 8. Протоколы и расширения



# Протокольные церемонии

И еще вы расширите свои знания! Поняли намек? Нет? Скоро поймете.

Мы уделяем очень большое внимание протоколам. Их соблюдение стало важной частью нашей повседневной работы. Крайне важно знать, чего следует ожидать.



Вы все знаете о классах и наследовании, но у Swift еще есть немало средств структурирования программ. Знакомьтесь: протоколы и расширения. Протоколы в Swift позволяют определить шаблон с перечнем методов и свойств, необходимых для некоторой цели или некоторого блока функциональности. Протокол включается классом, структурой или перечислением, в которых содержится его фактическая реализация. Типы, которые предоставляют необходимую функциональность, называются **поддерживающими** этот протокол. **Расширения** позволяют легко **добавлять новую функциональность** в существующие типы.

Мне нравятся классы и наследование. Я осознала их силу, но что, если во всех моих субклассах должна присутствовать некоторая функциональность, которая не имеет смысла в родительском классе?

**Отличный вопрос. Мы очень рады, что вы его задали.**

В Swift существует совершенно иной механизм построения взаимосвязанной логики и объектов с использованием протоколов и расширений. Это особый, в чем-то уникальный подход *в стиле Swift*.

**Протоколы определяют набор функциональных средств, которые что-то могут *включать* без предоставления реализации.**

**Протокол можно рассматривать как своего рода контракт:** если вы определяете протокол, который требует наличия свойства с именем `color` для хранения строки, то любой тип, соответствующий этому протоколу, безусловно обязан содержать свойство с именем `color` и типом `String`; в противном случае код не будет компилироваться. Тип может поддерживать несколько протоколов; это означает, что он будет реализовывать несколько блоков функциональности.

Поняли что-нибудь? Скоро поймете.

**Расширения позволяют добавлять полностью реализованные новые методы и свойства к существующим типам без необходимости изменять реализацию типов.**

Расширения предоставляют отличную возможность для добавления полезной функциональности к классам и структурам, которые вы *не* писали (например, к встроенным типам Swift). Кроме того, они являются отличным способом четкой изоляции и структурирования вашего кода.

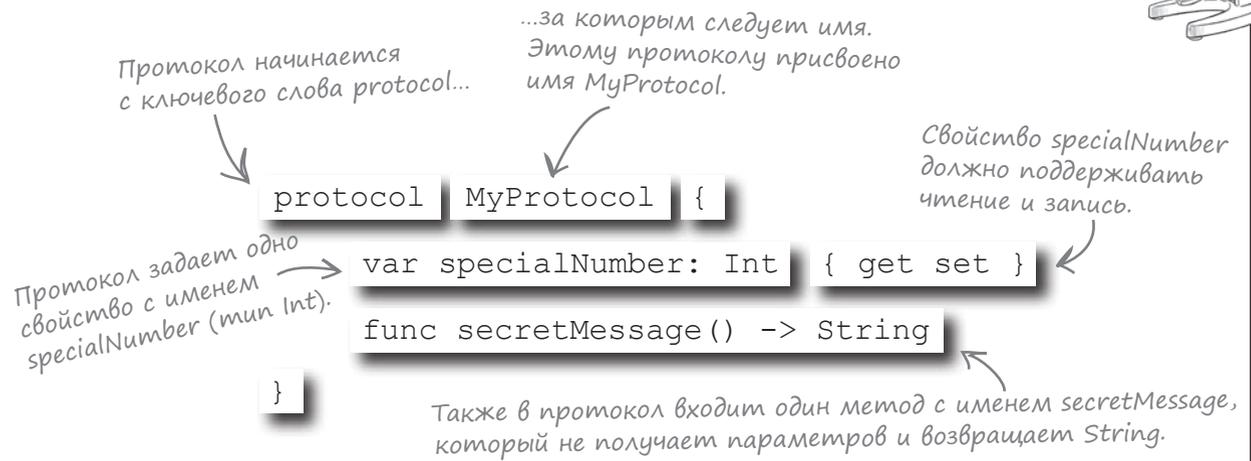
Начнем с **протоколов**.



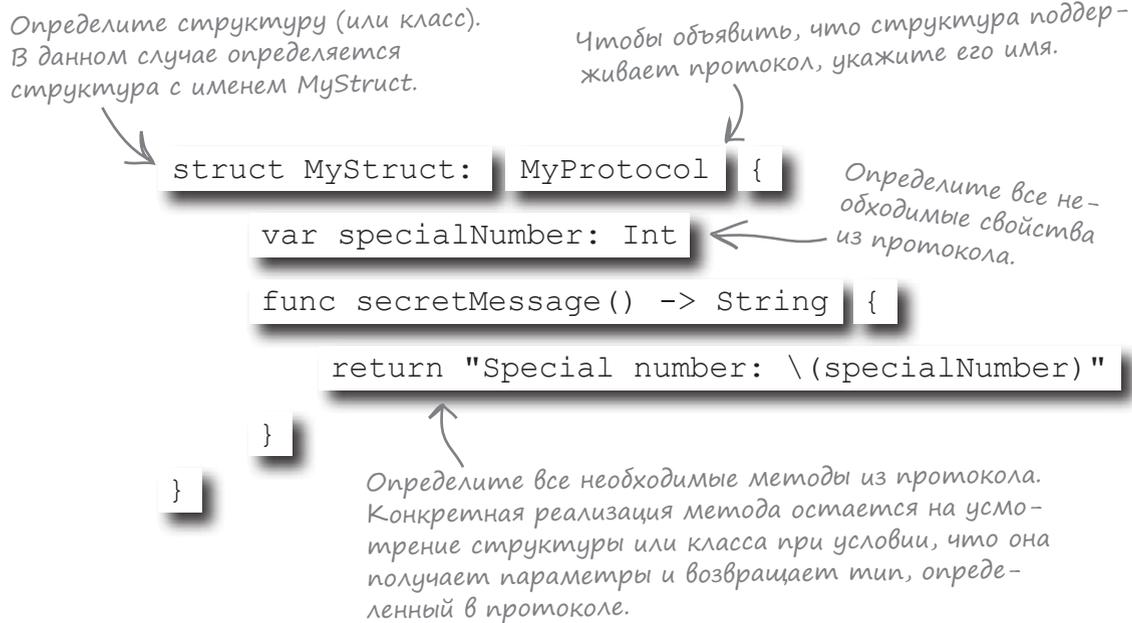
## Анатомия протокола

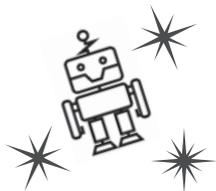


### Определение протокола



### Поддержка протокола





## Фабрика роботов

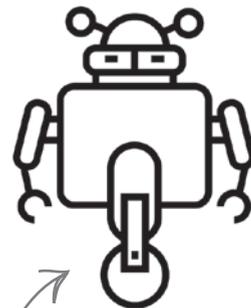
### Задача:

Нужно написать код для представления разных видов роботов, которые могут производиться на Фабрике Роботов. Однако роботы, выпускаемые фабрикой, сильно отличаются друг от друга.

У одних роботов две ноги и две антенны.

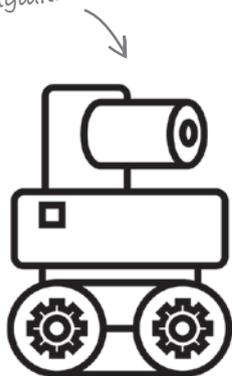


У других роботов нет ног, а есть две антенны и две руки, и они неподвижны.

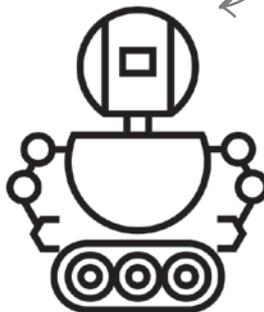


У третьих роботов одно колесо, две руки и две антенны.

У некоторых роботов есть лазерная пушка и два колеса.



Есть роботы с двумя руками и тремя колесами.



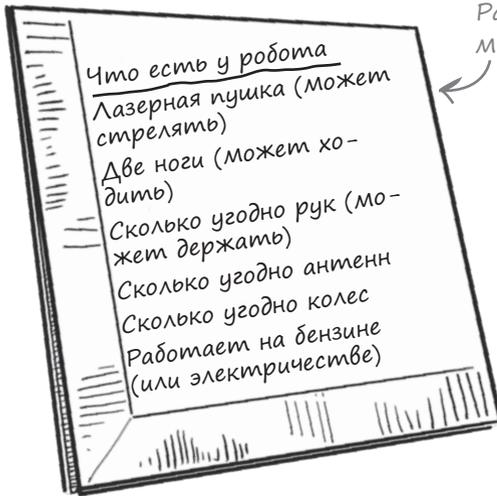
(Роботы от iconcheese из проекта Noun Project.)

**Решение:**

Это не единственное решение, потому что задачу можно решить несколькими способами.

Если составить список всех частей, которыми может быть оснащен робот, мы можем реализовать протокол для каждой части и использовать эти протоколы для реализации разновидностей роботов. Существует миллион разных вариантов реализации этого кода, но рассмотренный подход особенно соответствует *стилю Swift*.

**1 Составление списка возможных частей роботов**



Разобравшись с возможными частями роботов, можно переходить к построению списка.

**2 Создание протоколов для всех частей**

```
protocol HasLaserGun {
 func fireLaserGun()
}

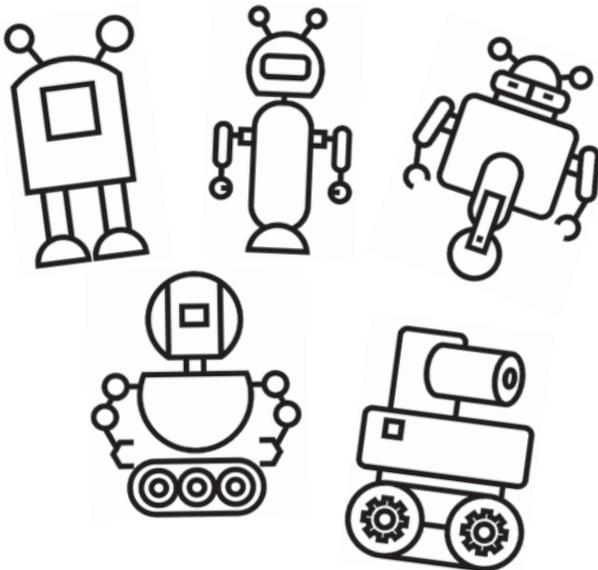
protocol Bipedal {
 func walk()
}

protocol HasArms {
 var armCount: Int { get }
 func gripWith(arm: Int)
}

protocol HasAntenna {
 var antennaCount: Int { get }
}

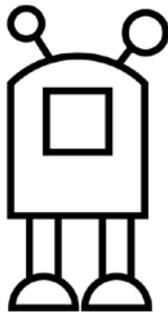
protocol HasWheels {
 var wheelCount: Int { get }
}

protocol PetrolPowered {
 var petrolPercent: Int { get set }
}
```



### 3 Определение типов роботов

Для каждой разновидности роботов создается структура, поддерживающая соответствующие протоколы. После этого можно создавать конкретные экземпляры роботов.

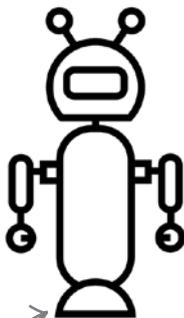


*У этого робота две ноги и есть антенны.*

```
struct RobotOne: Bipedal, HasAntenna {
 var antennaCount: Int

 func walk() {
 print("Robot is now walking, using its legs.")
 }
}

var myFirstRobot = RobotOne(antennaCount: 2)
myFirstRobot.walk()
```



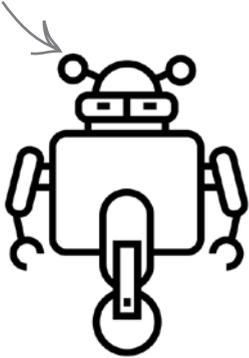
*У этого робота руки и антенны, и он работает на бензине.*

```
struct RobotTwo: HasArms, HasAntenna, PetrolPowered {
 var armCount: Int
 var antennaCount: Int
 var petrolPercent: Int

 func gripWith(arm: Int) {
 print("Now gripping with arm number \(arm)")
 }
}

var mySecondRobot = RobotTwo(armCount: 2,
 antennaCount: 2,
 petrolPercent: 100)
mySecondRobot.gripWith(arm: 1)
```

У этого робота есть колеса, руки и антенны.



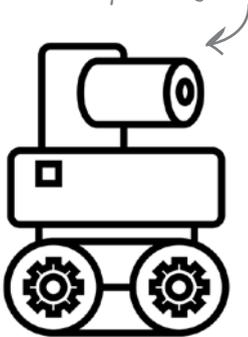
```
struct RobotThree: HasWheels, HasArms, HasAntenna {
 var wheelCount: Int
 var armCount: Int
 var antennaCount: Int

 func gripWith(arm: Int) {
 print("A RobotThree type robot is
 now gripping with arm number \(arm)")
 }
}

var myThirdRobot = RobotThree(wheelCount: 1,
 armCount: 2,
 antennaCount: 2)

myThirdRobot.gripWith(arm: 2)
```

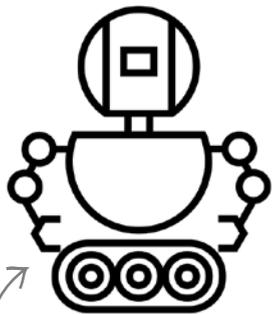
Этот робот оснащен колесами и лазерной пушкой.



```
struct RobotFour: HasLaserGun, HasWheels {
 var wheelCount: Int

 func fireLaserGun() {
 print("RobotFour type robot is firing laser!")
 }
}

var myFourthRobot = RobotFour(wheelCount: 2)
myFourthRobot.fireLaserGun()
```



У этого робота есть колеса и руки.

```
struct RobotFive: HasArms, HasWheels {
 var armCount: Int
 var wheelCount: Int

 func gripWith(arm: Int) {
 print("Now gripping with arm number \(arm)")
 }
}

var myFifthRobot = RobotFive(armCount: 2, wheelCount: 3)
myFifthRobot.gripWith(arm: 2)
```



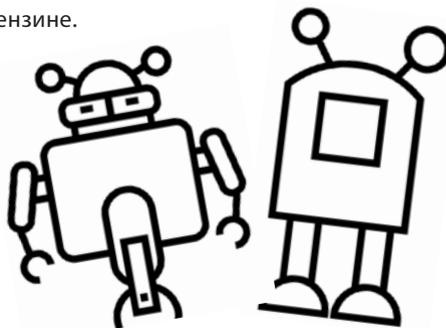
## Мозговой Штурм

Реализуйте протоколы и создайте структуры для следующих роботов:

- Робот с лазерной пушкой, колесами и руками, работающий на бензине.
- Робот с руками, работающий на бензине.
- Робот на двух ногах с лазерной пушкой.
- Робот с колесами и антеннами.
- Робот с колесами, работающий на бензине.

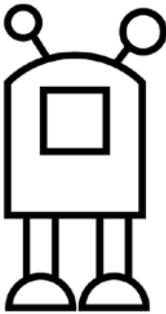
Создайте экземпляр каждого из этих роботов.

Создайте совершенно новый протокол для какой-нибудь части робота. Добавьте его к одному из существующих роботов.



## Наследование протоколов

Помните этого робота?  
RobotOne.



Для наличия нужной функциональности необходимо поддерживать эти два протокола.

```
struct RobotOne: Bipedal, HasAntenna {
 var antennaCount: Int

 func walk() {
 print("Robot is now walking, using its legs.")
 }
}

var myFirstRobot = RobotOne(antennaCount: 2)
myFirstRobot.walk()
```

Пример использования его возможностей:

Это означает, что экземпляры данного робота могут создаваться так.

Оказывается, структуру протоколов можно расширить в глубину и построить протоколы, наследующие от других протоколов.

Иногда бывает проще объявить протокол...

...который поддерживает другие протоколы.

```
protocol ReportingRobot: Bipedal, HasAntenna { }
```

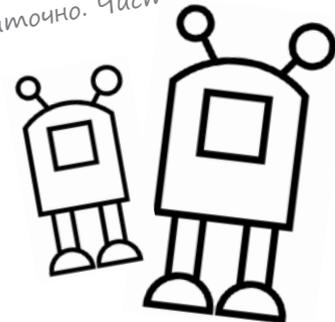
Новый протокол AssemblyRobot можно расширить дополнительными возможностями, но в данном случае этого делать не нужно, поэтому фигурные скобки останутся пустыми.

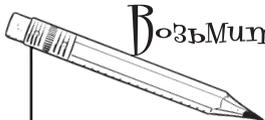
```
struct RobotOne: ReportingRobot {
 var antennaCount: Int

 func walk() {
 print("Robot is now walking, using its legs.")
 }
}
```

Тогда достаточно обеспечить поддержку нового протокола, который наследует от других... и этого вполне достаточно. Чистое и удобно.

```
var myFirstRobot = RobotOne(antennaCount: 2)
myFirstRobot.walk()
```





## Возьмите в руку карандаш

Для следующего набора протоколов, представляющих разные способы передвижения, создайте новое транспортное средство в виде структуры, которое включает все четыре протокола (а следовательно, может летать, плавать и передвигаться по суше и работает на бензине):

```
protocol Aircraft {
 func takeOff()
 func land()
}

protocol Watercraft {
 var buoyancy: Int { get }
}

protocol PetrolPowered {
 var petrolPercent: Int { get set }
 func refuel() -> Bool
}

protocol Landcraft {
 func drive()
 func brake() -> Bool
}
```

→ Ответ на с. 247.

## Ключевые моменты

- Протоколы описывают набор свойств и методов, которые должна содержать структура или класс.
- Структуры или классы могут включать любое количество протоколов. В этом случае они обязаны реализовать свойства и методы этих протоколов.
- Протоколы не используются напрямую. Они всего лишь описывают нечто, что вы можете создать в программе.
- Протокол может наследовать от одного или нескольких протоколов. Это называется наследованием протоколов.

## Изменяющие методы

Если в протокол нужно включить метод, изменяющий экземпляр, которому он принадлежит, используйте ключевое слово `mutating`.

*Определите протокол.*

```
protocol Increaser {
 var value: Int { get set }
 mutating func increase()
}
```

*Добавьте ключевое слово `mutating` во все методы, которые изменяют экземпляры любого типа, поддерживающего протокол.*

```
struct MyNumber: Increaser {
 var value: Int
```

*Объявите о поддержке протокола.*

```
 mutating func increase() {
 value = value + 1
 }
}
```

*Обеспечьте поддержку протокола и реализуйте изменяющий метод. Метод должен изменять экземпляр (например, изменять некоторые из его свойств).*

```
var num = MyNumber(value: 1)
num.increase()
```

*Перечисления тоже могут поддерживать протоколы. В этом случае также необходимо использовать ключевое слово `mutating`, но мы доберемся до этого позднее.*



*Будьте осторожны!*

### **Ключевое слово `mutating` применимо только к структурам.**

*Если метод экземпляра протокола объявляется изменяемым, необходимо использовать ключевое слово `mutating` со структурами, соответствующими протоколу.*

*Классы тоже могут включать протоколы, но использовать ключевое слово `mutating` с ними не обязательно.*



Пхоже, протоколы могут использоваться как типы... своего рода? Как это работает? Когда я могу использовать протокол как тип?

**Протоколы могут использоваться как типы.**

Хотя протоколы не реализуют реальную функциональность, они могут использоваться как типы. Когда вы указываете протокол как тип, можно использовать любой экземпляр, поддерживающий протокол.

```
protocol ReportingRobot: Bipedal, HasAntenna { }
```

← Еще не забыли ReportingRobot?

```
struct SecurityBot: ReportingRobot {
 // Здесь размещается реализация.
}
```

← Возьмем следующую ситуацию: есть два разных робота, которые поддерживают ReportingRobot, но используются для разных целей. Один обеспечивает безопасность, а другой следит за тем, чтобы еда не подгорела.

```
struct FoodMonitorBot: ReportingRobot {
 // Здесь размещается реализация.
}
```

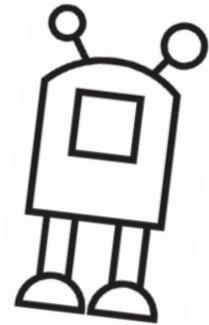
```
class Situation {
 var robot: ReportingRobot

 init(robot: ReportingRobot) {
 self.robot = robot
 }

 func observeSituation() {
 robot.walk()
 }
}
```

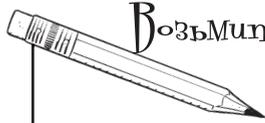
Если использовать протокол ReportingRobot как тип...

← Вы точно знаете, что некоторая функциональность всегда будет доступна.



← Независимо от ситуации, в которой она используется.

```
var securityRobot = SecurityBot(antennaCount: 2)
var foodmonitorRobot = FoodMonitorBot(antennaCount: 1)
var securityMonitoring = Situation(robot: securityRobot)
var cookingMonitoring = Situation(robot: foodmonitorRobot)
securityMonitoring.observeSituation()
cookingMonitoring.observeSituation()
```



Возьмите в руку карандаш

Решение

С. 244

```
struct FutureVehicle: Aircraft, Watercraft, PetrolPowered, Landcraft {

 var buoyancy: Int = 0
 var petrolPercent: Int = 0

 func takeOff() {
 print("Taking off!")
 }

 func land() {
 print("Landing!")
 }

 mutating func refuel() -> Bool {
 petrolPercent = 100
 return true
 }

 func drive() {
 print("Driving!")
 }

 func brake() -> Bool {
 print("Stopped!")
 return true
 }
}

var futureCar: FutureVehicle = FutureVehicle(buoyancy: 10, petrolPercent: 100)
futureCar.drive()
```

## Протоколы как типы и коллекции

Так как протоколы ведут себя как типы, их можно использовать как типы в коллекциях: *= super useful*

```
protocol Animal {
 var type: String { get }
}
```

*← Протокол Animal требует присутствия переменной типа String.*

```
struct Dog: Animal {
 var name: String
 var type: String

 func bark() {
 print("Woof!")
 }
}
```

*← Это тип Dog, поддерживающий протокол Animal.*

```
struct Cat: Animal {
 var name: String
 var type: String

 func meow() {
 print("Meow!")
 }
}
```

*← Это тип Cat, он также поддерживает протокол Animal.*

*Теперь можно создать экземпляры Dog и Cat.*

```
var buntly = Cat(name: "Buntly", type: "British Shorthair")
var nigel = Cat(name: "Nigel", type: "Russian Blue")
var percy = Cat(name: "Percy", type: "Manx")
var argos = Dog(name: "Argos", type: "Whippet")
var apollo = Dog(name: "Apollo", type: "Lowchen")
```

```
var animals: [Animal] = [buntly, nigel, percy, argos, apollo]
```

*← Так как Cat и Dog поддерживают протокол Animal, мы можем создать массив с элементами Animal и сохранить в нем экземпляры как Dog, так и Cat. Очень удобно!*

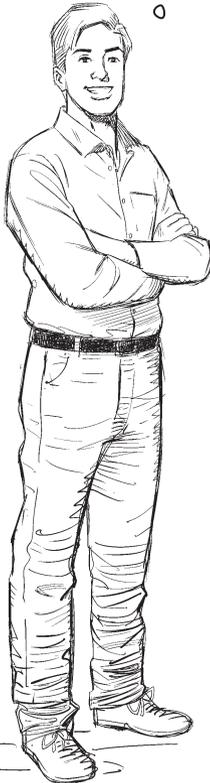


## Мозговой Штурм

Создайте цикл, который перебирает элементы массива `Animal`. Как обеспечить вызов `bark` или `meow` в зависимости от того, к какому типу относится элемент массива — `Dog` или `Cat`?

Когда это будет сделано, создайте новый экземпляр, поддерживающий `Animal`. Включите метод с именем, которое определяется звуком, издаваемым животным, добавьте его в массив `Animal` и добейтесь того, чтобы он работал в цикле.

А если мне понадобится протокол, который может поддерживаться только теми типами, которые я укажу?



**Вы можете создать протокол, который может использоваться только классами (но не структурами или чем-нибудь еще).**

Чтобы ограничить протокол только типами классов, вы можете добавить протокол `AnyObject` в его список наследования:

```
protocol SecretClassFeature: AnyObject {
 func secretClassFeature()
}
```

При попытке включить протокол `SecretClassFeature` во что-либо, кроме класса, вы получите сообщение об ошибке:

```
struct NotAClass: SecretClassFeature {
}
```

Non-class type 'NotAClass' cannot conform to class protocol 'SecretClassFeature'

```
class AClass: SecretClassFeature {
 func secretClassFeature() {
 print("I'm a class!")
 }
}
```



Кажется, протоколы — полезная штука. Но было бы неплохо иметь возможность добавлять методы в существующие типы. Как заставить классы и структуры делать то, что не было в них изначально запрограммировано?

Как ни странно, в Swift для этого существует специальный механизм: расширения.

Расширения позволяют добавить новую функциональность в существующие классы, структуры и протоколы. При помощи расширений можно добавить вычисляемые свойства, методы или инициализаторы или обеспечить поддержку протокола чем-либо.

## Анатомия расширения



### Объявление расширения

Расширение объявляется ключевым словом `extension`.

Указывается тип, который вы хотите расширить. Это расширение расширяет тип `Int`.

```
extension Int {
 func cubed() -> Int {
 return self * self * self
 }
}
```

Здесь размещается функциональность, которую вы хотите добавить.

### Использование функциональности из расширения

```
var number: Int = 5
```

Создайте экземпляр того типа, который вы расширяете.

```
number.cubed()
```

Используйте добавленную функциональность — в данном случае метод с именем `cubed`.

## Возьмите в руку карандаш



Попробуйте определить, является ли каждый из следующих примеров правильным расширением Swift. Если не является, то почему? Если является, то что делает это расширение?

- ```
extension Int {
    var even: Bool {
        return self % 2 == 0
    }
}
```
- ```
extension String {
 override func makeHaha() -> String {
 return "Haha!"
 }
}
```
- ```
extension Int {
    func cubed() -> Int {
        print(self*self*self)
    }
}
```
- ```
extension Bool {
 func printHello() {
 print("Hello!")
 }
}
```

→ Ответ на с. 253.

## Ключевые моменты

- Расширения позволяют добавлять новую функциональность в типы, которые были созданы не вами.
- Вся функциональность, добавленная в тип через расширение, работает точно так же, как если бы она была встроена в исходный тип.
- Расширения могут добавлять методы и вычисляемые свойства, но не хранимые свойства.
- Расширения также используются для организации кода. Отделение кода от функциональности типа, созданной с использованием расширений, упрощает чтение и понимание кода.

## Вычисляемые свойства в расширениях



Помните, как мы добавили вычисляемое свойство `lactoseFree` в структуру `Pizza`, когда помогали шеф-повару пиццерии?

А теперь представьте, что мы забыли это сделать. Код стал недоступен, но с ним необходимо взаимодействовать и добавить свойство `lactoseFree`. Вычисляемые свойства можно добавить при помощи **расширения**.



Перед вами структура `Pizza` без свойства `lactoseFree`:

```
struct Pizza {
 var name: String
 var ingredients: [String]
}
```

И несколько экземпляров `Pizza`:

```
var hawaiian = Pizza(name: "Hawaiian", ingredients: ["Pineapple", "Ham", "Cheese"])
var vegan = Pizza(name: "Vegan", ingredients: ["Red Pepper", "Tomato", "Basil"])
```

Свойство `lactoseFree` будет реализовано при помощи расширения.

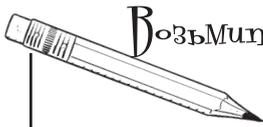
### 1 Объявление расширения

```
extension Pizza {
 var lactoseFree: Bool {
 if(ingredients.contains("Cheese")) {
 return false
 } else {
 return true
 }
 }
}
```

### 2 Использование функциональности из нашего расширения

```
print(hawaiian.lactoseFree)
print(vegan.lactoseFree)
```

↙  
false  
true



Возьмите в руку карандаш

## Решение

С. 251

Попробуйте определить, является ли каждый из следующих примеров правильным расширением Swift. Если не является, то почему? Если является, то что делает это расширение?



```
extension Int {
 var even: Bool {
 return self % 2 == 0
 }
}
```

Работает. Позволяет использовать свойство even для проверки четности целого числа.



```
extension String {
 override func makeHaha() -> String {
 return "Haha!"
 }
}
```

Не работает. Тип `String` обычно не содержит функцию `makeHaha`, поэтому переопределить ее невозможно.



```
extension Int {
 func cubed() -> Int {
 print(self*self*self)
 }
}
```

С точки зрения функциональности в основном нормально. Но поскольку функция должна возвращать значение `Int`, но не возвращает его, работать не будет.



```
extension Bool {
 func printHello() {
 print("Hello!")
 }
}
```

Работает. Позволяет вывести сообщение `hello` из `Boolean` (для чего бы это ни было нужно).



## Полезные расширения и Вы

Иногда бывает непросто понять полезность расширений или разобраться, как они интегрируются в код, который необходимо написать. Мы постараемся вам помочь.

### Полезные инициализаторы

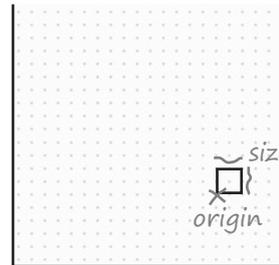
Возьмем следующие структуры, которые могут использоваться для представления прямоугольника.

```
struct Size {
 var w = 0
 var h = 0
}

struct Point {
 var x = 0
 var y = 0
}

struct Rectangle {
 var origin = Point()
 var size = Size()
}
```

```
var smallSquare =
 Rectangle(
 origin: Point(x: 15, y: 5),
 size: Size(w: 2, h: 2)
)
```

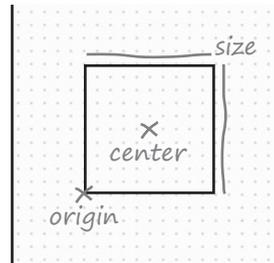


### Добавление инициализатора (с расширением)

Можно воспользоваться расширением, чтобы создать новый инициализатор для структуры Rectangle в дополнение к поэлементному инициализатору. Это позволяет создавать новые экземпляры Rectangle для представления прямоугольника конкретного размера относительно заданной центральной точки (вместо базовой точки в углу):

```
extension Rectangle {
 init(center: Point, size: Size) {
 let origin_x = center.x - (size.w/2)
 let origin_y = center.y - (size.h/2)
 self.init(origin: Point(x: origin_x, y: origin_y), size: size)
 }
}

var bigSquare =
 Rectangle(
 center: Point(x: 10, y: 10),
 size: Size(w: 10, h: 10)
)
```



## Возьмите В руку карандаш

В следующем коде встречаются пропуски. Сможете ли вы определить, что они должны содержать? Заполните пропуски и запустите код. Дополнительное задание: добавьте какие-либо методы и свойства к расширяемому типу.

Какими еще возможностями вы сможете дополнить этот тип?

```
extension ____ {

 var square : Int{

 }

 _____ -> Int{
 return self * self * self
 }

 _____ func incrementBy10() {

 }

 _____ {
 return "This Int contains the value \ (self) "
 }
}

var myInt: Int = 100
print(myInt.square)
print(myInt.cube())
myInt.incrementBy10()
print(myInt.description)
```

→ Ответ на с. 258.

## Расширение протокола

Расширения также могут использоваться для прямого добавления реализованных методов, вычисляемых свойств и инициализаторов к протоколам. Помните наш протокол `Animal`?

```
protocol Animal {
 var type: String { get }
}
```

Расширение может использоваться для прямого расширения протокола `Animal` с добавлением нового метода `eat`. При расширении протокола вы можете определить поведение в самом протоколе (вместо того, чтобы определять его в каждом типе, поддерживающем этот протокол):

```
extension Animal {
 func eat(food: String) {
 print("Eating \(food) now!")
 }
}
```

Здесь предоставляется реализация, несмотря на то что это расширение предназначено для протокола. Все типы, поддерживающие этот протокол, получают этот метод без необходимости реализовать его самостоятельно.

Если у вас имеется тип, поддерживающий расширенный протокол, в данном случае `Animal`:

```
struct Horse: Animal {
 var name: String
 var type: String
}
```

все поддерживающие типы получают реализацию метода, и вам не приходится записывать ее внутри типа:

```
var edward = Horse(name: "Edward", type: "Clydesdale")
edward.eat(food: "hay")
```

Eating hay now!



Будьте осторожны!

При помощи расширений нельзя заставить протокол расширять другой протокол или наследовать от него.

Если вы хотите заставить протокол наследовать от другого протокола, это возможно, но делать это необходимо внутри самого протокола.

## Композиция протоколов

## Полезные протоколы и вы



Иногда бывает удобно потребовать, чтобы что-то поддерживало сразу несколько протоколов. Как делалось ранее, можно создать новый протокол, который поддерживает сразу несколько других протоколов:

```
protocol ReportingRobot: Bipedal, HasAntenna { }
```

Но также можно указать сразу несколько протоколов в одном требовании к типу:

```
func doSomethingWith(robot: HasArms & HasLaserGun) {
 print("Gripping with arm!")
 robot.gripWith(arm: 1)
 print("Firing laser!")
 robot.fireLaserGun()
}
```

В параметре robot должен передаваться тип, поддерживающий как HasArms, так и HasLaserGun.

Так как мы знаем, что robot гарантированно поддерживает протокол HasArms, можно вызвать gripWith.

Аналогичным образом, так как robot заведомо поддерживает HasLaserGun, можно вызвать fireLaserGun.

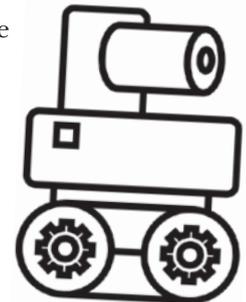
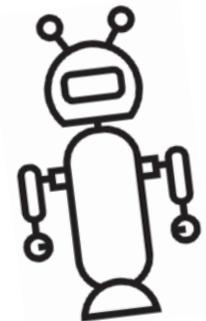
Также можно создать псевдоним типа с использованием композиции протоколов:

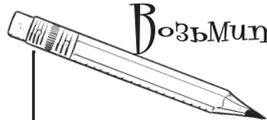
```
typealias RobotQ = Bipedal & PetrolPowered
```

```
func doSomethingElseWithA(robot: RobotQ) {
 print("Walking!")
 robot.walk()
 print("Petrol percent is \(robot.petrolPercent)")
}
```

Если тип, передаваемый методам, поддерживает оба требуемых протокола, такое решение будет работать.

Иногда бывает удобно объединять протоколы посредством композиции, чтобы упростить чтение кода и улучшить изоляцию логики. Намного проще понять, что здесь происходит, когда логика разбита на отдельные шаги. Впрочем, не стоит злоупотреблять этим.





Возьмите в руку карандаш

Решение

C.255

```
extension Int {

 var square : Int{
 return self*self
 }

 func cube() -> Int{
 return self * self * self
 }

 mutating func incrementBy10() {
 self = self + 10
 }

 var description: String {
 return "This Int contains the value \$(self)"
 }
}

var myInt: Int = 100
print(myInt.square)
print(myInt.cube())
myInt.incrementBy10()
print(myInt.description)
```



А как было бы замечательно, если для поддержки протокола можно было бы воспользоваться расширением!

Вдали от проторенных дорог



**У нас хорошие новости — это возможно!**

Используя расширение, можно обеспечить поддержку протокола существующим типом. Ничего особенно волшебного здесь нет: расширение может добавлять к типам свойства и методы, а это означает возможность выполнения любых требований, которые предъявляются протоколом.

```

struct Dog {
 var name: String
 var age: Int
}

var argos = Dog(name: "Argos", age: 7)

protocol Bark {
 func bark()
}

extension Dog: Bark {
 func bark() {
 print("Woof!")
 }
}

argos.bark()

```

Структура Dog представляет собаку.

Экземпляр Dog на базе нового типа.

Протокол Bark требует наличия метода bark.

Это расширение Dog поддерживает протокол Bark и добавляет функциональность протокола Bark в Dog.

Теперь для Dog можно вызывать bark.

**Расширения — отличный механизм организации кода для улучшения его удобочитаемости.**



**Мозговой Штурм**

Как вы думаете, что произойдет, если вы используете расширение для поддержки протокола, но расширяемый тип уже соответствует требованиям протокола (даже если он не поддерживал протокол явно до создания расширения)? Проверьте свой ответ.

## Поддержка протоколов Swift



### Последовательности

Swift предоставляет много полезных протоколов, которые вы можете поддерживать, чтобы добавлять новую функциональность в ваши собственные типы.

Например, при переборе коллекции (скажем, массива) в цикле `for-in` Swift использует специальную встроенную систему, называемую итератором, для преобразования созданного вами цикла `for-in` в скрытый внутренний цикл `while` с итератором. Swift многократно использует итератор, пока итератор не вернет `nil`, а цикл `while` не завершится. Итератор отвечает за перебор коллекции. Вы можете реализовать те же протоколы, которые Swift использует в своей внутренней реализации, чтобы предоставить ту же функциональность в ваших типах коллекций.

Для этого вам понадобятся два протокола: `Sequence` (последовательность, то есть нечто, поддерживающее последовательный перебор) и `IteratorProtocol` (нечто, способное выполнять перебор последовательности). Поддержка обоих протоколов одним типом означает, что вы должны реализовать метод `next`, который возвращает следующее значение для вашей последовательности.

### Поддержка протоколов Sequence и IteratorProtocol:

```
struct MySequence: Sequence, IteratorProtocol {
 var cur = 1

 mutating func next() -> Int? {
 defer {
 cur = cur * 5
 }
 return cur
 }
}
```

← Наш тип `MySequence` поддерживает `Sequence` и `IteratorProtocol`...

← ...это означает, что мы должны предоставить метод `next`, возвращающий следующий элемент.

← Код, включенный в `defer { }`, будет выполнен при завершении текущей области видимости. В приведенном примере он вернет `cur`, а затем обновит `cur` значением `cur`, умноженным на 5.

### Использование новой последовательности:

```
var myNum = 0
let numbers = MySequence()
for number in numbers {
 myNum = myNum + 1
 if myNum == 10 {
 break
 }
 print(number)
}
```

```
1
5
25
125
625
3125
15625
78125
390625
```

**Информацию обо всех встроенных протоколах (и о том, что необходимо для их поддержки) можно найти в документации Swift.**



## Equatable

Другой полезный встроенный протокол — Equatable — обеспечивает возможность сравнения объектов типов, определяемых пользователем, с помощью оператора ==.

### Задача

```
enum DogSize {
 case small
 case medium
 case large
}
struct Dog {
 var breed: String
 var size: DogSize
}
var whippet = Dog(breed: "Whippet", size: .medium)
var argos = Dog(breed: "Whippet", size: .medium)
var trevor = Dog(breed: "Greyhound", size: .large)
var bruce = Dog(breed: "Labrador", size: .medium)

if(whippet == argos) {
 print("Argos is a whippet!")
}
```

*Перечисление DogSize с разными вариантами размеров собак*

*Структура Dog, представляющая собаку*

*Some Dogs...*

*Проблема!*

Binary operator '==' cannot be applied to two 'Dog' operands

### Поддержка Equatable:

```
struct Dog: Equatable {
 var breed: String
 var size: DogSize
}
if(whippet == argos) {
 print("Argos is a whippet!")
}
```

*Поддержка Equatable с помощью типа Dog позволяет сравнивать объекты.*

Argos is a whippet!

### Реализация ==

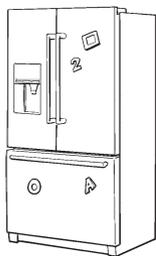
```
struct Dog: Equatable {
 var breed: String
 var size: DogSize

 static func ==(lhs: Dog, rhs: Dog) -> Bool {
 return lhs.size == rhs.size
 }
}
if(bruce == argos) {
 print("Argos and Bruce are the same size!")
}
```

*Сравниваться должны только размеры (size), а не породы (breed).*

Argos and Bruce are the same size!

Реализация Equatable, которая используется Swift по умолчанию, проверяет все свойства типа. Таким образом, если вы хотите проверить только часть свойств или некоторые из свойств еще не поддерживают Equatable, вам придется реализовать собственную версию ==.



## Развлечения с магнитами

Магниты, из которых была выложена программа Swift, перепутались на холодильнике. Сможете ли вы переставить фрагменты и создать работоспособную программу, которая генерирует результат, приведенный на следующей странице? В коде используются концепции, которые были подробно описаны ранее, а также те, что еще не рассматривались.

*Переставьте эти магниты, чтобы программа работала.*



```
var myCollection = GameCollection(gamesList: videoGames)
```

```
var videoGames = ["Mass Effect", "Deus Ex", "Pokemon Go",
"Breath of the Wild", "Command and Conquer", "Destiny 2",
"Sea of Thieves", "Fallout 1"]
```

```
struct GameCollection: EnumerateCollection {
```

```
myCollection.
enumerateCollection()
```

```
videoGames.describe()
```

```
var gamesList: [String]
```

```
}
```

```
extension Collection {
 func describe() {
 if count == 1 {
 print("There is 1 item in this collection.")
 } else {
 print("There are \(count) items in this collection.")
 }
 }
}
```

```
func enumerateCollection() {
 print("Games in Collection:")
 for game in gamesList {
 print("Game: \(game)")
 }
}
```

```
protocol EnumerateCollection {
 func enumerateCollection()
}
```

## Развлечения с магнитами, продолжение

Расставьте здесь магниты с предыдущей страницы.



There are 8 items in this collection.  
Games in Collection:  
Game: Mass Effect  
Game: Deus Ex  
Game: Pokemon Go  
Game: Breath of the Wild  
Game: Command and Conquer  
Game: Destiny 2  
Game: Sea of Thieves  
Game: Fallout 1

Результат. Сможете ли вы расставить фрагменты кода в правильном порядке? В решении должен использоваться каждый фрагмент.

→ Ответ на с. 265.



## СТАТЬ КОМПИЛЯТОРОМ Swift

На этой и следующей странице приведены фрагменты кода Swift. Представьте себя на месте компилятора Swift и определите, будут ли работать каждый фрагмент или нет. Если какие-либо фрагменты не компилируются, то как бы вы их исправили?

A

```
protocol Starship {
 mutating func performBaryonSweep()
}

extension Starship {
 mutating func performBaryonSweep() {
 print("Baryon Sweep underway!")
 }
}
```

C

```
protocol Clean { }
protocol Green { }

typealias EnvironmentallyFriendly =
 Clean & Green

struct Car: EnvironmentallyFriendly {
 func selfDrive() {
 print("Beep boop")
 }
}
```

B

```
struct Hat {
 var type: String
}

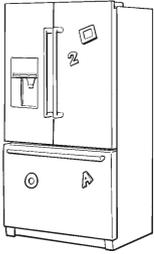
var bowler = Hat(type: "Bowler")

protocol Wearable {
 func placeOnHead()
}

extension Hat: Wearable {
 func placeOnHead() {
 print("Placing \(self.type) on head.")
 }
}

bowler.placeOnHead()
```

→ Ответ на с. 266.



## Развлечения с магнитами. Решение

С. 263

```
extension Collection {
 func describe() {
 if count == 1 {
 print("There is 1 item in this collection.")
 } else {
 print("There are \(count) items in this collection.")
 }
 }
}

protocol EnumerateCollection {
 func enumerateCollection()
}

struct GameCollection: EnumerateCollection {
 var gamesList: [String]

 func enumerateCollection() {
 print("Games in Collection:")
 for game in gamesList {
 print("Game: \(game)")
 }
 }
}

var videoGames = ["Mass Effect", "Deus Ex", "Pokemon Go", "Breath of the Wild", "Command and Conquer", "Destiny 2", "Sea of Thieves", "Fallout 1"]
videoGames.describe()
var myCollection = GameCollection(gamesList: videoGames)
myCollection.enumerateCollection()
```

There are 8 items in this collection.  
 Games in Collection:  
 Game: Mass Effect  
 Game: Deus Ex  
 Game: Pokemon Go  
 Game: Breath of the Wild  
 Game: Command and Conquer  
 Game: Destiny 2  
 Game: Sea of Thieves  
 Game: Fallout 1



# СТАТЬ компилятором Swift

## Решение

С. 264

Фрагменты А, В и С правильны!

А ничего не выводит, потому что никакие экземпляры не создаются.

В выводит «Placing Bowler on head».

С не выводит ничего, потому что никакие экземпляры не создаются.

## 9. Опциональные типы, распаковка, обобщение и другое

# Неизбежные ОПЦИОНАЛЬНЫЕ ТИПЫ



Обработка несуществующих данных может быть весьма непростым делом. К счастью, в Swift для этого существует решение: **опциональные типы**. В Swift опциональный тип позволяет работать со значением или выполнить действия **при отсутствии значения**. Это одно из многих проявлений безопасности при проектировании Swift. Ранее вы уже встречались с опциональными типами в коде, а теперь мы изучим их более подробно. **Опциональные типы улучшают безопасность Swift**, потому что с ними снижается риск написания кода, который перестает работать при отсутствии данных, или возврата значения, которое в действительности значением не является.



## ИНТЕРВЬЮ С ОПЦИОНАЛЬНЫМ ТИПОМ

### Сегодняшний собеседник: опциональный тип

**Head First:** Спасибо, что присоединились к нам. Все наши слушатели желают познакомиться с вами поближе.

**Опциональный тип:** Всегда рад. Я сейчас довольно популярен, но никогда не отказываюсь от общения. Не возражаете, если буду делать заметки во время беседы?

**Head First:** Обо мне? Но мне хотелось провести интервью, чтобы побольше узнать о вас! Хотя почему бы и нет. Как пожелаете.

**Опциональный тип:** Спасибо! Ваше любимое число?

**Head First:** У меня его нет. Извините. Говорят, вы одна из самых мощных возможностей Swift. Можете пояснить, о чем идет речь?

**Опциональный тип:** Понятно, любимого числа нет. Отмечаем. Мощных? Ну, не знаю, но у меня есть кое-какие способности. С моей помощью можно представить отсутствие данных.

**Head First:** Каких данных?

**Опциональный тип:** Любых. Вообще любых.

**Head First:** Строка?

**Опциональный тип:** Да, например, строка. Я могу представлять строку, которая либо есть, либо ее нет.

**Head First:** А это не то же самое, что пустая строка?

**Опциональный тип:** Нет. Пустая строка — это строка, которая не содержит символов. А я представляю

значение, которое может быть строкой, может быть пустой строкой или полным отсутствием строки. Понятно?

**Head First:** Кажется, да. Так говорите, любой тип данных может быть опциональным в Swift?

**Опциональный тип:** Да! Опциональный тип `Int` может принимать значения `5`, `10`, `1000` или `nil` — иначе говоря, он может вообще не существовать.

**Head First:** Таким образом, опциональное нечто может содержать любое возможное значение этого типа или `nil`?

**Опциональный тип:** Да. Представьте простой тип `Boolean`. Он может принимать только значения `true` или `false`. Но необязательный тип `Boolean` может принимать значения `be true`, `false` или `nil`.

**Head First:** И вы уверены в своей полезности?

**Опциональный тип:** Абсолютно уверен. Вскоре вы поймете. Ваш год рождения?

**Head First:** Эээ... А это важно?

**Опциональный тип:** Нет, но ведь он у вас есть, верно?

**Head First:** Да...

**Опциональный тип:** Хорошо, хорошо.

**Head First:** Что ж, спасибо за беседу!

**Опциональный тип:** Пожалуйста.



### Синтаксис под увеличительным стеклом

В синтаксисе опционального типа центральное место занимает оператор `?`. Он используется для обозначения того, что нечто является опциональным типом. Например, чтобы создать опциональное целое число, введите следующую команду:

```
var number: Int? = nil
```

Пока `number` не содержит никакого реального значения — только `nil`, потому что тип является опциональным. Если позднее вы захотите сохранить целое число в `number`, ничто не мешает это сделать:

```
number = 42
```

## Обработка отсутствующего значения

### Задача

Вы пишете программу Swift, которая предлагает пользователю ввести его любимую цитату:

```
var favoriteQuote: String = "Space, the final frontier..."
```

```
if favoriteQuote != "" {
 print("My favorite quote is: \"\(favoriteQuote)\"")
} else {
 print("I don't have a favorite quote.")
}
```

Любимая цитата хранится в виде строки.

Код проверяет введенный текст, и если он присутствует — выводит его.

Давайте начистоту... Я не вижу никаких проблем. Кажется, этот код просто нормально работает. Я что-то упускаю из виду?



**Опциональные типы позволяют чисто обработать отсутствующее или потенциально отсутствующее значение.**

**Код будет работать.**

**И к правильности нет никаких претензий.**

Проблема в том, что такой подход не соответствует стилю Swift.

Если вы запустите этот код в том виде, в каком он приведен выше, он будет работать. Он выведет строку favoriteQuote String, и все будет нормально:

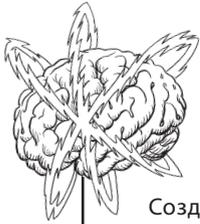
```
My favorite quote is: 'Space, the final frontier...'
```

Если у вас нет любимой цитаты и строка favoriteQuote осталась пустой, программа выведет сообщение:

```
I don't have a favorite quote.
```

Такое решение не соответствует стилю Swift. И его безопасность оставляет желать лучшего. Пустая строка на самом деле не является любимой цитатой, это просто пустая строка.

Опциональные типы решают проблему — они способны представить **потенциальное отсутствие** чего-либо.



# Мозговой Штурм

Создайте среду Playground и попробуйте выполнить код с предыдущей страницы. Он должен нормально работать. Что произойдет, если полностью удалить определение favoriteQuote? Как тогда поведет себя команда if?

## Опциональные типы в контексте



# Для чего могут понадобиться опциональные типы

Иногда бывает непросто разобраться в том, как работают опциональные типы и как встроить их в код, над которым вы работаете. Мы поможем вам с этим.



**Тун Person**

```
struct Person {
 var name: String
 var coffeesConsumed: Int
}
```

← Структура Person представляет человека.

Это Джош. Он относится к типу Person.

Это Том. Он тоже относится к типу Person.



Джош пьет много кофе. Сегодня он выпил уже 5 порций (и возможно, выпьет еще!) Том вообще не пьет кофе, поэтому лучше, что можно сделать при создании структуры, — сохранить в ней 0.

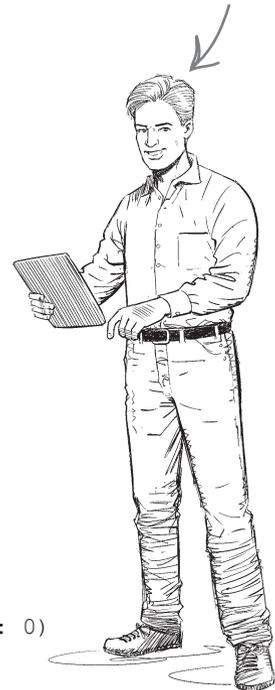


Джош в виде структуры Person.

```
var josh = Person(name: "Josh", coffeesConsumed: 5)
```

А это Том в виде структуры Person.

```
var tom = Person(name: "Tom", coffeesConsumed: 0)
```



Структура вполне неплохо представляет Джоша: в ней хранится имя «Josh» и количество выпитых чашек кофе (5).

```
var josh = Person(name: "Josh", coffeesConsumed: 5)
```

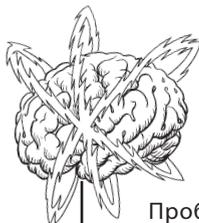


Однако структура, представляющая Тома, оставляет желать лучшего. Том не выпил 0 чашек кофе; он вообще не пьет кофе.

```
var tom = Person(name: "Tom", coffeesConsumed: 0)
```



Формально это правильно: Том не выпил ни одной чашки кофе, но существует и более точный способ представления.



## Мозговой Штурм

Проблему можно было бы решить включением в структуру логического свойства `consumesCoffee` или чего-нибудь в этом роде. Попробуйте реализовать этот путь и исправить код, прежде чем мы перейдем к рассмотрению опциональных типов.

## Возьмите в руку карандаш



Перед вами список возможных данных. Подумайте, какие из них хорошо бы подошли для хранения с использованием опциональных типов.

Если подходит, то почему? Если не подходит, то почему?

- Столица страны пользователя, регистрирующегося в вашем приложении
- Код страны для международного звонка в Австралию
- Дата рождения друга в списке контактов
- Список языков, которыми владеет пользователь
- Цвет волос пользователя
- Длина имени в символах
- Возраст собаки
- Количество страниц в книге

→ Ответ на с. 275.

## Опциональные типы и обработка отсутствующих данных

**Person**

```
struct Person {
 var name: String
 var coffeesConsumed: Int?
}
```

Небольшое изменение: `coffeesConsumed` теперь относится к опциональному типу `Int`.

**Алекс и Том**

```
var alex = Person(name: "Alex", coffeesConsumed: 5)
var tom = Person(name: "Tom")
```

А Алекс пьет кофе.

Том не пьет кофе, и задавать значение `coffeesConsumed` вообще не нужно, потому что оно относится к опциональному типу `Int`.



Ладно, все понятно, но ведь это все равно `Int`? Мне нужно сделать что-то особенное, чтобы добраться до `coffeesConsumed` — теперь, когда тип стал опциональным?

**Да, чтобы получить доступ к значению опционального типа, его необходимо распаковать.**

Так как опциональный тип (для конкретности будем говорить об опциональном `Int`, как в случае с `coffeesConsumed`) может содержать либо `Int` (например, 5), либо `nil` (то есть вообще ничего), его необходимо **распаковать перед использованием**.

С `Int` можно выполнять ряд ожидаемых операций (например, арифметических), которые не могут выполняться с `nil`. Даже если бы Swift разрешал такие операции, это было бы небезопасно.

Итак, чтобы получить доступ к сочной сердцевине опционального типа, его сначала нужно распаковать!

**Распаковка опциональных типов играет ключевую роль в их использовании. Чтобы безопасно использовать опциональные типы, их всегда необходимо распаковывать.**

## Распаковка опциональных типов



Чтобы понять, как выполняется распаковка опциональных типов, взгляните на новую улучшенную структуру Person:

```
struct Person {
 var name: String
 var coffeesConsumed: Int?
}
```

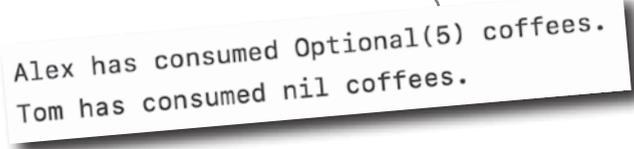
Экземпляры, созданные на базе этой структуры:

```
var alex = Person(name: "Alex", coffeesConsumed: 5)
var tom = Person(name: "Tom")
```

А теперь подумайте, что произойдет, если вывести свойства этих двух экземпляров структуры Person:

```
print("\(alex.name) consumed \(alex.coffeesConsumed) coffees.")
print("\(tom.name) consumed \(tom.coffeesConsumed) coffees.")
```

Вместо того чтобы вывести 5, программа выводит Optional(5) — далеко не идеальный результат. Нужно число присутствует, но оно упаковано в конструкцию Optional().



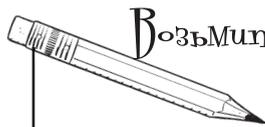
Это значение относится к типу Optional<Int>, а не Int. Если вы хотите работать с Int, необходимо распаковать опциональный тип:

```
if let coffees = alex.coffeesConsumed {
 print("The unwrapped value is: \(coffees)")
} else {
 print("Nothing in there.")
}
```

Использование *if let* распаковывает переменную с проверкой условия.

Если здесь присутствует значение, оно будет присвоено coffees.

Если в распаковываемом типе значение отсутствует, будет выполнено условие else.



Возьмите в руку карандаш

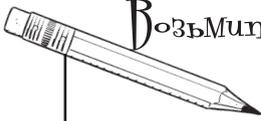
## Решение

С. 212

- Столица страны пользователя, регистрирующегося в вашем приложении  
*Нельзя исключать, что пользователь введет несуществующую страну. Хороший кандидат для опционального типа.*
- Код страны для международного звонка в Австралию  
*Всегда существует — плохой кандидат.*
- День рождения друга в списке контактов  
*День рождения может быть неизвестен. Хороший кандидат для применения опционального типа.*
- Список языков, которыми владеет пользователь  
*Каждый говорит на каком-нибудь языке. Плохой кандидат.*
- Цвет волос пользователя  
*А если пользователь лысый? Хороший кандидат.*
- Длина имени в символах  
*У каждого имени есть длина. Плохой кандидат.*
- Возраст собаки  
*Возраст определенно существует. Плохой кандидат.*
- Количество страниц в книге  
*В любой книге есть страницы. Плохой кандидат для опционального типа.*

## Ключевые моменты

- Опциональные типы позволяют обрабатывать факт отсутствия данных.
- Любой другой тип (включая типы, созданные вами) может создаваться в опциональной версии.
- Опциональная версия типа создается оператором `?.`
- Опциональный тип может содержать все, что может содержать обычный тип, или же значение `nil`, представляющее полное отсутствие какого-либо значения.
- Обращение к данным в опциональном типе требует дополнительного шага, который называется распаковкой.



## Возьмите В руку карандаш

Попробуйте определить, что выводит каждый из следующих фрагментов (и выводит ли). Если фрагмент ничего не выводит, то какие минимальные изменения нужно внести, чтобы он что-то выводил?

- ```

var magicNumber: Int? = nil
magicNumber = 5
magicNumber = nil
if let number = magicNumber {
    if(number == 5) {
        print("Magic!")
    }
} else {
    print("No magic!")
}

```

- ```

var soupOfTheDay = "French Onion"
if let soup = soupOfTheDay{
 print("The soup of the day is \(soup)")
} else {
 print("There is no soup of the day today!")
}

```
  
- ```

let mineral: String? = "Quartz"
if let stone = mineral {
    print("The mineral is \(stone)")
}

```

- ```

var name: String? = "Bob"
if let person = name {
 if(person=="Bob") {
 print("Bye Bob!")
 }
}

```

→ Ответ на с. 279.

# Распаковка опциональных

## типов с ключевым словом guard

Опциональные типы в контексте



### 1 Иногда требуется и далее использовать распакованное значение

Иногда требуется быть полностью уверенным в том, что вы получили распакованное значение, чтобы и далее использовать его в оставшейся части области видимости, в которой оно было распаковано.

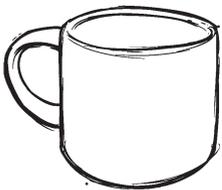
### 2 Ключевое слово guard позволяет это сделать

Заменяв `if` ключевым словом `guard` в синтаксисе, который использовался ранее для распаковки опционального типа, вы сможете распаковать опциональный тип или выйти из текущей области видимости, если это невозможно.

### Ключевое слово guard позволяет распаковать опциональный тип:

```
func order(pizza: String?, quantity: Int) {
 guard let unwrappedPizza = pizza else {
 print("No specific pizza ordered.")
 return
 }
 var message = "\(quantity) \(unwrappedPizza) pizzas were ordered."
 print(message)
}
```

Если опциональное значение `pizza` не содержит значения, управление выходит за пределы функции `order`.



### Расслабьтесь

Вероятно, в большинстве случаев для распаковки опциональных типов вы будете использовать `if let`.

Конструкцию `guard let` удобно использовать для распаковки опциональных типов в начале функции или метода. После проверки вы точно знаете, что можно продолжать работу с данными, а для любого опционального типа, который не содержит реального значения, управление было выведено за пределы метода. Вы можете расслабиться, потому что все необходимые значения заведомо присутствуют. Другими словами, `guard let` позволяет проверить выполнение некоторых условий, связанных с опциональными типами, прежде чем продолжать выполнение. `guard let` не будет работать без ключевого слова `return`.

## Принудительная распаковка



Я слышала истории об операторе Swift, который настолько опасен, что... его вообще не стоило бы использовать. Его называют «катастрофическим оператором». О чем идет речь?

**Катастрофический оператор — не миф. Он существует.**

И на самом деле он называется **принудительной распаковкой**. Как и следовало ожидать, этому оператору присвоено обозначение `!`.

Оператор принудительной распаковки позволяет пропустить все проверки и процессы, которые обычно необходимы для того, чтобы добраться до желанного значения внутри опционального типа.

Допустим, у вас имеется опциональный тип `String`:

```
let greeting: String? = "G'day mates!"
```

Его можно принудительно распаковать оператором `!`:

```
print(greeting!) → G'day mates!
```

С другой стороны, если вообще не применять распаковку, тип останется опциональным:

```
print(greeting)
```

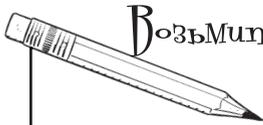
```
Optional("G'day mates!")
```



Будьте осторожны!

**Оператор называется «катастрофическим» не просто так.**

*Применяйте принудительную распаковку только в том случае, если вы абсолютно уверены, что значение присутствует и может использоваться так, как вы намерены его использовать.*



# Возьмите в руку карандаш

## Решение

С. 276

```

 var magicNumber: Int? = nil
 magicNumber = 5
 magicNumber = nil
 if let number = magicNumber {
 if (number == 5) {
 print("Magic!")
 }
 } else {
 print("No magic!")
 }

```

Выводит сообщение «No magic!»

**No magic!**

```

 var soupOfTheDay = "French Onion"
 if let soup = soupOfTheDay {
 print("The soup of the day is \(soup)")
 } else {
 print("There is no soup of the day today!")
 }

```

Ничего не делает, так как использует if let с обычной, а не опциональной строкой.

Чтобы исправить ошибку, сделайте soupOfTheDay опциональным типом String.

```

 let mineral: String? = "Quartz"
 if let stone = mineral {
 print("The mineral is \(stone)")
 }

```

Выводит сообщение «The mineral is Quartz!»

**The mineral is Quartz**

```

 var name: String? = "Bob"
 if let person = name {
 if (person=="Bob") {
 print("Bye Bob!")
 }
 }

```

Выводит сообщение «Bye Bob!»

**Bye Bob!**



# Безопасность в Swift

Сегодняшний урок

## Принудительная распаковка опциональных типов

Пример ситуации, в которой можно применить принудительную распаковку, — когда вы включаете в программу фиксированное значение, которое заведомо будет работать, и сохраняете его в виде опционального типа.

Возьмем следующий код:

```
let linkA = URL(string:"https://www.oreilly.com")
```

URL — класс из библиотеки Apple Foundation Library, представляющий URL-адрес (например, ссылку на веб-сайт). В данном случае в нем хранится URL-адрес известного издателя технической литературы. Это правильно сформированный URL-адрес (неважно, работает он или нет): он содержит символы / и точки в нужных местах.

Но класс URL не знает, что вы всегда будете передавать строку с правильно сформированным URL-адресом. Ведь вы можете поступить и так:

```
let linkB = URL(string: "I'm a lovely teapot")
```

Так как «I'm a lovely teapot» — абсолютно нормальная строка, которая не является действительным, правильно сформированным URL-адресом, в URL будет значение store nil. Присваивание linkB означает, что в linkB будет храниться опциональный URL-адрес (потому что если URL может возвращать либо URL-адрес, либо nil, будет логично возвращать опциональную версию URL).

При попытке использования переменной:

```
print(linkB) → nil
```

...выводится nil, потому что linkB содержит nil.

Но при попытке использования linkA:

```
print(linkA) → Optional(https://www.oreilly.com)
```

...вы получите опциональный URL.

При попытке принудительно распаковать URL при присваивании произойдет ошибка:

```
let linkC = URL(string: "I'm a lovely teapot")!
```

Unexpectedly found nil while unwrapping an Optional value

Дело в том, что использование оператора ! заверяет Swift, что опциональный тип определенно содержит значение, а не nil, тогда как эта переменная содержит nil.

Если вы проведете принудительную распаковку URL с действительным URL-адресом (потому что вы точно знаете, что с ним все нормально), решение будет работать:

```
let linkC = URL(string:"https://www.oreilly.com")!
print(linkC) → https://www.oreilly.com
```



## СТАТЬ КОМПИЛЯТОРОМ Swift

Представьте себя на месте компилятора Swift, рассмотрим следующий код с использованием опциональных типов и определим, будет ли он компилироваться или же использование (или отсутствие) опционального типа приведет к ошибке/фатальной ошибке. Не торопитесь. Предполагается, что все фрагменты выполняются независимо и ничего не знают друг о друге.

typealias Moolah = Int

**A**

let bankBalanceAtEndOfMonth: Moolah? = 764

var statementBalance = bankBalanceAtEndOfMonth!

**B**

var moolah = 100

func addMoney(amount moolah: Int) -> Int? {

    return nil

}

var myMoney = (moolah + addMoney(amount: 10)!)

**C**

func countPasswordChars(password: String?) -> Int {

    let pass = password!

    return pass.count

}

print(countPasswordChars(password: "IAcceptTheRisk"))

**D**

struct Starship {

    var shipClass: String

    var name: String

    var assignment: String?

}

let ship1 =

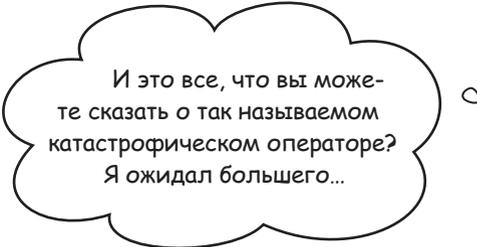
    Starship(shipClass: "GSV", name: "A Very Bad Idea", assignment: "Contact")

let ship2 =

    Starship(shipClass: "GSU", name: "Lack of Morals", assignment: nil)

print("Assignment of \(ship2.shipClass) \(ship2.name) is: \(ship2.assignment!)")

→ Ответ на с. 289.



Вообще-то с ним можно выполнять и другие операции, например неявную распаковку.

Неявная распаковка позволяет определить значение, которое будет вести себя так, словно оно вообще не является опциональным и не нуждается в распаковке — но при этом остается опциональным.

Для этого можно добавить оператор ! после типа в объявлении:

```
var age: Int! = nil
```

Когда вы используете опциональный тип с неявной распаковкой, он будет вести себя так, словно он уже был распакован. Но при попытке использовать его, когда он содержит nil, программа аварийно завершится. Будьте осторожны:

```
print(age) → nil
age = age + 1 → Unexpectedly found nil while unwrapping an Optional value
age = 42
print(age) → Optional(42)
age = age + 1
print(age) → Optional(43)
```



*Будьте осторожны!*

**По возможности используйте обычные опциональные типы.**

*Использовать опциональные типы с неявной распаковкой очень удобно, если вы абсолютно уверены, что это не создаст проблем, потому что наличие значения гарантировано. Но, скорее всего, это будет только вашим предположением.*

*Будьте осторожны, используйте классические опциональные типы и распаковывайте их самостоятельно, если это возможно.*



## Анатомия сцепления опциональных типов

Сцепление опциональных типов — одна из многих полезных возможностей Swift, формально представляющая собой сокращенную запись для функциональности, которую можно реализовать другим, более длинным способом.

### Задача

*Вы пишете код, который работает с массивами данных о посещениях концертов.*

```
var bobsConcerts = ["Queen at Live Aid", "Roger Waters — The Wall"]
var tomsConcerts = [String]()
```

*По какой-то причине вы хотите прочитать из массива первый посещенный концерт для каждого из этих людей и сохранить его как часть сообщения, но при этом преобразовать к ВЕРХНЕМУ РЕГИСТРУ. Можно поступить так...*

```
var message: String
if let bobFirstConcert = bobsConcerts.first {
 message = "Bob's first concert was \(bobFirstConcert.uppercased())"
}
```

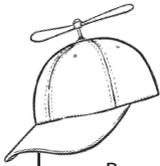
### Решение

Сцепление опциональных типов предоставляет сокращенную запись для операций, выполняемых между другими опциональными типами (или сцепленными с ними). Для этого между другими переменными вставляется оператор `?:`:

```
let bobsFirstConcert = bobsConcerts.first?.uppercased()
let tomsFirstConcert = tomsConcerts.first?.uppercased()
```

*Swift проверит, содержит ли средний компонент цепочки (значение, возвращаемое при вызове `.first` для массива `array`) определенное значение (то есть что он отличен от `nil`). Если он содержит `nil`, то оставшаяся часть цепочки (вызов `uppercased()` в данном случае) выполняться не будет. Если он содержит значение, то цепочка продолжит выполняться так, словно была выполнена распаковка.*

**Сцепление опциональных типов позволяет пройти через несколько уровней опциональности в одной понятной строке кода. Если окажется, что какой-либо из опциональных уровней содержит `nil`, то и вся строка даст результат `nil`. Удобно, правда?**



## Серьезное программирование

Взгляните на приведенный ниже код. В нем определяются структуры `Person` и `Song`. У каждого человека (`Person`) есть имя (`name`), любимая песня (`favoriteSong`) и любимая песня для караоке (`favoriteKaraokeSong`). Однако значение `favoriteKaraokeSong` является опциональным, потому что не все любят караоке. Также мы создали для вас несколько экземпляров `Person`.

```
struct Person {
 var name: String
 var favoriteSong: Song
 var favoriteKaraokeSong: Song?
}
```

*Оператор ? обозначает опциональный тип.*

```
struct Song {
 var name: String
}

let paris =
 Person(name: "Paris",
 favoriteSong: Song(name: "Learning to Fly – Pink Floyd"),
 favoriteKaraokeSong: Song(name: "Africa – Toto"))
```

```
let bob =
 Person(name: "Bob",
 favoriteSong: Song(name: "Shake It Off – Taylor Swift"))
```

```
let susan =
 Person(name: "Susan",
 favoriteSong: Song(name: "Zombie – The Cranberries"))
```

Напишите код, который выводит значение `favoriteKaraokeSong` для трех экземпляров `Person`, которые мы создали для вас, с использованием команды `if`. Если значение отсутствует, выведите соответствующее сообщение.

Когда это будет сделано, создайте дополнительные экземпляры структуры `Person` и посмотрите, удастся ли вам преобразовать ваш код в функцию.

Опциональные типы можно применить для создания объектов, которые не будут созданы?



**Да, это называется «инициализатором с возможным отказом».**

Представьте, что вы пишете структуру Shape для представления геометрической фигуры. В ней должно храниться количество сторон... и все.

Структура содержит метод printShape(), который когда-нибудь будет рисовать фигуру на экране, а пока просто выводит количество сторон у фигуры.

Допустим, ваша структура Shape выглядит так:

```
struct Shape {
 var sides: Int
 func printShape() {
 print("Shape has \(sides) sides.")
 }
}
```

А если вы хотите запретить создание экземпляров Shape, имеющих менее 3 сторон? Создать фигуру с таким количеством сторон все равно затруднительно, поэтому, вероятно, такие ситуации следует предотвращать. И здесь на помощь приходит инициализатор с возможным отказом! Если вы добавите такой инициализатор в структуру Shape, он может возвращать nil при попытке создания экземпляра Shape с менее чем 3 сторонами:

```
init?(sides: Int) {
 guard sides >= 3 else { return nil }
 self.sides = sides
}
```

У инициализатора с возможным отказом за ключевым словом init следует символ ?.

Команда guard используется для возвращения nil, если значение sides меньше 3.

Если выполнение дошло до этой точки, то нормальное присваивание значений свойствам в инициализаторе будет безопасным.

Теперь в программе можно безопасно создавать как действительные, так и недействительные фигуры:

```
var box = Shape(sides: 4)
var triangle = Shape(sides: 3)
var triquandle = Shape(sides: -4)
```

Вернет Optional<Shape> (с 4 сторонами).

Вернет Optional<Shape> (с 3 сторонами).

Вернет nil.



**Мозговой Штурм**

Попробуйте изменить инициализатор с возможным отказом, чтобы он завершился неудачей при попытке создания фигуры, имеющей более 9 сторон.



## Приведение опциональных типов

В Swift есть полезное ключевое слово **as?**, с помощью которого можно выполнить приведение типа. Если приведение прошло удачно, возвращается преобразованный тип, а при неудаче возвращается `nil`. Ключевое слово `as?` проверяет, относится ли значение к некоторому опциональному типу; если относится, его можно использовать, а если нет, вы получите `nil`.

```
class Bird {
 var name: String
}

class Singer: Bird {
 func sing() {
 print("\(self.name) is singing! Singing so much!")
 }
}

class Nester: Bird {
 func makeNest() {
 print("\(self.name) made a nest.")
 }
}

let birds = [Bird(name: "Cyril"), Singer(name: "Lucy"),
 Singer(name: "Maurice"), Nester(name: "Cuthbert")]

for bird in birds {
 if let singer = bird as? Singer {
 singer.sing()
 }
}
```

← Класс Bird. Не содержит ничего, кроме свойства name.

← Субкласс для певчих птиц Singer

← Метод sing для певчих птиц

← Субкласс для птиц, вьющих гнезда, — Nester

← Метод makeNest для птиц, вьющих гнезда

← Массив разных субклассов Bird и обычных экземпляров Bird

← Метод sing должен вызываться только для экземпляров Singer.

← Ключевое слово `as?` используется для приведения каждого элемента массива к типу `Singer`. Если это `Singer`, то мы получаем значение, а если нет, получаем `nil`, и метод не вызывается. Удобно и безопасно.

```
Lucy is singing! Singing so much!
Maurice is singing! Singing so much!
```



## Анатомия оператора объединения с nil

Пора познакомиться с новым оператором! Знакомьтесь: оператор объединения с nil. Он полезен в тех ситуациях, когда вы работаете с опциональными типами и хотите быть полностью уверенными в том, что он содержит значение.

### Задача

Если у вас имеется опциональный тип с некоторыми данными...

```
var dogBreed: String? = "Beagle"
print("Look at that cute \(dogBreed)!")
```

... и вы применяете оператор ! для его принудительной распаковки при использовании, все нормально.

Look at that cute Beagle!

Но если у вас имеется опциональный тип, не содержащий значения, в нем хранится nil...

```
var dogBreed: String?
print("Look at that cute \(dogBreed)!")
```

...и вы захотите использовать оператор ! для его принудительной распаковки, то столкнетесь с проблемой.

Unexpectedly found nil while unwrapping an Optional value

### Решение

```
var dogBreed: String?
print("Look at that cute \(dogBreed ?? "doggo")!")
```

Оператор объединения с nil — ?? — распаковывает опциональный тип, и если он содержит nil, то предоставляет заданное значение по умолчанию.

```
var dogBreed: String? = "Beagle"
print("Look at that cute \(dogBreed ?? "doggo")!")
```

Look at that cute Beagle!

## Обобщения

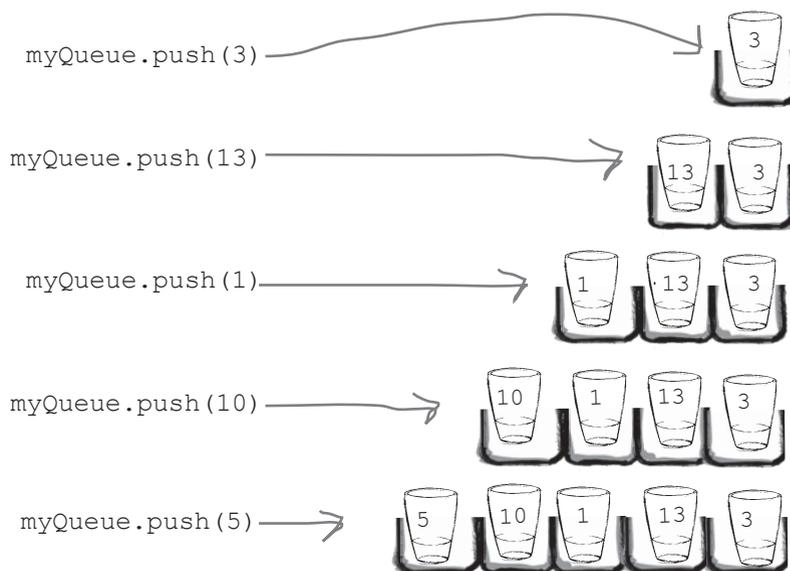
Очень важно, чтобы написанный вами код был гибким и подходил для повторного использования. Обобщения чрезвычайно эффективно работают на достижение желанных целей гибкости и возможности повторного использования. **Обобщения позволяют писать функции и типы, способные работать буквально с любым типом, который удовлетворяет определенным вами требованиям.**

Прекрасный пример такого рода – создание новых типов данных, которые могут работать по-разному в зависимости от конкретных потребностей. Типы коллекций Swift, такие как `Array`, действуют именно так, и вы уже неоднократно использовали функциональность Swift, основанную на обобщениях. А сейчас давайте создадим собственный тип коллекции.

Построим **очередь**: тип данных, работающий по принципу «первым зашел, первым вышел» (как и очереди в реальном мире). Элементы всегда добавляются в конец очереди, а удаление всегда выполняется в ее начале.

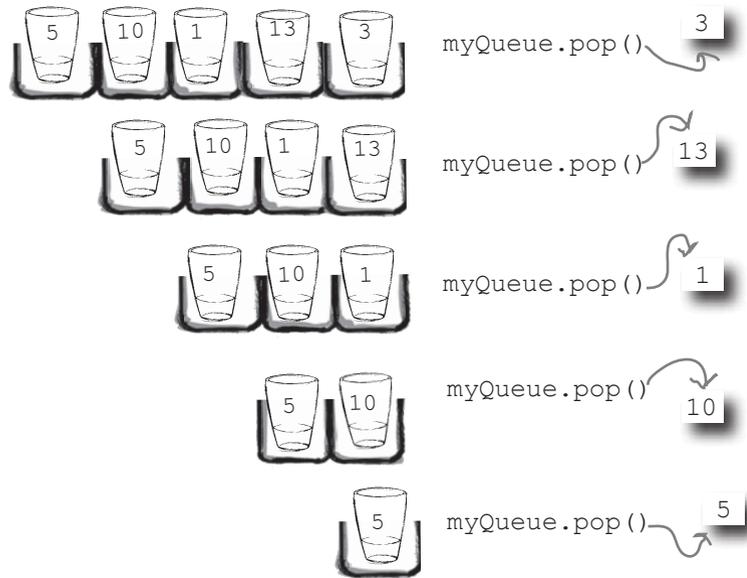
### Требования к очереди

#### 1 Занесение новых элементов в конец очереди



## Очередь с обобщениями

### 2 Извлечение элементов в начале очереди (с удалением)



### 3 Возможность получения количества элементов в очереди

```
myQueue.push(5)
myQueue.push(11)
myQueue.count
```

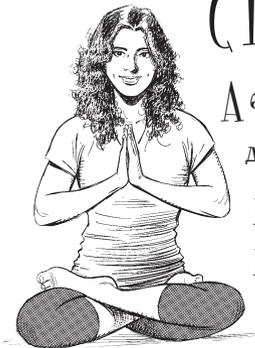
2

### 4 Возможность создания очереди с элементами любого типа (Int, String и т. г.)

... и здесь на помощь приходят обобщения!

## СТАТЬ КОМПИЛЯТОРОМ Swift. Решение

С. 281



А будет работать. В не будет работать, потому что принудительная распаковка возвращенного значения попытается распаковать значение nil. С будет работать. D не будет работать, потому что принудительная распаковка второго присваивания попытается распаковать значение nil.

## Новый тип Queue

Это и есть обобщение. Имя условного типа-заполнителя заключается в угловые скобки < и >.

В нашей очереди реализованы все четыре требования.

```

struct Queue<T> {
 private var arrayRepresentation = [T]()

 var count: Int {
 return arrayRepresentation.count
 }

 mutating func push(_ item: T) {
 arrayRepresentation.append(item)
 }

 mutating func pop() -> T? {
 if arrayRepresentation.count > 0 {
 return arrayRepresentation.removeFirst()
 } else {
 return nil
 }
 }
}

```

T представляет обобщенный тип. Вместо него может быть подставлен любой другой тип.

В реализации Queue здесь будет использоваться массив с элементами типа, представленного обозначением T.

Свойство count реализации Queue возвращает количество элементов во внутреннем массиве.

Функция push позволяет добавить элемент в очередь — элемент типа T.

А функция pop возвращает элемент типа T из начала внутреннего массива (если он есть).

### Использование Queue

```

var stringQueue = Queue<String>()
stringQueue.push("Hello")
stringQueue.push("Goodbye")
print(stringQueue.pop()!)

```

Создание очереди (с элементами String в данном случае)

В очередь заносятся два элемента String.

Извлечение (и вывод) элемента в начале очереди

Оptionальный результат подвергается принудительной распаковке, потому что в данном случае мы полностью уверены в наличии значения. Но в реальной программе следовало бы действовать по правилам.



И... это все? Или обобщения могут делать что-то еще?

**Отличный вопрос. Классический пример использования обобщений относится к функциям.**

Обобщенные функции — это функции, способные работать с любым типом.

Следующая функция меняет местами два значения `Int`:

```
func switchInts(_ one: inout Int, _ two: inout Int) {
 let temp = one
 one = two
 two = temp
}
```

Теперь представьте, что вам нужно создать аналогичную функцию, которая меняет местами два значения `Double`, два значения `String` или любого другого типа. Для каждого случая придется писать практически одинаковые функции. На помощь приходят обобщения.

Та же функция, переписанная в обобщенном виде:

```
func switchValues<T>(_ one: inout T, _ two: inout T) {
 let temp = one
 one = two
 two = temp
}
```

Теперь функция `switchValues` может поменять местами любые пары значений независимо от их типа:

```
var a = 5
var b = 11
switchValues(&a, &b)
print(a)
print(b)
```

**Обозначать обобщенный тип именем `T` необязательно. Вы можете использовать любой тип по своему усмотрению. `T` — всего лишь общепринятое соглашение. Проявите фантазию!**

Часть  
Задаваемые  
Вопросы

**В:** Кажется, я понял, что такое опциональные типы. Они позволяют представить нечто такое, что может присутствовать и относиться к определенному типу либо может вообще отсутствовать?

**О:** Да. Опциональные типы представляют наличие или отсутствие значения. Они понятны и просты в использовании.

**В:** И опциональный тип нужно распаковать, чтобы получить доступ к фактическому значению?

**О:** Да, для получения доступа к значению опционального типа его необходимо распаковать. Для этого можно использовать конструкцию `if let` или `guard let`.

**В:** Можно ли использовать принудительную распаковку опциональных типов?

**О:** Да, для принудительной распаковки опционального типа можно воспользоваться оператором `!`. Но если вы выполните принудительную распаковку и получите `nil` (то есть отсутствие значения), то в программе произойдет фатальная ошибка.

**В:** А неявно распакованные опциональные типы, по сути, автоматически распаковывают сами себя?

**О:** Да. Вы можете создать опциональный тип, который распаковывается неявно. Все выглядит так, словно распаковка происходит автоматически, но, по сути, он просто лишается специальных средств безопасности опциональных типов.

**В:** А что со сцеплением опциональных типов? Насколько я понимаю, мы делаем что-то с опциональным типом и игнорируем остаток кода, если в опциональном типе окажется `nil`?

**О:** Все точно. Сцепление опциональных типов — безопасный механизм выполнения операций с опциональным типом, которые могут быть заключены между другими переменными, не являющимися опциональными.

**В:** И объединение с `nil` — всего лишь способ предоставить заведомо безопасное значение по умолчанию на случай, если опциональный тип содержит `nil`?

**О:** Верно. Объединение с `nil` — всего лишь хитроумный способ сказать «а здесь предоставляется значение по умолчанию для `nil` в опциональном типе».

**В:** А если я создаю объект, который использует опциональные типы, я могу создать инициализатор с возможным отказом, чтобы обеспечить его безопасность?

**О:** Да, суть инициализатора с возможным отказом заключается в том, что вы можете добавить оператор `?` к ключевому слову `init` при определении инициализатора. Тем самым гарантируется, что весь объект не может быть создан с некорректными входными данными.

**В:** И наконец, приведение типа... Это просто преобразование одного типа к другому?

**О:** Верно. Приведение типа сообщает Swift, что вы хотите получить некоторое значение в виде другого типа.

**В:** Кажется, опциональные типы — очень полезная штука. Почему их нет в других языках программирования?

**О:** Хороший вопрос. Мы понятия не имеем.

## Ключевые моменты

- `nil` представляет отсутствие значения.
- Использование оператора `!` с типом (например, `Int!`) предполагает, что значения этого типа не обязательно распаковывать перед использованием — при условии, что соответствующая переменная не содержит `nil`.
- Добавление оператора `?` к ключевому слову `init` позволяет создать инициализатор с возможным

отказом, то есть инициализатор, который вообще не завершится, если в нем произойдет сбой из-за некорректных данных.

- Опциональные типы являются ключевым аспектом безопасности Swift. Благодаря им вы можете безопасно работать с данными, которые могут отсутствовать.

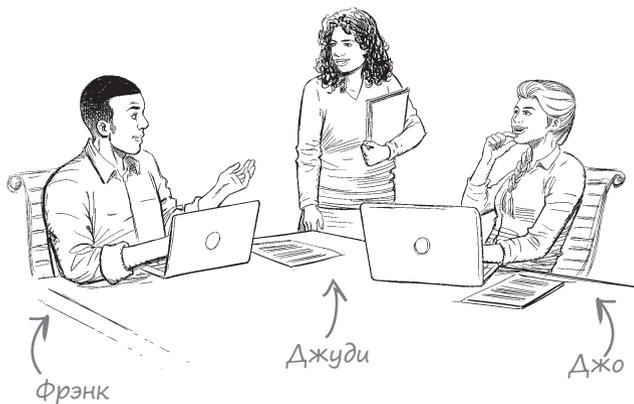
## 10. Знакомство со SwiftUI

# Пользовательские интерфейсы



Пришло время применить на практике все приемы, возможности и компоненты **Swift**, о которых вы узнали в книге: мы займемся построением пользовательских интерфейсов. В этой главе мы сведем все воедино для построения первого настоящего пользовательского интерфейса. Он будет строиться на основе **SwiftUI**, **UI-фреймворка для платформ Apple**. Мы по-прежнему будем использовать Playgrounds (по крайней мере, на первом этапе), но все, что здесь будет делаться, заложит фундамент для реальных приложений iOS. Приготовьтесь: в этой главе будет много кода и новых концепций. Вдохните поглубже и переверните страницу, чтобы с головой погрузиться в **SwiftUI**.

## А что это вообще такое — UI-фреймворк?



**Фрэнк:** Кто-нибудь знает, что такое UI-фреймворк?

**Джуди:** Ну это... фреймворк... для пользовательских интерфейсов?

**Джо:** Это набор средств для прорисовки визуальных элементов на экране. В нашем случае он называется SwiftUI, а при его создании использовались фундаментальные составляющие Swift, которые изучались ранее.

**Фрэнк:** И как он работает?

**Джуди:** И насколько он удобен?

**Джо:** Вы строите пользовательский интерфейс на программном уровне, используя структуры, свойства и все такое.

**Джуди:** Как построить пользовательский интерфейс из структур?

**Фрэнк:** Полагаю, мы будем использовать протоколы? Чтобы структуры соответствовали протоколу, предоставляемому SwiftUI?

**Джо:** Точно. Именно так оно работает.

**Джуди:** И какие элементы пользовательского интерфейса можно использовать? Кнопки? Что еще?

**Джо:** Кнопки, конечно. Текстовые поля для ввода текста. Списки. Графические поля и множество других полезных элементов.

**Фрэнк:** И как правильно разместить элементы, если пользовательский интерфейс строится на программ-

ном уровне? Я хочу сказать, что, когда я строил пользовательские интерфейсы в других средах программирования, у меня была возможность строить макеты в визуальном редакторе. А как правильно разместить элементы, если это делается в коде?

**Джо:** Элементы можно легко позиционировать и строить макеты при помощи таких инструментов, как `NavigationView`, `VStack` для вертикальных контейнеров, `HStack` для горизонтальных контейнеров, и т. д. Встраивая эти представления в другие представления, можно достаточно точно управлять позицией элементов.

**Джуди:** И для каких платформ можно строить пользовательские интерфейсы на базе SwiftUI?

**Фрэнк:** Я думаю, этот фреймворк работает в iOS, macOS, tvOS и watchOS.

**Джуди:** Стало быть, все платформы Apple...

**Джо:** Вот именно. Этот фреймворк предназначен только для платформ Apple, но он очень хорош. И он позволяет чрезвычайно легко строить чистые, хорошо спроектированные приложения.

**Фрэнк:** Ладно, убедил. С чего мне начинать изучение SwiftUI?

**Джуди:** И меня убедил. За дело!

**Джо:** К счастью, у меня для вас есть несколько советов вводного уровня. Вы найдете их на следующей странице...

Я уже устал от текстовых программ.  
Хочу графики, хочу нормальных  
пользовательских интерфейсов.  
Что для этого потребуется?



### Для этого потребуется SwiftUI.

SwiftUI позволяет создавать графические интерфейсы (иногда называемые GUI) в простом и ясном коде Swift.

Перед вами **полный код программы** с пользовательским интерфейсом (на базе SwiftUI):

```
import SwiftUI
import PlaygroundSupport

struct ContentView: View {
 @State var count = 0

 var body: some View {
 VStack {
 Button(action: { self.count += 1 }, label: {
 Text("Press Here")
 .padding()
 })

 if count > 0 {
 Text("The button has been pressed \(count) times.")
 } else {
 Text("The button has not been pressed.")
 }
 }
 }
}

PlaygroundPage.current.setLiveView(ContentView())
```



## Тест-драйв

Создайте новую среду Playgrounds и выполните приведенный выше код.  
Что вы видите?

## Ваш первый пользовательский интерфейс SwiftUI UI

```
import SwiftUI
import PlaygroundSupport

struct HuzzahView: View {
 var body: some View {
 Text("Huzzah!")
 }
}

PlaygroundPage.current.setLiveView(ContentView())
```

Помните команды `import` с предыдущей страницы?

Чтобы создать представление, следует объявить его тип (в данном случае `HuzzahView`), поддерживающий протокол `View`.

Переменная `body`, наличие которой требует протокола, является вычисляемым свойством. Мы используем ее для предоставления контента, который должен быть выведен на экран.

В вычисляемом свойстве `body` мы объединяем примитивные представления, предоставляемые фреймворком SwiftUI, чтобы вывести на экран нужные элементы.

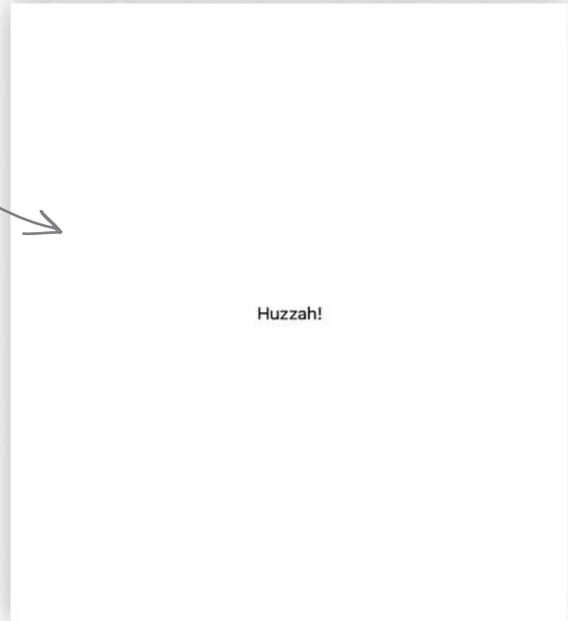
В данном случае создается текстовое представление (`Text`), инициализированное текстом «Huzzah».

А теперь нужно приказать Playground вывести представление, которое должно появиться на экране. В данном случае это представление `HuzzahView`.

**Запустите и посмотрите, что произойдет.**  
Создайте новую среду Swift Playground и введите в ней приведенный выше код. Затем выполните код Playground.



Некрасиво, остается слишком много свободного места. Но работает!



**SwiftUI — декларативный UI-фреймворк, отличающийся высокой эффективностью. Код легко читается, и по нему можно в целом представить, как будет выглядеть полученный интерфейс.**

## Анатомия представления



### 1 Строка import

SwiftUI необходимо импортировать. Строка import сообщает Swift, что вы хотите использовать фреймворк SwiftUI.

```
import SwiftUI
import PlaygroundSupport

struct HuzzahView: View {
```

### 2 Структура с именем HuzzahView

Создается структура с именем HuzzahView (имя выбрано произвольно), поддерживающая протокол View. View — протокол SwiftUI для всего, что отображается на экране.

```
 var body: some View {
 Text("Huzzah!")
 }
}
```

### 3 Свойство body

Вычисляемое свойство body имеет тип some View. Это означает, что оно возвращает (содержит) нечто, поддерживающее протокол View.

### 4 Содержимое представления

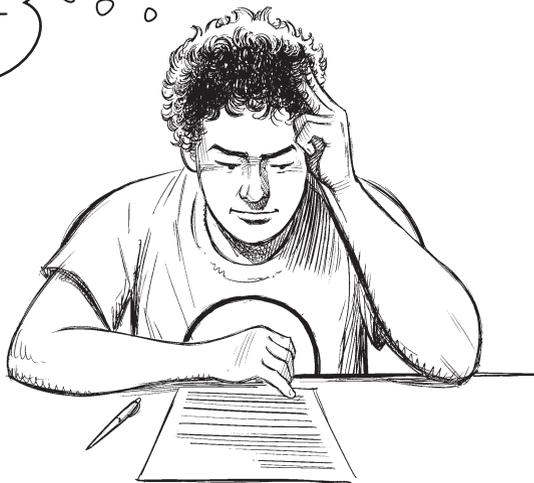
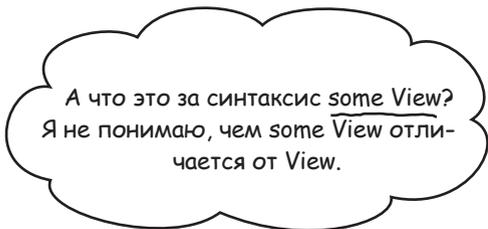
Здесь выводится текст с использованием представления Text. Это соответствует требованию о том, что свойство body должно возвращать some View.



Будьте осторожны!

**Фреймворк SwiftUI прост в использовании, но в работе с ним легко ошибиться. Впрочем, исправить ошибку очень легко!**

*SwiftUI намного капризнее самого кода Swift. Легко допустить ошибку, из-за которой вы получаете какое-то невразумительное сообщение, а возникшая ошибка не всегда будет напрямую связана со строкой, в которой эта ошибка была допущена. Не торопитесь и будьте осторожны.*



**Это так называемый непрозрачный возвращаемый тип.**

Чтобы понять, почему используется эта конструкция, проще рассмотреть пример кода. Вместо `some View` возвращаемым типом `body` может быть конкретный тип, *поддерживающий View*, например `Text`:

```
struct MyView: View {
 var body: Text {
 Text("Hello!")
 }
}
```

Но поскольку фреймворк SwiftUI разрабатывался так, чтобы вы могли собирать представления из нескольких представлений, реальное представление SwiftUI не ограничивается представлением `Text`, даже если в нем выводится только представление `Text`, где обычно находится элемент `VStack`, `HStack` или нечто похожее для его позиционирования.

И здесь мы возвращаемся к **обобщениям** Swift. Все типы, поддерживающие `View`, которые вы используете, являются обобщенными типами. Конкретный тип изменяется в зависимости от того, что содержится внутри обобщенного типа.

Например, если у вас имеется `VStack` с представлением `Text` и представлением `Image`, то тип `VStack` будет автоматически определен как `VStack<TupleView<(Text, Image)>>`.

Но если заменить представление `Image` представлением `Text`, автоматически определяемый тип определяется как `VStack<TupleView<(Text, Text)>>`.

Таким образом, если бы мы использовали конкретный тип для представления, возвращенного `body`, то нам пришлось бы вручную изменять его каждый раз, когда мы что-то изменяем в `body`.

Гораздо проще сказать, что `body` возвращает **some view**.

*Распаковка многоуровневых иерархий типов стремительно усложняется. К счастью, вам не придется беспокоиться об этом!*

```
var body: VStack<TupleView<(Text,Text)>> {
 VStack {
 Text("Hello!")
 Text("I'm more text!")
 }
}
```

## Часть Задаваемые Вопросы

**В:** Я не понимаю, SwiftUI является частью Swift?

**О:** SwiftUI — фреймворк для Swift. Он был построен компанией Apple и спроектирован так, чтобы вы могли строить пользовательские интерфейсы для платформ Apple: iPadOS, iOS, tvOS, watchOS и macOS. SwiftUI — фреймворк с закрытым кодом, доступный только при использовании Swift на iPad с Playgrounds или на машине с macOS, использующей Playgrounds или Xcode. В будущем ситуация может измениться, но сейчас она такова.

**В:** Выходит, SwiftUI не распространяется с открытым кодом, как Swift?

**О:** Верно. SwiftUI распространяется с закрытым кодом.

**В:** Означает ли это, что я не смогу использовать SwiftUI для построения приложений для Linux или Windows?

**О:** Пока да. SwiftUI может использоваться только для построения приложений на платформе Apple.

**В:** Я слышал о фреймворке UIKit. Как он связан с SwiftUI?

**О:** UIKit — другой UI-фреймворк компании Apple для iOS и iPadOS. Он все еще существует, все еще обновляется и все еще прекрасно работает, но не является платформенным фреймворком на базе Swift (хотя и может использоваться со Swift). На него распространяются те же ограничения, обусловленные закрытостью кода и платформенными ограничениями. Время от времени мы будем использовать представления UIKit из SwiftUI в этой книге, но книга не научит вас пользоваться UIKit.

**В:** Значит, SwiftUI заменяет UIKit?

**О:** Нет! Отдельные части фреймворка SwiftUI построены на базе UIKit (но не все). Иногда при использовании SwiftUI вы на самом деле используете UIKit. Но знать это вам не обязательно. SwiftUI — полнофункциональный фреймворк, мощный и полностью построенный на платформе Swift. Вас вообще не должно интересовать, использует ли SwiftUI фреймворк UIKit для отображения созданных вами пользовательских интерфейсов, так как вы просто используете SwiftUI.

**В:** А насколько быстро работает SwiftUI?

**О:** SwiftUI работает очень быстро. Во внутренней реализации все представления, которые вы создаете с SwiftUI, оптимизируются для максимально быстрой и эффективной прорисовки, а многие используют сверхбыстрый фреймворк Metal от Apple для прорисовки с использованием GPU.

**В:** Придется ли мне в какой-то момент заняться изучением UIKit?

**О:** Если вы намерены углубленно заниматься разработкой приложений для платформ Apple, то после чтения этой книги вам стоит освоить UIKit. Но начинать стоит с изучения SwiftUI.

**В:** Ранее вы назвали SwiftUI «декларативным» фреймворком. Что это значит?

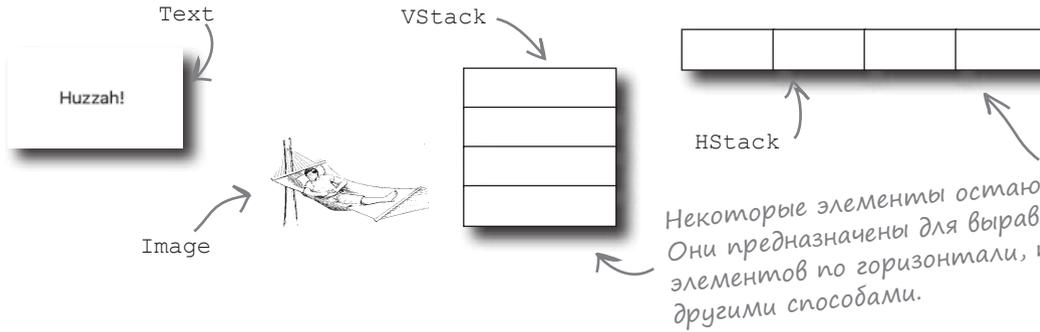
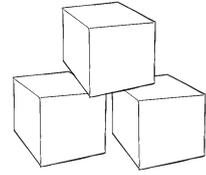
**О:** SwiftUI действительно является декларативным UI-фреймворком (еще бывают императивные). В этой главе вы узнаете, что это значит, а пока в двух словах: вы определяете набор правил и состояний и то, как переходить между ними, а SwiftUI определяет все остальное.

## Ключевые моменты

- Для использования SwiftUI необходимо импортировать SwiftUI.
- SwiftUI представляет собой структуру, поддерживающую протокол View. Протокол View предоставляется SwiftUI.
- Пользовательский интерфейс состоит из нескольких, обычно вложенных представлений SwiftUI.
- Код и поведение пользовательского интерфейса очень тесно связаны друг с другом, и SwiftUI широко использует такие концепции Swift, как протоколы.
- Некоторые представления SwiftUI предлагают популярные UI-элементы (например, кнопки), другие предоставляют невидимые элементы формирования макета, такие как `VStack` для вертикального выравнивания и `HStack` для горизонтального выравнивания.
- Простой пользовательский интерфейс с минимумом кода строится очень легко и быстро. Тем не менее с усложнением интерфейса код стремительно усложняется. Не спешите и будьте внимательны.

## Строительные блоки пользовательских интерфейсов

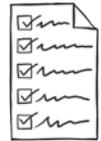
Как и любой хороший инструментарий, SwiftUI содержит обширный набор инструментов, из которых можно построить фантастический интерфейс. Вот лишь некоторые из них:



*Некоторые элементы остаются невидимыми. Они предназначены для выравнивания других элементов по горизонтали, по вертикали или другими способами.*

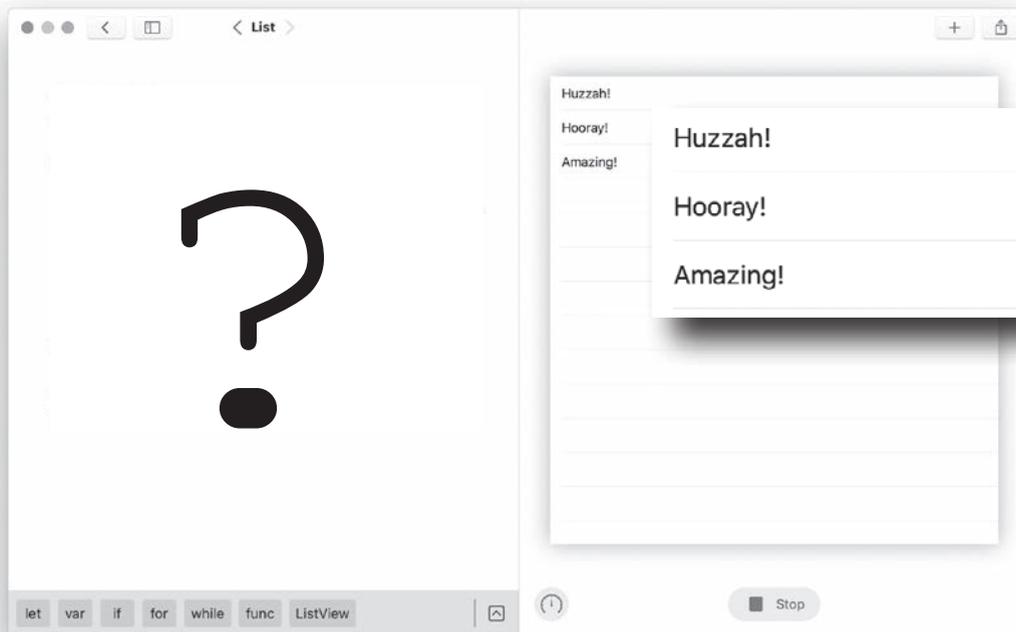
|                  |                                                                                                                                                                                                                                           |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Text</b>      | Представление <code>Text</code> выводит одну или несколько строк текста. Текст представления <code>Text</code> доступен только для чтения, он не может редактироваться пользователем.                                                     |
| <b>TextField</b> | Представление <code>TextField</code> предоставляет текстовое поле с возможностью редактирования. Оно позволяет связать значение с представлением, которое обновляется в соответствии с данными, введенными пользователем в представлении. |
| <b>Image</b>     | Представление <code>Image</code> предназначено для вывода графического изображения.                                                                                                                                                       |
| <b>Button</b>    | Представление <code>Button</code> создает и отображает кнопку. С кнопкой связывается надпись (текст, выводимый на кнопке) и действие — метод или замыкание, вызываемое при касании или щелчке на кнопке.                                  |
| <b>Toggle</b>    | Представление <code>Toggle</code> может переключаться между двумя состояниями: установленным (включенным) и сброшенным (отключенным). С ним можно связать логическое свойство, отражающее состояние переключателя.                        |
| <b>Picker</b>    | Представление <code>Picker</code> отображает представление, содержащее набор взаимоисключающих значений. Одно из значений может быть выбрано пользователем, и эту информацию можно передать нужному приемнику.                            |
| <b>Slider</b>    | Представление <code>Slider</code> создает элемент для выбора значения из ограниченного диапазона. С элементом можно связать значение, которое автоматически обновляется при перемещении ползунка.                                         |
| <b>Stepper</b>   | Представление <code>Stepper</code> отображает элемент, при помощи которого пользователь может последовательно увеличивать или уменьшать значение.                                                                                         |

## Создаем список (и неоднократно возвращаемся к нему, чтобы довести до совершенства)



Теперь, когда вы немного узнали о том, как работает SwiftUI (а вскоре узнаете гораздо больше), мы займемся построением чуть более сложного интерфейса.

**Пользовательский интерфейс, который мы строим:**



### Последовательность действий для построения UI:

- 1 **Создание структуры для представления**  
Структура представления должна поддерживать протокол `View`.
- 2 **Реализация требования протокола: свойства `body`**  
Протокол `View`, который мы включаем, требует, чтобы структура представления содержала вычисляемое свойство с именем `body`.
- 3 **Создание списка**  
SwiftUI предоставляет контейнер `List`, который автоматически предоставляет строки данных, выстроенные в один столбец.
- 4 **Включение представлений `Text` в список**  
Используя представления `Text`, мы создаем столько блоков текста, сколько должно отображаться в списке.



## Мозговой Штурм

Создадим представление List с предыдущей страницы средствами SwiftUI. При этом нам понадобятся некоторые концепции, описанные ранее в книге. Для заполнения некоторых пробелов мы будем использовать SwiftUI.

### Подготовка

Как обычно, вся работа будет выполняться в среде Swift Playground. Создайте такую среду, а в ней импортируйте SwiftUI и PlaygroundSupport.



### Шаг 1: Создание представления

Перейдем к созданию списка. Первое, что для этого понадобится, — структура, которая послужит представлением для списка. Эта структура должна включать протокол View.

- \* Создайте структуру и присвойте ей имя, подходящее для представления, содержащего список.

- \* Настройте структуру, чтобы она включала протокол View.

### Шаг 2: Объявление body

Чтобы на экране действительно что-то появилось, нам понадобится свойство body. Свойство body необходимо для соблюдения требований протокола View, поддерживаемого структурой.

- \* Создайте переменную с именем body, которая возвращает View. Пока замыкание остается пустым.

### Шаг 3: Создание списка

Главное, что необходимо разместить в нашем представлении, — список. Таким образом, на этом шаге фактически достаточно добавить этот список в замыкание свойства body.

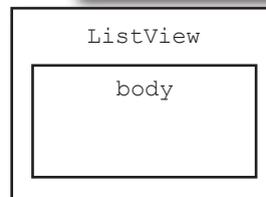
- \* Создайте контейнер List, пока оставьте его пустым (мы заполним его на следующем шаге).

### Шаг 4: Включение данных в список

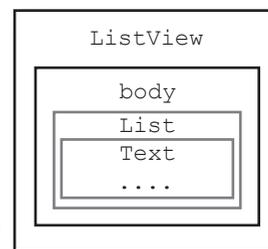
Список, который мы хотим построить, содержит три текстовых блока. На этой стадии в контейнер List будут включены следующие элементы:

- \* Добавьте представление Text с текстом "Huzzah!"
- \* Добавьте представление Text с текстом "Hooray!"
- \* Добавьте представление Text с текстом "Amazing!"

Наша структура представления будет называться ListView. Но вы можете присвоить ей любое имя по своему усмотрению.



Замыкание свойства body возвращает «some View» — оно должно возвращать один конкретный тип, поддерживающий протокол View, но нас не интересует, какой именно.



Контейнер List будет создавать статический список с элементами, поддерживающими протокол View.

И не забудьте приказать среде Playground отобразить представление (ListView в данном случае).

## Пользовательские интерфейсы с состоянием

Чтобы от пользовательского интерфейса была хоть какая-то польза, он должен отражать состояние чего-либо: некоторой логики, вычислений, данных от пользователя, данных с удаленного сервера... **некоторого состояния, критичного для пользовательского интерфейса.**

Согласно философии SwiftUI, *представление является функцией своего состояния.* В SwiftUI код, который создает представление, тесно связан с кодом, который это состояние интерпретирует. Более того, часто это вообще один и тот же код.

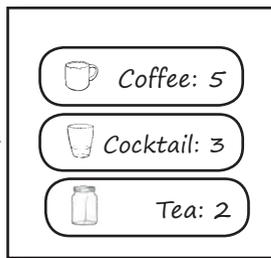
Представьте, что вы строите приложение для отслеживания потребления кофе, чая и коктейлей.

Требования выглядят так:

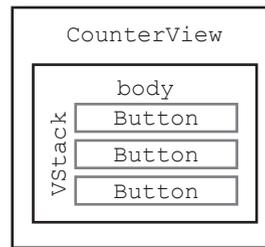
- \* На экране отображаются три кнопки: на одной кнопке выводится текст «Coffee», на другой – «Cocktail» и на третьей «Tea».
- \* На каждой кнопке выводится число, которое является частью текста на кнопке.
- \* Число представляет количество напитков каждого типа. При нажатии кнопки число увеличивается на единицу.

Приложение может выглядеть примерно так:

Каждая кнопка увеличивает свой счетчик.



Реализация представления Counter может выглядеть примерно так:



**Представление является функцией своего состояния.**

Это просто означает, что при прорисовке представления учитывается состояние, соответствующее этому представлению.



Будьте осторожны!

### Состояние поддерживается только во время выполнения кода.

Здесь речь идет о состоянии кода во время его выполнения. Мы не говорим о сохранении состояния приложения, когда выполнение кода прерывается.

Состояние хранится в оперативной памяти устройства, на котором выполняется код. Оно никогда не записывается на диск. Если код будет перезапущен, происходит возврат к состоянию по умолчанию.

## Как работают кнопки

Прежде чем разбираться в том, как построить это приложение, необходимо знать, как работают представления Button в SwiftUI. На самом деле они работают примерно так же, как и представления Text:

Создаем кнопку.

```
Button(action: {
 print("The button was pressed!")
}) {
 Text("This is a Button")
}
```

С кнопкой должно быть связано действие: код, который выполняется при ее нажатии.

Замыкание предоставляется как действие.

В данном случае действием является вызов print, который сообщает о нажатии кнопки.

Представление также реализуется в виде замыкания. В сущности, это и есть кнопка.

Кнопка будет представлена на экране текстом. Внутри кнопки может содержаться практически любое представление, так что вместо него, например, можно вывести графическое изображение.

Также существует сокращенный синтаксис для создания чисто текстовой кнопки — скорее всего, это будет самой распространенной разновидностью кнопок, которую вы будете создавать чаще всего. Он выглядит так:

```
Button("I'm also a Button") {
 print("The other button was pressed!")
}
```

Кнопке нужен текст, который будет отображаться на экране.

Замыкание предоставляется как действие.

Снова создается кнопка.

Действие — еще один вызов print, который сообщает о нажатии кнопки.

### Возьмите в руку карандаш

Подумайте, как бы вы написали код для создания описанного выше приложения. Как бы вы структурировали его в SwiftUI на основании того, что вам уже известно? Какие представления SwiftUI вам для этого понадобятся?

А когда это будет сделано, обратитесь к реальному коду и поэкспериментируйте в среде Swift Playgrounds. Шаги, которые вам могут понадобиться:

- Создайте структуру представления SwiftUI и присвойте ей имя, подходящее для приложения. Добавьте требуемое вычисляемое свойство body, чтобы реализовать внутри него нужное вам представление.
- Внутри структуры представления создайте три переменные для хранения трех счетчиков. Проследите за тем, чтобы изначально они содержали значение 0.
- Внутри body создайте панель VStack, чтобы элементы пользовательского интерфейса были выстроены по вертикали, один над другим.
- Внутри VStack создайте три представления Button. В каждой кнопке текст надписи должен строиться из эмодзи, названия напитка и счетчика для этого названия. Действие каждой кнопки должно увеличивать соответствующий счетчик на единицу.

Для формирования надписи можно воспользоваться строковой интерполяцией.

После этого (не забудьте использовать PlaygroundSupport для вывода представления) переверните страницу, и мы обсудим ваше решение. Не огорчайтесь, если на этой стадии у вас что-то не работает.

# Возьмите в руку карандаш

## Решение

Вероятно, ваше решение будет в той или иной степени похоже на следующий код. Сравните его со своей реализацией. Похоже? **А почему этот код не работает?**

```
import SwiftUI
import PlaygroundSupport

struct DrinkCounterView: View {
 var coffeeCount = 0
 var cocktailCount = 0
 var teaCount = 0

 var body: some View {
 VStack {
 Button("☕ Coffee: \(coffeeCount)") {
 self.coffeeCount += 1
 }
 Button("🍸 Cocktails: \(cocktailCount)") {
 self.cocktailCount += 1
 }
 Button("🍵 Tea: \(teaCount)") {
 self.teaCount += 1
 }
 }
 }
}

PlaygroundPage.current.setLiveView(DrinkCounterView())
```

← Все необходимые команды импортирования

← Структура представления SwiftUI

← Внутри представления определяются переменные для хранения состояния всех трех счетчиков. Каждый счетчик инициализируется нулем.

← Свойство body

← Панель VStack с тремя кнопками. Элементы внутри панели выстраиваются по вертикали.

← В тексте каждой кнопки строковая интерполяция используется для вывода соответствующего счетчика...

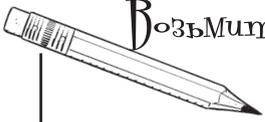
← ...А соответствующая переменная-счетчик увеличивается.

← Приказываем среде Playground вывести созданное нами представление счетчика.

Казалось бы, все должно работать: с учетом того, что вы узнали о SwiftUI к настоящему моменту, все выглядит логично. Представление создается правильно, мы правильно используем VStack для размещения трех представлений Button друг над другом. Каждая кнопка выводит переменную count, которая находится внутри структуры представления, и правильно увеличивает переменную. Так почему же код не работает?

Ответ кроется в сообщении об ошибке, которое вы получите при попытке выполнить этот код.

Left side of mutating operator isn't mutable: 'self' is immutable



## Возьмите в руку карандаш

### Решение, продолжение

Сообщение об ошибке указывает, что значение `self` является неизменяемым — в данном случае `self` является структурой `BirdCounterView`, которую мы создали.

Так как `BirdCounterView` является структурой, ее можно было создать как константу. Значение константы не может изменяться, и это распространяется на значения свойств структуры, которая была создана как константа.

Если бы наша структура содержала метод, который мог изменять значения свойств, то этот метод должен быть объявлен с ключевым словом `mutating` (как вы узнали в главе 6).

Но так как мы работаем с представлением SwiftUI, `body` является *вычисляемым свойством*. К вычисляемым свойствам ключевое слово `mutating` применяться не может.

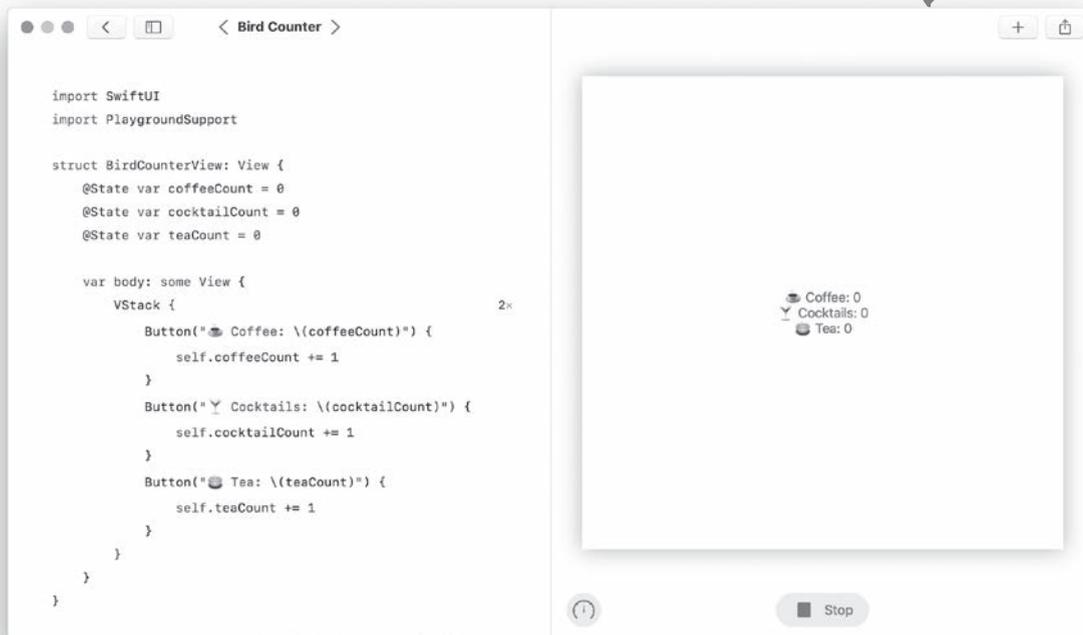
Однако вам необходимо иметь возможность изменять значения переменных, которые являются частями состояния. Как же решить эту проблему?

Для этого можно пометить свойства специальной *оберткой свойства* с именем `@State`:

```
@State private var coffeeCount = 0
@State private var cocktailCount = 0
@State private var teaCount = 0
```

С атрибутом обертки свойства `@State` Swift будет хранить значение помеченного свойства отдельно, причем так, чтобы его можно было изменять. Также следует добавить тег управления доступом `private`, так как предполагается, что свойства будут использоваться только внутри этого представления.

Если вы внесете в написанный ранее код необходимые изменения, добавьте атрибут `@State` и метку управления доступом `private`, то вы сможете запустить этот код, а кнопки будут увеличивать значения правильных счетчиков.

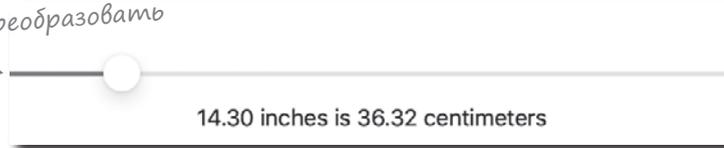


## Давайте посмотрим, насколько далеко мы зашли

Вы изучили основы программирования на Swift, а также освоили основные принципы и структурные элементы SwiftUI. Пришло время сделать что-то практичное!

Наша цель — написать программу SwiftUI для простого перевода дюймов в сантиметры. Результат ее работы должен выглядеть примерно так:

Ползунок для выбора значения в дюймах, которое требуется преобразовать



Текст с исходным значением в дюймах и преобразованным значением в сантиметрах

И в дюймах, и в сантиметрах!

Вы уже знаете, как использовать обертку свойства `@State` для работы со значением, задействованным в представлении. Здесь есть только одна новая концепция — использование элемента SwiftUI `Slider`.

Ползунок (представление `Slider`) в SwiftUI создается следующим образом:

Также понадобится замкнутый диапазон, представляющий шкалу перемещения ползунка.

```
Slider(value: $sliderValue, in: 1...10, step: 1)
```

Ползунок должен быть связан с переменной. В этой переменной хранится текущее значение элемента.

И еще приращение `step` — величина изменения значения при смещении ползунка на один шаг.

Представление `Slider` можно настраивать и оформлять по тем же принципам, что и большинство представлений SwiftUI. Например, если понадобится заменить цвет оформления ползунка на красный, это можно сделать так:

```
Slider(value: $sliderValue, in: 1...10, step: 1)
```

```
.accentColor(Color.red)
```

Это называется модификатором...



Синтаксис `$` используется в Swift для связывания значений с элементами. Он широко применяется в SwiftUI.

Ползунок также можно заключить в рамку:

```
Slider(value: $sliderValue, in: 1...10, step: 1)
```

```
.border(Color.red, width: 3)
```

И это тоже модификатор.



## Мозговой Штурм

Вы уже умеете создавать представления SwiftUI, поэтому мы создали за вас представление с именем `SliderView`. Вам остается лишь реализовать пользовательский интерфейс SwiftUI, описанный на предыдущей странице, в среде Swift Playground.

Сначала необходимо добавить в `SliderView` свойство, помеченное оберткой свойства `@State`. Оно будет использоваться для хранения значения в дюймах, представленного ползунком.

В реализации `body` внутри `VStack` создается представление `Slider`. Присвойте `value` созданное свойство, `range` — какое-нибудь разумное значение (например, от 0 до 100, без отрицательных значений), а `step` — приращение при каждом перемещении ползунка (например, 0.1).

Наконец, также нужно добавить текст для описания преобразования. Вам остается указать имя свойства для хранения дюймов; все остальное мы сделали за вас.

```
import SwiftUI
import PlaygroundSupport

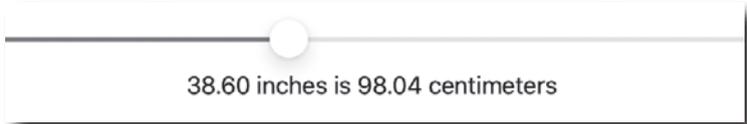
struct SliderView: View {
 [redacted]

 var body: some View {
 VStack {
 [redacted]
 Text("\([redacted], specifier: "%.2f") inches is
 \([redacted] * 2.54, specifier: "%.2f")
 centimeters")
 }
 }
}

PlaygroundPage.current.setLiveView(SliderView())
```

*Этот спецификатор форматирует результат с двумя знаками в дробной части double, чтобы вывод был более аккуратным.*

Вы поймете, что решение работает, когда сможете запустить код в среде Playground, выбрать значение при помощи ползунка и получить значение в сантиметрах:



# Настройка представлений при помощи модификаторов

## Модификаторы в контексте



- 1** Элементы представлений, используемые Swift по умолчанию, смотрятся неплохо  
И все же обычно при выборе оформления нам хочется выйти за их пределы. Мы хотим, чтобы внешний вид элементов соответствовал дизайну и теме наших программ. Мы хотим изменять и настраивать их.
- 2** Модификаторы представлений позволяют... модифицировать представления  
Модификатор представления берет представление, к которому он применен, изменяет его в соответствии с некоторой спецификацией и возвращает представление нужного вам вида.

### Использование модификаторов представлений:

```
Text("Big Red Text")
 .font(.headline)
 .foregroundColor(.red)
 .padding()
```

Big Red Text

### Порядок важен (иногда):

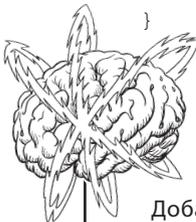
Big Red Text

```
Text("Big Red Text")
 .font(.headline)
 .foregroundColor(.red)
 .background(Color.green)
 .padding()
```

Big Red Text

```
Text("Big Red Text")
 .font(.headline)
 .foregroundColor(.red)
 .padding()
 .background(Color.green)
```

Фон занимает больше места, потому что сначала вызывается padding.

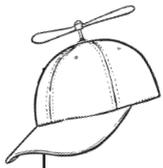


## Мозговой Штурм

Добавьте другой цвет фона (не зеленый) после padding в первом примере с цветом фона. Попробуйте предположить, где будет выводиться новый фон и как он будет соотноситься с другим (зеленым) фоном. Запустите код и проверьте свое предположение.

Мне вечно не хватает времени. У меня в списке задач есть пункт «Написать приложение для ведения списка задач», только я об этом постоянно забываю, потому что записываю свой список задач на салфетке. Напишите мне приложение! Пожалуйста!



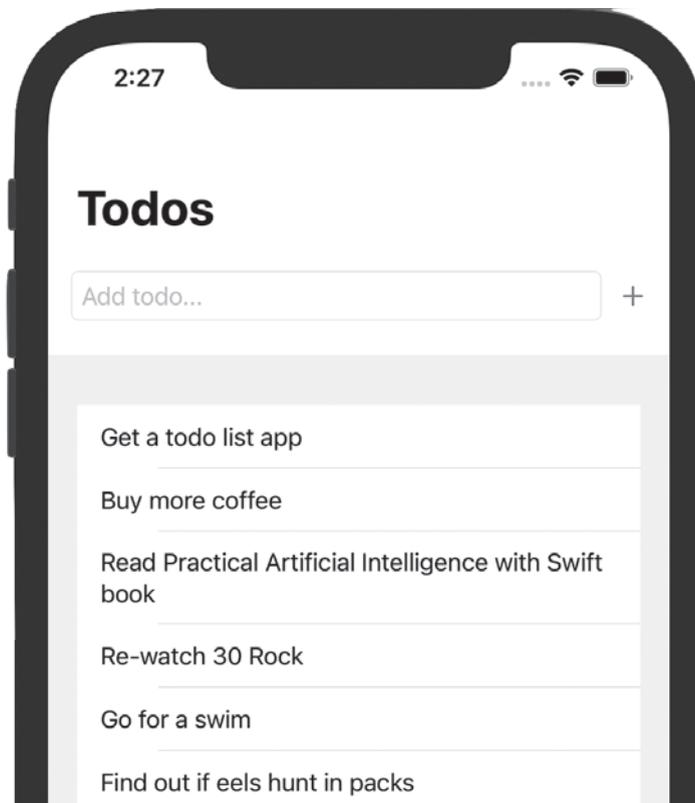


## ЗАДАЧА: Написать приложение на базе SwiftUI

В работе над приложением будет использоваться Xcode — мощная среда разработки для macOS, предназначенная для Swift (и других языков).

- Создайте в XCode новый проект SwiftUI для iOS.
- Создайте новый тип для хранения отдельной задачи из списка.
- Позаботьтесь о том, чтобы каждый элемент списка задач имел уникальный идентификатор.
- Создайте пользовательский интерфейс для приложения: текстовое поле для добавления новых задач, кнопку для их сохранения и список для вывода всех задач.
- Реализуйте функцию сохранения списка задач, чтобы он не терялся между запусками приложения.

*Повторите эти шаги и постройте приложение со списком задач! На нескольких ближайших страницах мы покажем, как это делается.*



*Приложение, которое мы строим. Неплохо, не правда ли?*

*Поле для ввода новых задач*

*Список всех добавленных задач*

## Создание в Xcode нового проекта SwiftUI для iOS

Хватит разговоров. Перейдем к построению списка задач. Прежде всего нужно создать проект Xcode. (Если у вас еще не установлена среда Xcode, начните с ее установки.)



- 1 Запустите Xcode.
- 2 На экране *Welcome to Xcode* выберите вариант **Create a new Xcode project**.



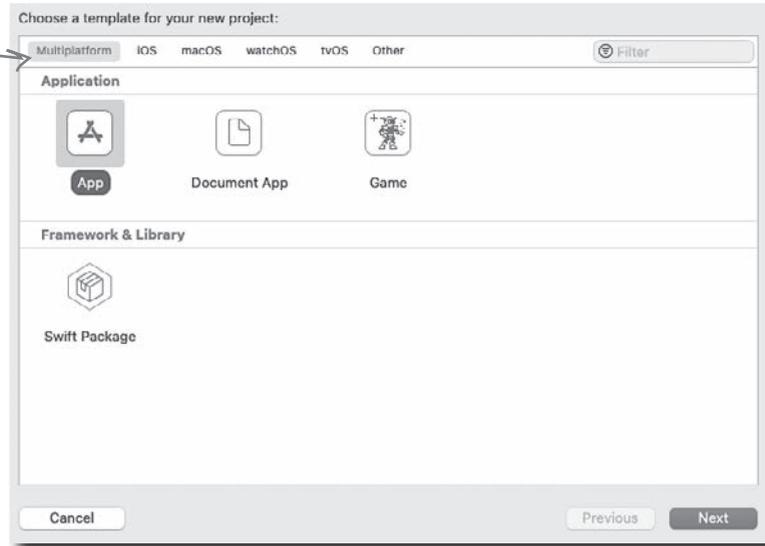
Если вы не видите экран «Welcome to Xcode», его можно открыть командой *New -> Project...* из меню *File*.

### Ключевые моменты

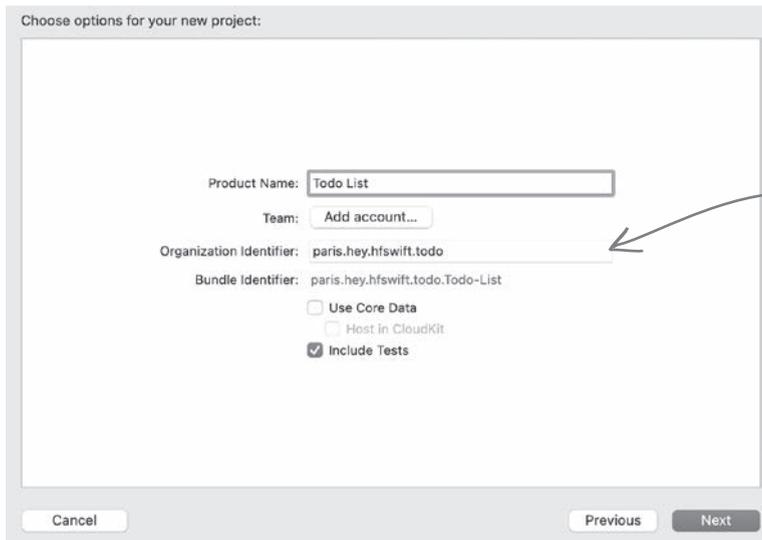
- Xcode — более масштабная, более сложная и обладающая более широкими возможностями версия Playgrounds.
- Xcode может компилировать приложения, предназначенные для платформ Apple: iOS, iPadOS, macOS, tvOS, watchOS и других платформ, а также веб-приложения.
- Проект Xcode необходимо сохранить в конкретном месте (он не появится в списке того, что было сделано ранее, как в Playgrounds).
- Проект Xcode представляет собой папку, в которой находится файл *.xcodeproj*, а также набор внутренних папок со специальными именами.
- Вам не придется работать с папкой проекта Xcode напрямую. Вместо этого вы будете работать с ней через интерфейс Xcode.
- Полное описание функциональности Xcode выходит за рамки книги, но если не торопиться и придерживаться знакомых возможностей, общих для Playgrounds и Xcode, все будет хорошо.

- 3 На экране выбора шаблона найдите шаблон App, выберите его и щелкните на кнопке Next.

Выберите платформу для вашего приложения: будет ли оно многоплатформенным (Multiplatform) или предназначенным для iOS, macOS или одной из других платформ Apple. Приложение будет работать на большинстве этих платформ, но мы рекомендуем оставить выбранным вариант Multiplatform.



- 4 Введите имя своего продукта и идентификатор организации, щелкните на кнопке Next.



Идентификатор организации обычно состоит из доменного имени вашего веб-сайта, записанного в обратном порядке, за которым следует имя приложения. Если у вас нет веб-сайта, введите произвольное имя на свое усмотрение.

- 5 Выберите каталог для сохранения проекта и щелкните на кнопке Create.

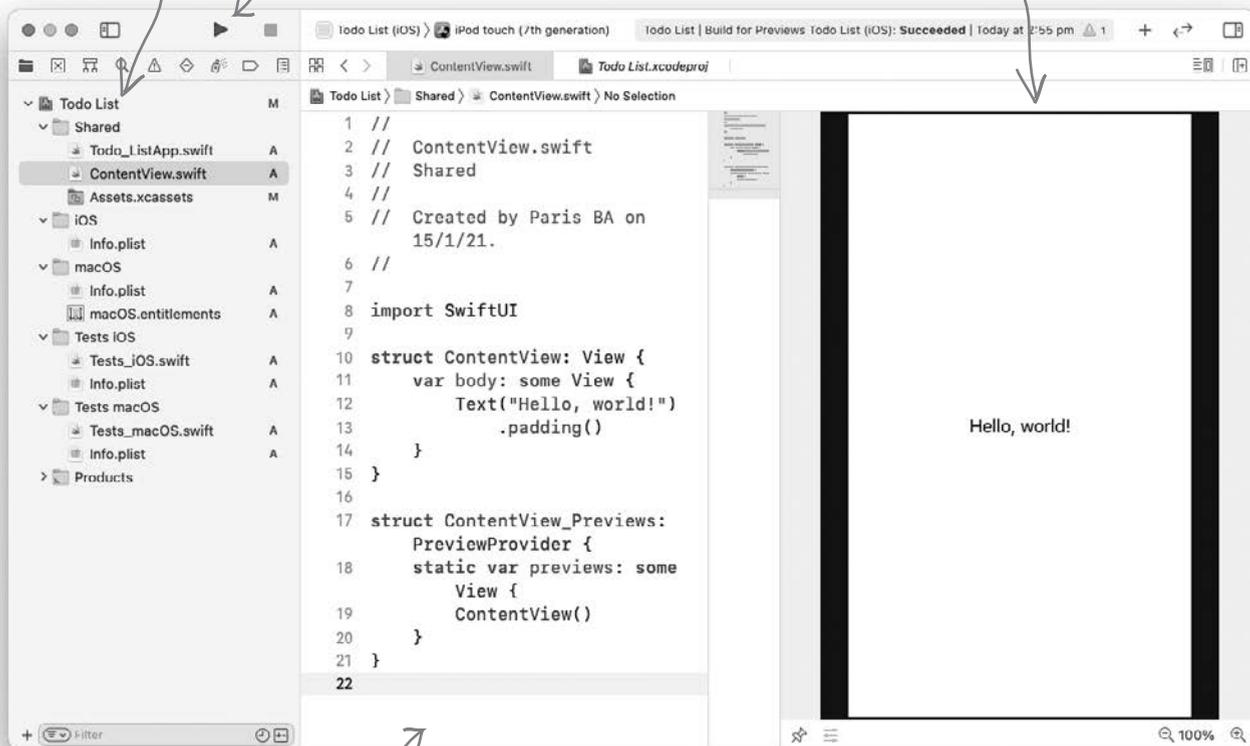
Создайте в Xcode новый проект SwiftUI для iOS.

## Ваша среда Xcode должна выглядеть примерно так

Здесь размещаются файлы с программным кодом, входящие в проект

Приложение можно запустить этой кнопкой.

Здесь выводится предварительное изображение пользовательского интерфейса текущего кода (если он есть).



Здесь редактируется код текущего активного файла.



Будьте осторожны!

**На первых порах разобраться в Xcode может быть непросто.**

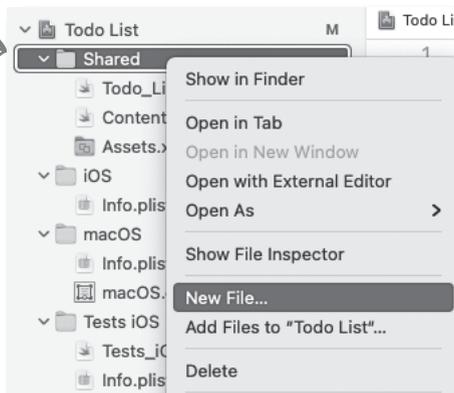
Но не беспокойтесь. Xcode — важный шаг вперед по сравнению с Playgrounds. Многие возможности Playgrounds поддерживаются и в Xcode, а для их обозначения используются те же элементы интерфейса. Программируйте, запускайте свой код кнопкой Play, и все будет хорошо.

## Создание нового типа для хранения элемента списка задач

Работа над приложением начинается с создания нового типа для хранения всех элементов списка задач.

- 1 Создайте новый файл Swift с именем `Item.swift`, в котором будет храниться код нового типа.

Чтобы добавить файл в проект, щелкните правой кнопкой мыши на папке `Shared` на панели `Project`...

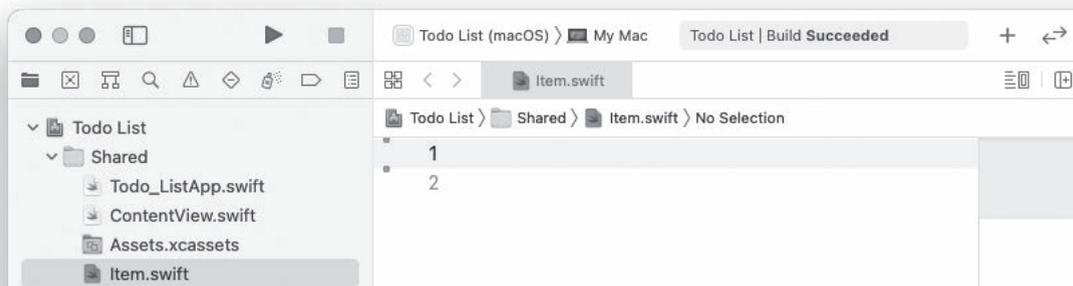


... и выберите в меню команду `New File...`

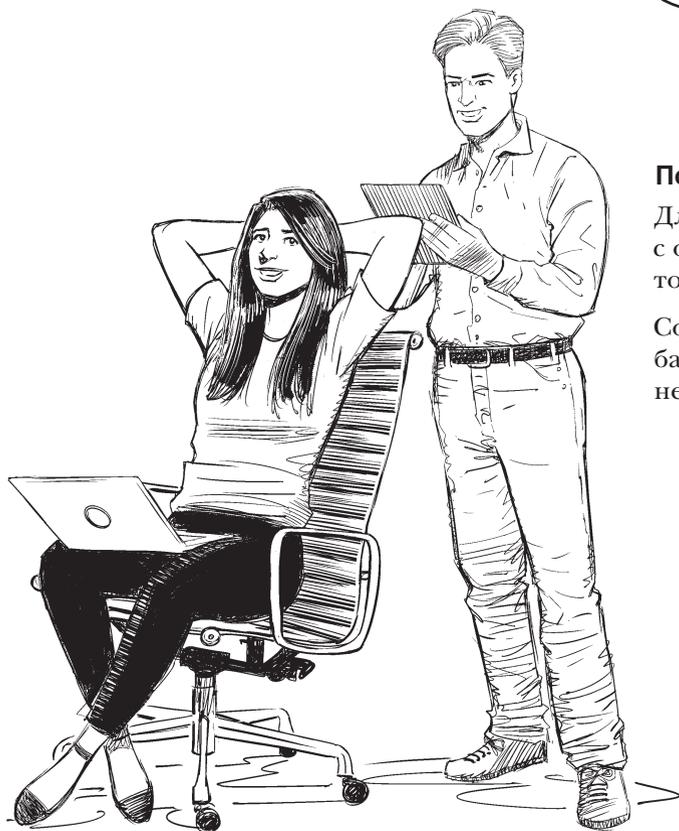


Выберите `Swift File` из предложенных вариантов. На этой стадии неважно, какую платформу вы выберете, поэтому оставьте платформу `iOS`.

- 2 В файле `Item.swift` определите новый тип с именем `Item`.
- 3 В типе `Item` будет храниться только одно поле данных: строка с описанием задачи.



Почему мы создали специальный тип для хранения задач? Почему просто не использовать String?



**Потому что хранить нужно не только String.**

Для каждой задачи должна храниться строка с описанием, а также уникальный идентификатор (но это будет сделано позже).

Создание собственного типа позволит нам добавить в задачу новые возможности, которые не поддерживает String.

**Возьмите в руку карандаш**



Теперь ваша очередь. Требуется реализовать новый тип. Мы уже описали основные необходимые шаги, так что попробуйте реализовать его самостоятельно в своем новом проекте.

Вероятно, новый тип стоит создать на базе структуры, и в нем (пока) будет храниться строка с описанием задачи.

А когда тип будет создан, мы добавим возможность однозначной идентификации каждого элемента в списке задач.

## Возьмите в руку карандаш



### Решение

Мы создаем тип для представления задачи из списка.

Лучше всего для этого воспользоваться структурой:

```
struct Item {

}
```

В новой структуре должна храниться строка с описанием задачи, поэтому в нее добавляется свойство:

```
struct Item {
 var todo: String
}
```

Вот и все!



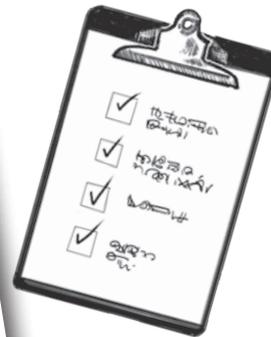
Создайте новый тип для хранения отдельной задачи из списка.

### Структуры

Структуры являются типами-значениями. Каждый раз, когда вы создаете или присваиваете значение структурной переменной, вы создаете новую структуру или копируете ее.

Благодаря этой особенности структуры идеально подходят для типа Item.

Мы можем легко создавать элементы списка задач, а потом так же легко удалять их после завершения работы с ними.



```
var todoItem1 = Todo(todo: "Rewatch 30 Rock")
var todoItem2 = Todo(todo: "Learn Spanish")
var todoItem3 = Todo(todo: "Eat a good souvlaki")
```

## Однозначная идентификация каждого элемента списка задач

Так как данные списка должны сохраняться между запусками приложения, необходимо позаботиться о том, чтобы каждый элемент списка задач можно было отличить от других. Иначе говоря, необходимо позаботиться о том, чтобы каждый экземпляр нового типа `Item` обладал уникальным идентификатором.

К счастью, Apple позволяет легко это сделать: протокол `Identifiable`, включенный в Swift, используется для создания уникальных идентификаторов.

- 1 Откройте файл `Item.swift` и найдите реализацию `Item`:

```
struct Item {
 var todo: String
}
```

← Это будет нетрудно, потому что в файле ничего другого быть не должно.

- 2 Объявите, что структура `Item` будет поддерживать протокол `Identifiable`:

```
struct Item: Identifiable {
 var todo: String
}
```

- 2 Реализуйте свойство, необходимое для поддержки протокола `Identifiable`:

```
struct Item: Identifiable {
 let id = UUID()
 var todo: String
}
```

← Функция `UUID()` является частью библиотеки `Foundation` компании Apple. Она возвращает универсально-уникальное значение, которое может использоваться в качестве идентификатора `Item`.



Позаботьтесь о том, чтобы каждый элемент списка задач имел уникальный идентификатор.



### Поддержка `Identifiable` под увеличительным стеклом

Поддержка протокола `Identifiable` означает, что структура должна содержать свойство `id`, которое будет использоваться для идентификации экземпляра. Если тип является структурой, то вам придется реализовать это свойство самостоятельно, как мы делаем здесь.

Функция `UUID`, которую мы вызываем для получения идентификатора, генерирует длинный целочисленный уникальный идентификатор. Ничего сверхъестественного.

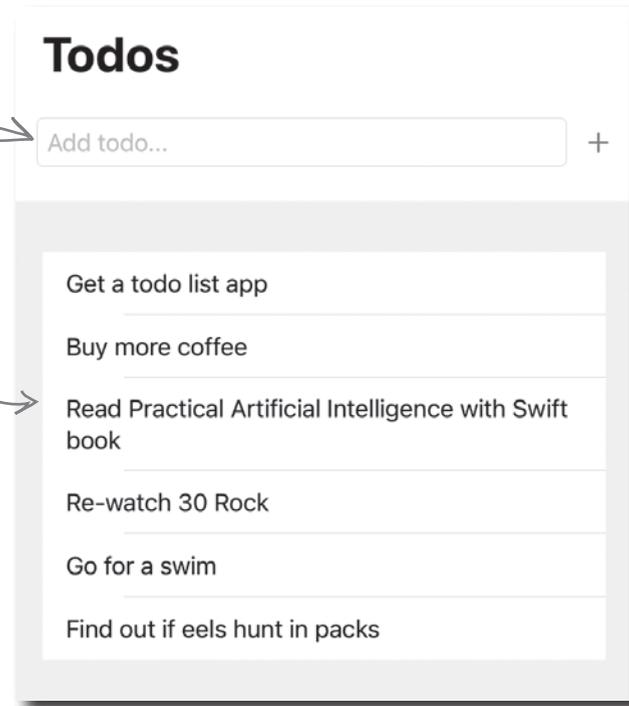
## Создание пользовательского интерфейса приложения

А теперь мы создадим пользовательский интерфейс приложения:

- 1 Необходимо создать переменные `@State` для хранения состояния приложения. Чтобы определить, какую информацию состояния необходимо хранить, стоит заранее продумать планируемый интерфейс приложения.

Нам потребуется переменная для хранения новой задачи.

А также массив для всех элементов в списке.



- 2 Добавим две переменные для хранения этого состояния в структуру `ContentView` в файле `ContentView.swift`:

```
struct ContentView: View {
```

```
 @State private var currentTodo = ""
```

```
 @State private var todos: [Item] = []
```

```
 var body: some View {
```

```
 Text("Hello, world!")
```

```
 .padding()
```

```
 }
```

```
}
```

Для хранения только что созданной задачи. Текст новой задачи содержит пустую строку.

Массив экземпляров `Item` (наш тип для хранения задачи) будет использоваться для `List`.

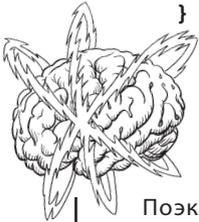
## Создание пользовательского интерфейса приложения (продолжение)

На этом шаге придется повозиться. В теле `ContentView` (удалите шаблонный код, сгенерированный при создании проекта) нам понадобится представление `NavigationView`, внутри которого находится панель `VStack`, внутри которой находится панель `HStack`, внутри которой находится поле `TextField` (в котором вводятся новые задачи) и представление `Button` для сохранения новой задачи. Также необходимо будет заполнить заголовок панели `NavigationView` осмысленным текстом.

3 Обновите тело `ContentView` и приведите его к следующему виду:

```
var body: some View {
 1 NavigationView {
 2 VStack {
 3 HStack {
 4 TextField("New todo..", text: $currentTodo)
 .textFieldStyle(RoundedBorderTextFieldStyle())

 5 Button(action: {
 guard !self.currentTodo.isEmpty else { return }
 self.todos.append(Item(todo: self.currentTodo))
 self.currentTodo = ""
 }) {
 Image(systemName: "text.badge.plus")
 }
 .padding(.leading, 5)
 }.padding()
 }
 .navigationBarTitle("Todo List")
 }
}
```



Мозговой  
Штурм

Поэкспериментируйте с разными параметрами `textFieldStyle` для стилизованного оформления `TextField`. Попробуйте использовать значения `SquareBorderTextFieldStyle`, `DefaultTextFieldStyle` и `PlainTextFieldStyle`. Какой из вариантов вам больше нравится?

## Что происходит в этом интерфейсе?

1

### NavigationView

Представление позволяет представить пользовательский интерфейс в виде серии представлений. Кроме того, оно предоставляет простую возможность вывести содержательное имя в верхней части экрана, для чего `navigationBarTitle` присваивается значение "Todo List".

2

### VStack

Все, что находится в `NavigationView`, будет размещаться внутри панели `VStack`. Это делается для того, чтобы пользовательский интерфейс был выстроен от верхнего края экрана к нижнему.

3

### HStack

Пока в `VStack` будет находиться только один элемент `HStack`. Мы воспользуемся панелью `HStack` для размещения элементов по горизонтали вблизи друг от друга. Кроме того, после `HStack` будет создан отступ.

4

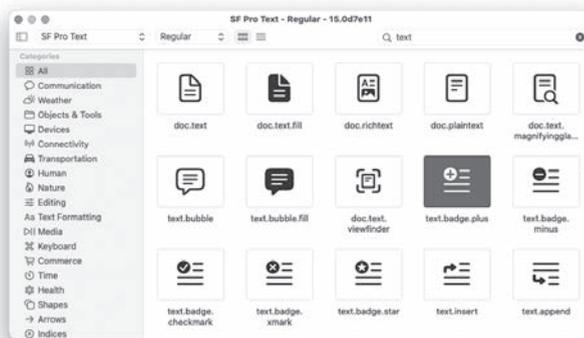
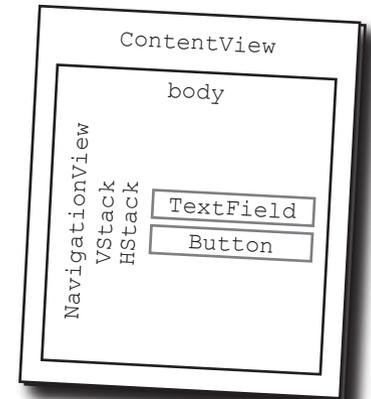
### TextField

Сначала внутри `HStack` мы поместим элемент `TextField`, в котором пользователь вводит текст новой задачи. Элемент помечается надписью "New todo..." и связывается с переменной состояния `currentTodo`, которая была создана ранее. Кроме того, параметру `textFieldStyle` присваивается значение `RoundedBorderTextFieldStyle()`.

5

### Button

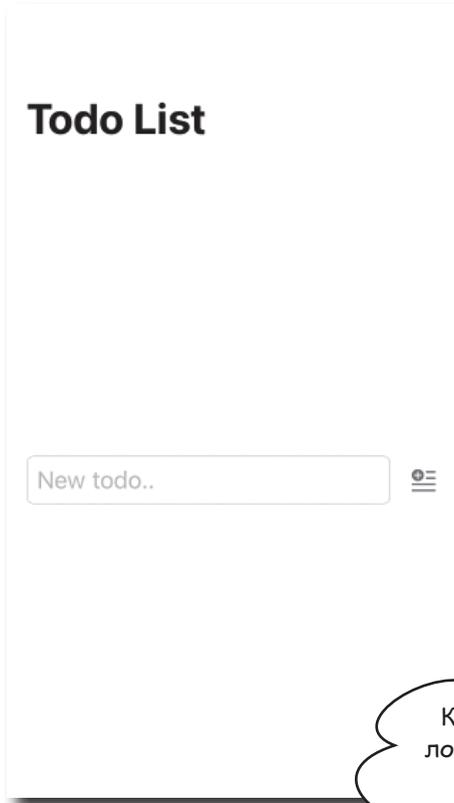
Затем нам понадобится кнопка `Button` для включения текущей задачи в список. В качестве действия `Button` назначается замыкание, которое проверяет, не пуста ли переменная `newTodo` (что означало бы, что пользователь не ввел текст), и если не пуста, присоединяет содержимое `newTodo` как элемент массива `todos`, созданного ранее. Наконец, на кнопке выводится элемент `Image`. Кнопке `Button` также назначается начальный отступ.



### Полезные значки

Значок, который загружается вызовом `Image(systemName: "text.badge.plus")` в `Button`, взят из полезной библиотеки изображений, распространяемой компанией Apple, — эта библиотека называется `SF Symbols`. `SF Symbols` содержит сотни разных значков, которые можно свободно использовать в приложениях для платформ Apple.

Посетите страницу <https://developer.apple.com/sf-symbols> и загрузите приложение для просмотра значков!



На этой стадии ваше приложение (если вы запустите его) выглядит примерно так. Теперь необходимо добавить список...



Чтобы запустить приложение в эмуляторе iOS, щелкните на кнопке Play в левом верхнем углу Xcode.

Проследите за тем, чтобы здесь был выбран правильный тип устройства iOS для вашего эмулятора.

Как идет работа над моим приложением? Я не могу ждать вечно, знаете ли. У меня все больше незавершенных дел.



## Создание пользовательского интерфейса для приложения (продолжение)

Наше приложение выглядит каким-то пустым без его главной особенности: списка `List`. Пора добавить его. Список будет размещаться в элементе `VStack`, который был создан ранее, но вне элемента `HStack` и после него.

### 4 Обновите тело `ContentView` и добавьте список `List`:

```
var body: some View {
 NavigationView {
 VStack {
 HStack {
 TextField("New todo..", text: $currentTodo)
 .textFieldStyle(RoundedBorderTextFieldStyle())

 Button(action: {
 guard !self.currentTodo.isEmpty else { return }
 self.todos.append(Item(todo: self.currentTodo))
 self.currentTodo = ""
 }) {
 Image(systemName: "text.badge.plus")
 }
 .padding(.leading, 5)
 }.padding()

 List {
 Text("This is something in my list!")
 Text("This is also in my list!")
 Text("And another thing!")
 }
 }
 .navigationBarTitle("Todo List")
 }
}
```

*List сам по себе является представлением.*

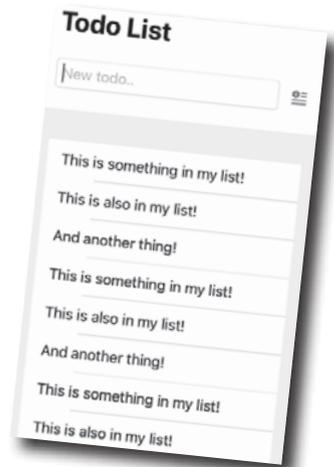
*Все, что находится внутри, выводится в List.*

## Вывод списка задач в List

- 5 Чтобы в List действительно выводились задачи, необходимо использовать блок `ForEach`, который перебирает элементы массива `todos` и выводит каждый элемент в представлении `Text` внутри `List`. Начнем с добавления `ForEach`:

```
List {
 ForEach(todos) { todoEntry in
 Text("This is something in my list!")
 Text("This is also in my list!")
 Text("And another thing!")
 }
}
```

Одни и те же три строки текста повторяются каждый раз, когда `ForEach` обнаруживает элемент в массиве `todos`. Выглядит поинтереснее, но все еще довольно бесполезно?



- 6 Если запустить приложение в этой точке, вы увидите, как одни и те же три строки текста повторяются снова и снова для каждого элемента, который добавляется в список задач при помощи поля `TextField` и кнопки `Button`. Не бойтесь, попробуйте: приложение не кусается. К сожалению, кое-чего вы не увидите: содержимого каждой задачи в списке. Давайте исправим ситуацию. Удалите три представления `Text` из `ForEach` и добавьте:

```
List {
 ForEach(todos) { todoEntry in
 Text(todoEntry.todo)
 }
}
```

Так как `todos` (переменная состояния, в которой хранится массив с элементами типа `Item`) содержит наш список задач, необходимо перебрать его содержимое.

Текущая позиция перебора в `ForEach` обозначается именем `todoEntry`.

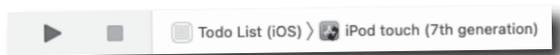
А поскольку `todos` содержит экземпляры `Item`, мы знаем, что в них присутствует свойство `todo`, которое содержит строку, представляющую задачу. Эта строка выводится в представлении `Text` внутри `List`.



Создайте пользовательский интерфейс для приложения: текстовое поле для добавления новых задач, кнопку для их сохранения и список для вывода всех задач.

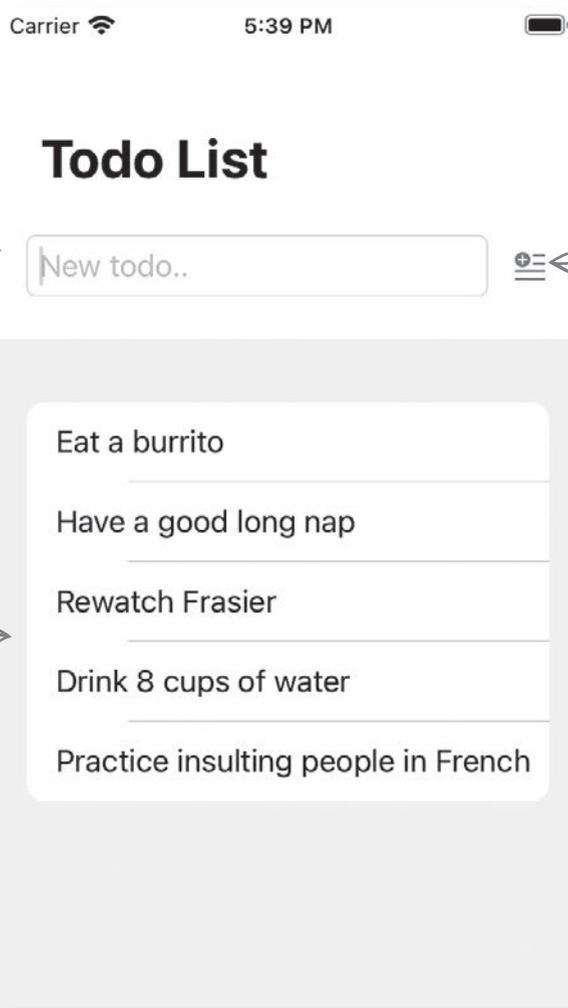


## Тест-драйв



Запустите приложение и посмотрите, что произойдет...

Чтобы добавить задачу в список, введите ее здесь...



...а затем нажмите эту кнопку.

Ваши задачи появляются в списке!

## Реализация сохранения списка задач

- 1 Теперь необходимо вернуться к файлу *Item.swift* и обновить тип *Item*, чтобы кодировать и декодировать текст (это означает *сохранять и загружать* на техническом жаргоне). Для этого необходимо добавить в тип *Item* поддержку протокола *Codable*:

```
struct Item: Identifiable, Codable {
 let id = UUID()
 var todo: String
}
```



За сценой



*Codable* является псевдонимом типа для *Decodable* и *Encodable*, чтобы вы могли легко включить преобразование вашего типа во внешнее представление и обратно. Иначе говоря, вы приказываете Swift сделать так, чтобы тип можно было кодировать и декодировать из сериализованного формата (такого, как JSON) — чаще всего для сохранения на диске.

За кулисами Swift пытается сгенерировать код кодирования и декодирования экземпляра типа, поддерживающего *Codable*.

Чтобы тип мог поддерживать *Codable*, его хранимые свойства тоже должны поддерживать *Codable*. К счастью, стандартные типы Swift — *String*, *Int*, *Double*, *Array* и т. д. — уже поддерживают *Codable*.

Если ваши свойства поддерживают *Codable*, то ваш тип будет автоматически поддерживать *Codable* — достаточно просто объявить об этом.

Этот фрагмент:

```
struct Item: Identifiable, Codable {
 let id = UUID()
 var todo: String
}
```

в точности эквивалентен следующему:

```
struct Item: Identifiable, Encodable, Decodable {
 let id = UUID()
 var todo: String
}
```

Впрочем, если вам нужно только что-то одно (кодирование или декодирование), можно выбрать только *Encodable* или *Decodable*.

Кодирование данных пригодится для записи их на диск, отправки по сети или для передачи функциям API.

2 Теперь вернитесь к *ContentView.swift* и создайте три метода в структуре *ContentView*:

1 **save()**

Метод, который кодирует все элементы *Item* в массиве *todos* и сохраняет их.

```
private func save() {
 UserDefaults.standard.set(
 try? PropertyListEncoder().encode(self.todos), forKey: "myTodosKey"
)
}
```

*UserDefaults* позволяет сохранить небольшой объем данных как часть профиля пользователя на устройстве. Этот механизм хорошо подходит для быстрого сохранения простых данных. Синтаксис выглядит так:

```
UserDefaults.standard.set("Сохраняемые данные", forKey: "имяКлюча")
```

Таким образом, метод *save* использует *try* для того, чтобы попытаться получить закодированную форму массива *todos* и сохранить ее с ключом *myTodosKey*.

2 **load()**

Также понадобится парный метод, который загружает все элементы *Item* из сохранения в массив *todos*.

```
private func load() {
 if let todosData = UserDefaults.standard.value(forKey: "myTodosKey") as? Data {
 if let todosList
 = try? PropertyListDecoder().decode(Array<Item>.self, from: todosData) {
 self.todos = todosList
 }
 }
}
```

Наш метод *load* читает данные из *UserDefaults* с тем же ключом *myTodosKey* и запрашивает их в формате *Data*. Получив данные, он пытается при помощи *PropertyListDecoder* декодировать данные, хранящиеся в *UserDefaults*, и поместить их обратно в массив *todos*.



Тип Swift *Data* позволяет сохранять неструктурированные байтовые блоки. Он удобен для сохранения и загрузки данных или передачи их по сети.

**2 delete(at offset: IndexSet)**

Наконец, нам понадобится метод для удаления отдельной задачи как из списка, так и из сохраненной копии.

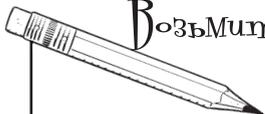
```
private func delete(at offset: IndexSet) {
 self.todos.remove(atOffsets: offset)
 save()
}
```

Метод `delete method` получает параметр `offset` типа `IndexSet`. В `IndexSet` хранится коллекция уникальных целых значений, представляющих индексы элементов другой коллекции. Фактически это отсортированная коллекция целых чисел, представляющих позиции в другой коллекции.

В данном случае мы берем параметр `IndexSet – offset` – и используем его для удаления одного или нескольких элементов массива `todos`.

После этого вызывается метод `save`. И всё!

**Возьмите в руку карандаш**



Снова ваш ход. Попробуйте определить, в какой точке должны вызываться методы, которые мы только что определили.

Метод `save ()` должен вызываться при добавлении новой задачи.

Метод `load ()` должен вызываться при отображении `NavigationView`.

Метод `delete ()` должен вызываться при удалении элемента `List`.

Чтобы вам было немного проще, учтите следующее:

- < У `List` есть метод с именем `onDelete`, который получает параметр с именем `perform`. В нем может передаваться метод, которому при вызове неявно передается набор `IndexSet`, представляющий удаляемый элемент.
- < Аналогичным образом представление `NavigationView` (и все остальные представления) содержит метод `onAppear` с параметром `perform`, в котором передается метод.

→ Ответ на с. 332.

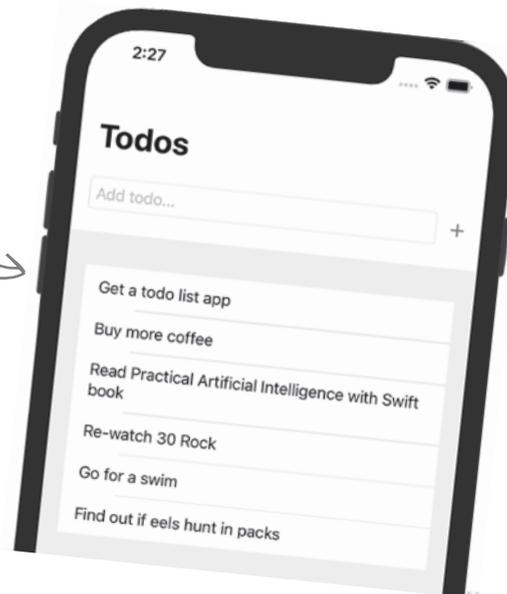


Реализуйте функцию сохранения списка задач, чтобы он не терялся между запусками приложения.

**Вот и всё. Можно переходить к тестированию замечательного нового приложения.**

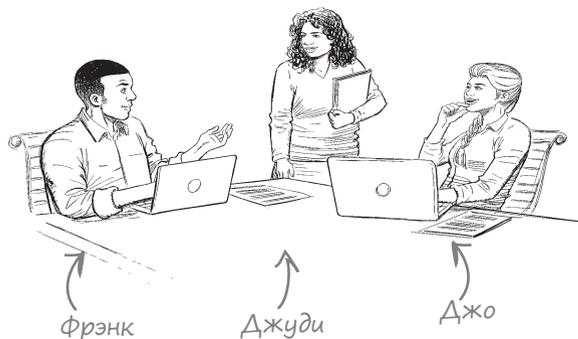
Запустите приложение и протестируйте его!

Вы можете добавлять задачи, а также удалять задачи — для этого проведите по одной из них справа налево и нажмите кнопку удаления. А если закрыть и перезапустить приложение, в нем будет отображаться список задач без потери данных! Потрясающе.



Неплохо, да? Интересно, что еще вы можете сделать для меня. У меня много друзей, которым тоже нужны приложения.

## Значит, это и есть UI-фреймворк?



**Фрэнк:** Вот, значит, что такое UI-фреймворк?

**Джуди:** Меня не особо впечатляет.

**Джо:** А меня впечатляет. В смысле меня и раньше впечатляло, а теперь впечатляет еще больше. Очень удобно.

**Джуди:** И что со всем этим делать дальше?

**Фрэнк:** Да, что делать дальше? У меня голова идет кругом. Кажется, я знаю достаточно, чтобы создать проблемы, но слишком мало, чтобы построить что-то серьезное.

**Джо:** Не бойтесь. В следующий раз мы построим что-то побольше списка задач. И может быть, даже с подключением к интернету! Вы увидите, как SwiftUI масштабируется и становится более сложным, при этом не теряя фирменного стиля Swift.

**Джуди:** Я очень хочу узнать о подключении к интернету, это очень полезно. Можно загружать всякое...

**Фрэнк:** И я тоже!

**Джо:** Прекрасно. Продолжаем учиться!

### Ключевые моменты

- SwiftUI — декларативный UI-фреймворк, который проектировался с учетом сильных сторон и возможностей Swift.
- SwiftUI можно использовать для построения приложений с пользовательским интерфейсом для платформ Apple.
- Чтобы использовать фреймворк SwiftUI, его необходимо импортировать.
- Представление SwiftUI создается объявлением структуры (которой по общепринятым соглашениям присваивается имя `<ИмяView>`), поддерживающей протокол `View`.
- Протокол `View` является одной из важнейших частей SwiftUI. Он обязательно включается всеми элементами SwiftUI, которые должны прорисовываться на экране.
- В структуре представления объявляется вычисляемое свойство с именем `body` и типом `some View`.
- `some View` означает, что возвращаться будет нечто поддерживающее протокол `View`, но всегда должна возвращаться одна и та же разновидность `View` (это не может быть иногда один тип, поддерживающий `View`, а иногда — другой тип, поддерживающий `View`).
- Элементы SwiftUI кодируются в вычисляемом свойстве `body`. Например, `Text("Hello")` отображает представление `Text` с сообщением «Hello».
- Некоторые элементы представлений SwiftUI невидимы, например `VStack` и `HStack`; они используются исключительно для размещения других элементов.
- SwiftUI — мощный фреймворк, но его эффективность сильно зависит от правильного размещения элементов. Будьте внимательны и следите за тем, чтобы все элементы находились на своих местах.

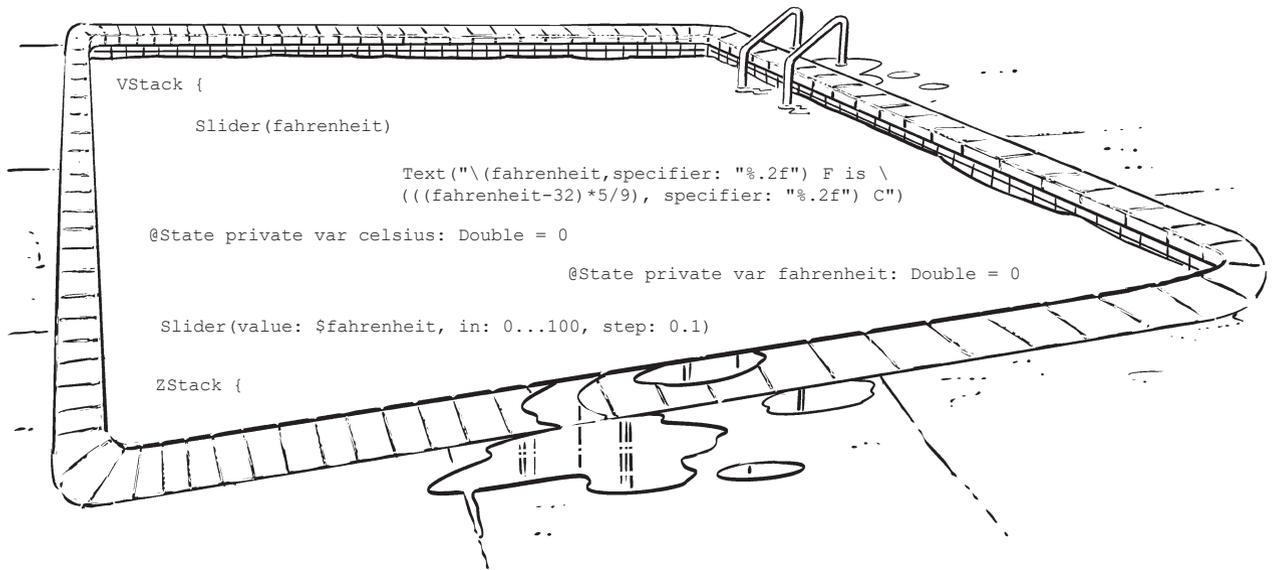


## У бассейна

**Внимание: каждый предмет из бассейна может использоваться только один раз! Или не использоваться вовсе.**

Выловите из бассейна строки кода и разместите их в пустых строках среды Playground. Каждая строка может использоваться только один раз; использовать все строки не обязательно. Ваша задача: построить код, который генерирует приведенный результат.

78.30 F is 25.72 C



```
VStack {
 Slider(fahrenheit)

 Text("\(fahrenheit, specifier: "%.2f") F is \
 (((fahrenheit-32)*5/9), specifier: "%.2f") C")

 @State private var celsius: Double = 0
 @State private var fahrenheit: Double = 0

 Slider(value: $fahrenheit, in: 0...100, step: 0.1)

 ZStack {

import SwiftUI
import PlaygroundSupport

struct TemperatureConvert: View {

 var body: some View {

 }
}

PlaygroundPage.current.setLiveView(TemperatureConvert())
```

Ответ на с. 333.



Возьмите в руку карандаш

## Решение

C. 328

```

var body: some View {
 NavigationView {
 VStack {
 HStack {
 TextField("New todo..", text: $currentTodo)
 .textFieldStyle(RoundedBorderTextFieldStyle())

 Button(action: {
 guard !self.currentTodo.isEmpty else { return }
 self.todos.append(Item(todo: self.currentTodo))
 self.currentTodo = ""
 self.save()
 }) {
 Image(systemName: "text.badge.plus")
 }
 .padding(.leading, 5)
 }.padding()

 List {
 ForEach(todos) { todoEntry in
 Text(todoEntry.todo)
 }.onDelete(perform: delete)
 }
 }
 .navigationBarTitle("Todo List")
 }.onAppear(perform: load)
}

```



## У бассейна. Решение

С. 331.

```
import SwiftUI
import PlaygroundSupport

struct TemperatureConvert: View {
 @State private var fahrenheit: Double = 0

 var body: some View {
 VStack {
 Slider(value: $fahrenheit, in: 0...100, step: 0.1)
 Text("\(fahrenheit, specifier: "%.2f") F is \
 ((fahrenheit-32)*5/9), specifier: "%.2f") C")
 }
 }
}

PlaygroundPage.current.setLiveView(TemperatureConvert())
```

## СТАТЬ КОМПИЛЯТОРОМ Swift



Каждый фрагмент кода Swift на этой и следующей странице определяет представление SwiftUI. Вообразите себя на месте компилятора Swift и определите, будут ли выполняться каждый из этих фрагментов. Если какой-либо фрагмент не компилируется, то что с ним не так?

*Предполагается, что каждый фрагмент выполняется в среде Playgrounds, импортирует SwiftUI и использует эту команду.*

↙ PlaygroundPage.current.setLiveView(MyView())

A

```
struct MyView: View {
 var dogs = ["Greyhound", "Whippet", "Italian Greyhound"]
 @State private var selectedDog = 0

 var body: some View {
 VStack {
 Text("Please select your favorite dog breed:")
 Picker(selection: $selectedDog, label: Text("Dog")) {
 ForEach(0..
```

B

```

struct MyView: View {
 var body: some View {
 Image(systemName: "star.fill")
 .imageScale(.large)
 .foregroundColor(.yellow)
 }
}

```

D

```

struct MyView: View {
 var body: some View {
 List {
 Text("This is a list!")
 Text("Hello")
 Text("I'm a list!")
 }
 }
}

```

F

```

struct MyView: View {
 @State private var coffeesConsumed = 0

 var body: some View {
 Stepper(value: coffeesConsumed) {
 Text("Cups of coffee consumed: \(coffeesConsumed)")
 }
 }
}

```

C

```

struct MyView: View {
 var body: View {
 VStack {
 Text("Hello")
 Text("SwiftUI!")
 }
 }
}

```

E

```

struct MyView: View {
 @State private var lovelyDayStatus = true

 var body: some View {
 Toggle(isOn: $lovelyDayStatus) {
 Text("Is it a lovely day?")
 }

 if(lovelyDayStatus) {
 Text("It's a lovely day!")
 } else {
 Text("It's not a lovely day.")
 }
 }
}

```

→ Ответ на с. 336.

## СТАТЬ компилятором Swift. Решение

С. 334



A, B, D и E работают нормально.

C не работает, потому что у body отсутствует ключевое слово some.

F не работает, потому что при связывании coffeesConsumed в Stepper пропущен символ \$.

## 11. Практическое применение SwiftUI

# Круги, таймеры, кнопки — выбирайте!

И все это — UI-элементы.

В SwiftUI много элементов, и когда вы начнете строить сложные интерфейсы, будет казаться, что вам приходится жонглировать ими всеми.



**SwiftUI** **вовсе не ограничивается кнопками и списками.** В интерфейсах также можно использовать геометрические фигуры, анимации и многое другое! В этой главе будут рассмотрены некоторые **расширенные возможности построения пользовательских интерфейсов в SwiftUI** и их связывания с источниками данных, содержимое которых не генерируется пользователем (как в случае со списком задач). SwiftUI позволяет строить **пользовательские интерфейсы** с обработкой **событий**, поступающих из разных источников. Мы будем работать в Xcode, среде разработки от компании Apple, а основное внимание будет уделяться приложениям для iOS, но все, что вы узнаете в этой главе, в равной степени применимо к SwiftUI для iPadOS, macOS, watchOS и tvOS. Вперед, глубины SwiftUI ожидают вас!

## Что интересного можно сделать с UI-фреймворком?



**Фрэнк:** Итак, мы узнали, что такое UI-фреймворк...

**Джуди:** И даже построили неплохое приложение для ведения списка задач...

**Джо:** Но что еще можно сделать?

**Фрэнк:** Полагаю, мы можем построить другое приложение!

**Джуди:** Да, можем. И построим. Но чего мы еще не знаем? Наверняка же мы еще не успели изучить все, верно?

**Джо:** Да, конечно, у SwiftUI есть более сложные возможности, которые мы еще не рассматривали.

**Джуди:** Может, построим что-нибудь с одной-двумя из них?

**Джо:** Мне тоже хотелось бы увидеть, как построить веб-сайт на основе Swift. Я слышал, что это не так уж сложно.

**Фрэнк:** Да, веб-сайты! А мы можем использовать SwiftUI для построения веб-сайтов?

**Джуди:** Нет, не можем, но существует фреймворк `Varo`, который позволяет создать веб-сайт на Swift. Думаю, вскоре мы до этого доберемся.

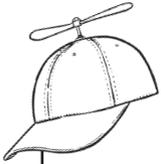


Люблю расслабиться на природе. А вы мне поможете написать приложение для отслеживания оставшегося времени встречи, чтобы я знал, когда я смогу вернуться в свой гамак?

**SwiftUI поможет построить таймер для встреч.**

В SwiftUI существуют разнообразные структурные элементы, которые могут использоваться для построения таймера.

Одно из удобных изображений таймера — кольцо, непрерывно уменьшающееся с течением времени.



## Серьезное программирование

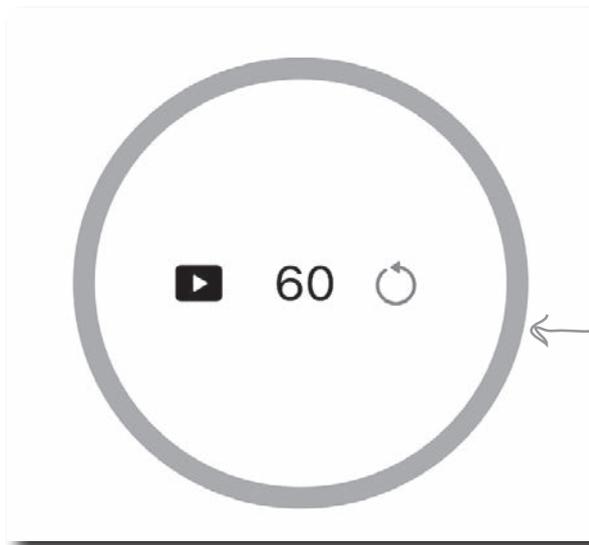
### Создание таймера

*Отсчет времени запущен... (вообще-то нет — думайте сколько душе угодно).*

Мы построим приложение Executive Timer для нашего знакомого. В этом случае также будет использоваться Xcode — интегрированная среда разработки для Swift и SwiftUI от компании Apple. Чтобы создать приложение, необходимо выполнить следующие действия:

- Создайте в Xcode новый проект SwiftUI для iOS.
- Создайте основные элементы приложения Executive Timer.
- Выполните необходимую подготовку, чтобы интерфейс мог нормально функционировать.
- Создайте все части пользовательского интерфейса.
- Объедините все элементы!
- Обеспечьте обновление пользовательского интерфейса таймером.

*Повторите эти шаги самостоятельно и постройте собственное приложение Executive Timer.*



*Приложение Executive Timer должно выглядеть так:*

*Кольцо изображает отслеживаемый промежуток времени. Он постепенно уменьшается.*

*А теперь за дело...*

## Создание нового проекта SwiftUI для iOS

И снова начнем с создания проекта Xcode. Это делается практически так же, как и в предыдущей главе (собственно, никаких отличий вообще нет).



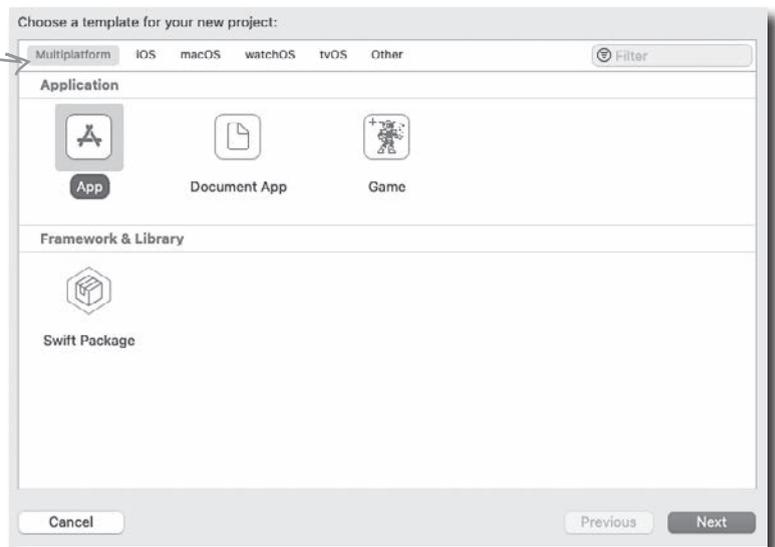
- 1 Запустите Xcode.
- 2 Выберите вариант **Create a new Xcode project** на экране *Welcome to Xcode*.



Если вы не видите экран «Welcome to Xcode», его можно открыть командой *New -> Project...* из меню *File*.

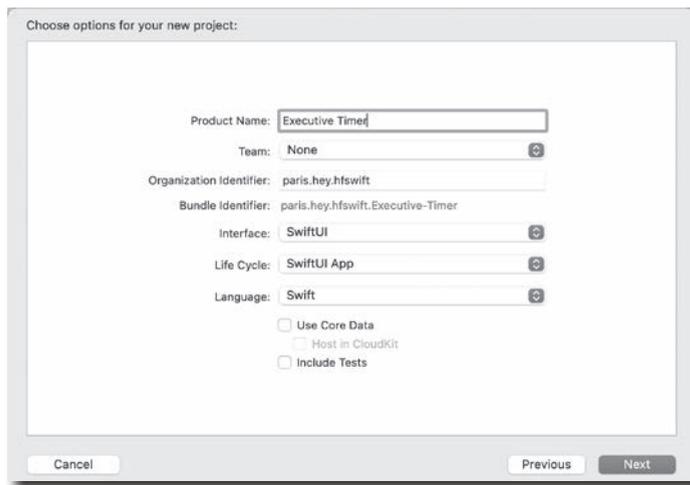
- 3 На экране выбора шаблона найдите шаблон *App*, выберите его и щелкните на кнопке *Next*.

Выберите платформу для вашего приложения — будет ли оно многоплатформенным (*Multiplatform*), или предназначенным для *iOS*, *macOS* или одной из других платформ *Apple*. Приложение будет работать на большинстве этих платформ, но мы рекомендуем оставить выбранным вариант *Multiplatform*.



## Создание нового проекта SwiftUI для iOS (продолжение)...

- 4 Введите имя своего продукта и идентификатор организации, щелкните на кнопке Next.



- 5 Выберите папку для сохранения проекта и щелкните на кнопке Create.

Создайте в Xcode новый проект SwiftUI для iOS.

### Ключевые моменты

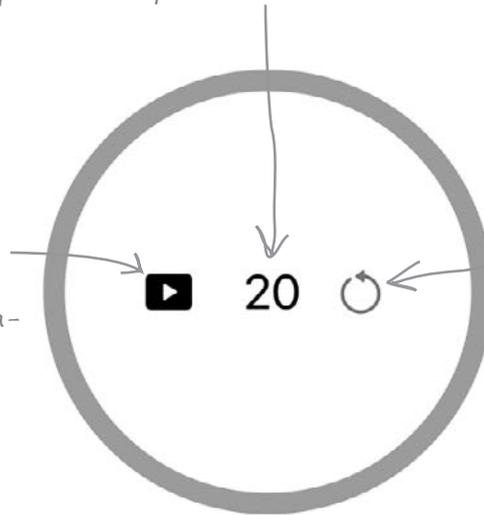
- При выборе проекта SwiftUI в Xcode ничего особенного не происходит. Каждый тип проекта в Xcode предоставляет разумный набор значений по умолчанию, подходящий для того, что вы пытаетесь построить. Так как мы хотим создать приложение iOS на базе SwiftUI, мы выбрали эти настройки для проекта.
- Также можно было создать приложение-таймер для macOS или даже для watchOS с точно таким же кодом (со SwiftUI это делается очень просто). Так как эта книга старается излагать учебный материал как можно проще, мы будем придерживаться платформы iOS. Но через какое-то время вы сможете разобраться в том, как сделать приложение-таймер кросс-платформенным — это станет хорошим учебным упражнением!
- Наше приложение-таймер будет использовать простые геометрические фигуры для представления частей его пользовательского интерфейса. Дело в том, что таймер должен иметь форму круга, а в SwiftUI нет встроенных элементов, естественным образом представляющих круглый таймер: нам придется построить его самостоятельно. И мы это сделаем.

## Пользовательский интерфейс и функциональность Executive Timer

### Пользовательский интерфейс Executive Timer выглядит и работает так:

В середине отображается оставшееся время в секундах.

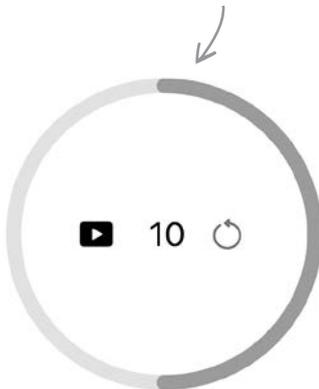
Эта кнопка запускает отсчет времени на таймере. Во время работы таймера эта кнопка приостанавливает работу таймера.



Кнопка сбрасывает таймер и возвращается к исходному промежутку времени.

### А вот как будет выглядеть интерфейс в различных состояниях:

Зеленое кольцо уменьшается с каждой секундой прошедшего времени.



Когда остается от 4 до 6 секунд, кольцо окрашивается в желтый цвет.



А на нескольких последних секундах кольцо становится красным.

## Создание основных элементов приложения

Для создания приложения Executive Timer необходимо выполнить кое-какие подготовительные действия. Мы начнем с создания некоторых мест для хранения состояния, а затем самого таймера. После этого мы подготовим пользовательский интерфейс для непосредственной прорисовки таймера на экране.

- 1 Создайте область памяти для хранения времени по умолчанию, другими словами, времени, от которого будет вести отсчет таймер. Для этой цели будет использоваться тип `CGFloat`; это обычный тип `Float`, но для операций, связанных с графикой (например, создания пользовательских интерфейсов):

```
let defaultTime: CGFloat = 20
```

*Время в секундах* ↗

- 2 Создайте свойства с ключевым словом `@State` для хранения элементов состояния, относящихся к пользовательскому интерфейсу:

```
@State private var timerRunning = false
@State private var countdownTime: CGFloat = defaultTime
```

↖ Это логическое значение показывает, активен ли таймер.

↗ *CGFloat хранит время, оставшееся до остановки таймера.*

↖ *defaultTime выбирается в качестве исходного значения: оставшееся время изначально равно максимальному промежутку времени, на который рассчитан таймер.*

- 3 Создайте экземпляр `Timer`, чтобы приложение могло отсчитывать время:

```
let timer = Timer.publish(every: 1, on: .main, in: .common).autoconnect()
```



За сценой

Экземпляр `Timer` создается методом `publish`, которому передается интервал отсчета (1 секунда); таймер выполняется в главном цикле приложения. Так как таймер был создан методом `publish`, данные будут публиковаться (генерироваться) с течением времени (через каждую секунду).

Данные, генерируемые публикаторами в Swift, могут приниматься разными способами, включая представления SwiftUI, которые могут работать с обновлениями при каждом их получении.

Публикаторы являются частью библиотек Apple, которая называется **Combine**. Мы не будем подробно рассматривать Combine, но эта библиотека предоставляет много полезных возможностей для получения данных из других частей приложения.



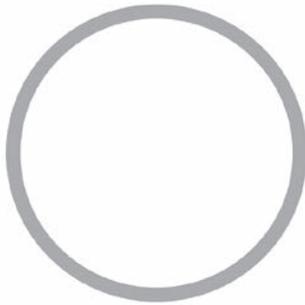
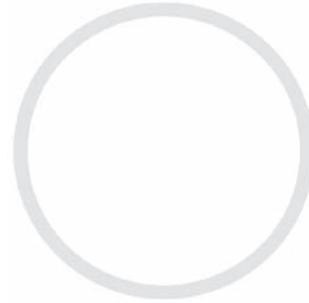
Создайте основные элементы приложения Executive Timer.

## Составляющие пользовательского интерфейса

### Серый круг

Прежде всего нам понадобится серый круг. Он будет находиться под всеми остальными элементами и будет постепенно раскрываться по мере того, как основной круг будет постепенно уменьшаться в процессе отсчета времени таймером.

1



### Основной круг

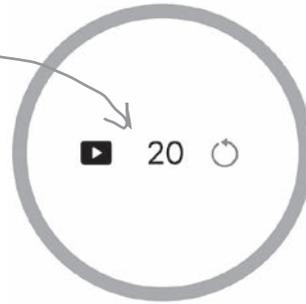
Также понадобится основной круг, который будет уменьшаться до нужного размера в зависимости от оставшегося времени.

2

### Время и кнопки

И наконец, нам понадобится кнопка запуска/приостановки и кнопка сброса, а также индикатор оставшегося времени в секундах.

3



## Сейчас мы их запрограммируем...



### Мозговой Штурм

Создайте среду Playground и выведите на экран несколько фигур средствами SwiftUI. Мы рекомендуем начать с треугольника:

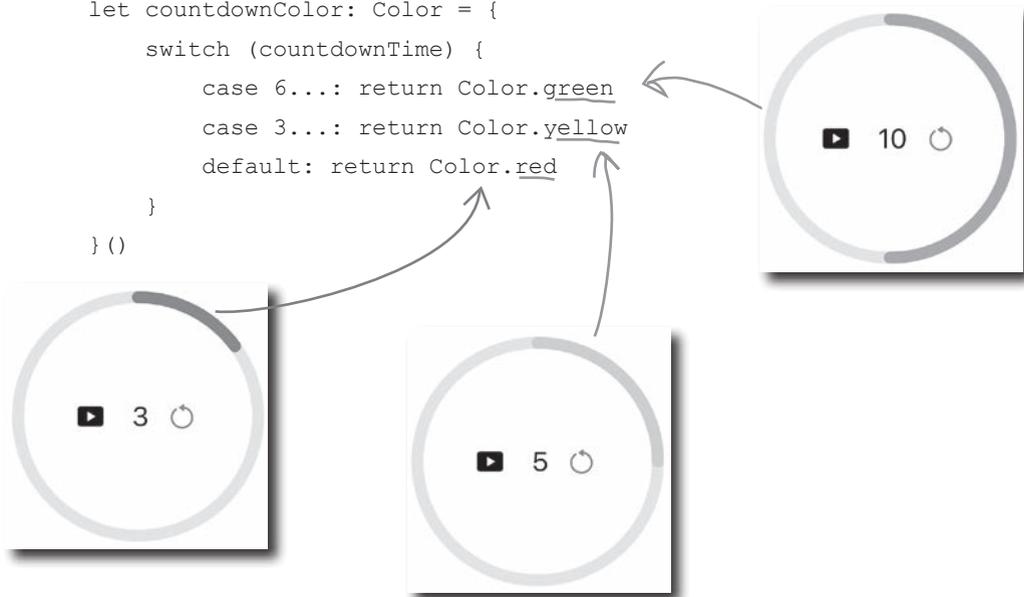
```
Triangle()
 .stroke(Color.green, style:
 StrokeStyle(lineWidth: 5, lineCap: .round, lineJoin: .round))
 .frame(width: 300, height: 300)
```

## Настройка пользовательского интерфейса приложения

Теперь займемся настройкой пользовательского интерфейса для рисования таймера.

- 1 Создайте замыкание, которое возвращает подходящий цвет в зависимости от значения `countdownTime`:

```
let countdownColor: Color = {
 switch (countdownTime) {
 case 6...: return Color.green
 case 3...: return Color.yellow
 default: return Color.red
 }
}()
```



- 2 Определите константу `strokeStyle`, чтобы упростить последующие эксперименты с оформлением таймера:

```
let strokeStyle = StrokeStyle(lineWidth: 15, lineCap: .round)
```



- 3 Определите константу `buttonIcon`, чтобы кнопка запуска/приостановки таймера переключалась на правильный значок в подходящее время в зависимости от состояния `timerRunning`:

```
let buttonIcon = timerRunning ? "pause.rectangle.fill" : "play.rectangle.fill"
```

*Тернарный оператор*



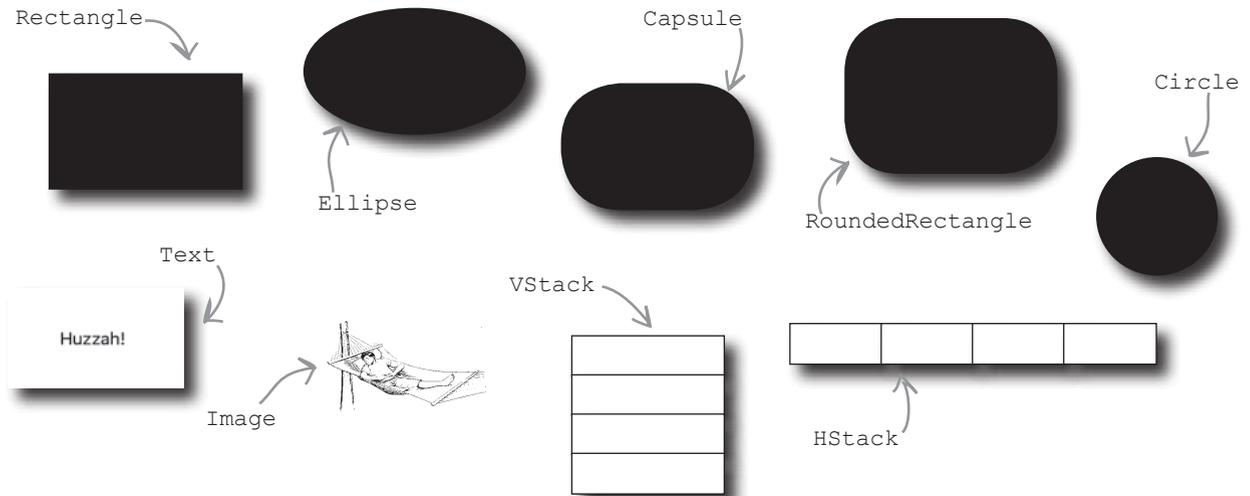
Выполните необходимую подготовку, чтобы интерфейс мог нормально функционировать.

# Программирование составляющих пользовательского интерфейса

## 1 Серый круг

Чтобы нарисовать серый круг, необходимо нарисовать... серый круг! Так как мы рисуем статический элемент интерфейса, лежащий под всеми остальными, ничего делать ему не нужно; круг просто должен существовать.

SwiftUI содержит много полезных представлений, таких как `Text`, `Image`, `VStack` и `HStack`. Но во фреймворке также можно найти много примитивных геометрических фигур, которыми можно пользоваться в разработке, таких как `Rectangle`, `Ellipse`, `Capsule`, `RoundedRectangle` и `Circle`.



### Чтобы нарисовать серый круг

Создайте представление `Circle`.

```
Circle()
 .stroke(Color.gray.opacity(0.2), style: strokeStyle)
```

*.stroke задает цвет и стиль линии...*

*strokeStyle — заранее определенный экземпляр StrokeStyle, который мы создали ранее.*

*... и вы получаете вот это!*

## 2 Программирование основного круга

Основной круг чуть сложнее серого. Он должен уменьшать свое кольцо в зависимости от того, сколько времени осталось, а его уменьшение должно быть реализовано анимацией.

```

Circle()
 .trim(from: 0, to: 1 - ((defaultTime - countdownTime) / defaultTime))
 .stroke(countdownColor, style: strokeStyle)
 .rotationEffect(.degrees(-90))
 .animation(.easeInOut)

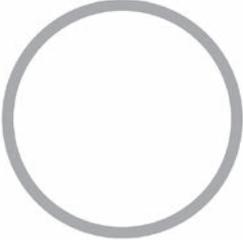
```

*.trim отсекает часть фигуры по ее представлению в виде траектории.*

*.stroke задает цвет контура фигуры.*

*.rotationEffect поворачивает элемент на заданный угол.*

*.animation задает темп анимации в представлении.*



## 3 Программирование показаний таймера и кнопок

Текущее время и кнопки не отображаются с использованием геометрических фигур, в отличие от двух кругов, нарисованных ранее. Для них мы воспользуемся элементами, уже использовавшимися ранее при построении списка текущих задач.

Прежде всего все элементы заключаются в панель HStack с заданными интервалами.

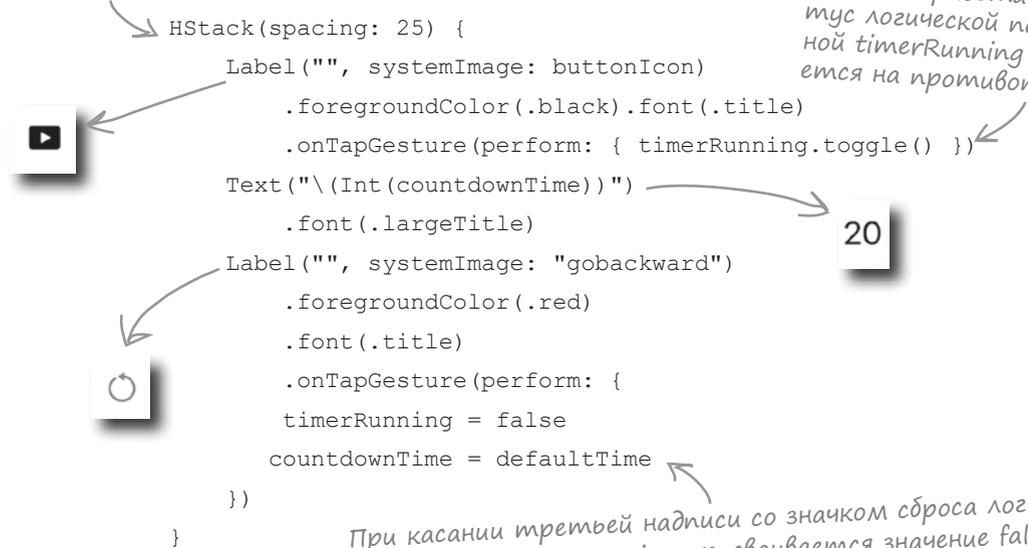
```

HStack(spacing: 25) {
 Label("", systemImage: buttonIcon)
 .foregroundColor(.black).font(.title)
 .onTapGesture(perform: { timerRunning.toggle() })
 Text("\ (Int (countdownTime)) ")
 .font(.largeTitle)
 Label("", systemImage: "gobackward")
 .foregroundColor(.red)
 .font(.title)
 .onTapGesture(perform: {
 timerRunning = false
 countdownTime = defaultTime
 })
}

```

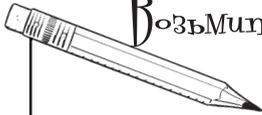
*Когда пользователь касается первой метки со значком запуска/приостановки, статус логической переменной timerRunning переключается на противоположный.*

*При касании третьей надписи со значком сброса логической переменной timerRunning присваивается значение false, а переменная countdownTime возвращается к значению defaultTime.*



Создайте все части пользовательского интерфейса.

## Возьмите В руку карандаш



Взгляните на следующие фрагменты кода SwiftUI. Для каждого фрагмента нарисуйте то, что он, по вашему мнению, должен выводить на экран, а если он не должен выводить ничего, попробуйте описать, что в нем не так и как исправить проблему.

Предполагается, что SwiftUI и PlaygroundSupport импортированы, а в среде Playground для прорисовки представления используется следующий код:

```
PlaygroundPage.current.setLiveView(ShapeView())
```

struct ShapeView: View {  
     var body: some View {  
         VStack {  
             Rectangle().padding()  
             Ellipse().padding()  
             Capsule().padding()  
             RoundedRectangle(cornerRadius: 25).padding()  
         }  
     }  
}

struct ShapeView: View {  
     var body: some View {  
         VStack {  
             Circle()  
             Circle()  
         }  
     }  
}

struct ShapeView: View {  
     var body: some View {  
         VStack {  
             RoundedRectangle(cornerRadius: 100).padding()  
         }  
     }  
}

→ Ответ на с. 353.

## Объединение трех элементов

Чтобы создать нужное представление, необходимо объединить все три элемента по порядку внутри `ZStack`:

```
ZStack {
 1 Circle()
 .stroke(Color.gray.opacity(0.2), style: strokeStyle)
 2 Circle()
 .trim(from: 0, to: 1 - ((defaultTime - countdownTime) / defaultTime))
 .stroke(countdownColor, style: strokeStyle)
 .rotationEffect(.degrees(-90))
 .animation(.easeInOut)
 HStack(spacing: 25) {
 Label("", systemImage: buttonIcon)
 .foregroundColor(.black).font(.title)
 .onTapGesture(perform: { timerRunning.toggle() })
 Text("\(Int(countdownTime))")
 .font(.largeTitle)
 Label("", systemImage: "gobackward")
 .foregroundColor(.red)
 .font(.title)
 .onTapGesture(perform: {
 timerRunning = false
 countdownTime = defaultTime
 })
 }
}.frame(width: 300, height: 300)
```

Подумайте, что мог бы делать элемент `ZStack`. Вскоре мы обсудим этот вопрос.

Ограничиваем `ZStack` аккуратным квадратом `300x300`, чтобы круг был не слишком большим.

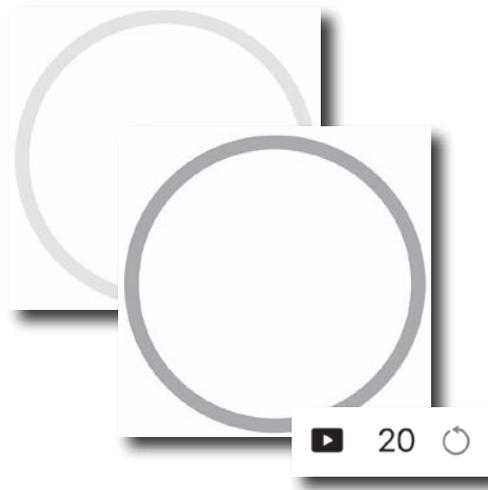
Объедините все элементы!

Погодите, а что такое `ZStack`? Я использовала `HStack` для горизонтальных макетов, `VStacks` для вертикальных макетов, но я не знаю, что делает `ZStack`!



**ZStack** работает по тому же принципу, что и **HStack** или **VStack**, но вместо размещения по горизонтали или вертикали элементы в этой панели размещаются по оси **z** (то есть оси глубины).

В приложении **Executive Timer** необходимо было использовать **ZStack**, потому что серый круг должен находиться под всем остальным, основной круг должен размещаться над ним (чтобы серый круг становился видимым при уменьшении основного круга), а показания таймера и кнопки — над всем остальным.



## Мозговой шторм

Создайте новую среду **Swift Playground**, добавьте в нее несколько представлений **Text** (возможно, в **HStack** и/или **VStack**) и графическое изображение, если вам захочется проявить фантазию. Затем поэкспериментируйте с упаковкой всех элементов в **ZStack** и посмотрите, что произойдет. Поэкспериментируйте с вложением панелей **ZStack**, чтобы каждая панель содержала несколько представлений **Text**.

Вы быстро поймете, как работают элементы **ZStack**!

## Завершающие штрихи

Осталось заполнить последний фрагмент головоломки: добавить к представлению действие для события `onReceive`. Это позволит представлению реагировать на события таймера, созданного ранее, и выполнять необходимую обработку этой информации:

*Здесь размещаются три визуальных фрагмента пользовательского интерфейса, созданные выше. Мы не приводим их, чтобы код поместился на странице.*

```
ZStack {
 1 2 3
 .frame(width: 150 * 2, height: 150 * 2)
 .onReceive(timer, perform: { _ in
 guard timerRunning else { return }
 if countdownTime > 0 {
 countdownTime -= 1
 } else {
 timerRunning = false
 countdownTime = defaultTime
 }
 })
}
```

*.onReceive* — метод экземпляра, доступный для представлений, позволяющих задать публикатор, на который вы хотите реагировать, и метод, который должен выполняться при генерации данных этим публикатором.

Параметр `perform` задает метод, который должен выполняться при выдаче данных заданным публикатором (в нашем примере таймером).

- Обеспечьте обновление пользовательского интерфейса таймером.

## И теперь всё готово к тестированию приложения!

Выполняемый метод представляет собой замыкание, которое отслеживает `timerRunning` (то есть оно происходит, пока работает таймер). Оно уменьшает `countdownTime` на единицу, если таймер еще не достиг нуля, или отключает таймер и сбрасывает время к значению по умолчанию при обнулении таймера.

### Работа с публикаторами

Наш таймер фактически представляет собой изменяемый блок состояния, который определяет, что должен делать пользовательский интерфейс приложения: он должен отображать количество секунд, оставшихся на таймере, а также обновлять изображение кольца в соответствии с оставшимся временем.

Метод `onReceive` сообщает SwiftUI, что мы хотим получать генерируемые события заданного публикатора (в данном случае таймера) и выполнять функцию при их получении.

Метод `perform` обновляет значение `countdownTime`, которое применяется пользовательским интерфейсом для прорисовки своего состояния.

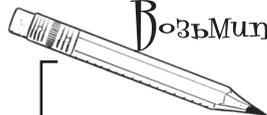
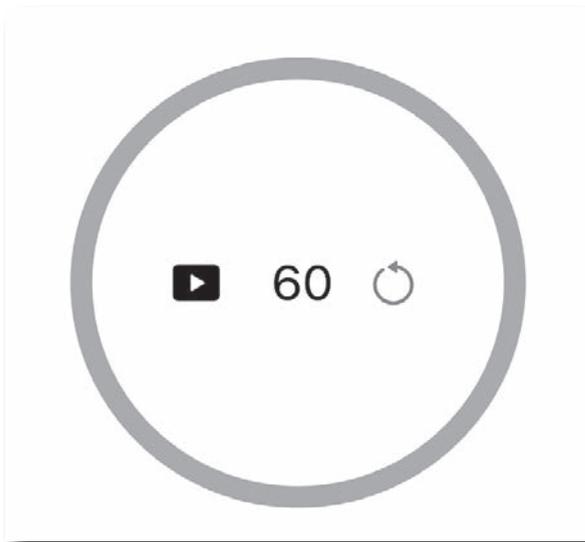


# Тест-драйв



Запустите приложение и посмотрите, что произойдет...

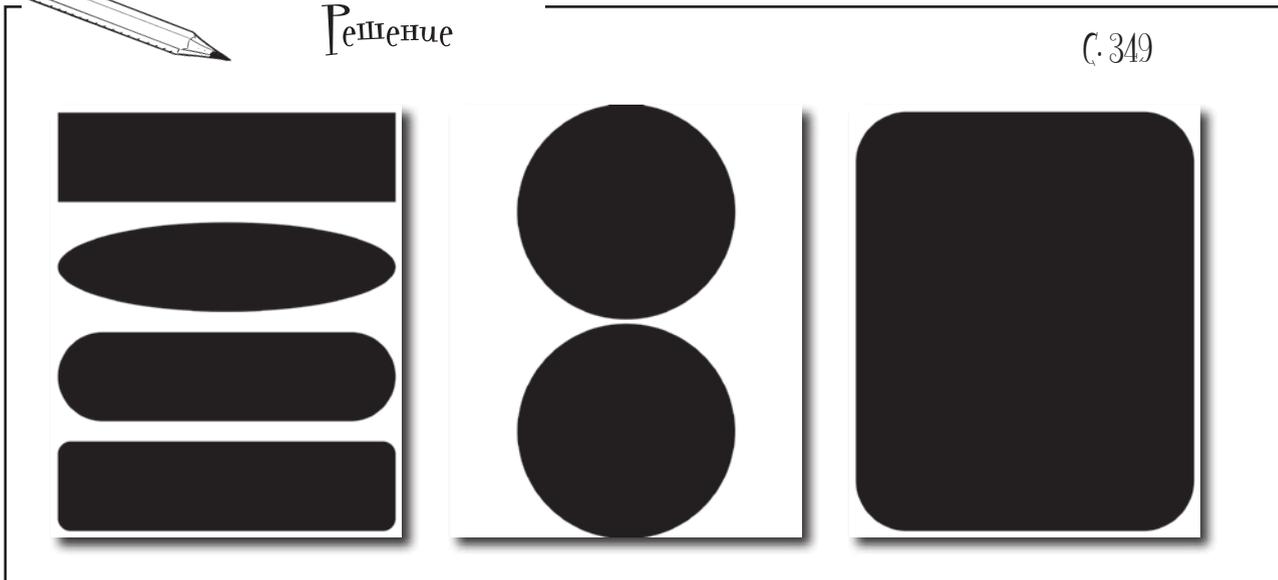
Результат должен выглядеть примерно так:



Возьмите в руку карандаш

Решение

С. 349



## Представления с вкладками



Во многих моих любимых приложениях для iOS предусмотрен симпатичный интерфейс с набором вкладок. Как мне реализовать эту возможность в SwiftUI? Вообще не представляю, как это будет работать. Помогите!



**SwiftUI** содержит полезные контейнерные представления, с которыми любое представление можно легко разместить на отдельной вкладке.

Одно из таких полезных **контейнерных представлений** — `TabView` — позволяет разместить разные представления на разных вкладках вашего приложения.

Чтобы использовать `TabView`, необходимо выполнить следующие шаги:

**1** **Создайте представления, которые вам нужны**

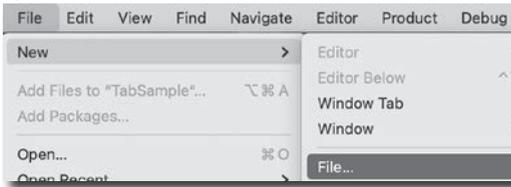
На каждой вкладке должно что-то выводиться. Позаботьтесь о том, чтобы у вас были представления для размещения на вкладках.

**2** **Постройте представление `TabView`, содержащее ваши представления**

Используйте контейнерное представление SwiftUI `TabView` для размещения вкладок и вывода пользовательского интерфейса с вкладками.

# 1 Создайте все представления (которые должны существовать на вкладках)

Добавьте новый файл в проект командой меню File.



Выберите шаблон SwiftUI View из предложенных вариантов.



Введите имя представления.



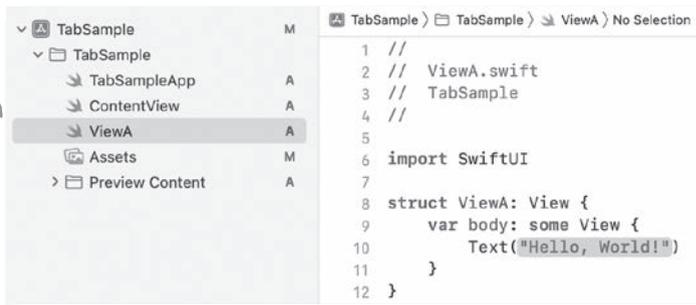
Создайте несколько представлений (например, ViewA, ViewB и ViewC), которые будут размещаться на вкладках TabView.

Добавьте какой-нибудь текст, чтобы отличать представления друг от друга.

```
struct ViewA: View {
 var body: some View {
 Text("This is ViewA!")
 }
}
```

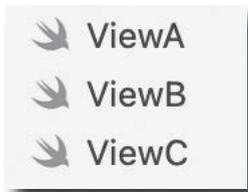
Щелкните на кнопке Create, чтобы добавить представление в проект.

Новое представление появляется на панели Project.



## 2 Постройте представление TabView с вашими представлениями

Когда представления будут готовы...



Заклучите все представления в TabView.

В том представлении, в котором должны отображаться вкладки:

```

struct ContentView: View {
 var body: some View {
 TabView {
 1 ViewA()
 .padding()
 .tabItem {
 Image(systemName: "a.square")
 Text("View A")
 }
 .tag(1)
 ViewB()
 .padding()
 .tabItem {
 Image(systemName: "b.square")
 Text("View B")
 }
 .tag(2)
 3 ViewC()
 .padding()
 .tabItem {
 Image(systemName: "c.square")
 Text("View C")
 }
 .tag(3)
 }
 }
}

```

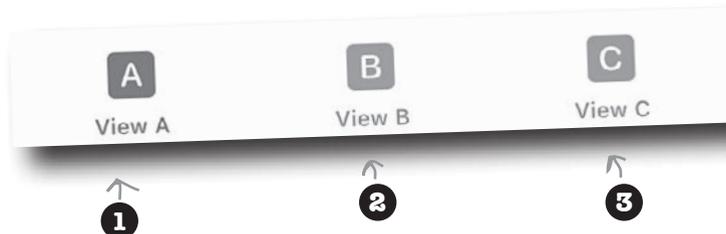
И вызовите каждое представление (в данном случае ViewA).

С подходящим отступом.

Информация, которая должна выводиться на вкладке: в данном случае изображение из SF Symbols и некий текст.

А также числовые теги, определяющие порядок вкладок.

Представления, помещенные в TabView, будут выводиться на вкладках. Модификатор tabItem добавляет информацию, которая будет отображаться на вкладке.



## Полученные Вкладки

Если вы создали три представления и содержащий их элемент `TabView`, результат будет выглядеть примерно так:



### Знакомство с Label

Вместо того чтобы объединять представления `Image` и `Text`, можно воспользоваться представлением `Label`. `Label` объединяет графическое изображение и текст.

Таким образом, одна из вкладок может содержать вызов:

```
Label("View B", systemImage: "b.square")
```

ВМЕСТО:

```
Image(systemName: "b.square")
```

```
Text("View B")
```



### Тест-драйв

Обновите код трех вкладок, чтобы в них использовались представления `Label` вместо двух представлений `Image` и `Text`.



Легко! Просто упакуйте представление с таймером и новое представление **Settings** в **TabView**.

Вы уже знаете, как сделать все необходимое, — предполагается, что представление **Settings** пусто (вскоре мы доберемся до этого).



## Упражнение

Обновите приложение **Executive Timer** так, чтобы оно отображало две вкладки: одну для **Timer**, другую для представления **Settings**. Для этого необходимо выполнить следующие действия:

- Переименуйте представление **ContentView**, в котором отображается таймер, в **TimerView**. Переименовать необходимо как структуру **ContentView** (в **TimerView**), так и файл (на боковой панели Xcode).
- Включите в проект новый файл **SwiftUI View**, присвойте ему имя **ContentView**.
- В новом представлении **ContentView** добавьте элемент **TabView** и вкладку для отображения **TimerView**.
- Включите в проект новый файл **SwiftUI View**, присвойте ему имя **TimerSettingsView**.
- В представлении **ContentView** добавьте в **TabView** вторую вкладку **TimerSettingsView**.



## Упражнение Решение

### Переименуйте существующий файл ContentView:

Выберите ContentView на панели Project в левой части окна Xcode.

Измените имя ContentView.swift на TimerView.swift в правой части окна Xcode на панели File Inspector.



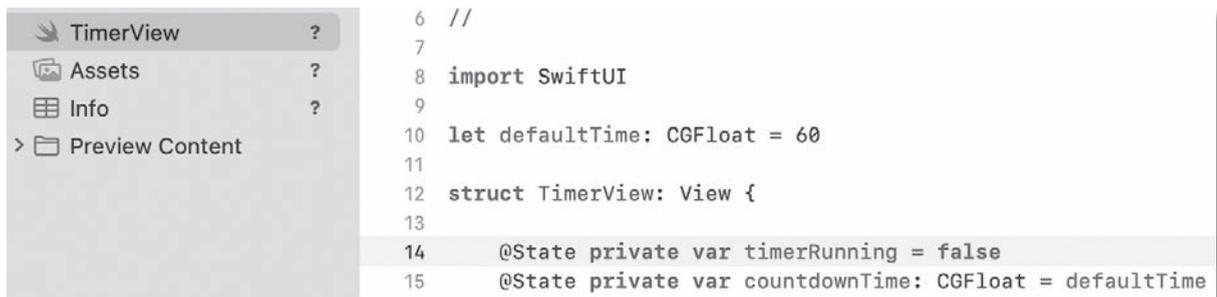
### Переименуйте существующую структуру ContentView:

В коде из только что переименованного файла TimerView.swift переименуйте структуру представления из ContentView в TimerView.

```
struct ContentView: View {
```

```
struct TimerView: View {
```

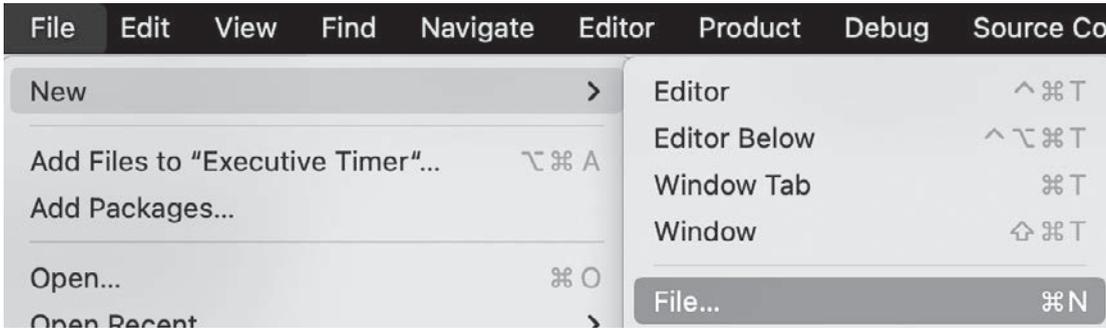
### Переименованное представление TimerView должно выглядеть так:



## Создание нового представления ContentView с вкладками

Чтобы отобразить набор вкладок, нам понадобится новое представление, содержащее TabView. Это новое представление будет называться ContentView, и оно заменит предыдущее представление ContentView (которое мы только что переименовали в TimerView).

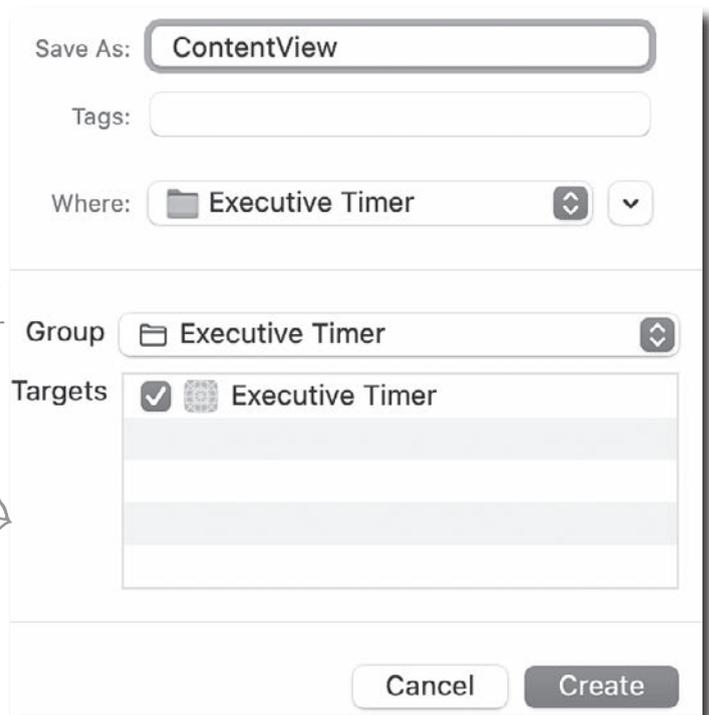
Выберите команду File... из подменю New меню File среды Xcode.



На следующем экране выберите тип файла, который добавляется в проект, — SwiftUI View.



Присвойте новому файлу имя ContentView и щелкните на кнопке Create, чтобы добавить файл в проект Executive Timer.



## Создание вкладок и TabView

В новом представлении ContentView необходимо создать представление TabView. Добавьте его:

```
import SwiftUI

struct ContentView: View {
 var body: some View {
 TabView {

 }
 }
}
```

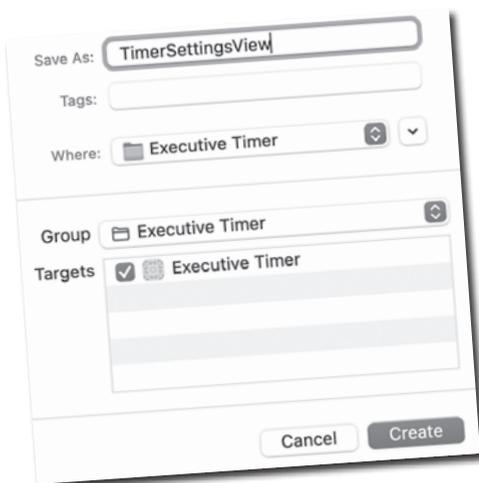
Внутри TabView добавьте элемент Tab для отображения таймера:

```
TimerView()
 .padding()
 .tabItem {
 Image(systemName: "timer")
 Text("Timer")
 }
 .tag(1)
```

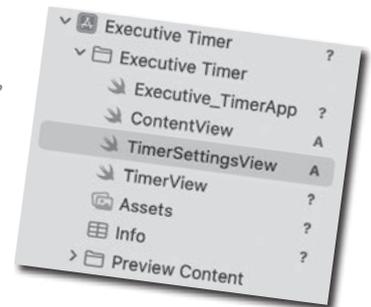
*Создает экземпляр TimerView.*



### 1 Создайте пустое представление TimerSettingsView



Вам не придется изменять код, который мы здесь приводим, если только вы не захотите, чтобы он выводил что-то другое. На следующем шаге мы займемся созданием настроек таймера.



```
import SwiftUI

struct TimerSettingsView: View {
 var body: some View {
 Text("Hello, World!")
 }
}
```

## 2 Добавьте вторую вкладку в TabView из ContentView

В настоящее время представление ContentView выглядит так, как показано ниже, и содержит только одну вкладку (TimerView). Добавим вкладку TimerSettingsView (в настоящее время она пуста):

```
struct ContentView: View {
 var body: some View {
 TabView {
 TimerView()
 .padding()
 .tabItem {
 Image(systemName: "timer")
 Text("Timer")
 }
 .tag("Timer")
 }
 }
}
```



```
TimerSettingsView()
 .padding()
 .tabItem {
 Image(systemName: "gear")
 Text("Settings")
 }
 .tag("Settings")
```

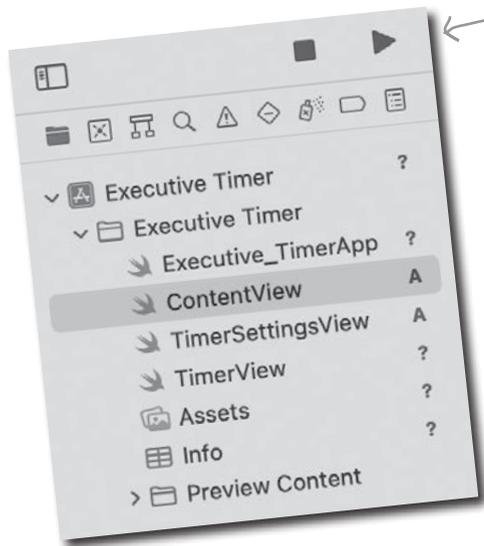
Добавьте TimerSettingsView как вкладку с отступом и тегом (вкладка 2) и набором модификаторов .tabItem, сходных с модификаторами первой вкладки.

Возможно, вы заметили, что мы обновили модификатор .tag на первой вкладке с типом String (и использовали String для второй вкладки). Это упрощает будущие обращения к вкладкам в нашем коде по имени тега.

### Работа подошла к концу! Теперь приложение Executive Timer использует вкладки.

- Переименуйте представление ContentView, в котором отображается таймер, в TimerView. Переименовать необходимо как структуру ContentView (в TimerView), так и файл (на боковой панели Xcode).
- Включите в проект новый файл SwiftUI View, присвойте ему имя ContentView.
- В новом представлении ContentView добавьте элемент TabView и вкладку для отображения TimerView.
- Включите в проект новый файл SwiftUI View, присвойте ему имя TimerSettingsView.
- В представлении ContentView добавьте в TabView вторую вкладку TimerSettingsView.

## Запустите новую версию Executive Timer



Запустите Executive Timer и оцените мощь вкладок!



На второй вкладке выводится незавершенное представление TimerSettingsView.



На первой вкладке выводится представление TimerView.

И это все, что мне  
нужно знать о SwiftUI?  
Я могу идти?

**Да... и нет.**

Чтобы описать SwiftUI во всей красе, нам пришлось бы написать отдельную книгу из серии Head First.

Мы всего лишь постарались рассказать понемногу обо всем, чтобы дать вам представление о SwiftUI: вы ознакомились с основами, и теперь, когда вы продолжите изучать другие части SwiftUI, вы уже будете знать, что делаете.

Неплохо придумано, верно?



## 12. Приложения, Веб-программирование и Все такое

# Собирая все вместе

Проведем вместе еще одну главу...



**Вы значительно продвинулись в изучении Swift.** Вы освоили Playgrounds и Xcode. Было понятно, что когда-нибудь нам придется **попрощаться**, и сейчас этот момент настал. Расставаться нелегко, но мы знаем, что вы справитесь. В этой главе — последней, в которой мы будем вместе с вами (в этой книге), — мы еще раз **пройдемся по многим концепциям**, которые вы изучили, и совместно построим несколько приложений. Мы убедимся в том, что ваши навыки Swift закреплены, и дадим некоторые рекомендации относительно того, **что делать дальше**, — своего рода домашнее задание, если хотите. Это будет интересно, и мы расстанемся на высокой ноте.

Эту главу можно сравнить со сценой в детективном романе, в которой великий сыщик собирает всех в гостиной и раскрывает подоплеку дела.

## Путешествие должно завершиться...

Вы узнали много всего о Swift, но еще осталось время, и мы проведем вместе еще одну главу...



Переменные

Константы

Типы

Операторы

Строки

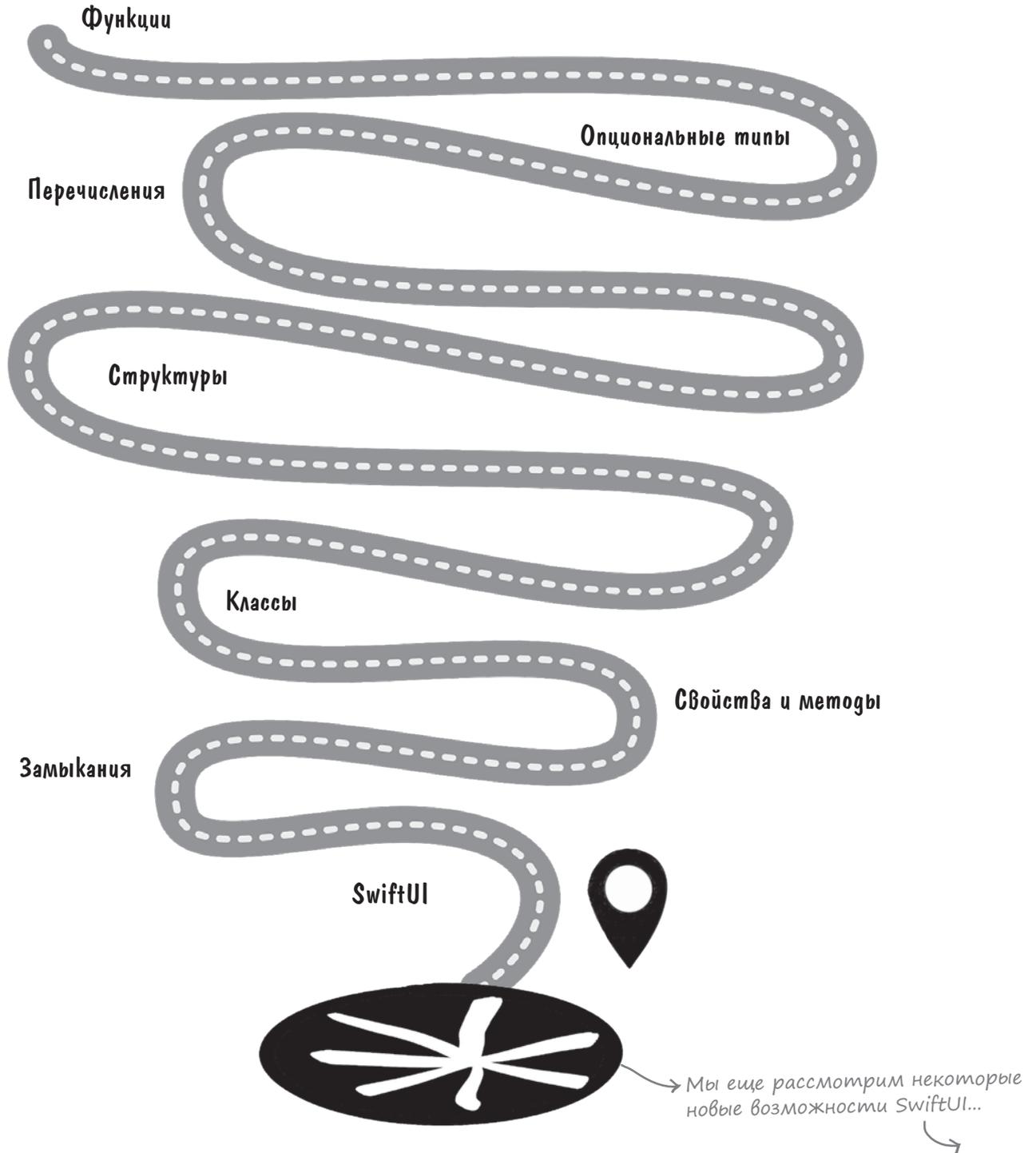
Коллекции

Условные команды

Циклы

Команды выбора

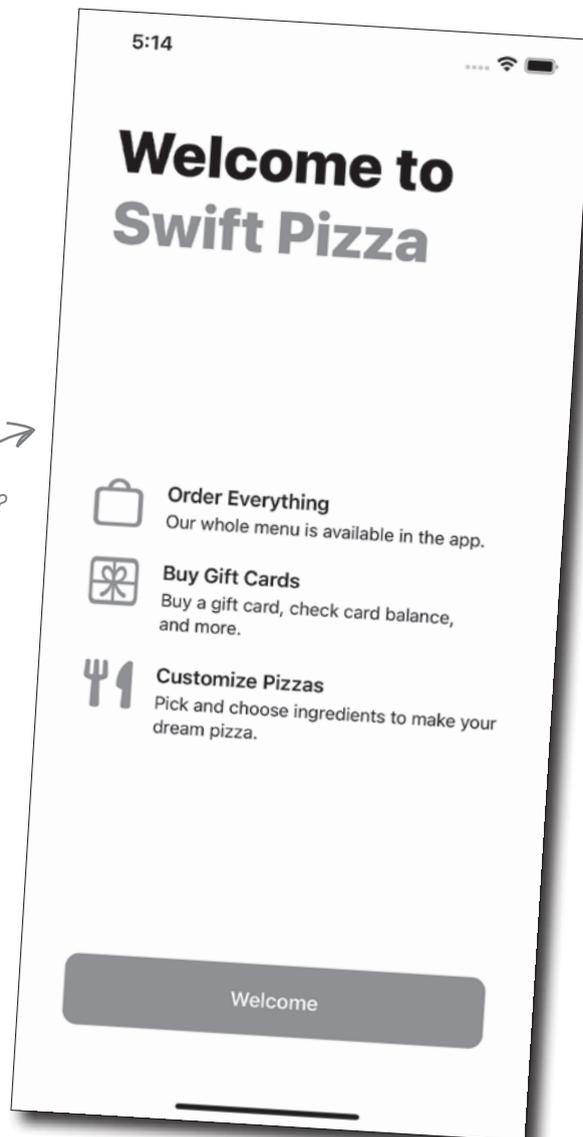


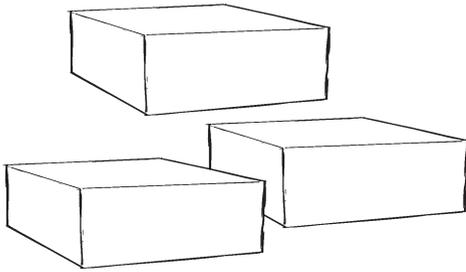




Что-то в этом роде?

Прекрасно, что вы узнали столько всего о Swift, но не могли бы вы сделать красивую заставку для моего приложения с пиццей? Знаете, из тех, которые есть в любом крупном приложении!





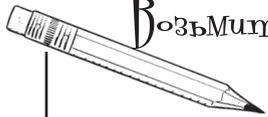
**SwiftUI позволяет легко сконструировать красивую заставку из блоков.**

Давайте посмотрим, как использовать структурные элементы SwiftUI для построения заставки.

Заставка состоит из тех же основных элементов, которые вы уже использовали в SwiftUI (и не один раз).

Среди них нет ничего незнакомого: элементы `Text`, `Spacer`, несколько `HStack` и `VStack`, несколько элементов `Image` и `Button`.

## Возьмите в руку карандаш



Как вы думаете, из каких основных элементов будет строиться экран заставки?

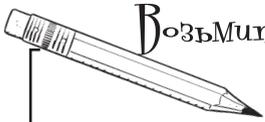
Мы нарисовали эскиз такого экрана. Попробуйте определить, из каких разновидностей структурных элементов SwiftUI вы бы его построили.

Мы оставили несколько подсказок, чтобы вам было проще начать работу.

Большое приветственное сообщение строится из представлений `Text` в `VStack`, так что они располагаются друг над другом. Одно из них будет окрашено в синий цвет.

Что должно заполнять оставшуюся часть заставки? Не забудьте о `VStack`, `HStack` и `Spacer`!





Возьмите в руку карандаш

Решение

Еще один элемент Text, также с крупным шрифтом и цветом

Text с большим размером шрифта

VStack для расположения всех элементов по вертикали

VStack для вертикального размещения строк текста

Text с жирным шрифтом

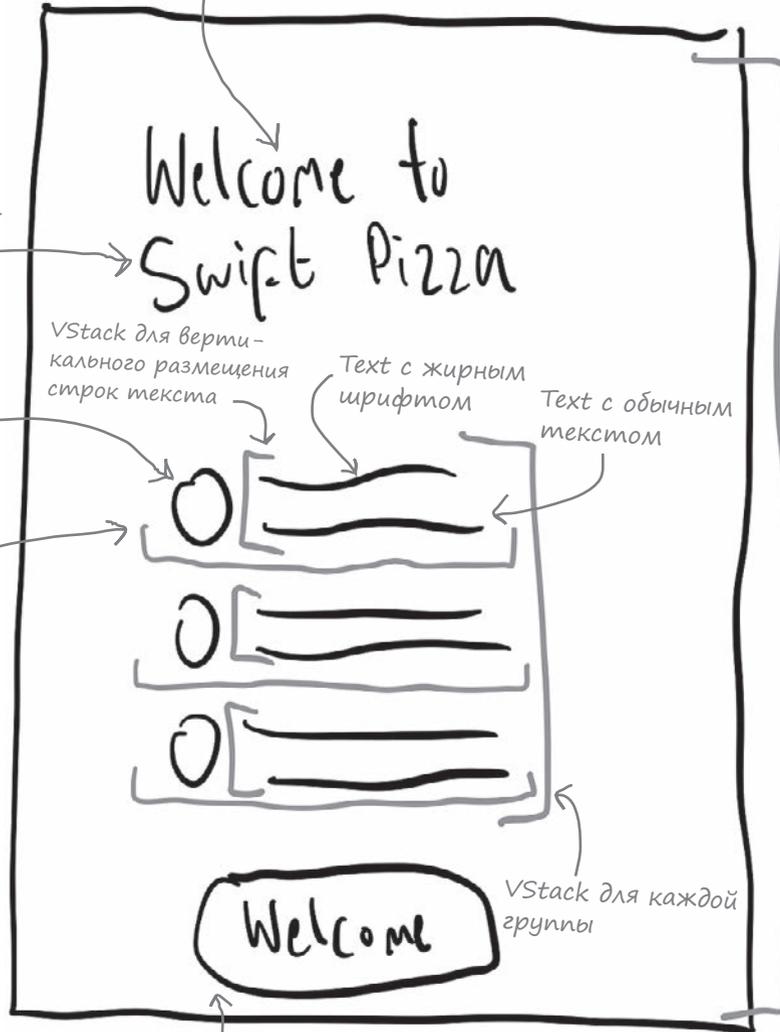
Text с обычным текстом

Image

HStack для размещения Image рядом с Text

VStack для каждой группы

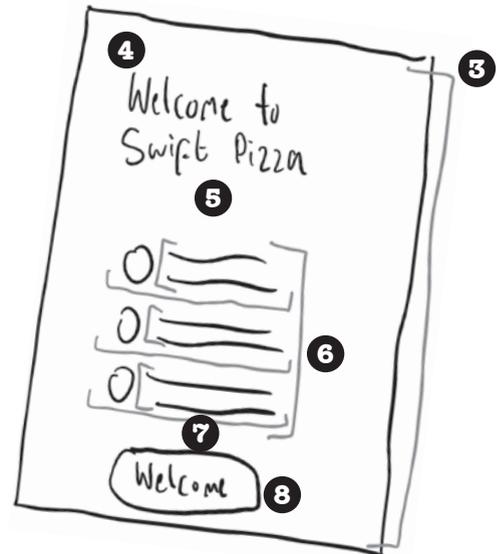
Кнопка Button, красивая и синяя (наверное)



Как же построить этот экран со SwiftUI? Все концепции вроде бы знакомы, но мне нужна помощь!

## Построение заставки

- 1 Убедитесь в том, что структура `ContentView` готова к использованию. Мы оставим ей имя `ContentView`, которое SwiftUI default использует по умолчанию для первого представления в проекте Xcode.
- 2 Также убедитесь в том, что свойство `body` готово к работе. Ничего таинственного, все просто.
- 3 Создайте элемент `VStack`, в котором будет размещаться все остальное, снабдите его отступом.
- 4 Добавьте два представления `Text` для вывода сообщения «Welcome to Swift Pizza» и оформите их соответствующим образом.
- 5 Добавьте представление `Spacer`.
- 6 Добавьте представления для нескольких элементов: `VStack` для хранения всего остального и трех идентичных панелей `HStack`, каждая из которых содержит `Image`, и еще одну панель `VStack` с двумя представлениями `Text`.
- 7 Добавьте еще одно представление `Spacer`.
- 8 И наконец, добавьте кнопку `Button` с соответствующим оформлением.



## Пошаговая сборка экрана заставки

### ContentView

```
1 struct ContentView: View {

}
```

### body

```
2 var body: some View {

}
```

### VStack для размещения всего

```
3 VStack (alignment: .leading) {

}.padding(.all, 40)
```

Добавляет 40 единиц отступа ко всему содержимому VStack.

### Текст приветствия

```
4 Text("Welcome to")
 .font(.system(size: 50)).fontWeight(.heavy)
 .foregroundColor(.primary)
Text("Swift Pizza")
 .font(.system(size: 50)).fontWeight(.heavy)
 .foregroundColor(Color(UIColor.systemBlue).opacity(0.8))
```

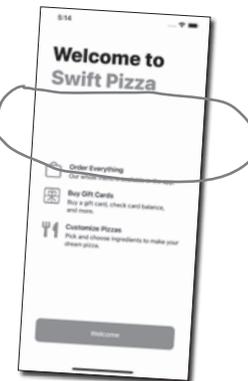
### Spacer

```
5 Spacer()
```

Элемент *Spacer* будет занимать все доступное место по той оси, по которой он может расширяться. Так как элемент *Spacer* находится внутри панели *VStack*, содержащей все остальное, он расширяется по вертикали настолько, насколько это возможно. Это создаст необходимое пустое место на заставке.

Вы можете присвоить *ContentView* любое имя, но шаблон приложения *SwiftUI* по умолчанию ищет представление с именем *ContentView*. (Впрочем, имя *body* изменить нельзя.)

Выбор режима выравнивания *.leading* означает, что содержимое будет выравниваться по левому краю в локальных контекстах с направлением слева направо и по правому краю в локальных контекстах с направлением справа налево.



## 6 Основные возможности

Все три группы основных возможностей выстроены по вертикали.

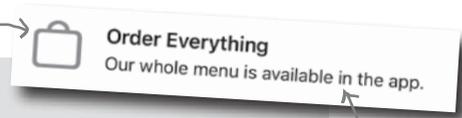
Каждая группа представляет собой элемент HStack, что позволяет нам разместить изображение слева от текста.

```
VStack (alignment: .leading, spacing: 24) {
```

```
 HStack (alignment: .top, spacing: 20) {
```

```
 Image(systemName: "bag").resizable()
 .frame(width: 40, height: 40)
 .foregroundColor(Color(UIColor.systemBlue).opacity(0.8))
```

```
 VStack (alignment: .leading, spacing: 4) {
 Text("Order Everything").font(.headline).bold()
 Text("Our whole menu is available in the app.")
 .font(.subheadline)
 }
```



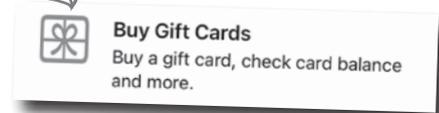
```
 }
```

Эта структура дублируется для всех трех групп.

```
 HStack (alignment: .top, spacing: 20) {
```

```
 Image(systemName: "giftcard").resizable()
 .frame(width: 40, height: 40)
 .foregroundColor(Color(UIColor.systemBlue).opacity(0.8))
```

```
 VStack (alignment: .leading, spacing: 4) {
 Text("Buy Gift Cards").font(.headline).bold()
 Text("Buy a gift card, check card balance, and more.")
 .font(.subheadline)
 }
```



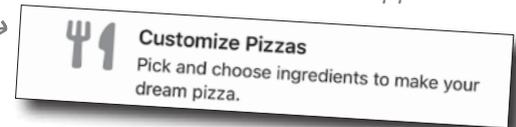
```
 }
```

Используемые изображения взяты из библиотеки SF Symbols компании Apple.

```
 HStack (alignment: .top, spacing: 20) {
```

```
 Image(systemName: "fork.knife").resizable()
 .frame(width: 40, height: 40)
 .foregroundColor(Color(UIColor.systemBlue).opacity(0.8))
```

```
 VStack (alignment: .leading, spacing: 4) {
 Text("Customize Pizzas").font(.headline).bold()
 Text("Easily View Stock Options, Quotes, Charts etc.")
 .font(.subheadline)
 }
```



```
 }
```

```
}
```

```
}
```

# Spacer или Padding

## Еще один Spacer



Возможно, вас интересует, чем использование Spacer отличается от добавления отступов вокруг представлений SwiftUI при помощи padding. Ответ прост. Spacer расширяется, чтобы заполнить все свободное пространство между представлениями SwiftUI (в том направлении, в котором они могут расширяться, например в зависимости от того, находятся ли они внутри VStack или HStack).

Отступы позволяют точнее управлять интервалами между объектами: вы можете запросить конкретную величину отступов с конкретной стороны представления. Если вы запросите создание отступа без указания конкретной величины, система постарается подобрать оптимальный вариант за вас.

## Button

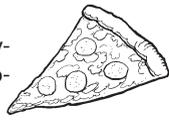
```

8 Button(action: {}) {
 Text("Welcome").foregroundColor(.white).bold()
}.frame(width: 350, height: 60)
 .background(Color(UIColor.systemBlue))
 .opacity(0.8).cornerRadius(12)

```

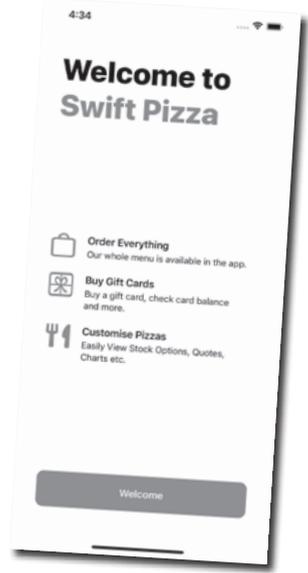
## Серьезное программирование

Итак, мы рассмотрели все элементы. Теперь запустите Xcode и посмотрите, удастся ли вам воспроизвести заставку Swift Pizza.



Не бойтесь экспериментировать с использованием новых средств выделения текста, выбором других значков или творческим использованием цветов!

Создайте новый проект Xcode с пустым проектом iOS.



**SwiftUI автоматически поддерживает темный режим! В большинстве случаев вам вообще ничего делать не придется.**

Но возможности автоматической поддержки не безграничны. Иногда вам придется вмешиваться в происходящее и вносить исправления.

Допустим, в вашем приложении имеется следующий код для отображения карты, в центре которой находится площадь Таймс-сквер в Нью-Йорке:

```
import SwiftUI
import MapKit

struct ContentView: View {
 @State private var region: MKCoordinateRegion =
 MKCoordinateRegion(center:
 CLLocationCoordinate2D(latitude: 40.75773, longitude: -73.985708),
 span: MKCoordinateSpan(latitudeDelta: 0.05, longitudeDelta: 0.05))

 var body: some View {
 Map(coordinateRegion: $region)
 .edgesIgnoringSafeArea(.all)
 }
}
```

Чтобы работать с картой, необходимо импортировать MapKit.

Переменная состояния используется для определения области карты, которую вы хотите просматривать.

var body: some View {

Map(coordinateRegion: \$region)

.edgesIgnoringSafeArea(.all)

Указывает на площадь Таймс-сквер в Нью-Йорке.  
Отображает карту с регионом, определенным выше. Дополнительный параметр приказывает iOS растянуть карту до краев отображаемой области.



Запустив эту программу в эмуляторе, вы увидите примерно такой результат:



Если вы воспользуетесь функцией Toggle Appearance в эмуляторе, изображение переключится в темный режим.



...И карта изменится. Волшебство!



## Мозговой Штурм

Попробуйте создать собственный пользовательский интерфейс на базе SwiftUI для тестирования автоматической реализации темного режима. Это настолько просто, что вряд ли этому можно посвятить упражнение, но все равно попробуйте — это просто потрясающе!

### Эксперименты с темным режимом

- Создайте новое приложение iOS в Xcode.
- В пустом представлении `ContentView`, которое было сгенерировано за вас, создайте элемент `VStack`.
- В представлении `VStack` добавьте элемент `Text`, несколько изображений `Image` из `SF Symbols`, несколько кнопок `Button` и вообще все, что вам захочется.
- Добавьте карту `Map`, если хочется приключений.
- Запустите приложение в эмуляторе iOS и посмотрите, как оно выглядит.
- Переключитесь в темный режим командой меню `Toggle Appearance`. Как оно выглядит теперь?



Что выглядит не так?

А что выглядит так, как надо?

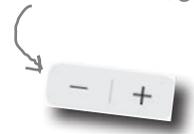


### Представления SwiftUI являются функциями своего состояния.

На практике это означает, что когда вы хотите, чтобы представление изменилось, вы изменяете данные — состояние, — которые используются для формирования представления. Представление автоматически обновляется, отражая эти изменения. Если говорить более конкретно, SwiftUI предоставляет некоторые средства для хранения этого состояния.

Вы уже работали с одним из этих средств — оберткой свойства `@State`. Вот как это выглядит:

Представление Stepper из SwiftUI



```
struct ContentView: View {
 @State private var number: Int = 0

 var body: some View {
 Stepper(value: $number, in: 0...10, label: {Text("Number is \(number)")})
 }
}
```

Создаем свойство для хранения числа с пометкой `@State`.

Представление Stepper, связанное с числом

Stepper — элемент для пошагового увеличения или уменьшения значения.

Использование `@State` перед объявлением свойства (которое в остальном представляет собой обычное значение `int`) сообщает Swift, что вы хотите управлять этим свойством и хранить его в памяти, пока существует представление, которому это состояние принадлежит. Так как свойство имеет пометку `@State`, SwiftUI автоматически перерисовывает представление при изменении свойства.

← Это важно, потому что представления SwiftUI являются структурами, а как вы знаете из предыдущей работы со структурами, структуры изменяться не могут.

`@State` позволяет эффективно работать с любым свойством, принадлежащим одному конкретному представлению; свойство неактуально и не используется за пределами представления. Это одна из причин, по которым оно помечается ключевым словом `private`.

← Состояние никогда не выходит за пределы представления, которому оно принадлежит. Из-за этого состояние объявляется приватным.

## Совместное использование состояния

### @State

@State — самый простой представитель этой группы. Вам уже знакомо это ключевое слово; это простой источник информации для представлений в ваших приложениях. Оно используется с простыми значениями (Int, String, Bool и т. д.) и не предназначается для сложных ссылочных типов — или типов, которые вы создаете сами на базе классов и структур.

@State означает, что переменная body представления, связанного с ней, будет пересчитываться заново при каждом обновлении свойства.

### @StateObject

@StateObject чуть сложнее @State. Этот модификатор сообщает SwiftUI, что переменная body должна пересчитываться заново при изменении любых свойств с оберткой свойства @Published в объекте, помеченном оберткой свойства @StateObject.

Любой объект с пометкой @StateObject должен поддерживать протокол ObservableObject.

### @ObservedObject

@ObservedObject позволяет отслеживать изменения в уже созданных экземплярах @StateObject.

Когда вам понадобится передавать @StateObject между представлениями, все представления (кроме тех, которые его создали) снабжают данное свойство пометкой @ObservedObject.

@ObservedObject сообщает Swift, что объект уже был создан, и вам нужно, чтобы (дочернее) представление, которое к нему обращается, получало доступ к тем же данным без того, чтобы создавать объект заново.

### @EnvironmentObject

@EnvironmentObject относится к экземпляру @StateObject, который был создан где-то в ваших представлениях и присоединен к конкретному представлению. Когда вам нужно использовать его в дочернем представлении где-то вдали от представления, которому он принадлежит, вы используете @EnvironmentObject для обращения к нему. При поиске учитывается тип, а это означает, что в одном дереве представлений не может быть двух @EnvironmentObject с одинаковыми типами.

## На помощь приходит старый знакомый...

надуманный пример

↳ @StateObject и @ObservedObject

### Тип, представляющий счет в игре:

```
class GameScore: ObservableObject {
 @Published var numericalScore = 0
 @Published var piecesCaptured = 0
}
```

GameScore поддерживает протокол ObservableObject. Это позволяет использовать экземпляры GameScore внутри представления, а представление будет перезагружаться при изменении GameScore.

Обертка свойства @Published приказывает SwiftUI инициализировать обновление представлений, когда изменяется значение одного из таких свойств.

### Несколько представлений, использующих наш класс:

Создаваемый здесь объект должен поддерживать ObservableObject. К счастью, он поддерживает.

```
struct ContentView: View {
 @StateObject var score = GameScore()

 var body: some View {
 VStack {
 Text("Score is \(score.numericalScore),
 \(score.piecesCaptured) pieces captured.")
 ScoreView(score: score)
 }
 }
}
```

Наше свойство score, экземпляр созданного нами типа GameScore, использует обертку свойства @StateObject. @StateObject означает, что представление готово перезагружаться при изменении любых свойств с пометкой @Published внутри ObservableObject.

Здесь создается второе представление (ScoreView), которому передается свойство score.

```
struct ScoreView: View {
 @ObservedObject var score: GameScore

 var body: some View {
 Button("Bigger score!") {
 score.numericalScore += 1
 }
 Button("More pieces!") {
 score.piecesCaptured += 1
 }
 }
}
```

Нашему свойству score в этом представлении, которое будет использоваться внутри ContentView, не нужно создавать новый экземпляр score; ему достаточно отслеживать свойство, экземпляр которого уже был создан (с использованием обертки свойства @StateObject). По этой причине score здесь объявляется с модификатором @ObservedObject.

Обновление свойств score приводит к обновлению исходного @StateObject.



А как было бы замечательно, если бы я могла передавать информацию между представлениями...

## Построение приложения с несколькими представлениями, совместно использующими состояние

→ @StateObject  
и @EnvironmentObject

### Первый экран нашего гипотетического приложения:

Кнопка для увеличения счета...

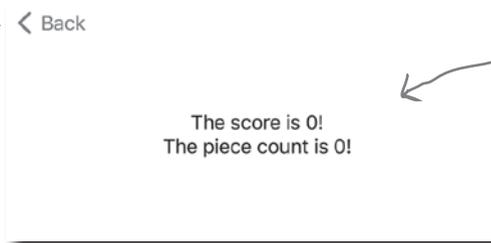
... и для увеличения счетчика фигур.

А также кнопка для перехода к другому представлению, в котором мы сможем просматривать эти значения.



### Второй экран гипотетического приложения:

Можно вернуться к первому представлению...



Выводится значение счета и счетчик фигур.



Мозговой  
Штурм

Удастся ли вам построить это приложение (или хотя бы его часть)? Начните с пустого приложения SwiftUI для iOS в среде Xcode, ограничиваясь одним исходным файлом (по умолчанию *ContentView.swift*). Вам придется создать класс, который представляет экземпляр *GameScore* со свойствами *score* и *pieces* (с поддержкой *ObservableObject*). Используйте свои знания состояния для создания всего остального. Если вам не удастся придумать, как разбить приложение на два представления, между которыми можно переключаться, разместите все на одном экране.

## Упражнение



# Построение приложения с двумя представлениями

Вот что нам предстоит сделать:

- Реализуйте класс `GameScore` для хранения счета и счетчика фигур. Класс должен поддерживать протокол `ObservableObject`.
- Постройте первое представление `ContentView`, содержащее `@StateObject` с типом `GameScore` и все необходимые компоненты.
- Постройте второе представление `ScoreView`, используя `@EnvironmentObject` для получения доступа к `@StateObject` с типом `GameScore` из первого представления, а также все необходимые компоненты.
- Реализуйте возможность перехода между представлениями.

## ObservableObject

- 1 Нам понадобится новый тип для хранения двух значений: счета и счетчика фигур (и то и другое — целые числа).

```
class GameScore {
 var score = 0
 var pieces = 0
}
```

- 2 Тип должен поддерживать протокол `ObservableObject`, что позволяет публиковать его свойства при помощи обертки свойства `@Published`.

```
class GameScore: ObservableObject {
 @Published var score = 0
 @Published var pieces = 0
}
```

*Что-то подозрительно просто...*

- Реализуйте класс `GameScore` для хранения счета и счетчика фигур. Класс должен поддерживать протокол `ObservableObject`.

**Класс предназначен для хранения двух целых чисел и обновления любых представлений SwiftUI, использующих объект этого типа как свойство, при изменении одного из этих двух целых чисел.**

## Первое представление

Поначалу представление ContentView устроено весьма тривиально: @StateObject (с GameScore), две кнопки Button и представление Text в VStack. Ничего сложного.

```

struct ContentView: View {
 @StateObject var gameScore = GameScore()

 var body: some View {
 VStack {
 1 Button("Add score") {
 gameScore.score += 1
 }
 .buttonStyle(.bordered)
 .padding()

 2 Button("Add pieces") {
 gameScore.pieces += 1
 }
 .buttonStyle(.bordered)
 .padding()

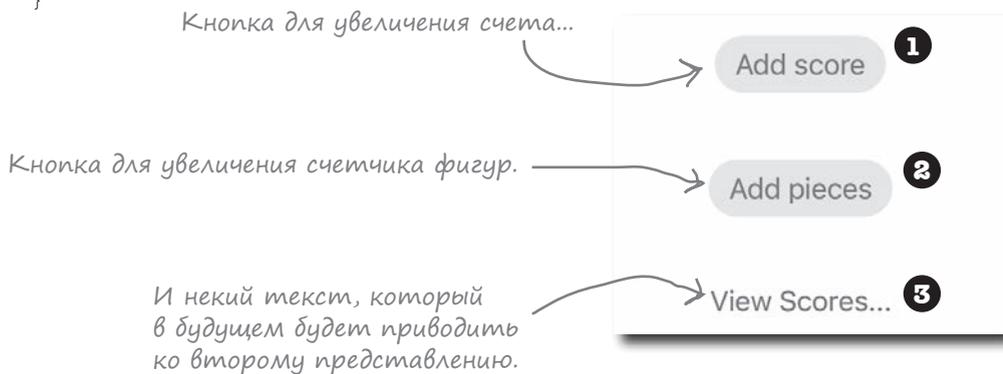
 3 Text("View Scores...")
 .padding()
 }
 }
}

```

Наш экземпляр GameScore создается с оберткой свойства @StateObject, так как мы собираемся использовать его в разных представлениях.

Мы увеличиваем свойства score и pieces, хранящиеся в экземпляре GameScore, при нажатии соответствующих кнопок.

В конечном итоге это станет ссылкой на второе представление, но пока ничего особенного — просто представление Text.



- Постройте первое представление ContentView, содержащее @StateObject с типом GameScore и все необходимые компоненты.

## Второе представление

Представление `ScoreView` тоже вполне тривиально. Мы используем `@EnvironmentObject` для обращения к `GameScore`, а остальное — это просто `VStack` с парой представлений `Text`, в которых выводится счет и счетчик фигур, и элемент `Spacer`.

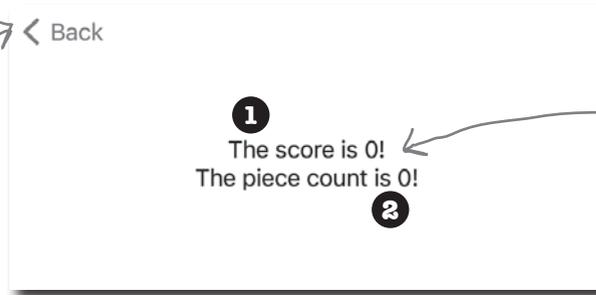
```
struct ScoreView: View {
 @EnvironmentObject var gameScore: GameScore
 var body: some View {
 VStack {
 Text("The score is \(gameScore.score)!")
 Text("The piece count is \(gameScore.pieces)!")
 Spacer()
 }
 }
}
```

*@EnvironmentObject позволяет обратиться к экземпляру GameScore с пометкой @StateObject, который был создан другим представлением.*

1  
2

*Свойства score и pieces экземпляра gameScore выводятся в представлениях Text.*

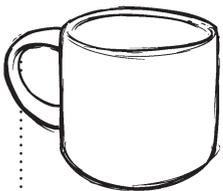
Возможно, вы заметили, что кнопка `Back` еще не реализована... сейчас мы этим займемся.



В текстовых полях выводится счет и счетчик фигур.



Постройте второе представление `ScoreView`, используя `@EnvironmentObject` для получения доступа к `@StateObject` с типом `GameScore` из первого представления, а также все необходимые компоненты.



### Расслабьтесь

Для одного этапа сделать придется довольно много, но в этом задании используются концепции, которые уже неоднократно использовались ранее. Просто на этот раз они будут объединяться новыми способами.

Все остальные системы совместного использования состояния, по сути, работают по тому же принципу, что и `@State`; просто они обладают большей гибкостью. А переключаться между представлениями не сложнее, чем упаковать элементы в `VStack` или `HStack`.

## И снова первое представление

- 1 Необходимо обновить представление `ContentView`, чтобы от него можно было переходить ко второму представлению (`ScoreView`). Для этого мы добавим элементы `NavigationView` и `NavLink`.

```

struct ContentView: View {
 @StateObject var gameScore = GameScore()

 var body: some View {
 NavigationView {
 VStack {
 Button("Add score") {
 gameScore.score += 1
 }
 .buttonStyle(.bordered)
 .padding()

 Button("Add pieces") {
 gameScore.pieces += 1
 }
 .buttonStyle(.bordered)
 .padding()

 NavLink(destination: ScoreView()) {
 Text("View Scores...")
 .padding()
 }
 }
 }
 }
}

```

*NavigationView — специальное контейнерное представление, которое позволяет управлять другими представлениями.*

*Для перехода от одного представления к другому внутри NavigationView используется элемент NavLink — особая разновидность Button, которая заставляет другое представление появиться поверх текущего представления и автоматически добавляет в новое представление кнопку Back для возврата.*

*Указываем, что целью должен быть элемент ScoreView, и оставляем то же представление Text, которое было ранее. Теперь Text будет поддерживать касания.*

`NavigationView` можно рассматривать как аналог других контейнерных представлений, таких как `VStack` и `HStack`. Вместо того чтобы группировать представления на экране, `NavigationView` строит коллекцию представлений, между которыми может перемещаться пользователь. На малых экранах (например, на iPhone) представления замещают друг друга. Но на больших экранах (например, iPad и Mac) первое представление может отображаться в левом столбце, а второе — в правом столбце.

## Первое представление (продолжение)

- 2 Чтобы данные, хранящиеся в свойстве `GameScore`, были доступны для второго представления, необходимо добавить модификатор `.environmentObject()` в `ContentView`. В результате данные становятся доступными для любых внутренних представлений (в данном случае для `ScoreView`).

```
struct ContentView: View {
 @StateObject var gameScore = GameScore()

 var body: some View {
 NavigationView {
 VStack {
 Button("Add score") {
 gameScore.score += 1
 }
 .buttonStyle(.bordered)
 .padding()

 Button("Add pieces") {
 gameScore.pieces += 1
 }
 .buttonStyle(.bordered)
 .padding()

 NavigationLink(destination: ScoreView()) {
 Text("View Scores...")
 .padding()
 }
 }
 }
 }
}

```

Добавляем модификатор `.environmentObject()` к `NavigationView`, так как данные `gameScore` должны быть доступны для всех внутренних представлений (это будет представление `ScoreView`, к которому мы переходим через `NavigationLink`).

Реализуйте возможность перехода между представлениями.

- 3 Если вы повторяли наши действия в Xcode, на этой стадии вы сможете запустить приложение, и оно должно работать! Вы сможете увеличивать как счет, так и количество фигур, а также переходить к другому представлению для просмотра данных.

## Ключевые моменты

- Обертка свойства `@State` может использоваться для совместного доступа из представлений к простым видам данных, таким как `Int`, `String` и `Bool`. Она пригодна для совместного использования самых примитивных видов состояния: чисел, переключателей «вкл/выкл» и т. д.
- `@StateObject` может использоваться для совместного использования более сложных типов. Любые данные, совместно используемые с `@StateObject`, должны поддерживать `ObservableObject`, а любые свойства внутри этого типа, которые должны инициировать обновление представления, должны иметь обертку свойства `@Published`.
- Протокол `ObservableObject` позволяет объекту оповещать другие объекты об изменениях значений любых его свойств, помеченных оберткой свойства `@Published`.
- Обертка свойства `@StateObject` необходима при первом создании экземпляра.
- `@ObservedObject` используется для обращения к данным уже созданного экземпляра, который поддерживает `ObservableObject` и был создан родителем текущего представления с использованием обертки свойства `@StateObject`.
- `@EnvironmentObject` может обращаться к данным `@StateObject`, поддерживающим `@ObservableObject`, из любой позиции иерархии представлений, если экземпляр `@StateObject` был присоединен к представлению, в котором он был создан, с модификатором `.environmentObject()`.
- `NavigationView` — особая разновидность контейнерного представления, которая позволяет переключаться между несколькими представлениями.
- `NavigationLink` — особая разновидность `Button`, которая позволяет инициировать перемещение между представлениями внутри `NavigationView`.
- iOS (и другие операционные системы, поддерживаемые SwiftUI) автоматически добавляет кнопки `Back` и другие визуальные признаки для прямого и обратного перемещения между представлениями.



Хмм.. Как мне использовать графику в моих приложениях SwiftUI, если изображения берутся не из коллекции SF Symbols?

### SwiftUI позволяет легко работать с локальными изображениями.

Впрочем, вы, наверное, уже догадались. В большинстве приложений в какой-то момент возникает необходимость в выводе локальных изображений. Локальные изображения не хранятся в интернете и не загружаются в процессе работы приложения. Они поставляются вместе с приложением.

В SwiftUI работать с ними действительно удобно, но вам понадобится небольшая помощь от Xcode.



## Добавление локальных изображений под увеличительным стеклом

Чтобы добавить локальное изображение в ваше приложение, необходимо добавить его в каталог ресурсов вашего приложения в Xcode. После этого вы сможете обращаться к нему по имени в коде Swift.

### 1 В проекте Xcode

Найдите категорию Assets в области Project на левой панели.

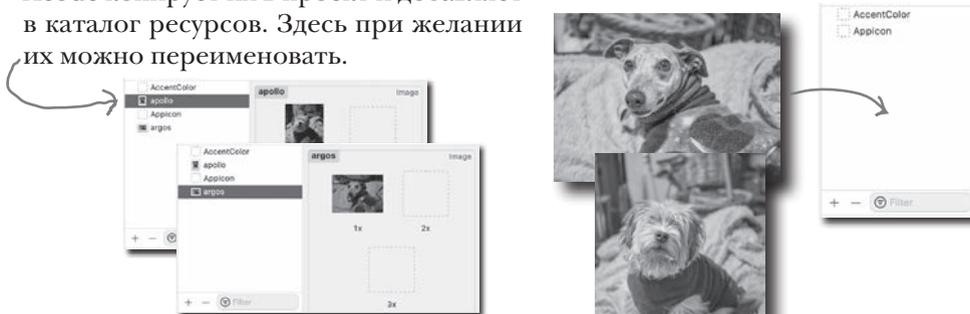


### 2 Перетащите изображения из Finder

Перетащите нужные изображения из Finder в каталог ресурсов в Xcode.

### 3 Убедитесь в том, что изображения находятся на месте

Xcode копирует их в проект и добавляет в каталог ресурсов. Здесь при желании их можно переименовать.



### 4 Используйте изображения в коде

В коде Swift вы ссылаетесь на изображение по имени в представлении Image.

```
struct ContentView: View {
 var body: some View {
 VStack {
 Image("argos")
 .resizable()
 .aspectRatio(contentMode: .fit)
 Spacer()
 Image("apollo")
 .resizable()
 .aspectRatio(contentMode: .fit)
 }
 }
}
```

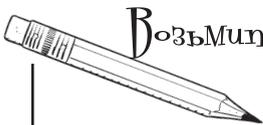
*Этот модификатор приказывает Swift автоматически изменять размер изображения, чтобы оно заполняло все доступное пространство.*

*Этот модификатор приказывает Swift сохранять пропорции изображения. Также можно использовать значение .fill, если нужно игнорировать пропорции изображения и растянуть его до границ, в которых оно находится.*



## Ключевые моменты

- Представление SwiftUI Image позволяет выводить графические изображения по имени.
- Изображения, которые можно выводить по имени, должны находиться в каталоге ресурсов проекта. Чтобы добавить изображения в каталог ресурсов, перетащите их из Finder в каталог Assets.
- По умолчанию изображения выводятся в полном размере, и они могут оказаться слишком большими для того представления, в котором они используются.
- Чтобы изображение более разумно заполняло всю доступную экранную область, добавьте к нему модификатор `.resizable()`.
- Чтобы сохранить у изображения исходные пропорции (избежать его искажений), примените модификатор `.aspectRatio(contentMode: .fit)`.
- Также можно позиционировать изображение в специальном фрейме, используя модификатор `.frame(width: 100, height: 100)` с соответствующими значениями.
- Модификаторы выравнивания `.top`, `.bottom`, `.right`, `.left` и `.center` смещают изображение в некотором направлении: `.frame(width: 100, height: 100, alignment: .top)`.
- Изображение можно обрезать по определенной форме, для чего следует добавить модификатор `.clipShape()` с указанием нужной фигуры, например `.clipShape(Circle())`.
- Как и с любым представлением SwiftUI, к изображению можно добавить рамку при помощи модификатора `.border` с соответствующим значением, например `.border(Color.red, width: 2)`.



### Возьмите в руку карандаш

Попробуйте воспроизвести это представление, используя свои навыки SwiftUI. Если потребуется, замените этих невероятно милых собак собственными изображениями.

→ *Обратитесь на с. 392.*



**Порядок** применения модификаторов в SwiftUI может повлиять на то, как будет выглядеть результат.



Может, вопрос и глупый, но как мне загрузить изображение из интернета?

**SwiftUI позволяет легко загрузить изображение из интернета.**

И вопрос вовсе не глупый. Приложений, загружающих графику по URL-адресу, так много, что это можно считать неотъемлемой частью SwiftUI!

Знакомьтесь: `AsyncImage` – еще один полезный структурный элемент, предоставляемый SwiftUI.

*AsyncImage работает примерно так же, как Image.*

```
AsyncImage(url: URL(string: "https://cataas.com/cat?type=square"))
 .frame(width: 300, height: 300)
```

Но может получать URL изображения.

Я знаю, что такое URL... но что такое URL в Swift? Тип?

**URL — тип, предоставляемый Swift.**

URL можно создать на базе String. Многие компоненты Swift могут использовать URL для загрузки разных данных, и URL может ссылаться на удаленный или локальный файл.

Такой же тип, как любой другой.

Получает строковый параметр.

```
let myLink = URL(string: "https://secretlab.games/")
```

Но также может создаваться относительно другого URL.

```
let mySecondLink = URL(string: "headfirstswift/", relativeTo: myLink)
```

И от него можно получить (опциональную) строку с абсолютным URL-адресом.

```
let firstLinkString = myLink?.absoluteString
let secondLinkString = mySecondLink?.absoluteString
```

The first link is: `https://secretlab.games/`

```
print("The first link is: \(firstLinkString!)")
print("The second link is: \(secondLinkString!)")
```

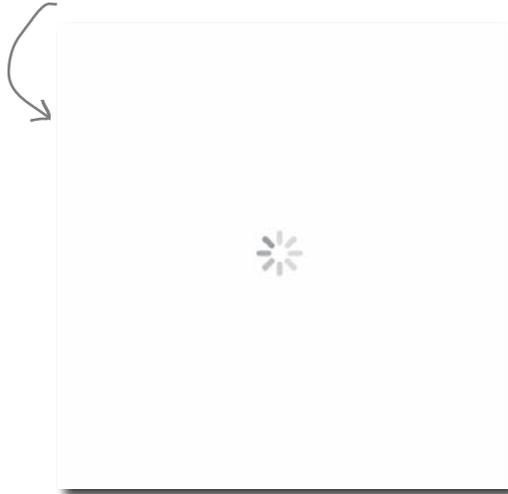
The second link is: `https://secretlab.games/headfirstswift/`

## AsyncImage с наворотами

Иногда при загрузке графики из интернета требуется проявить чуть больше фантазии, вывести на экран симпатичный индикатор прогресса, который отражает ход загрузки, и в конечном итоге заменить его этим изображением.

### Примерно так:

Пока изображение загружается, на экране отображается круговой индикатор...



После завершения загрузки заменяется изображением.



### Это делается так:

```
struct ContentView: View {

 var body: some View {
 VStack {
 AsyncImage(url: URL(string: "https://cataas.com/cat?type=square")) { image in
 image.resizable() ←
 } placeholder: {
 ProgressView() ←
 }
 .frame(width: 300, height: 300)
 }
 }
}
```

Когда AsyncImage завершит загрузку изображения, оно помещается в image. Здесь image — элемент SwiftUI Image, с которым вы уже умеете работать.

Здесь можно просто применить модификатор .resizable(), потому что в этой точке это просто обычное изображение.

ProgressView — удобное представление SwiftUI, которое отображает круговой индикатор с бесконечной анимацией. Чтобы он исчез с экрана, вы должны явно убрать его. Без этого он так и будет крутиться бесконечно.

При желании ProgressView() можно заменить чем-то другим, например элементом Text("Image loading!"). Все, что находится в замыкании, будет отображаться на экране, пока AsyncImage не завершит загрузку изображения.



## Упражнение

Взгляните на приведенный ниже код. Это еще более навороченная версия того, что вы уже научились строить. В коде остались пропуски, и мы хотим, чтобы вы их заполнили.

```
struct ContentView: View {
```

```
 var imageURL = URL(string: "https://cataas.com//cat?type=square")
```

```
 var body: some View {
```

```
 VStack {
```

```
 AsyncImage(url: imageURL) { phase in
```

Выбираем по значению phase...

```
 switch phase {
```

Это означает, что изображение все еще загружается.

В этом случае получаем изображение.

А это означает, что загрузка завершилась неудачей.

И если происходит что-то непредвиденное, получаем EmptyView.

```
 case .empty:

```

```
 case .success(let image):


```

```
 case .failure:

```

```
 default:

```

```
 EmptyView()
 }
 }
 }
}
```

AsyncImage может работать с замыканием, получающим экземпляр AsyncImagePhase, который представляет состояние загрузки изображения.

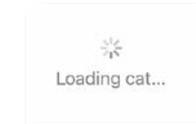
Здесь необходимо поместить элемент ProgressView, который должен отображаться во время загрузки. В ProgressView можно поместить текст: ProgressView("Like this...")

Этот синтаксис связывает значение (ассоциированное с перечислением, из которого берется .success) с константой, которой присвоено имя image. Это позволяет нам получить image.

Здесь используется image: изображение отображается с модификатором .resizable() и подгоняется по размеру представления во фрейме 300x300.

Здесь выводится изображение через SF Symbols. Мы рекомендуем использовать shippingbox, но никто не запрещает вам проявить фантазию.

EmptyView — представление SwiftUI, которое вообще ничего не содержит. Оно полезно в ситуациях, когда вам понадобится действительно пустое представление.



.empty

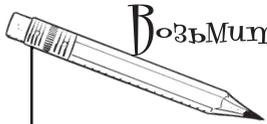


.success



.failure

ответ на с. 393.



# Возьмите в руку карандаш

## Решение

```
struct ContentView: View {
 var body: some View {
 VStack {
 Image("argos")
 .resizable()
 .frame(width: 300, height: 300)
 .aspectRatio(contentMode: .fill)
 .border(Color.green, width: 3.0)
 Spacer()
 Image("apollo")
 .resizable()
 .aspectRatio(contentMode: .fit)
 .clipShape(Capsule())
 }
 }
}
```

С. 388



Говорят, на Swift можно строить веб-сайты. Как это делать? Я уже вполне уверенно освоилась со Swift, но не знаю, как создать веб-сайт...



### Пришло время познакомиться с Varog (пусть и ненадолго).

Varog – выразительный, управляемый событиями, протоколно-ориентированный фреймворк веб-разработки, написанный на Swift и для Swift. Он предоставляет все необходимое для построения современных веб-служб и веб-сайтов с использованием Swift.

Varog не является проектом Apple; это часть энергичного сообщества Swift-разработки с открытым кодом.



## Упражнение Решение

С. 391

```

struct ContentView: View {

 var imageURL = URL(string: "https://cataas.com//cat?type=square")

 var body: some View {
 VStack {
 AsyncImage(url: imageURL) { phase in
 switch phase {
 case .empty:
 ProgressView("Loading cat...")
 case .success(let image):
 image.resizable()
 .aspectRatio(contentMode: .fit)
 .frame(maxWidth: 300, maxHeight: 300)
 case .failure:
 Image(systemName: "shippingbox")
 default:
 EmptyView()
 }
 }
 }
 }
}

```



## Vapor: веб-фреймворк для Swift



Предполагается, что фреймворк Vapor уже установлен на вашей машине. Чтобы получить проект Xcode, с которым вы можете работать, выполните процесс из трех шагов.

Установка Vapor здесь не рассматривается, потому что это довольно хлопотное дело, но за первыми инструкциями обращайтесь на сайт <https://vapor.codes> в вашем веб-браузере.

### 1 Новый проект Vapor можно создать следующей командой:

```
vapor new mysite -n
```

Когда это будет сделано, вы увидите нечто такое:



### 2 Перейдите в каталог проекта Vapor с помощью команды:

```
cd mysite
```



Будьте осторожны!

На iPad это сделать не получится.

Хотя все инструкции, приводившиеся ранее, в равной степени применимы к устройствам с iPadOS или macOS, Vapor может использоваться только для разработки на устройствах macOS.

Так что извините: для работы с Vapor вам придется включить Mac!

### 3 Затем откройте новый проект Vapor в Xcode с помощью команды:

```
open Package.swift
```



Fetching swift-nio (22%)...

Когда проект откроется в Xcode, менеджер пакетов Swift загрузит несколько пакетов.

Вы сможете наблюдать за тем, как пакеты загружаются и добавляются в проект, на боковой панели.

#### Swift Package Dependencies

- async-http-client
- async-kit
- console-kit
- multipart-kit (fetching...)
- routing-kit
- swift-backtrace
- swift-crypto
- swift-log
- swift-metrics
- swift-nio (fetching...)
- swift-nio-extras
- swift-nio-http2
- swift-nio-ssl
- vapor
- websocket-kit



#### Тест-драйв

Щелкните на кнопке запуска в Xcode и дождитесь построения проекта. Когда проект будет построен, загляните в область журнала Xcode и найдите в ней URL. Попробуйте открыть этот URL и посмотрите, что вы там обнаружите. Затем загляните в файл `routes.swift`. Удается ли вам разобраться в том, что там происходит?

[ NOTICE ] Server starting on  
http://127.0.0.1:8080

- mysite
  - Package.swift
  - Sources
    - App
      - Controllers
      - configure.swift
      - routes.swift
    - Run

Я понимаю, как маршруты соответствуют тому, что я могу увидеть. А если мне понадобится сделать что-то более сложное, чем просто вывести приветствие для пользователя?

### Маршрут вас куда-то приводит.

В `Varog` вы можете создавать маршруты, чтобы определять, что должно происходить в разных обстоятельствах. Здесь мы не будем рассматривать полноценный пример, но если вы откроете файл `routes.swift` в своем проекте Xcode и добавите в него новый маршрут:

```
app.get("swift", "awesome") { req ->
String in
 return "Swift is awesome!"
}
```

вы тем самым создадите новый маршрут.

Маршрут обрабатывает запрос `GET`. Он реализован в виде замыкания, а каждый параметр является параметром пути для URL-адреса, для которого создается маршрут (в данном случае `/swift/awesome`).

Теперь при посещении URL-адреса `/swift/awesome` на вашей локальной машине будет возвращаться текст, заданный для этого маршрута.

Полезно, не так ли?



## Отправка данных средствами Vapor

Вы также можете получать данные по URL-адресам при помощи запросов GET и использовать их для вывода специального приветствия в зависимости от имени, включенного в URL.

### 1 Создайте новое замыкание с динамическим параметром:

```
app.get("hello", ":name") { req -> String in
```

```
}
```

Формально это все еще строка. Но Vapor знает, что сначала нужно проверить символ `:` в начале, это означает, что этот параметр интерпретируется как динамический. Это соглашение Vapor, а не соглашение Swift.

### 2 Добавьте guard let для распаковки опционального параметра:

```
app.get("hello", ":name") { req -> String in
```

```
 guard let name = req.parameters.get("name") else {
```

```
 throw Abort(.internalServerError)
```

```
 }
```

```
}
```

Здесь мы используем guard let для распаковки опционального типа и выдаем ошибку в случае неудачи. Ошибки в книге почти не рассматривались, но этот код выдает ошибку из перечисления, предоставляемого Vapor.

### 3 Верните то, что требуется

```
app.get("hello", ":name") { req -> String in
```

```
 guard let name = req.parameters.get("name") else {
```

```
 throw Abort(.internalServerError)
```

```
 }
```

```
 return "Hello, \(name)!"
```

```
}
```

И наконец, мы возвращаем строку, включающую переменную name!

#### 4 Откройте новый URL-адрес



На странице выводится имя, переданное во втором компоненте URL!

### Ключевые моменты

- Vapor позволяет создавать современные веб-приложения и веб-сайты с использованием Swift.
- После установки Vapor можно использовать Xcode для определения маршрутов, действий, выполняемых разными страницами, и реализации других возможностей вашего веб-приложения.
- Vapor может возвращать текст или формат JSON и работать с пакетами из менеджера пакетов Swift для выполнения всевозможных трюков из области веб-программирования.
- Существуют пакеты, позволяющие использовать Vapor с чем угодно от MySQL до JQuery, Redis, Apple Push Notification Service и многого другого.



### Мозговой Штурм

Добавьте в веб-приложение маршруты для суммирования и умножения чисел на основании данных, переданных в URL (например, маршрут `/add/2/10` складывает 2 и 10 и выводит 12). Придумайте, где лучше всего разместить код сложения и как наиболее логично разделить код на осмысленные части.

Мы знаем, что Swift разрабатывается с открытым кодом. Но что именно это означает? Могу ли я работать над созданием самого языка Swift? Мне кажется, это стоящее дело...



## Процесс Swift Evolution — лучшая возможность поучаствовать в работе над Swift.

Процесс Swift Evolution разработан для привлечения новых людей, идей и опыта к сообществу Swift.

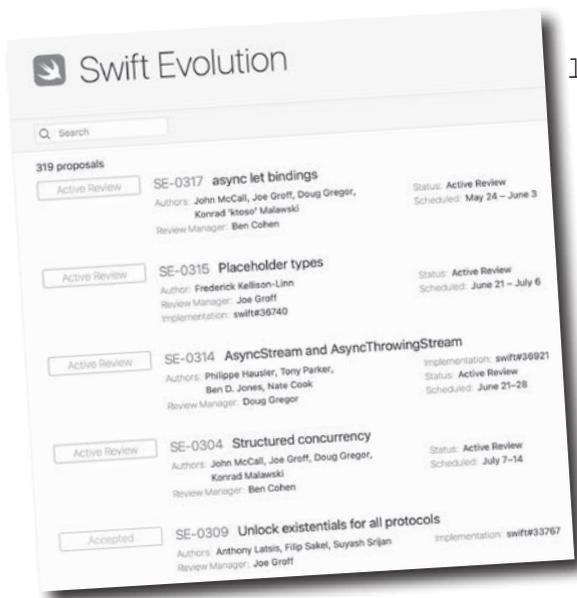
Swift — большой и сложный проект, и, несмотря на то что он распространяется с открытым кодом, автоматически принимать каждую предложенную идею просто невозможно.

Именно эту проблему решает Swift Evolution. На этом форуме все желающие приглашаются к внесению, обсуждению и рецензированию новых идей для Swift.

Формат публикации новых идей разрабатывался так, чтобы он был логически целостным и понятным.

Проект Swift отслеживает состояние идей в этом процессе, поэтому вы всегда можете наблюдать за их прогрессом.

Начните с посещения веб-сайта Swift.



```
let swiftWebsite =
 URL(string: "https://swift.org")
```

**Процесс Swift Evolution  
позволяет всем желающим  
оказывать прямое влияние  
на ход проекта.**

Код из этой книги, а также некоторые дополнительные упражнения можно загрузить по адресу [secretlab.com.au/books/head-first-swift](http://secretlab.com.au/books/head-first-swift)

И что теперь? Мы закончили со Swift? Мы все знаем?



**Фрэнк:** И все? Все закончилось?

**Джуди:** Закончилась эта книга, я полагаю.

**Джо:** И что теперь?

**Сэм:** Наверное, мы знаем все о Swift?

**Фрэнк:** Я так не думаю. Полагаю, мы узнали все необходимые структурные элементы и освоили все навыки, необходимые для дальнейшего изучения Swift.

**Джо:** Значит, это еще не ВСЕ, что нужно знать о Swift? Книга и так была длинная!

**Джуди:** Да, я думаю, что книга познакомила нас с различными частями Swift, а дальше нам придется действовать самостоятельно.

**Сэм:** Справимся! Вперед!