Чейрд ин’т Вейн

Swift. Подробно

Tjeerd in ‘t Veen

Swift in Depth

MANNING PUBLICATIONS
New York, 2019

Чейрд ин’т Вейн

Swift. Подробно

Москва, 2020

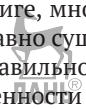
УДК 004.432
ББК 32.972.1
В26

В26 Чейрд ин’т Вейн
Swift. Подробно / Пер. с англ. Д. А. Беликова. – М.: ДМК Пресс, 2020. – 412 с.
ISBN 978-5-97060-780-0

Данная книга знакомит вас с навыками, необходимыми для создания профессионального программного обеспечения для платформ Apple, таких как iOS и MacOS. Вы освоите такие мощные методы, как обобщение, эффективная обработка ошибок, протокольно-ориентированное программирование и современные шаблоны Swift.

Издание рассчитано на программистов продвинутого и среднего уровней.

УДК 004.432
ББК 32.972.1



ISBN 978-5-97060-780-0 (рус.)
ISBN 978-1-61729-518-8 (анг.)

Содержание

ЛАНЬ®

Предисловие

Я начинал как разработчик iOS в 2011 году. Мне нравилось создавать приложения для iPhone, и я до сих пор занимаюсь этим. Помимо разработки мобильных приложений, я также занимался веб-разработкой, изучая Ruby. Мне нравился этот короткий мощный язык, и я хотел использовать компилируемый язык, такой как Objective-C, но с элегантностью и выразительностью Ruby.

Затем компания Apple представила Swift, и похоже, они услышали меня. Для меня Swift был новым подходом к программированию, сочетая элегантность динамического языка со скоростью и безопасностью языка статического. Мне никогда не нравился синтаксис Objective-C. Раньше я говорил что-то вроде: «Да, Objective-C многословен, но он делает свою работу». Однако при работе со Swift я снова нахожу чтение и написание кода очень приятным занятием, как в случае с Ruby. Наконец-то я мог использовать статический язык и продолжать работать, при этом любя язык, с которым я работаю. Для меня это была хорошая комбинация.

Однако это не была любовь с первого взгляда. До того, как я действительно стал наслаждаться Swift, я много с ним боролся. Swift выглядит очень дружелюбно, но, боже, иногда было тяжело. На этапе компиляции все должно быть безопасно, и я больше не мог смешивать и сопоставлять типы в массивах. Между тем Swift был только ранней версией и постоянно менялся; было трудно идти в ногу с ним. «Что такое генераторы? О, их теперь называют итераторами? А зачем использовать оператор guard? Разве нельзя вместо этого использовать оператор if? Пфф, опционалы переоценены; можно использовать простую проверку на nil?» и т. д. Я даже не и думал о работе с обобщениями.

Тем не менее я выстоял и начал принимать эти концепции Swift. Я понял, что это были более старые концепции из других языков программирования, но в новом обличье, что действительно помогло мне лучше программировать и качественнее выполнять свою работу. Со временем я начал любить язык Swift и его милый синтаксис.

Начиная с Swift 2 я имел возможность поработать в большой компании, где мы производили код Swift в больших масштабах, начиная с команды, состоящей из примерно 20 разработчиков и заканчивая более чем 40. После работы со Swift при таком количестве разработчиков и после участия в сотнях запросов на принятие изменений я заметил, что у других разработчиков была такая же борьба, как и у меня. Либо мои коллеги писали просто отличный код, но не понимали, что от них скрыта более элегантная или надежная альтернатива, ждущая своего часа. Несмотря на то что наш код был верным, иногда он мог бы быть несколько чище, лаконичнее или чуть безопаснее. Я также заметил, что мы все держались подальше от мощных методов, таких как обобщения или метод `flatMap`, потому что их было трудно понять. Или же нам нравилась идея обобщений, но мы сами не знали, зачем и когда применять ее.

После этих реализаций я начал вести записи. Поначалу эти каракули были для меня заметками о том, как правильно извлекать опционалы, как работают ленивые свойства, как обращаться с обобщениями и т. д. Затем эти заметки созрели,

и, прежде чем я это понял, у меня было достаточно содержания для некоторых глав. Пришло время превратить эти заметки во что-то более сложное: книгу по программированию, которая может помочь другим сократить их путешествие по Swift. Имея на руках несколько сырых глав, я задавался вопросом, стоит ли мне выложить электронную книгу онлайн. Тем не менее при наличии «впечатляющих» 200 человек, подписавшихся на меня в Twitter, и не имея популярного блога, я решил, что не найду нужную аудиторию. Более того, я подумал, что мне нужно узнать много неизвестного о написании книги.

Я решил обратиться к издателю, чтобы он помог мне превратить эти грубые главы в большую книгу. Я обратился в издательство Manning, и с тех пор мы вместе работаем над книгой. Полагаю, что эти «маленькие заметки» превратились в нечто особенное. С помощью друзей Manning и Swift больше года я проводил большую часть своего свободного времени, сочиняя, полируя и пытаясь сделать сложные концепции Swift более простыми для понимания.

Я надеюсь, что когда вы будете читать эту книгу, она поможет вам стать мастером Swift. Надеюсь, что эта книга сделает ваше путешествие по Swift легким и увлекательным.

Чейрд ин't Вейн

Благодарности

Спасибо издательству Manning за то, что помогло мне опубликовать мою первую книгу.

Я хочу выразить особую благодарность Майку Стивенсу за то, что он воспользовался шансом, взяв меня на борт. Спасибо Хелен Стергиус за то, что работала со мной на протяжении всего этого процесса. Спасибо Алену Куньо за великолепные технические обзоры моих глав; нелегко постоянно указывать на ошибки и возможные улучшения; тем не менее я очень ценю это. И я также хочу поблагодарить Александра Павлицкого за его креативные мультипликационные иллюстрации, использованные на протяжении всей книги.

Мне очень помогли остальные члены команды Manning: Александар Драгосавлевич, Кэндис Гилхулли, Ана Ромак, Шерил Вейсман, Дейрдре Хиам, Дотти Марсико, Николь Борода, Мэри Пирджи, Кэрол Шилдс, Даррен Мейсс, Мелоди Долаб и Мария Тудор (Aleksandar Dragosavljević, Candace Gillhoolley, Ana Romac, Cheryl Weisman, Deirdre Hiam, Dottie Marsico, Nichole Beard, Mary Piergies, Carol Shields, Darren Meiss, Melody Dolab, Marija Tudor). Я хочу выразить особую благодарность друзьям, коллегам и тем, кто были моими подопытными свинками и рецензировали (частично) эту книгу, в частности это: Барт ден Холландер, Димитар Гюров, Дарио де Роза, Рейн Спейккерман, Янина Кутын, Сидни де Конинг, Торбен Шульц и Эдвин Чун Винг Квок (Bart den Hollander, Dimitar Gyurov, Dario de Rosa, Rein Spijkerman, Janina Kutyn, Sidney de Koning, Torben Schulz, and Edwin Chun Wing Kwok). Также я бы хотел поблагодарить других рецензентов: Алессандро Кампейса, Али Накви, Акселя Реста, Дэвида Джейкобса, Густаво Гомеса, Хельмута Райтерера, Джейсона Пайка, Джона Монтгомери, Кента Р. Шпилнера, Ларса Петерсена,

Марсело Пиреса, Марко Джузеппе Салафию, Мартин Филипп, Моника Гимарайнш, Патрик Риган, Тайлер Слэйтер и Вейерт де Бур (Alessandro Campeis, Ali Naqvi, Axel Roest, David Jacobs, Gustavo Gomes, Helmut Reiterer, Jason Pike, John Montgomery, Kent R. Spillner, Lars Petersen, Marcelo Pires, Marco Giuseppe Salafia, Martin Philp, Monica Guimaraes, Patrick Regan, Tyler Slater, and Weyert de Boer).

Я знал, что написание книги – дело непростое, но это было гораздо сложнее, чем я себе представлял. У нас с моей невестой Дженикой только что родилась дочь, и было довольно сложно создавать новую семью, проводить бессонные ночи, сохранить работу на полную ставку и писать книгу по программированию на втором языке. Я бы не смог этого сделать, если бы мне не нравилось писать эту книгу при поддержке своей невесты. Спасибо, Дженика, за то, что ты так терпелива ко мне.

Об этой книге

Swift – это молодой язык. На момент написания данных строк вышла четвертая версия, и она все еще не является ABI-стабильной, а это означает, что после выхода Swift 5 будут существенные изменения. Так почему же эта книга может рассказать вам, как написать свой код?

Ваш скептицизм был бы оправдан, но, пожалуйста, не торопитесь. Несмотря на то что Swift – относительно новый язык, я думаю, было бы справедливо сказать, что некоторые решения работают лучше, чем другие. Это еще важнее понимать, если вы используете Swift для реальных производственных приложений.

Swift заимствует множество важных концепций из других языков программирования, таких как Haskell, Ruby, Rust, Python, C # и др. Следовательно, было бы разумно следить за этими концепциями.

Сочетая парадигмы программирования с реальным опытом, эта книга делится очень забавными и полезными рекомендациями, которые вы можете незамедлительно применить в своей работе.

Программируя более десяти лет на разных языках и в разных командах, я хотел бы поделиться советами, хитростями и рекомендациями, которые очень помогли моей карьере в Swift, чего и вам желаю.

Почему эта книга?

Честно говоря, многие программы в этом мире работают на «некрасивом» коде, и это совершенно нормально. Если ваш продукт делает то, что нужно, это – нравится вам это или нет – вполне подойдет для бизнеса.

Будучи разработчиком, вы должны убедиться, что ваш продукт работает, и работает хорошо. Но ваши пользователи не будут заглядывать под капот и указывать на страшных операторов if. Перфекционизм вреден для разработки программного обеспечения и является причиной большого количества незавершенных проектов.

Тем не менее существует большой разрыв между «Он делает то, что нужно» и проектом, в котором были приняты отличные решения, которые окупаются в долгосрочной перспективе.

Работая над многочисленными проектами, я высоко ценю написание кода, который ваши коллеги и ваши будущие сотрудники будут *четко* понимать, потому что элегантный код означает меньше шансов на ошибки, повышенное удобство в обслуживании, лучшее понимание для разработчиков, которые наследуют код, море счастья для программистов и много других преимуществ.

Другой аспект, который я ценю, – это надежность кода, то есть то, насколько стойкими к рефакторингу являются некоторые элементы. Сломается, если чихнуть на него? Или можно ли изменить код без хлопот?

В этой книге я поделюсь своими советами, хитростями и рекомендациями, которые отлично подошли мне и компаниям, в которых я работал. Кроме того, она заполняет значительные пробелы в знаниях, которые могут возникнуть при работе со Swift.

Хотя это книга о Swift, многие принципы, о которых здесь говорится, не ориентированы на Swift и также распространяются на другие языки программирования, потому что Swift заимствует множество идей и парадигм из других языков. После того как вы закончите читать книгу, вам, возможно, будет легко применять эти концепции в других языках. Например, вы узнаете много нового об опционалах или о том, как использовать метод `reduce` при работе с массивами. Позже вы можете решить изучать Kotlin, где можно сразу же применять опционалы и вышеуказанный метод, где он носит название `fold`. Вы также можете обнаружить, что Rust – и его аналогичные реализации обобщений – проще освоить.

Из-за мультипарадигмальной природы Swift эта книга без предпочтения переключается между парадигмами объектно-ориентированного, функционального и протокольно-ориентированного программирования – хотя, по общему признанию, я все же предпочитаю другие методы, нежели создание подклассов. Переключение между этими парадигмами предлагает множество инструментов и решений проблемы, а также понимание того, почему определенное решение работает хорошо или нет. Если вы застряли в колее или открыты для множества новых идей программирования, эта книга ставит перед вами задачу решать проблемы различными способами.

Подходит ли вам эта книга?

Предполагается, что вы создали одно или несколько приложений на Swift. Вы работаете в команде? Еще лучше. Эта книга покажет вам, как написать хороший, понятный код, который будет оценен в командах, и поможет вам улучшить запросы на принятие изменений от других разработчиков. Ваш код будет более надежным и потребует меньше обслуживания от вас и вашей команды.

Эта книга заполняет пробелы в знаниях как для новичков, так и для опытных разработчиков Swift. Возможно, вы освоили протоколы, но все еще боретесь с использованием метода `flatMap` при работе с типами или асинхронной обработкой ошибок. Или, может быть, вы создаете красивые приложения, но держитесь подальше от обобщений, потому что их трудно интерпретировать. Или, может быть, вы знаете, когда использовать структуру вместо класса, но не знаете, что перечисления иногда являются лучшей альтернативой. В любом случае, эта книга по-

может вам с этими темами. К концу работа с обобщениями должна быть такой же естественной, как и в случае с циклами. Вы будете уверены в том, что будете вызывать метод `flatMap` для опционалов, будете знать, как работать с ассоциированными типами, и с радостью будете использовать метод `reduce` в своем повседневном труде при работе с итераторами.

Если вы планируете в будущем пройти собеседование на должность программиста, чтобы получить новое место, вам понравится. Вы сможете ответить на множество важных вопросов, касающихся компромиссов и решений относительно разработки на языке Swift. Эта книга может даже помочь вам написать элегантный код в заданиях.

Если вам просто нужно приложение из магазина приложений, продолжайте делать то, что делаете; нет необходимости читать эту книгу! Но если вы хотите написать более надежный, понятный код, который увеличивает ваши шансы получить место, преуспеть на работе или дать качественные комментарии по запросам на принятие изменений, вы попали в нужное место.

Чем эта книга не является

Эта книга ориентирована на Swift. В основном в ней используются примеры без фреймворков, потому что речь идет не об обучении Cocoa, iOS, Kitura или другим платформам и фреймворкам.

В этой книге я часто пользуюсь Apple Foundation, чего трудно избежать, если вам нужны примеры из реальной жизни. Если вы работаете в Linux, то можете использовать swift.org, чтобы получить аналогичные результаты.

Особый акцент на практических сценариях

Эта книга очень практична и демонстрирует советы и приемы, которые вы можете сразу же применять при повседневном программировании.

Не волнуйтесь: это не теоретическая книга. Вы узнаете много теории, но только благодаря использованию реальных проблем, с которыми рано или поздно сталкивается любой разработчик Swift. Тем не менее она не дотягивает до академического уровня, где обсуждается представление LLVM или машинный код.

Кроме того, я позаботился о том, чтобы избежать своей личной любимой мозоли: я не делю подкласс «Animal» с «Dog» и не добавляю протокол «Flyable» в «Bird». Я также не добавляю «Foo» в «Bar». Вы будете иметь дело с реальными сценариями, такими как общение с API, загрузка локальных данных, рефакторинг и создание функций, и увидите полезные фрагменты кода, которые можно реализовать в своих проектах.

Дорожная карта

В следующих разделах представлен обзор книги, разделенный на главы. Книга довольно модульная, и вы можете начать с любой главы, которая вас интересует.

Некоторые главы я считаю важными. Глава 4 «Делаем опционалы второй натурой» является ключевой, потому что опционы очень распространены в Swift, и мы будем возвращаться к ним снова и снова в других главах.

Чтобы понять абстрактную сторону Swift, я настоятельно рекомендую прочитать главу 7 «Обобщения», главу 8 «Становимся профессионалами в протокольно-ориентированном программировании» и главу 12 «Расширения протоколов». Эти главы закладывают прочную основу для ключевых навыков Swift. Обязательно прочтите их!

В качестве бонуса, если вы заинтересованы в изучении методов функционального программирования, обратите внимание на главу 2 «Моделирование данных с помощью перечислений», главу 10 «map, flatMap и compactMap» и главу 11 «Асинхронная обработка ошибок с помощью типа Result».

Глава 1: Введение

Эта глава – своего рода разминка. В ней рассказывается о текущем состоянии Swift, о его положительных сторонах и недостатках и о том, чем мы будем заниматься в ходе прочтения данной книги. Она не очень техническая, однако подготовит вас к тому, о чем пойдет речь далее.

Глава 2: Моделирование данных с помощью перечислений

Эта глава отлично подойдет, если вы хотите напрячь свой мозг, по-другому взглянуть на моделирование данных и увидеть, насколько далеко могут зайти перечисления, чтобы помочь вам.

Вы узнаете, как моделировать данные с помощью структур и перечислений и преобразовывать структуры в перечисления, и наоборот.

Вам будет предложено отойти от обычного подхода к классам, подклассам и структурам и посмотреть, как вместо этого моделировать данные с помощью перечислений и зачем это нужно.

Вы также познакомитесь с другими интересными способами использования перечислений и узнаете, как использовать их для написания более безопасного кода.

К концу этой главы вы можете поймать себя на мысли, что пишете гораздо больше перечислений.

Глава 3: Написание более чистых свойств

Swift обладает богатой системой свойств со множеством опций на выбор. Вы научитесь выбирать правильный тип свойств для правильных типов ситуаций. Вы также создадите чистые вычисляемые свойства и хранимые свойства с поведением.

Потом вы узнаете, когда использовать ленивые свойства, которые могут привести к незначительным ошибкам, если с ними обращаться неаккуратно.

Глава 4: Делаем опционалы второй натурой

Эта глава не оставит камня на камне касательно опциональных типов.

Опциональные типы настолько распространены, что в этой главе они рассматриваются очень подробно. Данная глава изобилует передовыми методами, советами и хитростями, которые улучшат ваш обычный код. Она будет полезна как новичкам, так и опытным разработчикам Swift.

Глава охватывает множество случаев использования опциональных типов, например при обработке опциональных логических типов, опциональных строк и перечислений, неявно извлекаемых опционалов и принудительном извлечении.

Глава 5: Разбираемся с инициализаторами

Жизнь в мире программирования начинается с инициализаторов. Спрятаться от них в Swift невозможно, и, конечно, вы уже работаете с ними. Тем не менее в Swift есть множество странных правил и ловушек, касающихся структур и классов и того, как инициализируются их свойства. Эта глава раскрывает эти странные правила, чтобы помочь вам избежать поединка с компилятором.

Это не просто теория; вы увидите, как можно написать меньше кода для инициализации, чтобы поддерживать чистоту своей кодовой базы, и получите понимание того, как создавать подклассы и как там применяются правила инициализаторов.

Глава 6: Непринужденная обработка ошибок

В этой книге есть две главы, посвященные обработке ошибок, где идет речь о двух разных идиомах: одна для синхронной обработки ошибок, а другая для асинхронной.

Данная глава посвящена синхронной обработке ошибок. Вы откроете для себя передовые методы, связанные с генерацией ошибок, их обработкой и поддержанием рабочего состояния ваших программ. В ней также рассказывается о распространении, добавлении технической информации и информации о пользователях, о преобразовании в NSError.

Помимо этого, вы узнаете, как сделать свои API-интерфейсы более приятными, заставляя их генерировать меньше ошибок, соблюдая при этом целостность приложения.

Глава 7: Обобщения

Обобщения – своего рода обряд посвящения для разработчиков Swift. Поначалу их трудно понять или работать с ними. Однако как только вы освоитесь, у вас появится искушение часто их использовать. В этой главе вы узнаете, когда и как применять их, создавая обобщенные функции и типы.

Вы увидите, как с помощью обобщений можно сделать код полиморфным, чтобы иметь возможность писать многократно используемые компоненты и в то же время сокращать кодовую базу.

Обобщения становятся еще более интересными, когда вы ограничиваете их протоколами для специализированной функциональности. Вы познакомитесь

с основными протоколами `Equatable`, `Comparable` и `Hashable` и узнаете, как смешивать и сопоставлять с ними обобщения.

После прочтения этой главы обобщения не будут чем-то пугающим, я обещаю.

Глава 8: Становимся профессионалами в протокольно-ориентированном программировании

Протоколы – напоминающие классы типов в Haskell или типажи в Rust – святой Грааль Swift. Поскольку Swift можно считать протокольно-ориентированным языком, в этой главе мы рассмотрим, как с пользой применять протоколы.

В ней рассказывается об обобщениях и показано, как они справляются с использованием протоколов в качестве типов. Вы сможете четко делать выбор (или переключаться) между ними. Протоколы с ассоциированными типами можно считать расширенными протоколами. В этой главе вы поймете, почему и как они работают, чтобы вам не пришлось воздерживаться от их использования. Здесь будет смоделирована часть программы с протоколами и будут приведены недостатки, которые в конечном итоге решаются с помощью ассоциированных типов.

Затем вы узнаете, как передавать протоколы с ассоциированными типами в функции и типы, чтобы иметь возможность создавать чрезвычайно гибкий, но абстрактный код.

В этой главе много внимания уделено тому, как использовать протоколы на этапе компиляции (статическая отправка) и во время выполнения (динамическая отправка), а также связанным с ними компромиссам. Эта глава призвана обеспечить надежную основу для протоколов, чтобы вы могли использовать более сложные шаблоны в последующих главах.

Глава 9: Итераторы, последовательности и коллекции

Нередко в Swift создается структура данных, которая использует не только основные типы, такие как наборы, массивы и словари. Возможно, вам потребуется создать специальное хранилище для кеширования или систему разбивки на страницы при загрузке канала Twitter.

Структуры данных часто приводятся в действие протоколами `Collection` и `Sequence`. Вы увидите, что протокол `Sequence`, в свою очередь, использует протокол `IteratorProtocol`. Применяя сочетание этих протоколов, можно расширять и реализовывать основные функции в своих типах данных.

Вначале вы увидите, как работает итерация с протоколами `IteratorProtocol` и `Sequence`. Вы познакомитесь с полезными шаблонами итераторов, такими как `reduce()`, `reduce(into :)` и `zip`, а также увидите, как работают ленивые последовательности (`lazy sequences`).

Вы создадите структуру данных под названием сумка, также известную как мультимножество, используя протокол `Sequence`.

Затем вы познакомитесь с протоколом `Collection` и описанием всех протоколов коллекций, предлагаемых Swift.

В завершение вы создадите еще одну структуру данных и узнаете, как привести ее в соответствие с протоколом `Collection`. В этой части много практики, и вы сможете сразу же использовать эти же методы в своем коде.

Глава 10: map, flatMap и compactMap

В этой главе освещаются ключевые понятия, обычно встречающиеся не только в Swift, но и в других фреймворках и языках программирования.

Рано или поздно вы столкнетесь с методами `map`, `flatMap` и `compactMap` в массивах, опционалах, типах ошибок и, возможно, даже функциональном реактивном программировании, например RxSwift.

Вы получите правильное представление о том, как очистить код, применив методы `map` и `flatMap` к опциональным типам, а также узнаете, как использовать метод `map` при работе со словарями, массивами и другими типами коллекции, и познакомитесь с преимуществами применения метода `flatMap` при работе со строками.

Наконец, вы сможете ознакомиться с методом `compactMap` и его элегантной обработкой опциональных типов в коллекциях.

Знание методов `map`, `flatMap` и `compactMap` на более глубоком уровне является хорошей основой для понимания того, как читать и писать более краткий, но элегантный код, и для работы с типом `Result` в главе 11.

Глава 11: Асинхронная обработка ошибок с помощью типа Result

Обработка ошибок в Swift немного отстает от асинхронной обработки ошибок. Вы познакомитесь с этим поближе и узнаете, как обеспечить безопасность на этапе компиляции для асинхронного программирования, воспользовавшись так называемым типом `Result`, который неофициально предлагается компанией Apple через менеджер пакетов Swift.

Возможно, вы уже используете какую-то версию типа `Result`, найденную во фреймворках. Но даже если вы знакомы с ним, я готов поспорить, что в этой главе вы увидите новые и полезные методы.

Вы начнете с изучения недостатков традиционной обработки ошибок в стиле Cocoa и того, почему `Result` может помочь вам в этом. Затем увидите, как преобразовать традиционный вызов в тот, что использует `Result`.

Кроме того, вы увидите преобразование генерирующих функций в `Result` и обратно. Вы будете применять специальный тип `AnyError` для создания большей гибкости, избегая NSError и гарантируя повышенную безопасность на этапе компиляции.

Вы познакомитесь с типом `Never`, который является уникальным способом сообщить компилятору Swift, что у `Result` ничего не получается или он потерпел неудачу.

Наконец, вы будете использовать то, чему научились при применении методов `map` и `flatMap` с опционалами, чтобы понять, как применять метод `map` при работе со значениями и ошибками и даже как использовать метод `flatMap` с типом

`Result`. В результате вы получите так называемый монадический стиль обработки ошибок, который дает возможность очень аккуратно и элегантно распространять ошибки в стеке вызовов с очень небольшим количеством кода, сохраняя при этом повышенную безопасность.

Глава 12: Расширения протоколов

Эта глава полностью посвящена моделированию данных с помощью уменьшения связанности (decoupling). В ней предлагаются варианты реализации по умолчанию посредством протоколов. Мы будем использовать умные переопределения и узнаем, как расширить типы интересными способами.

Для начала вы узнаете о моделировании данных с использованием протоколов по сравнению с подклассами.

Затем вы будете моделировать данные, используя два подхода: один подход предполагает наследование протокола, а другой использует композицию протокола. У обоих есть свои плюсы и минусы, которые вы обнаружите, когда перейдете к рассмотрению связанных с ними компромиссов.

Кроме того, вы увидите, как работают расширения протокола, если они переопределены наследованием протокола и конкретными типами. Это несколько гипотетично, но полезно, чтобы понимать протоколы на более глубоком уровне.

Вы также увидите, как осуществлять расширение в двух направлениях. Одно направление – это расширение класса, чтобы соответствовать протоколу, а второе – расширение протокола и ограничение его классом. Это тонкое, но важное отличие.

В конце главы вы расширите протокол `Collection`, а затем пойдете дальше и расширите протокол `Sequence` для создания расширений с возможностью многократного использования. Вы познакомитесь с `ContiguousArray` и функциями с ключевым словом `rethrows`, а также создадите полезные методы, которые сможете напрямую применять в своих проектах.

Глава 13: Шаблоны Swift

Это, возможно, самая трудная глава в книге, но это вершина, которую стоит покорить.

Цель данной главы – справиться с распространенными препятствиями, с которыми вы можете столкнуться. Шаблоны, описанные здесь, не являются перефразировкой принципов SOLID – об этом написано множество книг! Вместо этого она фокусируется на современных подходах к современному языку.

Вы узнаете, как мокировать API с помощью протоколов и ассоциированных типов, – что часто бывает удобно, – чтобы иметь возможность создавать автономную и тестовую версии API.

Затем вы увидите, как работает условное соответствие касательно обобщенных типов и протоколов с ассоциированными типами. После этого создадите обобщенный тип и запустите его, используя мощную технику условного соответствия, которая является еще одним способом написания очень гибкого кода.

Далее вы будете иметь дело с проблемой, с которой можно столкнуться при попытке использовать протокол в качестве конкретного типа. Для борьбы с этим вы будете использовать два метода: один включает в себя перечисления, а второй – продвинутую технику под названием стирание типов.

Наконец, вы также проверите, можно ли считать протоколы хорошим выбором. Вопреки распространенному мнению протоколы – не всегда то, что нужно. Вы найдете альтернативный способ создания гибкого типа, включающий в себя структуру и функции высшего порядка.

Глава 14: Написание качественного кода на языке Swift

Это наименее ориентированная на код глава, но, возможно, одна из самых важных.

Здесь рассказывается о написании чистого, понятного кода, который создает меньше головной боли для всех, кто работает в вашей команде (если вы являетесь ее членом). Она ставит перед вами задачу создания соглашений об именах, добавления документации и комментариев, а также разделения больших классов на небольшие обобщенные компоненты. Вы также настроите SwiftLint, инструмент, который добавляет согласованность стилей и помогает избежать ошибок в проектах, а также познакомитесь с архитектурой и тем, как преобразовать большие классы со слишком большим количеством обязанностей в более мелкие обобщенные типы.

Эта глава – неплохая проверка, чтобы увидеть, соответствует ли ваш код стандартам и стилям, что поможет при создании запросов на принятие изменений или завершении задания при собеседовании на новой работе.

Глава 15: Что дальше?

На этом этапе ваши навыки в Swift будут достаточно сильными. Я поделюсь с вами несколькими советами о том, куда двигаться дальше, чтобы вы могли продолжить свое путешествие.

О коде

Эта книга содержит много примеров исходного кода, как в пронумерованных листингах, так и в обычном тексте. В обоих случаях исходный код форматируется `шрифтом фиксированной ширины, как этот`, чтобы отделить его от обычного текста. Иногда код также выделяется **жирным шрифтом**, чтобы выделить то, что изменилось по сравнению с предыдущими шагами в этой главе, например когда в существующую строку кода добавляется новая функция.

Во многих случаях оригинальный исходный код был переформатирован. Мы добавили разрывы строк и переработали отступы, чтобы разместить доступное пространство страницы в книге. В редких случаях даже этого было недостаточно, и списки содержат маркеры продолжения строки (➥). Кроме того, комментарии в исходном коде часто удалялись из списков при описании кода в тексте. Аннотации к кодам сопровождают многие списки, выделяя важные понятия.

Исходный код для всех листингов, приведенных в этой книге, можно загрузить с сайта издательства Manning по адресу https://www.manning.com/books/swift-in-depth и с GitHub на странице https://github.com/tjeerdintveen/manning-swift-in-depth. В каждую главу включен исходный код, за исключением глав 14 и 15.

Книжный форум

Приобретение данной книги включает в себя бесплатный доступ к частному веб-форуму, организованному Manning Publications, где можно оставить комментарии о книге, задать технические вопросы и получить помощь от авторов и других пользователей. Чтобы получить доступ к форуму, перейдите по ссылке https://forums.manning.com/forums/swift-in-depth. Вы также можете узнать больше о форумах Manning и правилах поведения на странице https://forums.manning.com/forums/about. Обязательство издательства по отношению к нашим читателям состоит в том, чтобы обеспечить пространство, где может иметь место содержательный диалог между отдельными читателями и между читателями и авторами. Это не обязательство какого-либо конкретного участия со стороны авторов, чей вклад в форум остается добровольным (и не оплачивается). Мы предлагаем вам попробовать задать авторам несколько сложных вопросов, чтобы их интерес не угас! Форум и архив предыдущих обсуждений будут доступны на сайте издателя, пока книга находится в печати.

Об авторе

Чейрд ин’т Вейн – заядлый фанат Swift и внештатный разработчик для iOS с опытом работы в агентствах. Он является соучредителем небольшого стартапа и на момент написания этих строк помогает ING расширять их мобильные разработки. Начав свою карьеру в качестве разработчика Flash в 2001 году, он разрабатывал для iOS с помощью Objective-C, занимался веб-разработкой на Ruby и немного освоил другие языки программирования.

Когда он не работает, то проводит время с двумя своими дочерьми, отпуская шутки и балуясь с акустической гитарой.

Вы можете найти его в Twitter (@tjeerdintveen).

Об иллюстрации на обложке

Изображение на обложке книги озаглавлено «Человек из Омишаля, остров Крк, Хорватия». Эта иллюстрация взята из репродукции, опубликованной в 2006 году, из коллекции костюмов и этнографических описаний XIX века под названием *Далмация* профессора Франэ Каррара (1812–1854), археолога и историка, первого директора Музея древности в Сплите, Хорватия. Иллюстрации были получены от услужливого библиотекаря из Этнографического музея (бывший Музей античности), который расположен в римском ядре средневекового центра Сплита: руинах

дворца императора Диоклетиана, датируемого примерно 304 г. н. э. Книга содержит прекрасные цветные иллюстрации жителей разных регионов Далмации, сопровождаемые описаниями костюмов и быта.

С XIX века дресс-код изменился, и разнообразие регионов, столь богатое в то время, исчезло. Сейчас трудно отличить жителей разных континентов, не говоря уже о разных городах или регионах. Возможно, мы обменяли культурное разнообразие на более разнообразную личную жизнь – конечно, на более разнообразную и быстро развивающуюся технологическую жизнь.

В то время когда трудно отличить одну компьютерную книгу от другой, издательство Manning празднует изобретательность и инициативу компьютерного бизнеса, используя обложки, основанные на богатом разнообразии жизни регионов двухвековой давности, оживленной иллюстрациями из коллекций, подобной этой.

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны. Вы можете написать отзыв прямо на нашем сайте `www.dmkpress.com`, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма. Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу `http://dmkpress.com/authors/publish_book/` или напишите в издательство по адресу `dmkpress@gmail.com`.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг — возможно, ошибку в тексте или в коде, — мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги. Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам

адрес копии или веб-сайта, чтобы мы могли применить санкции. Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы. Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Глава 1. Введение

В этой главе:

- краткий обзор популярности Swift и поддерживаемых платформ;
- преимущества Swift;
- более внимательный взгляд на тонкие недостатки Swift;
- о чем мы узнаем в этой книге.

Не секрет, что Swift поддерживается на многих платформах, таких как Apple iOS, macOS, watchOS и tvOS. Swift – это язык с открытым исходным кодом. Он также работает в Linux и набирает популярность на стороне сервера в таких веб-фреймворках, как Vapor, Perfect, Zewo и Kitura от IBM.

Кроме того, Swift медленно не только охватывает прикладное программирование (программное обеспечение для пользователей), но и начинает вводить системное программирование (программное обеспечение для систем), такое как SwiftNIO или инструменты командной строки. Swift превращается в мультиплатформенный язык. При изучении Swift перед вами открывается множество дверей.

Swift был самым популярным языком на сайте Stack overflow в 2015 году и остается в пятерке самых любимых языков в 2017 году. В 2018 году он поднялся на 6-е место (https://insights.stackoverflow.com/survey/2018).

Swift, безусловно, готов остаться, и если вы любите создавать приложения, веб-сервисы или другие программы, моя цель состоит в том, чтобы вы извлекли как можно больше пользы из этой книги как для себя, так и для своей команды.

Что мне нравится в Swift, так это то, что его легко изучать, но сложно овладеть. Всегда есть чему поучиться! Одна из причин заключается в том, что Swift включает в себя множество парадигм программирования, позволяя вам выбрать подходящее решение проблемы программирования, что вы и будете исследовать в этой книге.

1.1. «Золотая середина» SWIFT

Что мне нравится в динамическом языке, таком как Ruby, так это его выразительность. Ваш код говорит вам, чего вы хотите достичь, не слишком занимаясь управлением памятью и низкоуровневыми технологиями. Написание кода на Ruby один день и на Objective-C в другой убедили, что мне нужно было выбирать между скомпилированным языком, который работал хорошо, или экспрессивным, динамическим языком за счет более низкой производительности.

Тогда появился Swift и разрушил это заблуждение. Swift находит правильный баланс, когда он совместно использует множество преимуществ динамических языков, таких как Ruby или Python, предлагая дружественный синтаксис и сильный полиморфизм. Тем не менее Swift избегает некоторых недостатков динами-

ческих языков, в частности производительности, потому что Swift компилируется в машинный код с помощью компилятора LLVM. Поскольку Swift компилируется с помощью LLVM, вы получаете не только высокую производительность, но и тонны проверок безопасности, оптимизаций и гарантии того, что ваш код в порядке, прежде чем запускать его. В то же время Swift выполняет чтение как выразительный динамический язык, с которым приятно работать и легко выражать свои намерения.

Swift не может помешать вам делать ошибки или писать плохой код, но он помогает уменьшить программные ошибки на этапе компиляции, используя различные методы, включая статическую типизацию и сильную поддержку алгебраических типов данных (перечисления, структуры и кортежи), но не ограничиваясь ими. Swift также предотвращает ошибки null благодаря опциональным типам.

Недостатком некоторых статических языков является то, что вам всегда нужно определять типы. Swift делает этот процесс проще с помощью вывода типов и может выводить конкретные типы, когда это имеет смысл. Таким образом, вам не нужно явно указывать каждую переменную, константу и обобщение.

Swift представляет собой смесь различных парадигм программирования, поскольку независимо от того, используете ли вы объектно-ориентированный подход, функциональное программирование или привыкли работать с абстрактным кодом, Swift предлагает все это. В качестве основного убедительного аргумента Swift обладает надежной системой, когда речь заходит о полиморфизме, в форме обобщений и *протокольно-ориентированного программирования*, которое активно используется в качестве маркетингового инструмента как компанией Apple, так и разработчиками (см. рис. 1.1).

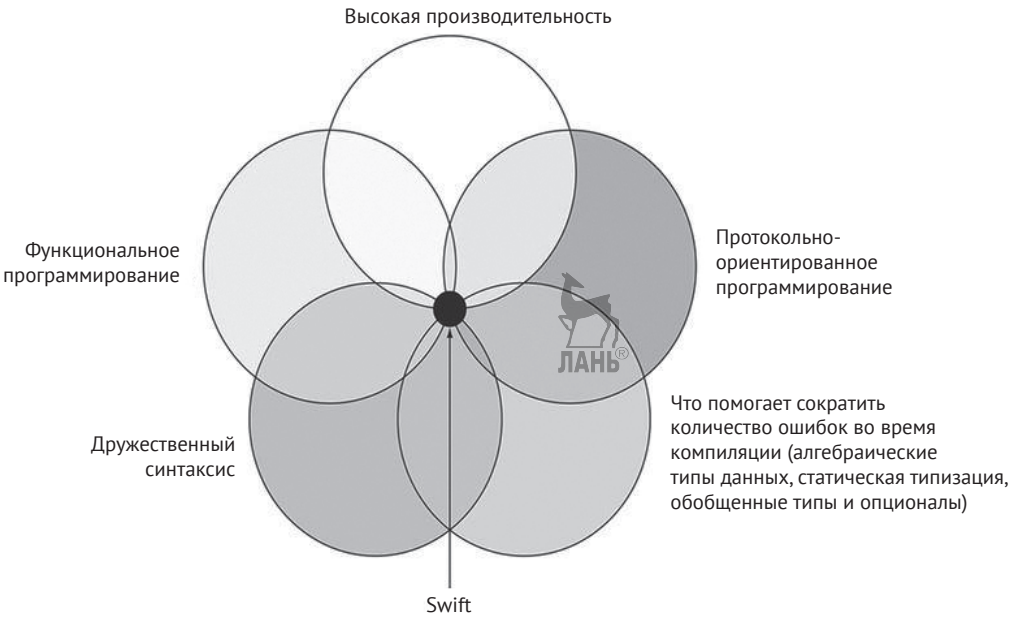


Рис. 1.1. «Золотая середина» Swift

1.2. Под поверхностью

Несмотря на то что Swift впечатляет своим дружественным синтаксисом и обещает создавать потрясающие приложения, это лишь верхушка айсберга. Соблазнительный вход в изучение Swift – это начать с разработки для iOS, поэтому вы узнаете, как создавать красивые приложения, которые состоят из важнейших компонентов из фреймворков Apple.

Но как только вам нужно будет самостоятельно доставить компоненты и приступить к созданию более сложных систем и структур, вы поймете, что Swift усердно работает над тем, чтобы скрыть многие сложности, и делает это успешно. Когда вам понадобится изучить эти сложности, и вы это сделаете, кривая сложности Swift возрастет в геометрической прогрессии. Даже самые опытные разработчики ежедневно обучаются новым трюкам и хитростям Swift!

Как только Swift вас зацепит, вы, скорее всего, столкнетесь со скачками скорости в форме обобщений и ассоциированных типов, и нечто такое «простое», как обработка строк, может вызвать больше проблем, чем вы могли ожидать (см. рис. 1.2).

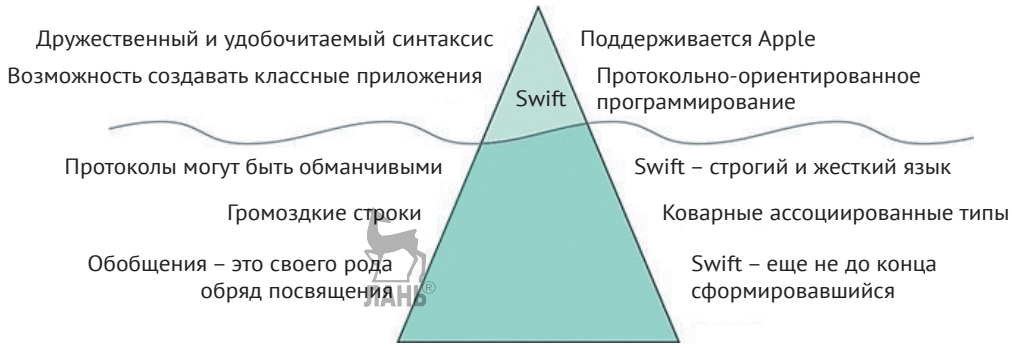


Рис. 1.2. Верхушка айсберга Swift

Эта книга помогает справиться с самыми распространенными сложностями. Вы будете освещать и решать любые проблемы и недостатки с помощью Swift и сможете использовать силы, которые привносят эти сложности, весело проводя время.

1.3. Минусы Swift

Swift – мой любимый язык программирования, но давайте не будем смотреть на него сквозь розовые очки. Как только вы признаете сильные и слабые стороны Swift, сможете адекватно решить, когда и как вы бы хотели его использовать.

1.3.1. Стабильность ABI

Swift все еще движется быстро и не является ABI-стабильным, а это означает, что код, написанный на Swift 4, не будет совместим с Swift 5, и наоборот. Представь-

те, что вы пишете фреймворк для своего приложения. Как только выйдет Swift 5, приложение, написанное на Swift 5, не сможет использовать ваш фреймворк, пока вы не обновите свой фреймворк до Swift 5. К счастью, Xcode сильно помогает в случае с миграцией, поэтому я ожидаю, что эта миграция не будет такой болезненной.

1.3.2. Строгость

Swift – строгий и жесткий язык, что является распространенной критикой статических языков, тем более при работе со Swift. Прежде чем освоиться, у вас может создаться впечатление, словно вы печатаете в наручниках. На практике вам нужно разрешить многие типы на этапе компиляции, например с помощью опциональных типов, смешивания значений в коллекциях или обработки перечислений.

Я бы сказал, что строгий характер Swift является одной из его сильных сторон. Как только вы пытаетесь компилировать, вы узнаете о коде, который не работает, а не о том, что клиент столкнулся с ошибкой во время выполнения. После того как вы сделали первоначальные инвестиции, чтобы освоиться со Swift, они станут естественными, и их ограниченность меньше будет вас останавливать. Эта книга поможет вам преодолеть себя, и Swift станет вашей второй натурой.

1.3.3. Сложность протоколов

Протоколы – это ключевой момент Swift. Но как только вы начнете работать с протоколами, будете задевать острые края и время от времени наносить себе порезы. То, что «должно просто работать», почему-то не работает. Протоколы хороши в своем текущем состоянии и уже достаточно хороши для создания качественного программного обеспечения, но иногда вы сталкиваетесь с трудностями, и вам приходится использовать обходные пути, о которых много говорится в этой книге.

Вот распространенный источник разочарования: если вам нужно передать типы, соответствующие протоколу `Equatable`, в функцию, чтобы увидеть, равны ли они, вас остановят. Например, можно попытаться проверить, равно ли значение всему массиву, как показано в приведенном ниже листинге. Вы увидите, что в Swift это не сработает.

Листинг 1.1. Попытка проверки равенства

```
areAllEqual(value: 2, values: [3,3,3,3])

func areAllEqual(value: Equatable, values: [Equatable]) -> Bool {
   guard !values.isEmpty else { return false }

   for element in values {
      if element != value {
         return false
      }
```

```
    }

    return true
}
```


Swift возвращает загадочную ошибку с неопределенным предложением относительно дальнейших действий:

```
error: protocol 'Equatable' can only be used as a generic constraint
because it has Self or associated type requirements
```

Вы поймете, почему это происходит и как избежать этих проблем, в главах 7 и 8.

Расширения протокола Swift – еще одно основное преимущество и одна из самых мощных функций, которые он может предложить. Протоколы могут действовать как интерфейсы. Расширения протокола предлагают реализации по умолчанию для типов, помогая вам избежать жесткого деления на подклассы.

Протоколы, однако, хитрее, чем может показаться, и могут удивить даже опытного разработчика. Например, допустим, у вас есть протокол под названием `FlavorType`, обозначающий продукт или напиток, который вы можете улучшить с помощью аромата, например кофе. Если вы расширите этот протокол с помощью реализации по умолчанию, которая *не* найдена в объявлении протокола, то можете получить удивительные результаты! Обратите внимание, что в следующем листинге у вас есть два типа `Coffee`, но они оба дают разные результаты при вызове для них `addFlavor` . Это небольшая, но важная деталь.

Листинг 1.2. Протоколы могут вас удивить

```
protocol FlavorType{
    // func addFlavor()
    // Мы получим разные результаты, если этот метод не существует.
}

extension FlavorType {
    func addFlavor() { // Создаем реализацию по умолчанию.
        print(«Adding salt!»)
    }
}

struct Coffee: FlavorType {
    func addFlavor() { // Структура Coffee предоставляет свою реализацию.
        print(«Adding cocoa powder»)
    }
}

let tastyCoffee: Coffee = Coffee() // tastyCoffee имеет тип 'Coffee'
tastyCoffee.addFlavor() // Добавляем какао-порошок

let grossCoffee: FlavorType = tastyCoffee // grossCoffee имеет тип FlavorType
grossCoffee.addFlavor() // Добавляем соль
```

Даже если вы имеете дело с кофе того же типа, сначала вы добавляете какао-порошок, а затем случайно добавляете соль, которая не помогает никому вставать по утрам. Какими бы эффективными ни были протоколы, иногда они могут вносить незначительные ошибки.

1.3.4. Параллелизм

Наши компьютеры и устройства – это параллельные машины, использующие несколько процессоров одновременно. Работая в Swift, вы уже можете выражать параллельный код через Grand Central Dispatch (GCD). Но языковая особенность параллелизма в Swift не существует.

Поскольку вы уже можете использовать GCD, не такая уж большая проблема, чтобы немного подождать подходящую модель параллелизма. Тем не менее GCD в сочетании со Swift не безупречен.

Во-первых, работа с GCD может создать так называемую пирамиду гибели, также известную как глубоко вложенный код, о чем свидетельствует кусок незавершенного кода в листинге 1.3.

Листинг 1.3. Пирамида гибели

```
func loadMessages(completion: (result: [Message], error: Error?) -> Void) {
   loadResource(«/user») { user, error in
      guard let data = data else {
         completion(nil, error)
         return
      }
      loadResource(«/messages/», user.id) { messages, error in
         guard let messages = messages else {
            completion(nil, error)
            return
         }
         storeMessages(messages) { didSucceed, error in
            guard let error != nil else
            {
               completion(nil, error)
               return
            }
            DispatchQueue.main.async { // При удалении элементов вы
            // уменьшаете их число
               completion(messages)
            }
         }
      }
   }
}
```

Во-вторых, вы не знаете, какая очередь вызывается асинхронным кодом. Если вам нужно вызвать `loadMessages`, у вас могут быть проблемы, если небольшое изменение перемещает блок завершения в фоновую очередь. Вы можете быть очень осторожны и завершить обратный вызов в главной очереди в месте вызова, но в любом случае есть компромисс.

В-третьих, обработка ошибок является неоптимальной и не соответствует модели Swift. И возвращаемые данные, и ошибка теоретически могут быть заполнены или равны нулю. Из кода не ясно, что это может быть только одно или другое. Мы рассмотрим это в главе 11. Вы можете ожидать модель асинхронного ожидания в Swift более поздних версий, возможно, версии 7 или 8, а это означает ожидание, пока эти проблемы не будут решены. К счастью, GCD более чем достаточно для большинства ваших потребностей, и вы можете обратиться к RxSwift в качестве альтернативы для реактивного программирования.

1.3.5. Отход от платформ Apple

Swift выходит на новую территорию для интернета и в качестве системного языка, особенно благодаря поддержке IBM в форме веб-сервера Kitura и переводу Swift в облако. Отказ от платформ Apple открывает захватывающие возможности, но имейте в виду, что вы можете столкнуться с нехваткой пакетов, которые могли бы помочь вам в этом. Для таких xOS-фреймворков, как iOS, watchOS, tvOS и macOS, можно использовать CocoaPods или Carthage, чтобы обрабатывать зависимости. За пределами семейства xOS вы можете использовать менеджер пакетов Swift, предлагаемый компанией Apple. Однако многие разработчики Swift сосредоточены на iOS, и вы можете столкнуться с заманчивым пакетом без поддержки диспетчера пакетов Swift.

Хотя нелегко соответствовать существующим экосистемам, таким как тысячи пакетов Python, npm из Node.js и Ruby gems, это также зависит от вашей перспективы. Отсутствие пакетов может быть сигналом того, что вы можете внести свой вклад в сообщество и экосистему, изучая Swift.

Хотя Swift – это язык с открытым исходным кодом, Apple держит руль. Вам не нужно беспокоиться о том, что Apple прекратит поддержку Swift, но не всегда можете соглашаться с направлением или скоростью обновлений Swift. Вы все еще должны зависеть от сторонних инструментов, чтобы получить зависимости, работающие для iOS и macOS, а Xcode пока плохо интегрируется с Swift Package Manager. К сожалению, обе эти проблемы, по-видимому, имеют низкий приоритет для Apple.

1.3.6. Время компиляции

Swift – высокопроизводительный язык. Но процесс компиляции может быть довольно медленным и отнимать много времени у разработчиков. Swift компилируется в несколько этапов с помощью компилятора LLVM. Хотя это дает оптимизацию при запуске кода, в повседневном программировании вы можете столкнуться с медленной сборкой, что может быть немного утомительно, если вы пытаетесь быстро протестировать работающий фрагмент кода. Также не каждый проект

Swift представляет собой отдельный проект. Как только вы включите несколько фреймворков, вы будете компилировать много кода для создания сборки, замедляя свой процесс.

В конце концов, у каждого языка программирования есть свои плюсы и минусы – это вопрос выбора правильного инструмента для работы. Я верю, что у Swift большое будущее, которое поможет вам создать красивое программное обеспечение с чистым, сжатым и высокопроизводительным кодом!

1.4. Чему вы научитесь

Цель этой книги – показать вам, как решать проблемы элегантными способами, применяя передовые методы.

Несмотря на то что на обложке этой книги написано Swift, я думаю, что одним из ее сильных сторон является то, что вы изучите концепции, которые легко перенести на другие языки. Вы узнаете о:

- концепциях функционального программирования, таких как `Reduce`, `FlatMap`, `Optional` и `Result`, и о том, как мыслить алгебраическими типами данных, используя структуры, кортежи и перечисления;
- многих реальных сценариях, к которым вы подходите с разных сторон, рассматривая плюсы и минусы каждого из подходов;
- обобщениях, охватывая полиморфизм на этапе компиляции;
- том, как написать более надежный, лаконичный и легко читаемый код;
- протокольно-ориентированном программировании, включая полное понимание ассоциированных типов, которые считаются самой сложной частью протоколов;
- работе Swift во время выполнения и на этапе компиляции с использованием обобщений, перечислений и протоколов;
- способе найти компромисс между функциональным, объектно-ориентированным и протокольно-ориентированным программированием.

В конце этой книги ваш пояс с инструментами будет тяжелым из-за всех этих вариантов, которые вам придется решать при программировании. После того как вы усвоите эти концепции, вы, возможно, обнаружите, что изучать другие языки, такие как Kotlin, Rust и подобные, которые разделяют схожие идеи и функциональность, намного проще.

1.5. Как извлечь максимум из этой книги

Отличный способ выучить язык программирования – выполнять упражнения перед чтением главы. Будьте честны и посмотрите, сможете ли вы *по-настоящему* закончить их, вместо того чтобы смотреть на них и думать, что вы уже знаете ответ. В иных упражнениях могут быть скрыты некоторые сложные ситуации, и вы увидите их, как только начнете над ними работать.

Выполнив упражнения, решите, хотите ли вы прочитать главу, чтобы узнать новый материал.

В этой книге материал идет потоком, и по мере прочтения его сложность растет. Тем не менее книга построена по модульному принципу. Можно читать главы не по порядку.

1.6. Минимальная квалификация

Эта книга не рассчитана на абсолютного новичка. Предполагается, что вы немного работали с Swift раньше.

Если вы считаете себя продвинутым новичком или на среднем уровне, большинство глав будет вам полезно. Если вы считаете себя опытным разработчиком, я все же полагаю, что многие главы будут полезны, дабы заполнить пробелы в ваших знаниях. Благодаря модульности этой книги вы можете выбирать главы, которые хотели бы прочитать, не читая предыдущих.

1.7. Версия Swift

Эта книга написана для Swift версии 4.2. Все примеры будут работать на этой версии либо в командной строке, либо в сочетании с Xcode 10.

Резюме

- Swift поддерживается на многих платформах.
- Swift часто используется для разработки под iOS, watchOS, tvOS и macOS и с каждым днем все больше и больше для веб-разработки, системного программирования, инструментов командной строки и даже машинного обучения.
- Swift проводит тонкую грань между высокой производительностью, удобочитаемостью и безопасностью на этапе компиляции.
- Swift легок в изучении, но его трудно освоить.
- Swift – безопасный и высокопроизводительный язык.
- В Swift нет встроенной поддержки параллелизма.
- Диспетчер пакетов Swift пока еще не работает для iOS, watchOS, tvOS или macOS.
- Swift включает в себя несколько стилей программирования, таких как функциональное программирование, объектно-ориентированное программирование и программирование протоколов.

Глава 2. Моделирование данных с помощью перечислений

В этой главе:

- почему перечисления являются альтернативой подклассам;
- использование перечислений для полиморфизма;
- почему перечисления являются типами «OR»;
- моделирование данных с помощью перечислений вместо структур;
- почему перечисления и структуры являются алгебраическими типами;
- преобразование структур в перечисления;
- безопасная обработка перечислений с необработанными значениями;
- преобразование строк в перечисления для создания надежного кода.

Перечисления являются основным инструментом, используемым разработчиками Swift. Перечисления позволяют определять тип, *перечисляя* его значения, например является ли HTTP- метод действием *get*, *put*, *post* либо *delete*, или указывает, имеет IP-адрес формат IPv4 либо IPv6.

Во многих языках есть реализация перечислений, с разными типами реализации для каждого языка. Перечисления в Swift, в отличие от C и Objective-C, не являются *всего лишь* представлениями целочисленных значений. Вместо этого Swift заимствует многие концепции из мира функционального программирования, что дает множество преимуществ, которые вы исследуете в этой главе.

На самом деле я бы сказал, что перечисления немного недоиспользуются в стране Swift. Я надеюсь изменить это и помочь вам понять, что перечисления могут быть удивительно полезны во многих отношениях. Моя цель – расширить ваш словарь перечислений, чтобы вы могли напрямую использовать эти методы в своих проектах.

Вначале вы увидите несколько способов моделирования ваших данных с помощью перечислений и их сопоставления со структурами и классами.

Перечисления – это способ предложить полиморфизм, то есть вы можете работать с одним типом, обозначающим другие типы. Мы пролили свет на то, как можно хранить несколько типов в одной коллекции, например в массиве.

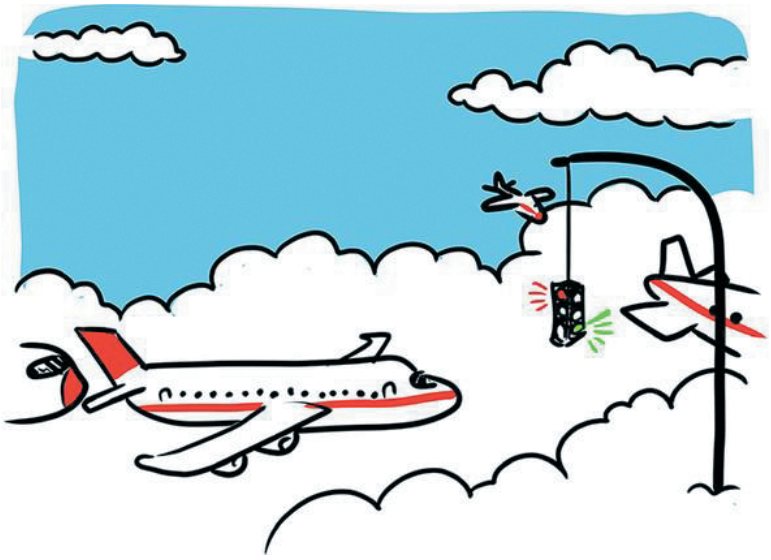
Затем вы увидите, что перечисления являются подходящей альтернативой для создания подклассов.

Мы немного углубимся в алгебраическую теорию, чтобы понять перечисления на более глубоком уровне; тогда вы увидите, как можно применить эту теорию и преобразовать структуры в перечисления и обратно.

В качестве вишенки на торте мы исследуем необработанные перечисления значений и то, как можно использовать их для аккуратной обработки строк. Прочитав эту главу, вы, возможно, обнаружите, что лучше моделируете данные, чуть чаще пишете перечисления и в итоге получаете в своих проектах более безопасный и чистый код.

2.1. OR в сравнении с AND

Перечисления можно рассматривать как тип «or». Перечисления могут быть только чем-то одним одновременно, например светофор может быть либо зеленым, либо желтым, либо красным. В качестве альтернативы кубик может быть либо шестигранным, либо двадцатигранным, но не то и другое одновременно.



Присоединяйтесь!

Будет более познавательно и веселее, если вы будете знакомиться с кодом по мере прохождения этой главы. Исходный код можно скачать по адресу `https://mng.bz/gNre`.

2.1.1. Моделирование данных с помощью структуры

Давайте начнем с примера, который показывает, как рассматривать типы «or» и «and» при моделировании данных.

В следующем примере вы будете моделировать данные сообщения в чат-приложении. Сообщение может быть текстом, которое может отправить пользователь, но еще может быть сообщением о присоединении или сообщением о выходе. Сообщение может даже быть сигналом для отправки воздушных шаров (см. рис. 2.1). А почему бы и нет? Apple тоже делает это в приложении Messages.

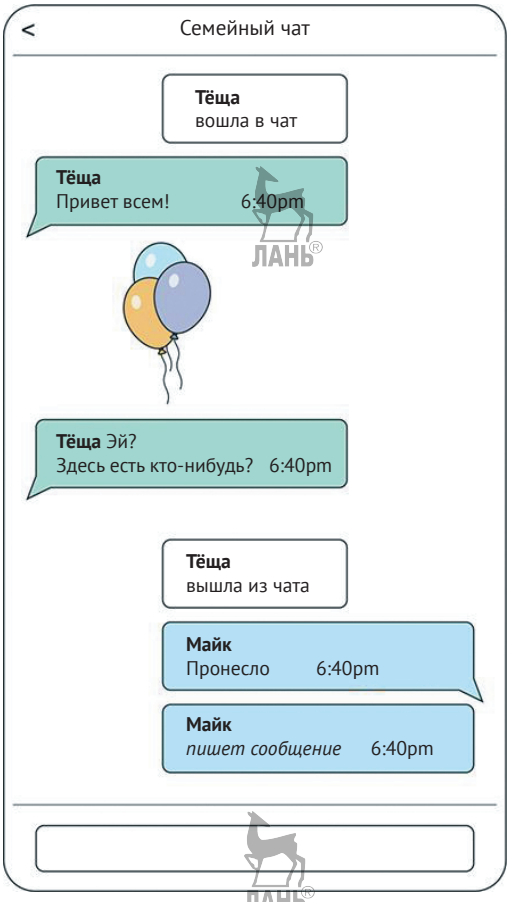


Рис. 2.1. Чат

Вот несколько типов сообщений, которые может поддерживать ваше приложение:

- сообщения о присоединении, такие как «Тёща вошла в чат»;
- текстовые сообщения, которые кто-то может написать, например «Привет всем!»;
- отправка воздушных шаров, что включает в себя анимацию и раздражающие звуки, которые другие могут видеть и слышать;
- сообщения о выходе, такие как «Тёща вышла из чата»;
- черновики, такие как «Майк пишет сообщение».

Давайте создадим модель данных для обозначения сообщений. Ваша первая идея может заключаться в использовании структуры для моделирования вашего сообщения. Вы начнете с этого и продемонстрируете проблемы, которые с этим связаны. Тогда вы решите эти проблемы, используя перечисление.

Вы можете создавать несколько типов сообщений в коде, например сообщение о присоединении, когда кто-то входит в чат.

Листинг 2.1. Сообщение о входе в чат

```
import Foundation //Необходимо для типа Date.

let joinMessage = Message(userId: «1»,
   contents: nil,
   date: Date(),
   hasJoined: true, // Устанавливаем значение логического типа данных
   hasLeft: false,
   isBeingDrafted: false,
   isSendingBalloons: false)
```

Вы также можете создать обычное текстовое сообщение.

Листинг 2.2. Текстовое сообщение

```
let textMessage = Message(userId: «2»,
   contents: «Hey everyone!», // Передаем сообщение
   date: Date(),
   hasJoined: false,
   hasLeft: false,
   isBeingDrafted: false,
   isSendingBalloons: false)
```

В вашем гипотетическом приложении для обмена сообщениями вы можете передавать данные этого сообщения другим пользователям. Структура `Message` выглядит так:

Листинг 2.3. Структура Message

```
import Foundation

struct Message {
   let userId: String
   let contents: String?
   let date: Date
   let hasJoined: Bool
   let hasLeft: Bool
   let isBeingDrafted: Bool
   let isSendingBalloons: Bool
}
```

Хотя это один небольшой пример, он выдвигает на первый план проблему. Поскольку структура может содержать несколько значений, вы можете столкнуться с ошибками, когда структура `Message` может быть текстовым сообщением, командой `hasLeft` и командой `isSendingBalloons`. Недопустимое состояние сообщения не сулит ничего хорошего, потому что сообщение может быть только тем или иным в бизнес-правилах приложения. Визуальные элементы также не будут поддерживать неверное сообщение.

В качестве иллюстрации вы можете получить сообщение в недопустимом состоянии. Он представляет собой текстовое сообщение, а также сообщение о входе и выходе из чата.

Листинг 2.4. Недопустимое сообщение с конфликтующими свойствами

```
let brokenMessage = Message(userId: "1",
   contents: "Hi there", // Текст для показа
   date: Date(),
   hasJoined: true, // Однако данное сообщение также сигнализирует о входе
   hasLeft: true, // ... и выходе
   isBeingDrafted: false,
   isSendingBalloons: false)
```

В небольшом примере столкнуться с неверными данными сложнее, но это неизбежно случается достаточно часто в реальных проектах. Представьте, что вы выполняете преобразование локального файла в `Message` или какую-либо функцию, которая объединяет два сообщения в одно. У вас нет никаких гарантий на этапе компиляции, что сообщение находится в правильном состоянии.

Вы можете подумать о проверке сообщения и генерировании ошибок, но затем обнаруживаете недопустимые сообщения во время выполнения (если они вообще есть). Вместо этого можно применить корректность на этапе компиляции, если мы моделируем структуру `Message`, используя перечисление.

2.1.2. Превращаем структуру в перечисление

Всякий раз, когда вы моделируете данные, посмотрите, можете ли вы найти *взаимоисключающие* свойства. Сообщение не может быть одновременно сообщением о присоединении и выходе. Сообщение также не может отправлять воздушные шары и одновременно быть черновиком.

Но сообщение может быть сообщением о присоединении *или* сообщением о выходе. Сообщение также может быть черновиком *или* обозначать отправку шаров. Когда вы обнаруживаете в модели операторы "or", перечисление может быть более подходящим выбором для вашей модели данных.

Использование перечисления для группировки свойств в кейсах делает данные намного понятнее для понимания. Давайте улучшим модель, превратив ее в перечисление.

Листинг 2.5. Message в качестве перечисления (без значений)

```
import Foundation

enum Message {
      case text
      case draft
      case join
      case leave
```

```
    case balloon
}
```


Но вы еще не закончили, потому что в кейсах нет значений. Их можно добавить с помощью кортежей. Кортеж – это упорядоченный набор значений, например (`userId: String`, `contents: String`, `date: Date`).

Комбинируя перечисление и кортежи, можно создавать более сложные структуры данных. Давайте теперь добавим кортежи.

Листинг 2.6. Message в качестве перечисления (со значениями)

```
import Foundation

enum Message {
   case text(userId: String, contents: String, date: Date)
   case draft(userId: String, date: Date)
   case join(userId: String, date: Date)
   case leave(userId: String, date: Date)
   case balloon(userId: String, date: Date)
}
```

После того как мы добавили кортежи, у кейсов есть так называемые *ассоциированные значения*, выражаясь терминами Swift. Кроме того, четко видно, какие свойства принадлежат друг другу, а какие нет.

Всякий раз, когда вам нужно создать `Message` в качестве перечисления, вы можете выбрать подходящий кейс с ассоциированными свойствами, не беспокоясь о смешивании и сопоставлении неправильных значений.

Листинг 2.7. Создание Message в качестве перечислений

```
let textMessage = Message.text(userId: "2", contents: "Bonjour!", ➥
   date: Date())
let joinMessage = Message.join(userId: "2", date: Date())
```


Когда вы хотите работать с сообщениями, можете использовать для них конструкцию `switch – case` и извлечь их внутренние значения.

Допустим, вы хотите регистрировать отправленные сообщения.

Листинг 2.8. Регистрация сообщений

```
logMessage(message: joinMessage) // Пользователь № 2 вошел в чат.
logMessage(message: textMessage) // Пользователь № 2 отправляет сообщение:
Добрый день!

func logMessage(message: Message) {
   switch message {
      case let .text(userId: id, contents: contents, date: date):
         print("[\(date)] User \(id) sends message: \(contents)")
      case let .draft(userId: id, date: date):
```

```
            print("[\(date)] User \(id) is drafting a message")
        case let .join(userId: id, date: date):
            print("[\(date)] User \(id) has joined the chatroom")
        case let .leave(userId: id, date: date):
            print("[\(date)] User \(id) has left the chatroom")
        case let .balloon(userId: id, date: date):
            print("[\(date)] User \(id) is sending balloons")
    }
}
```

Необходимость использования оператора `switch` для всех значений, приведенных в кейсах, на протяжении всего приложения, чтобы просто прочитать значение из одного сообщения, может быть сдерживающим фактором. Можно сэкономить время при наборе текста, используя комбинацию `if case let`, чтобы выполнить сопоставление для одного типа `Message`.

Листинг 2.9. Сопоставление для одного кейса

```
if case let Message.text(userId: id, contents: contents, date: date) =
    textMessage {
        print("Received: \(contents)") // Получено: Bonjour!
    }
```

Если вас не интересуют конкретные свойства при использовании сопоставления для перечисления, можно выполнить сопоставление с помощью подчеркивания, называемого *подстановочным знаком*, или, как мне нравится называть, оператором «мне все равно».

Листинг 2.10. Сопоставление с использованием подчеркивания

```
if case let Message.text(_, contents: contents, _) = textMessage {
    print("Received: \(contents)") // Получено: Bonjour!
}
```

2.1.3. Выбор между структурами и перечислениями

Получение преимуществ компилятора с помощью перечислений – значительное преимущество. Но если вы часто ловите себя на мысли, что используете сопоставление с образцом для единичного кейса, лучше использовать структуру.

Кроме того, имейте в виду, что ассоциированные значения перечисления – это контейнеры без дополнительной логики. У вас не будет бесплатных инициализаторов свойств; в случае с перечислениями нужно добавить их вручную.

В следующий раз, когда вы будете писать структуру, попробуйте сгруппировать свойства. Ваша модель данных может быть хорошим кандидатом для перечисления!

2.2. Перечисления для полиморфизма

Иногда вам нужна гибкость в форме *полиморфизма*. Полиморфизм означает, что одна функция, метод, массив, словарь – как угодно назовите – может работать с разными типами.

Однако если вы смешиваете типы в массиве, то получаете массив типа [Any] (как показано в следующем листинге), например когда вы помещаете Date, String и Int в один массив.

Листинг 2.11. Заполнение массива несколькими значениями

```
let arr: [Any] = [Date(), "Why was six afraid of seven?", "Because...", 789]
```

Массивы явно хотят быть заполненными одним и тем же типом. В Swift общее для этих смешанных типов то, что они являются типом Any.

Обработка типов Any часто не идеальна. Поскольку вы не знаете, что представляет собой Any на этапе компиляции, нужно проверить тип Any во время выполнения, чтобы увидеть, что он обозначает. Например, использовать сопоставление с шаблоном с помощью оператора *switch*.

Листинг 2.12. Сопоставление для значений Any во время выполнения

```
let arr: [Any] = [Date(), "Why was six afraid of seven?", "Because...", 789]

for element: Any in arr {
// element - это тип "Any"
   switch element {
      case let stringValue as String: "received a string: \(stringValue)"
      case let intValue as Int: "received an Int: \(intValue)"
      case let dateValue as Date: "received a date: \(dateValue)"
      default: print("I am not interested in this value")
   }
}
```


У нас по-прежнему есть возможность выяснить, что представляет из себя тип Any во время выполнения. Но мы не знаем, чего ожидать при сопоставлении; следовательно, мы также должны реализовать кейс по умолчанию, чтобы получить значения, которые нас не интересуют.

Работа с типами Any иногда необходима, когда вы не знаете, что происходит на этапе компиляции, например когда вы получаете неизвестные данные с сервера. Но если вы заранее знаете типы, с которыми имеете дело, то можете обеспечить безопасность на этапе компиляции с помощью перечисления.

2.2.1. Полиморфизм на этапе компиляции

Представьте, что вы хотите хранить два разных типа в массиве, например Date и диапазон из двух дат типа Range <Date>.

<Date> – что это за скобки?

Range – это тип, обозначающий нижнюю и верхнюю границы. Нотация <Date> указывает на то, что Range хранит обобщенный тип, который вы подробно будете изучать в главе 7.

Нотация Range <Date> сообщает вам, что вы работаете с диапазоном двух типов Date.

Можно создать DateType, обозначающий либо одну дату, либо диапазон дат. Затем можно заполнить массив, как показано ниже.

Листинг 2.13. Добавление нескольких типов в массив через перечисление

```
let now = Date()
let hourFromNow = Date(timeIntervalSinceNow: 3600)

let dates: [DateType] = [
   DateType.singleDate(now),
   DateType.dateRange(now..<hourFromNow)
]
```

Само перечисление просто содержит два кейса каждый со своим ассоциированным значением.

Листинг 2.14. Перечисление DateType

```
enum DateType {
   case singleDate(Date)
   case dateRange(Range<Date>)
}
```

Сам массив состоит только из экземпляров DateType. В свою очередь, каждый DateType содержит один из нескольких типов (см. рис. 2.2).

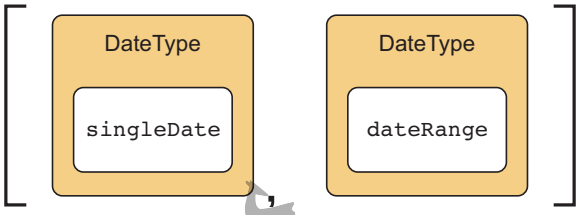


Рис. 2.2. Перечисления массива

Благодаря перечислению мы получаем массив, содержащий несколько типов, сохраняя при этом безопасность на этапе компиляции. Если бы мы читали значения из массива, то могли бы использовать оператор switch для каждого значения.

Листинг 2.15. Сопоставление для перечисления dateType

```
for dateType in dates {
   switch dateType {
```

```
        case .singleDate(let date): print("Date is \(date)")
        case .dateRange(let range): print("Range is \(range)")
    }
}
```

Компилятор также полезен, если вы измените перечисление. В качестве иллюстрации, если вы добавите в перечисление кейс `year`, компилятор сообщит вам, что вы забыли его обработать.

Листинг 2.16. Добавление кейса year

```
enum DateType {
    case singleDate(Date)
    case dateRange(Range<Date>)
    case year(Int) ❶
}
```

❶ Добавлен кейс `year`

Компилятор теперь выдает следующее.

Листинг 2.17. Компилятор уведомляет об ошибке

```
error: switch must be exhaustive
    switch dateType {
    ^

add missing case: '.year(_)'
    switch dateType {
```

Благодаря перечислениям вы можете вернуть безопасность на этапе компиляции при смешивании типов внутри массивов и других структур, таких как словари.

Конечно, вы должны знать заранее, какие кейсы вы ожидаете увидеть. Когда вы знаете, с чем работаете, дополнительная безопасность на этапе компиляции является прекрасным бонусом.

2.3. Перечисления вместо создания подклассов

Подклассы позволяют создавать иерархию данных. Например, у вас есть ресторан быстрого питания, где продают гамбургеры, картошку фри, как обычно. Для этого вы создадите суперкласс `FastFood` с такими подклассами, как `Burger`, `Fries` и `Soda`.

Одно из ограничений моделирования вашего программного обеспечения с помощью иерархий (подклассов) состоит в том, что это ограничивает вас в определенном направлении, которое не всегда соответствует вашим потребностям.

Например, вышеупомянутый ресторан получал жалобы от клиентов, которые хотели настоящие японские суши с картофелем фри. Они намереваются размес-

тить клиентов, но их модель подклассов не соответствует этому новому требованию.

В идеальном мире иерархическое моделирование данных имеет смысл. Но на практике вы иногда будете сталкиваться с крайними случаями и исключениями, которые могут не подходить для вашей модели.

В этом разделе мы рассмотрим эти ограничения моделирования данных с помощью создания подклассов в более реальных сценариях и решим их с помощью перечислений.

2.3.1. Формирование модели для приложения Workout

Далее мы создадим модельный уровень для приложения Workout, которое отслеживает пробежки и велосипедные сессии. Приложение включает в себя время начала, время окончания и дистанцию.

Вы создадите структуры `Run` и `Cycle`, обозначающие данные, которые моделируете.

Листинг 2.18. Структура Run

```
import Foundation ❶

struct Run {
   let id: String
   let startTime: Date
   let endTime: Date
   let distance: Float
   let onRunningTrack: Bool
}
```

❶ Необходимо для типа `Date`

Листинг 2.19. Структура Cycle

```
struct Cycle {

   enum CycleType {
      case regular
      case mountainBike
      case racetrack
    }
   let id: String
   let startTime: Date
   let endTime: Date
   let distance: Float
   let incline: Int
   let type: CycleType
}
```

Эти структуры являются хорошей отправной точкой для нашего уровня данных.

Следует признать, что создание отдельной логики в вашем приложении для типов `Run` и `Cycle` может быть обременительным. Давайте попробуем решить это с помощью создания подклассов. Позже вы быстро определите, какие проблемы сопутствуют подклассам, после чего увидите, как можно решить некоторые из этих проблем с помощью перечислений.

2.3.2. Создание суперкласса

Между `Run` и `Cycle` есть много общего, что на первый взгляд делает суперкласс хорошим кандидатом. Преимущество суперкласса заключается в том, что вы можете передавать его, например, в своих методах и массивах. Суперкласс избавляет вас от создания определенных методов и массивов для каждого подкласса тренировки.

Вы можете создать суперкласс под названием `Workout`, а затем превратить `Run` и `Cycle` в классы и сделать их подклассом `Workout`, который наследует от `Workout` (см. рис. 2.3). Иерархически структура создания подклассов имеет большой смысл, потому что у тренировок много значений.

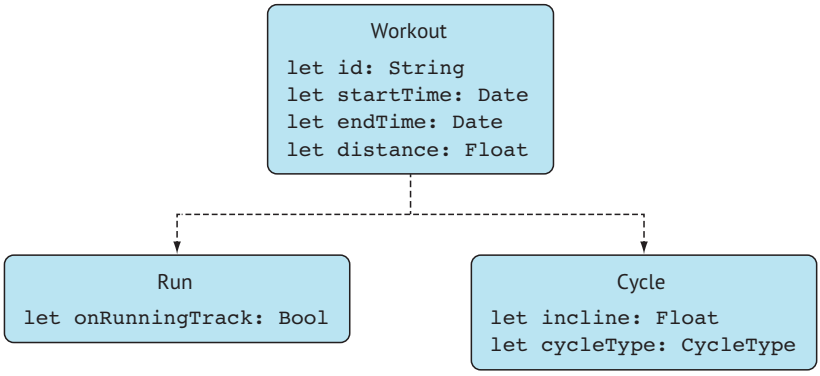


Рис. 2.3. Иерархия подклассов

Новый суперкласс `Workout` содержит свойства, которые совместно используются `Run` и `Cycle`, в частности `id`, `startTime`, `endTime` и `distance`.

2.3.3. Недостатки подклассов

Здесь мы вкратце коснемся проблем, связанных с созданием подклассов. Прежде всего вы вынуждены использовать классы. Классы могут быть удобными, но выбор между классами, структурами или другими перечислениями исчезает, когда вы создаете подклассы.

Однако необходимость использования классов не самая большая проблема. Давайте продемонстрируем еще одно ограничение, добавив новый тип под названием `Pushups`, в котором хранится несколько повторений (repetitions) и дата (date).

Листинг 2.20. Класс Pushups

```
class Pushups: Workout { ❶
   let repetitions: [Int]
   let date: Date
}
```

❶ Создается подкласс

Создание подклассов не работает должным образом, потому что некоторые свойства Workout не применимы к Pushups. Workout необходимы значения startTime, endTime и distance, которые не нужны Pushups.

Чтобы разрешить Pushups создавать подклассы Workout, необходимо провести рефакторинг суперкласса и всех его подклассов. Это можно сделать, переместив startTime, endTime и distance из Workout в классы Cycle и Run, поскольку эти свойства не являются частью класса Pushups (см. рис. 2.4).

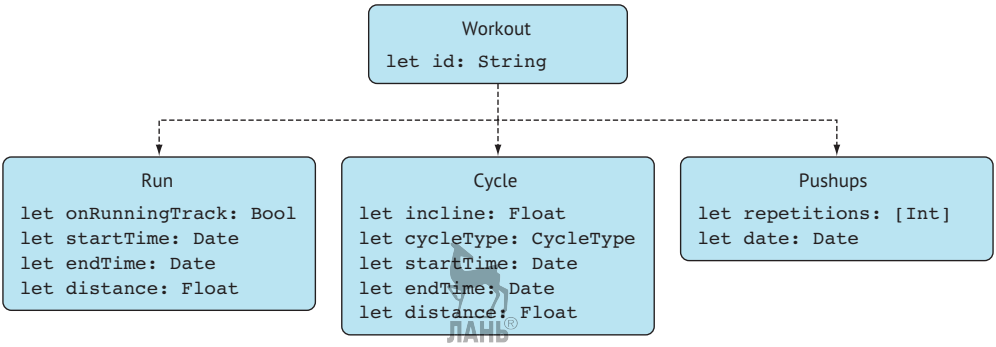


Рис. 2.4. Реорганизованная иерархия подклассов

Рефакторинг всей модели данных показывает проблему при создании подклассов. Как только вы вводите новый подкласс, рискуете реорганизовать суперкласс и все его подклассы, что существенно влияет на существующую архитектуру.

Давайте рассмотрим другой подход с использованием перечислений.

2.3.4. Рефакторинг модели данных с помощью перечислений

Используя перечисления, вы держитесь в стороне от иерархической структуры, но все равно можете сохранить возможность передавать Workout в своем приложении, а также сможете добавлять новые тренировки без необходимости рефакторинга уже существующих.

Это делается путем создания перечисления Workout вместо суперкласса. Внутри этого перечисления могут содержаться различные тренировки.

Листинг 2.21. Workout в качестве перечисления

```
enum Workout {
   case run(Run)
   case cycle(Cycle)
```

```
    case pushups(Pushups)
}
```

Теперь Run, Cycle и Pushups больше не будут делить Workout на подклассы. Фактически тип всех этих тренировок может быть любым, например структура, класс или даже еще одно перечисление.

Можно создать тренировку (workout), передав ее в тренировку Run, Cycle или Pushups. Например, можно преобразовать Pushups в структуру, инициализировать ее и передать в блок pushups внутри перечисления Workout.

Листинг 2.22. Создание тренировки

```
let pushups = Pushups(repetitions: [22,20,10], date: Date())
let workout = Workout.pushups(pushups)
```

Всякий раз, когда нам нужно извлечь тренировку, мы можем использовать сопоставление с образцом.

Листинг 2.23. Сопоставление с образцом

```
switch workout {
   case .run(let run):
      print("Run: \(run)")
   case .cycle(let cycle):
      print("Cycle: \(cycle)")
   case .pushups(let pushups):
      print("Pushups: \(pushups)")
}
```

Преимущество такого решения заключается в том, что вы можете добавлять новые тренировки без рефакторинга существующих. Например, если вы вводите тренировку Abs, то можете добавить ее в Workout, не касаясь Run, Cycle или Pushups.

Листинг 2.24. Добавление новой тренировки в перечисление Workout

```
enum Workout {
   case run(Run)
   case cycle(Cycle)
   case pushups(Pushups)
   case abs(Abs) ❶
}
```

❶ Введена новая тренировка

Отсутствие необходимости рефакторинга других тренировок для добавления новых является значительным преимуществом, и стоит рассмотреть возможность использования перечислений, вместо того чтобы создавать подклассы.

2.3.5. Подклассы или перечисления – что выбрать

Не всегда легко определить, подходят ли перечисления или подклассы вашей модели данных.

Когда типы совместно используют множество свойств и вы прогнозируете, что в будущем они не изменятся, можно продвинуться далеко вперед, используя классическое деление на подклассы. Но создание подклассов ведет к более жесткой иерархии. Кроме того, вы вынуждены использовать классы.

Когда сходные типы начинают расходиться или если вы хотите продолжать использовать перечисления и структуры (в отличие только от классов), создание включающего перечисления предлагает большую гибкость и может быть более предпочтительным вариантом.

Недостатком перечислений является то, что теперь ваш код должен соответствовать всем кейсам в приложении. Хотя может потребоваться дополнительная работа при добавлении новых кейсов, это также и мера предосторожности, когда компилятор гарантирует, что вы не забыли обработать кейс где-то у себя в приложении.

Еще одним недостатком перечислений является то, что на момент написания этих строк перечисления нельзя расширить с помощью новых кейсов. Перечисления ограничивают модель фиксированным числом кейсов, и если вы не владеете кодом, то не сможете изменить эту жесткую структуру. Например, возможно, вы предлагаете перечисление через стороннюю библиотеку, а теперь ее разработчики не могут ее расширить.

Это компромиссы, которые вам придется найти. Если вы можете привязать свою модель данных к фиксированному, управляемому количеству кейсов, перечисления могут стать хорошим вариантом.

2.3.6. Упражнения

1

Назовите два преимущества использования подклассов по сравнению с перечислениями с ассоциированными типами.

2

Назовите два преимущества использования перечислений с ассоциированными типами по сравнению с подклассами.

2.4. Алгебраические типы данных

Перечисления основаны на так называемых *алгебраических типах данных*. Это термин, который берет свое начало из функционального программирования. Алгебраические типы данных обычно выражают составные данные через так называемые типы-суммы и типы-произведения.

Перечисления – это *типы-суммы*; перечисление может быть только чем-то одним одновременно, отсюда и способ мышления «or», описанный ранее.

На другом конце спектра находятся *типы-произведения*, типы, которые содержат несколько значений, таких как кортеж или структура. Можно рассматривать тип-произведение как тип «and». Например, у структуры `User` может быть и имя, и идентификатор. Таким же самым образом у класса `Address` могут быть улица, номер дома и почтовый индекс.

Давайте использовать этот раздел, чтобы охватить немного теории, чтобы вы могли лучше представить себе, что такое перечисления. Затем мы перейдем к практическим примерам, где вы превратите перечисление в структуру, и наоборот.

2.4.1. Типы-суммы

Перечисления – это типы-суммы с фиксированным количеством значений, которые они могут обозначать. Например, приведенное ниже перечисление под названием Day обозначает любой день недели. Есть семь возможных значений, которые оно может обозначать.

Листинг 2.25. Перечисление Day

```
enum Day {
   case sunday
   case monday
   case tuesday
   case wednesday
   case thursday
   case friday
   case saturday
}
```

Чтобы узнать количество возможных значений перечисления, мы *добавляем* (суммируем) возможные значения типов внутри. В случае с перечислением `Day` общая сумма равна семи.

Еще один способ порассуждать о возможных значениях – это тип `UInt8`. В диапазоне от 0 до 255 общее число возможных значений составляет 256. Это не смоделировано именно так, но вы можете рассматривать `UInt8as` как перечисление с 256 кейсами.

Если бы вам нужно было написать перечисление с двумя кейсами и вы добавили `UInt8` в один из кейсов, возможные варианты этого перечисления увеличились бы с 2 до 257.

Например, у вас может быть перечисление `Age` (обозначающее чей-либо возраст), где возраст может быть неизвестным. Но если он известен, то содержит `UInt8`.

Листинг 2.26. Перечисление Age

```
enum Age {
   case known(UInt8)
   case unknown
}
```


Age теперь представляет 257 возможных значений, а именно кейс unknown (1) + кейс known (256).

2.4.2. Типы-произведения

На другом конце спектра находятся типы-произведения. Тип-произведение умножает возможные значения, которые он содержит. Например, если вам нужно хранить два логических типа данных внутри структуры, общее число вариаций является произведением (умножением) этих двух перечислений.

Листинг 2.27. Структура, содержащая два логических типа

```
struct BooleanContainer {
   let first: Bool
   let second: Bool
}
```

Первый логический тип (два возможных значения), *помноженный* на второй логический тип (два возможных значения), – это четыре возможных состояния, которые могут быть у этой структуры.

В коде это можно проверить, раскрыв все варианты.

Листинг 2.28. У BooleanContainer имеется четыре возможных варианта

```
BooleanContainer(first: true, second: true)
BooleanContainer(first: true, second: false)
BooleanContainer(first: false, second: true)
BooleanContainer(first: false, second: false)
```

Когда вы моделируете данные, следует помнить о количестве вариантов. Чем выше число возможных значений, которые имеют тип, тем сложнее рассуждать о возможных состояниях типа.

При наличии структуры с 1000 строк для свойств возможных состояний гораздо больше, нежели у структуры с одним логическим свойством.

2.4.3. Распределение суммы в перечислении

Я также не буду фокусироваться только на теории относительно типов-сумм и типов-произведений. Вы здесь не для того, чтобы писать сухую теоретическую дипломную работу, а для того, чтобы создавать прекрасное.

Представьте, что у вас есть перечисление PaymentType, содержащее три кейса, обозначающих три способа, которыми может расплатиться покупатель.

Листинг 2.29. Перечисление PaymentType

```
enum PaymentType {
   case invoice
   case creditcard
   case cash
}
```

Далее вы обозначите статус платежа. Структура – подходящий кандидат для хранения вспомогательных свойств, помимо перечисления PaymentType, например когда платеж завершен и касается ли это регулярного платежа.

Листинг 2.30. Структура PaymentStatus

```
struct PaymentStatus {
   let paymentDate: Date?
   let isRecurring: Bool
   let paymentType: PaymentType
}
```

Произведением всех вариантов будут все возможные даты, *умноженные* на 2 (логический тип данных) и *умноженные* на 3 (перечисление с тремя кейсами). У вас будет большое количество вариантов, потому что структура может хранить много вариантов дат.

Подобно сливочному сыру на бублике, вы размазываете свойства структуры по кейсам в перечислении, следуя правилам алгебраических типов данных (см. рис. 2.5).

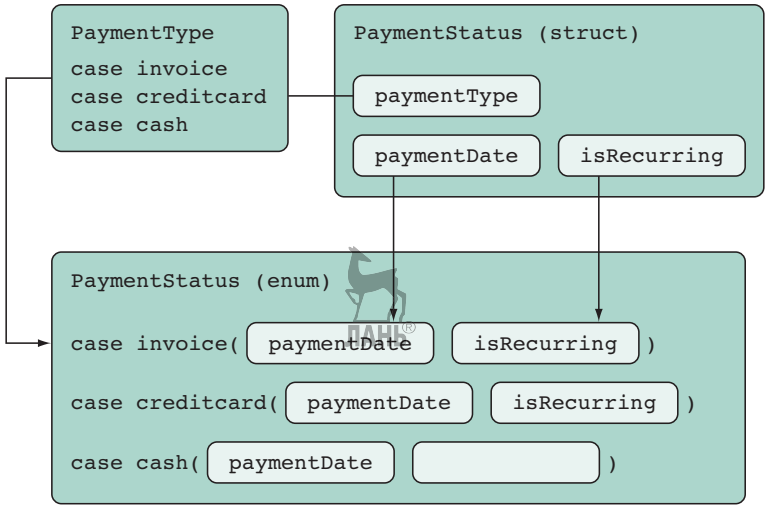


Рис. 2.5. Превращение структуры в перечисление

Вы получите перечисление с таким же именем, что и у структуры. Каждый кейс обозначает кейсы исходного перечисления со свойствами структуры внутри.

Листинг 2.31. Перечисление PaymentStatus, содержащее кейсы

```
enum PaymentStatus {
   case invoice(paymentDate: Date?, isRecurring: Bool)
   case creditcard(paymentDate: Date?, isRecurring: Bool)
   case cash(paymentDate: Date?, isRecurring: Bool)
}
```

Вся информация все еще там, и количество возможных вариаций остается прежним. За исключением того, что на этот раз вы вывернули типы наизнанку!

Выгода в том, что вы имеете дело только с одним типом. Для этого требуется повторение внутри каждого кейса. Тут нет таких понятий, как «правильно» или «неправильно». Это просто иной подход к моделированию одних и тех же данных. При этом то же количество возможных вариантов остается без изменений. Это хитрый трюк, который показывает алгебраическую природу типов и помогает моделировать перечисления несколькими способами. В зависимости от ваших потребностей перечисление может быть подходящей альтернативой для структуры, содержащей перечисление, или наоборот.

2.4.4. Упражнение

Имеется следующая структура данных:

```
enum Topping {
   case creamCheese
   case peanutButter
   case jam
}

enum BagelType {
   case cinnamonRaisin
   case glutenFree
   case oatMeal
   case blueberry
}

struct Bagel {
   let topping: Topping
   let type: BagelType
}
```

3

Каково количество возможных вариаций Bagel?

4

Превратите `Bagel` в перечисление, сохраняя то же количество возможных вариаций.

5

Имеется перечисление, представляющее игру-головоломку для определенного возрастного диапазона (например, младенец, ребенок, который учится ходить, или подросток), которое содержит фрагменты головоломки:

```
enum Puzzle {
   case baby(numberOfPieces: Int)
   case toddler(numberOfPieces: Int)
   case preschooler(numberOfPieces: Int)
   case gradeschooler(numberOfPieces: Int)
   case teenager(numberOfPieces: Int)
}
```

Как это перечисление будет представлено в виде структуры?

2.5. Более безопасное использование строк

Работа со строками и перечислениями довольно распространена. Давайте продолжим и уделим им дополнительное внимание, чтобы вы все делали правильно. В этом разделе пойдет речь об опасностях при работе с перечислениями, которые содержат необработанное значение типа `String`.

Когда перечисление определяется как тип необработанного значения, все кейсы в этом перечислении несут в себе какое-то значение.

Перечисления с необработанными значениями определяются добавлением типа в объявление перечисления.

Листинг 2.32. Перечисление с необработанными и строковыми значениями

```
enum Currency: String { ❶
   case euro = "euro" ❷
   case usd = "usd"
   case gbp = "gbp"
}
```

❶ String – это тип необработанного значения.

❷ Все кейсы содержат строковые значения.

Необработанные значения, которые могут храниться в перечислении, зарезервированы только для таких типов, как `String` (строка), `Chracter` (символьный тип), целое число и число с плавающей точкой.

Перечисление с необработанными значениями означает, что у каждого кейса есть значение, определенное на этапе компиляции. С другой стороны, перечисле-

ния с ассоциированными типами, которые вы использовали в предыдущих разделах, хранят свои значения во время выполнения.

При создании перечисления с необработанным типом `String` каждое необработанное значение принимает имя кейса. Не нужно добавлять строковое значение, если необработанное значение совпадает с именем кейса, как показано здесь.

Листинг 2.33. Перечисление с необработанными значениями, строковые значения опущены

```
enum Currency: String {
   case euro
   case usd
   case gbp
}
```

Поскольку перечисление по-прежнему имеет тип необработанных значений, например `String`, в каждом кейсе все еще содержатся необработанные значения.

2.5.1. Опасность необработанных значений

При работе с необработанными значениями следует проявлять осторожность, потому что, прочитав необработанные значения перечисления, вы лишитесь помощи компилятора.

Например, вы хотите установить параметры для гипотетического API-вызова. Эти параметры используются для запроса транзакций в той валюте, которую вы указали.

Вы будете использовать перечисление `Currency` для создания параметров вашего API-вызова. Вы можете прочитать необработанное значение перечисления, обратившись к свойству необработанного значения, и таким образом настроить параметры API.

Листинг 2.34. Установка необработанного значения внутри параметров

```
let currency = Currency.euro
print(currency.rawValue) // "euro"

let parameters = ["filter": currency.rawValue]
print(parameters) // ["filter": "euro"]
```

Чтобы внести ошибку, измените необработанное значение для кейса `euro` с «euro» на «eur» (опустив букву «o»), поскольку *eur* – это текущее обозначение валюты в евро.

Листинг 2.35. Переименование строки

```
enum Currency: String {
   case euro = "eur"
   case usd
```

```
        case gbp
}
```

Поскольку API-вызов полагался на необработанное значение, чтобы создать параметры, эти параметры теперь влияют на API-вызов.

Компилятор не уведомит вас об этом, потому что необработанное значение по-прежнему является допустимым кодом.

Листинг 2.36. Неожиданные параметры

```
let parameters = ["filter": currency.rawValue]
// Ожидается "euro", а получаем "eur"
print(parameters) // ["filter": "eur"]
```

Все еще компилируется. К сожалению, вы молча внесли ошибку в часть своего приложения.

Всегда обновляйте строку везде, где это может показаться очевидным. Но представьте, что вы работаете над большим проектом, в котором это перечисление было создано в совершенно другой части приложения или, возможно, было предложено из фреймворка. Безобидное изменение перечисления может принести вред в другом месте вашего приложения. Эти проблемы могут подкрасться к вам, и их легко пропустить, потому что вы не получаете уведомления на этапе компиляции.

Можно рискнуть и проигнорировать необработанные значения перечисления и использовать сопоставление. Как показано в приведенном ниже коде, когда вы устанавливаете параметры таким образом, то будете знать на этапе компиляции, когда меняется кейс.

Листинг 2.37. Явные необработанные значения

```
let parameters: [String: String]

switch currency {
   case .euro: parameters = ["filter": "euro"]
   case .usd: parameters = ["filter": "usd"]
   case .gbp: parameters = ["filter": "gbp"]
}

// Снова возвращаемся к использованию "euro"
print(parameters) // ["filter": "euro"]
```

Вы воссоздаете строки и игнорируете необработанные значения перечисления. Такой код может быть избыточным, но, по крайней мере, у вас будут именно те значения, которые вам нужны. Любые изменения в необработанных значениях не застанут вас врасплох, потому что теперь вам поможет компилятор. Вы можете даже полностью исключить необработанные значения, если ваше приложение это позволяет.

Возможно, так даже лучше, если вы все же используете необработанные значения, а при написании модульных тестов вы можете убедиться, что ничего не сломалось, и тем самым обезопасить себя. Таким образом вы получите своего рода страховочную сетку и преимущества использования необработанных ценностей.

Все это компромиссы, которые вам придется найти. Но полезно знать, что вы лишаетесь помощи компилятора, как только начинаете использовать необработанные значения из перечисления.

2.5.2. Сопоставление для строк

Всякий раз при использовании сопоставления с образцом для типа String (строка) вы открываете дверь для пропущенных кейсов. В этом разделе рассматриваются недостатки такого сопоставления и показано, как сделать из этого перечисление, чтобы дополнительно обезопасить себя.

В следующем примере мы будем моделировать пользовательскую систему управления изображениями, в которой клиенты могут хранить и группировать свои любимые фотографии, изображения и картинки. В зависимости от типа файла вам нужно знать, показывать или нет конкретный значок, указывающий на то, что это jpeg-файл, растровое изображение, gif или неизвестный тип.

В реальном приложении вы также проверяете реальные метаданные изображения, но при быстром и неаккуратном подходе вы будете смотреть только на расширение.

Функция `iconName` дает вашему приложению имя иконки, которое будет отображаться поверх изображения в зависимости от расширения файла. Например, у изображения в формате JPEG есть маленький значок. Имя этого значка – `assetIconJpeg`.

Листинг 2.38. Сопоставление для строк

```
func iconName(for fileExtension: String) -> String {
   switch fileExtension {
      case "jpg": return "assetIconJpeg"
      case "bmp": return "assetIconBitmap"
      case "gif": return "assetIconGif"
      default: return "assetIconUnknown"
   }
}

iconName(for: "jpg") // "assetIconJpeg"
```

Сопоставление работает, но при таком подходе возникают проблемы (в отличие от перечислений). Сделать опечатку легко, а следовательно, могут возникнуть неожиданности – например, когда вы будете ожидать формат «jpg», а получать «jpeg» или «JPG» из внешнего источника.

Функция возвращает неизвестный значок, как только вы отклонитесь совсем немного – например, передавая прописную строку.

Листинг 2.39. Неизвестная иконка

```
iconName(for: «JPG») // «assetIconUnknown», not favorable
```

Конечно, перечисление не решает сразу все проблемы, но если вы неоднократно выполняете сопоставление для одной и той же строки, шансы на опечатки возрастают.

Кроме того, если при сопоставлении возникают какие-либо ошибки, вы узнаете об этом во время выполнения. Однако в случае с перечислениями сопоставление является исчерпывающим. Если бы мы работали с перечислениями, то узнали бы об ошибках (например, забыли обработать кейс) на этапе компиляции.

Давайте создадим перечисление! Это можно сделать, используя перечисление с необработанным типом `String`.

Листинг 2.40. Создание перечисления с необработанным значением String

```
enum ImageType: String { ❶
   case jpg
   case bmp
   case gif
}
```

❶ Вводим перечисление

На этот раз при сопоставлении в функции `iconName` мы сначала превращаем строку в перечисление, передавая необработанное значение. Таким образом мы узнаем, добавляет ли `ImageType` еще один кейс. Компилятор сообщит нам, что `iconName` необходимо обновить, и обработает новый кейс.

Листинг 2.41. iconName создает перечисление

```
func iconName(for fileExtension: String) -> String {
   guard let imageType = ImageType(rawValue: fileExtension) else {
      return "assetIconUnknown" ❶
   }

   switch imageType { ❷
      case .jpg: return "assetIconJpeg"
      case .bmp: return "assetIconBitmap"
      case .gif: return "assetIconGif"
   }
}
```

❶ Функция пытается преобразовать строку в `ImageType`. Если ей это не удается, она возвращает «`assetIconUnknown`».

❷ `iconName` теперь использует сопоставление для перечисления, предоставляя преимущества компилятора, если мы пропустили кейс.

Однако мы до сих пор не решили проблему немного отличающихся значений, таких как «jpeg» или «JPEG». Если бы мы написали «jpg» прописными буквами, функция iconName вернула бы это: «assetIconUnknown».

Давайте позаботимся об этом сейчас, используя сопоставление для нескольких строк одновременно. Мы можем реализовать свой инициализатор, который принимает строку необработанного значения.

Листинг 2.42. Добавление пользовательского инициализатора в ImageType

```
enum ImageType: String {
   case jpg
   case bmp
   case gif

   init?(rawValue: String) {
      switch rawValue.lowercased() { ❶
         case "jpg", "jpeg": self = .jpg ❷
         case "bmp", "bitmap": self = .bmp
         case "gif", "gifv": self = .gif
         default: return nil
      }
   }
}
```

❶ Сопоставление теперь нечувствительно к регистру, что делает его более снисходительным.

❷ iconName сейчас использует сопоставление для нескольких строк одновременно, например "jpg" или "jpeg".

Опциональный инициализатор?

Инициализатор из ImageType возвращает опционал. Опциональный инициализатор указывает на то, что может произойти сбой. Когда инициализатор все же терпит неудачу, – когда вы даете ему непригодную строку, – он возвращает нулевое значение. Не волнуйтесь, если вам все еще не понятно. Мы будем подробно разбирать опционалы в главе 4.

Обратите здесь внимание на пару моментов. Мы устанавливаем кейс ImageType в зависимости от переданного необработанного значения (rawValue), но не перед тем, как превратить его в строку с нижним регистром, чтобы сделать сопоставление с образцом, нечувствительным к регистру. После этого мы задаем каждому кейсу несколько вариантов для сопоставления, например case «jpg», «jpeg», чтобы у него была возможность перехватывать больше кейсов. Можно было бы написать это, используя больше кейсов, но это простой способ сгруппировать сопоставление с образцом.

Теперь сопоставление строк стало более надежным, и мы можем использовать сопоставление для вариантов строк.

Листинг 2.43. Передача строк

```
iconName(for: "jpg") // "Received jpg"
iconName(for: "jpeg") // "Received jpg"
iconName(for: "JPG") // "Received a jpg"
iconName(for: "JPEG") // "Received a jpg"
iconName(for: "gif") // "Received a gif"
```

Если при преобразовании все же возникает ошибка, можно написать для нее контрольный пример и исправить перечисление только в одном месте, вместо того чтобы исправлять множественные сопоставления, разбросанные по всему приложению.

Работа со строками таким способом теперь более идиоматична. Код стал более безопасным и выразительным. Компромисс заключается в необходимости создания нового перечисления, которое может быть избыточным, если мы используем сопоставление с образцом только один раз.

Но когда вы снова и снова видите сопоставление для строки, преобразование ее в перечисление – неплохой вариант.

2.5.3. Упражнения

6

Какие необработанные типы поддерживаются перечислениями?

7

Необработанные значения перечисления устанавливаются на этапе компиляции или во время выполнения?

8

Ассоциированные значения перечисления устанавливаются на этапе компиляции или во время выполнения?

9

Какие типы могут входить в ассоциированное значение?

2.6. В заключение

Как вы убедились, перечисления – это больше, чем просто список значений. Как только вы начнете «думать перечислениями», получите взамен безопасность и надежность и сможете превращать структуры в перечисления и обратно.

Я надеюсь, что эта глава вдохновила вас применять перечисления на удивление весело и с пользой. Возможно, вы будете использовать перечисления чаще, чтобы объединять их со структурами и классами или менять на них.

В следующий раз в качестве домашнего проекта посмотрите, как далеко вы можете продвинуться, используя только перечисления и структуры. Ограничение себя перечислениями и структурами – это отличная тренировка, которая поможет вам мыслить типами-суммами и типами-произведениями.

Резюме

- Иногда перечисления являются альтернативой подклассам, что позволяет использовать гибкую архитектуру.
- Перечисления дают возможность перехватывать проблемы на этапе компиляции, а не во время выполнения.
- Можно использовать перечисления для группировки свойств.
- Перечисления иногда называют типами-суммами, основанными на алгебраических типах данных.
- Структуры можно распределять по перечислениям.
- Работая с необработанными значениями перечисления, вы избегаете проблем на этапе компиляции.
- Обработку строк можно сделать более безопасной путем преобразования их в перечисления.
- При преобразовании строки в перечисление группировка падежей и использование строки в нижнем регистре упрощают преобразование.

Ответы

1

Назовите два преимущества использования подклассов по сравнению с перечислениями с ассоциированными типами.

Суперкласс предотвращает дублирование. Не нужно объявлять одно и то же свойство дважды. При создании подклассов также можно переопределить существующую функциональность.

2

Назовите два преимущества использования перечислений с ассоциированными типами по сравнению с подклассами.

Не нужно ничего реорганизовывать, если вы добавляете еще один тип, тогда как при создании подклассов вы рискуете реорганизовать суперкласс и его подклассы. Во-вторых, вы не обязаны использовать классы.

3

Имеется структура данных. Каково количество возможных вариаций `Bagel`?

Двенадцать (3 начинки на 4 типа бублика).

4

Имеется структура данных. Превратите `Bagel` в перечисление, сохраняя при этом такое же количество возможных вариаций.

Два способа, потому что Bagel содержит два перечисления. Вы можете хранить данные в любом перечислении:

```
// Используем перечисление Topping в качестве кейсов.
enum Bagel {
   case creamCheese(BagelType)
   case peanutButter(BagelType)
   case jam(BagelType)
}

//С другой стороны используем перечисление BagelType в качестве кейсов.
enum Bagel {
   case cinnamonRaisin(Topping)
   case glutenFree(Topping)
   case oatMeal(Topping)
   case blueberry(Topping)
}
```

5

Имеется перечисление, представляющее игру-головоломку для определенного возрастного диапазона. Как это перечисление можно представить в виде структуры?

```
enum AgeRange {
   case baby
   case toddler
   case preschooler
   case gradeschooler
   case teenager
}

struct Puzzle {
   let ageRange: AgeRange
   let numberOfPieces: Int
}
```

6

Какие необработанные типы поддерживаются перечислениями?

Строковые, символьные, целочисленные и с плавающей точкой.

7

Необработанные значения перечисления устанавливаются на этапе компиляции или во время выполнения?

Необработанные значения типа устанавливаются на этапе компиляции.

8

Ассоциированные значения перечисления устанавливаются на этапе компиляции или во время выполнения?

Ассоциированные значения устанавливаются во время выполнения.

9

Какие типы могут входить в ассоциированное значение?

Все типы.

Глава 3. Написание более чистых свойств

В этой главе:

- как создавать вычисляемые свойства, использующие блоки get и set;
- когда (не нужно) использовать вычисляемые свойства;
- повышение производительности с помощью ленивых свойств;
- как ленивые свойства ведут себя со структурами и изменяемостью;
- обработка хранимых свойств с помощью поведения.

Чистое использование свойств может полностью упростить интерфейс ваших структур, классов и перечислений, делая ваш код более безопасным и легким для чтения и использования для других (и для вашего будущего Я). Поскольку свойства являются основной частью Swift, следование указателям в этой главе может помочь сразу же прочитать ваш код.

Сначала мы рассмотрим *вычисляемые* свойства, которые представляют собой функции, выглядящие как свойства. Вычисляемые свойства могут очистить интерфейс ваших структур и классов. Вы увидите, как и когда их создавать, а также когда их лучше избегать.

Затем мы исследуем *ленивые* свойства, которые можно инициализировать позже или не инициализировать вовсе. Ленивые свойства удобны по некоторым причинам, например когда вы хотите оптимизировать дорогостоящие вычисления. Вы также увидите, что ленивые свойства в структурах и классах ведут себя по-разному.

Кроме того, вы увидите, как можно запускать пользовательское поведение для свойств с помощью так называемых наблюдателей свойств, а также правил и особенностей, которые с ними связаны.

К концу этой главы вы сможете сделать правильный выбор между ленивыми, хранимыми и вычисляемыми свойствами, чтобы иметь возможность создавать чистые интерфейсы для своих структур и классов.

3.1. Вычисляемые свойства

Вычисляемые свойства – это функции, *маскирующиеся* под свойства. Они не хранят никаких значений, но снаружи выглядят так же, как хранимые свойства.

Например, у вас есть таймер обратного отсчета в приложении для приготовления пищи, и вам нужно свериться с ним, чтобы проверить, варятся ли ваши яйца, как показано в следующем листинге.

Листинг 3.1. Таймер обратного отсчета

```
cookingTimer.secondsRemaining // 411
```

Однако это значение изменяется со временем, как показано ниже.

Листинг 3.2. Значение изменяется со временем с вычисляемыми свойствами

```
cookingTimer.secondsRemaining // 409

// Немного подождем
cookingTimer.secondsRemaining // 404
// Немного подождем
cookingTimer.secondsRemaining // 392
```

Скажем по секрету, что это свойство является функцией, потому что значение постоянно меняется.

В оставшейся части этого раздела мы рассмотрим преимущества вычисляемых свойств. Вы увидите, как они могут очистить интерфейс ваших типов, а также узнаете, как вычисляемые свойства выполняют код при каждом обращении к свойству, придавая вашим свойствам динамический характер «всегда в курсе».

В разделе 3.2 будет показано, когда *не* следует использовать вычисляемые свойства и что иногда лучше отдать предпочтение ленивым свойствам.

3.1.1. Моделирование упражнения

Давайте рассмотрим беговое упражнение для нашего приложения. Пользователи смогут начинать пробежку и отслеживать текущий прогресс с течением времени, например прошедшее время в секундах.

Присоединяйтесь!

Будет более познавательно и веселее, если вы будете знакомиться с кодом по мере прохождения этой главы. Исходный код можно скачать по адресу `http://mng.bz/5N4q`.

Тип, обозначающий упражнение, называется `Run`. Он получает выгоду от использования вычисляемых свойств. У нас есть функция `elapsedTime()`, которая показывает, сколько времени прошло с момента начала пробежки. После ее завершения можно вызвать функцию `setFinished()`, чтобы завершить упражнение. Мы проверяем, что упражнение завершено, удостоверившись, что функция `isFinished()` возвращает значение `true`. Жизненный цикл пробежки показан в следующем примере.

Листинг 3.3. Жизненный цикл пробежки

```
var run = Run(id: "10", startTime: Date())
// Проверяем пройденное время.
print(run.elapsedTime()) // 0.00532001256942749
// Снова проверяем пройденное время.
print(run.elapsedTime()) // 14.00762099027634
run.setFinished() // Завершаем упражнение.
```

```
print(run.isFinished()) // Истинно
```

После введения Run вы увидите, как очистить его, преобразовав его функцию в вычисляемые свойства.

3.1.2. Преобразование функций в вычисляемые свойства

Давайте посмотрим на тип Run в приведенном ниже листинге, чтобы увидеть, как он составлен. После этого вы определите, как его реорганизовать, чтобы он использовал вычисляемые свойства.

Листинг 3.4. Структура Run

```
import Foundation

struct Run {
   let id: String
   let startTime: Date
   var endTime: Date?

   func elapsedTime() -> TimeInterval { ❶
      return Date().timeIntervalSince(startTime)
   }

   func isFinished() -> Bool { ❷
      return endTime != nil
   }

   mutating func setFinished() {
      endTime = Date()
   }

   init(id: String, startTime: Date) {
      self.id = id
      self.startTime = startTime
      self.endTime = nil
   }
}
```

❶ elapsedTime вычисляет продолжительность упражнения, измеренную в секундах.

❷ Проверяем, равен ли endTime нулю, чтобы определить, завершено ли упражнение.

Посмотрев на сигнатуры функций elapsedTime() и isFinished(), показанных в следующем листинге, вы увидите, что эти функции не принимают параметр, а возвращают значение.

Листинг 3.5. Кандидаты на вычисляемые свойства

```
func isFinished() -> Bool { ... }
func elapsedTime() -> TimeInterval { ... }
```

Отсутствие параметров и тот факт, что функции возвращают значение, показывают, что эти функции являются кандидатами на вычисляемые свойства.

Далее давайте преобразуем `isFinished()` и `elapsedTime()` в вычисляемые свойства.

Листинг 3.6. Запустить с вычисленными свойствами

```
struct Run {
   let id: String
   let startTime: Date
   var endTime: Date?

   var elapsedTime: TimeInterval { ❶
      return Date().timeIntervalSince(startTime)
   }

   var isFinished: Bool { ❷
      return endTime != nil
   }

   mutating func setFinished() {
      endTime = Date()
   }

   init(id: String, startTime: Date) {
      self.id = id
      self.startTime = startTime
      self.endTime = nil
   }
}
```

❶ `elapsedTime` теперь является вычисляемым свойством.

❷ `isFinished` также является вычисляемым свойством.

Вы преобразовали функции в свойства, предоставив свойству замыкание, которое возвращает значение.

На этот раз, когда вы захотите прочитать значение, можете вызвать вычисляемые свойства, как показано в следующем примере.

Листинг 3.7. Вызов вычисляемых свойств

```
var run = Run(id: "10", startTime: Date())
print(run.elapsedTime) // 30.00532001256942749
```

```
print(run.elapsedTime) // 34.00822001332744283
print(run.isFinished) // Ложно
```

Вычисляемые свойства могут динамически изменять значение, и таким образом они *рассчитываются*. В этом их отличие от хранимых свойств, которые являются фиксированными и их нужно явно обновлять.

В качестве проверки перед доставкой структуры или класса выполните быстрое сканирование, чтобы выяснить, можете ли вы преобразовать какие-либо функции в вычисляемые свойства для дополнительной удобочитаемости.

3.1.3. Завершение

Превратив функции в свойства, можно очистить интерфейс структуры или класса. Для тех, кто не в курсе, некоторые значения лучше общаются, будучи свойствами, чтобы передавать свои намерения. Вашим коллегам, например, не всегда нужно знать, что некоторые свойства втайне могут быть функцией.

Преобразование легковесных функций в свойства скрывает детали реализации для пользователей вашего типа, что является небольшим преимуществом, но может проявить себя в более крупных кодовых базах.

Давайте продолжим, и вы увидите, что вычисляемые свойства не всегда могут быть лучшим вариантом и как тут могут помочь ленивые свойства.

3.2. Ленивые свойства

Вычисляемые свойства не всегда являются лучшим вариантом, чтобы придать своим типам динамический характер. Когда вычисляемое свойство становится дорогим с точки зрения вычислений, приходится прибегать к другим средствам, таким как ленивые свойства.

Говоря кратко, ленивые свойства гарантируют, что вычисление происходит позднее (если вообще происходит) и только один раз.

В этом разделе вы узнаете, как ленивые свойства могут помочь, когда вы имеете дело с затратными вычислениями или когда пользователю приходится долго ждать.

3.2.1. Создание учебного плана

Давайте рассмотрим пример, когда вычисляемые свойства не всегда являются лучшим вариантом, и посмотрим, как лучше использовать для этого ленивые свойства.

Представьте, что вы создаете API, который может служить наставником для изучения новых языков. Используя причудливый алгоритм, он предоставляет клиентам индивидуальный распорядок дня для изучения языка. `LearningPlan` будет строить свое расписание на основе уровня каждого клиента, его родного языка и языка, который клиент хочет выучить. Чем выше уровень, тем больше

усилий требуется от клиента. Для простоты мы сосредоточимся на учебном плане, который составляет планы, связанные с изучением английского языка.

В этом разделе мы не будем реализовывать реальный алгоритм, но важно отметить, что на обработку этого гипотетического алгоритма уходит несколько секунд, что заставляет клиентов ждать. Несколько секунд может показаться не таким уж большим промежутком времени для одного отдельного случая, но это не подходит для создания множества расписаний, особенно если ожидается, что эти расписания будут быстро загружаться на мобильном устройстве.

Поскольку этот алгоритм затратен, вызывать его можно только один раз, когда это необходимо.

3.2.2. Когда вычисляемые свойства не помогают

Давайте выясним, почему вычисляемые свойства не лучший вариант для затратных вычислений.

Можно инициализировать `LearningPlan` с описанием и параметром уровня, как показано в приведенном ниже листинге. С помощью свойства `contents` можно прочитать план, который генерирует причудливый алгоритм.

Листинг 3.8. Создание LearningPlan

```
var plan = LearningPlan(level: 18, description: "A special plan for today!")

print(Date()) // 2018-09-30 18:04:43 +0000
print(plan.contents) // "Просмотр документального фильма на английском языке"
print(Date()) // 2018-09-30 18:04:45 +0000
```

Обратите внимание, что для вызова `contents` требуется две секунды, что объясняется вашим дорогостоящим алгоритмом.

Теперь давайте взглянем на структуру `LearningPlan`. Видно, что она создана с использованием вычисляемых свойств. Это наивный подход, который мы улучшим с помощью ленивого свойства.

Листинг 3.9. Структура LearningPlan

```
struct LearningPlan {

   let level: Int

   var description: String

   // contents - это вычисляемое свойство.
   var contents: String { ❶
      //В данном случае смоделирован умный алгоритм вычислений.
      print(«I'm taking my sweet time to calculate.»)
      sleep(2) ❷

      switch level { ❸
         case ..<25: return «Watch an English documentary.»
```

```
            case ..<50: return "Translate a newspaper article to English and
               transcribe one song."
            case 100...: return "Read two academic papers and translate them
            into your native language."
            default: return "Try to read English for 30 minutes."
        }
    }
}
```

❶ Contents – это то, где алгоритм вычисляет индивидуальный план.

❷ Имитация двухсекундной задержки для алгоритма.

❸ После выполнения расчетов вы возвращаете индивидуальный план обучения для клиента в зависимости от уровня.

Сопоставление с образцом

Можно использовать сопоставление с образцом для так называемых *односторонних диапазонов*, что означает отсутствие начального диапазона. Например, .. <25 означает все, что ниже 25, а 100 ... означает «100 и выше».

Обратите внимание, что каждый раз при вызове contents для его вычисления требуется две секунды.

Как показано далее, если вы вызываете его только пять раз (например, в цикле, показанном в следующем листинге), приложение будет работать десять секунд – не очень быстро!

Листинг 3.10. Вызываем contents пять раз

```
var plan = LearningPlan(level: 18, description: "A special plan for today!")

print(Date()) // Маркер начала
for _ in 0..<5 {
   plan.contents
}
print(Date()) // Маркер завершения
```

Будет выведен следующий список.

Листинг 3.11. Занимает десять секунд

```
2018-10-01 06:39:37 +0000
I'm taking my sweet time to calculate.
I'm taking my sweet time to calculate.
I'm taking my sweet time to calculate.
I'm taking my sweet time to calculate.
I'm taking my sweet time to calculate.
2018-10-01 06:39:47 +0000
```

Обратите внимание, что маркеры начальной и конечной дат находятся на расстоянии десяти секунд друг от друга. Поскольку вычисляемое свойство запускается всякий раз при вызове, использование вычисляемых свойств для затратных операций крайне не рекомендуется.

Далее мы заменим вычисляемое свойство ленивым свойством, чтобы сделать свой код более эффективным.

3.2.3. Использование ленивых свойств

Можно использовать ленивые свойства, чтобы гарантировать, что затратное с точки зрения вычислений или медленное свойство выполняется только один раз, и только когда вы его вызываете (если такое вообще случается). Давайте заменим вычисляемое свойство на ленивое в следующем листинге.

Листинг 3.12. LearningPlan и ленивое свойство

```
struct LearningPlan {

   let level: Int

   var description: String

   lazy var contents: String = { ❶
      // В данном случае смоделирован умный алгоритм вычислений.
      print("I'm taking my sweet time to calculate.")
      sleep(2)
      switch level {
         case ..<25: return "Watch an English documentary."
         case ..<50: return "Translate a newspaper article to English and
            transcribe one song."
         case 100...: return "Read two academic papers and translate them
            into your native language."
         default: return "Try to read English for 30 minutes."
      }
   }() ❷
}
```

❶ Теперь `contents` – это ленивое свойство.

❷ Замыканию свойства теперь нужны скобки «`()`».

Обращение к свойствам

Можно обращаться к другим свойствам из замыкания ленивого свойства. Обратите внимание, что `contents` может обращаться к свойству `level`.

Тем не менее этого недостаточно; ленивое свойство считается обычным свойством, и компилятор хочет, чтобы оно было инициализировано. Можно обойти

это, добавив пользовательский инициализатор, как показано в следующем листинге, где мы исключаем contents, тем самым удовлетворив компилятор.

Листинг 3.13. Добавление пользовательского инициализатора

```
struct LearningPlan {
   // ... Пропускаем часть кода
   init(level: Int, description: String) { // здесь нет contents!
      self.level = level
      self.description = description
   }
}
```

Все снова компилируется. Более того, самое приятное состоит в том, что при повторном вызове contents дорогостоящий алгоритм используется только один раз! Как только значение будет вычислено, оно сохраняется. Обратите внимание, что в следующем листинге, хотя к contents обращаются пять раз, ленивое свойство инициализируется только однажды.

Листинг 3.14. contents загружается только один раз

```
print(Date())
for _ in 0..<5 {
   plan.contents
}
print(Date())

// Будет выведено:
2018-10-01 06:43:24 +0000
I'm taking my sweet time to calculate.
2018-10-01 06:43:26 +0000
```

При первом обращении чтение занимает две секунды, но во второй раз contents возвращается немедленно. Общее время, потраченное алгоритмом, теперь составляет две секунды вместо десяти!

3.2.4. Делаем ленивое свойство устойчивым

Само по себе ленивое свойство не особенно надежное. Его можно легко сломать. Обратите внимание на следующий сценарий, в котором мы устанавливаем свойство contents из LearningPlan для использования в менее предпочтительном варианте.

Листинг 3.15. Переопределение содержимого LearningPlan

```
var plan = LearningPlan(level: 18, description: "A special plan for today!")
plan.contents = "Let's eat pizza and watch Netflix all day."
print(plan.contents) // " Давайте весь день есть пиццу и смотреть Netflix."
```

Как видно, алгоритм можно обойти, установив ленивое свойство, что делает его несколько хрупким.

Но не волнуйтесь – уровень доступа к свойствам можно ограничить. Добавляя ключевое слово `private(set)` к свойству, как показано в следующем листинге, можно указать, что ваше свойство доступно для чтения, но может быть установлено (изменено) только его владельцем, которым является `LearningPlan`.

Листинг 3.16. Делаем contents свойством private(set)

```
struct LearningPlan {
   lazy private(set) var contents: String = {
   // ... Пропускаем часть кода.
}
```

Теперь свойство нельзя изменить за пределами структуры. Для внешнего мира `contents` доступно только для чтения. Ошибка, выданная компилятором, подтверждает это.

Листинг 3.17. Нельзя установить свойство contents

```
error: cannot assign to property: 'contents' setter is inaccessible
plan.contents = «Let's eat pizza and watch Netflix all day.»
~~~~~~~~~~~~~ ^
```

Таким образом, вы можете предоставить доступ к свойству как изменяемому только для владельца и доступному только для чтения для остальных, делая ленивое свойство более устойчивым.

3.2.5. Изменяемые и ленивые свойства

Как только вы инициализируете ленивое свойство, его нельзя будет изменить. Следует проявлять особую осторожность при использовании изменяемых свойств, также известных как свойства `var`, в сочетании с ленивыми свойствами. Это особенно справедливо при работе со структурами.

Давайте рассмотрим сценарий, в котором, казалось бы, невинная случайность может привести к появлению незначительных ошибок.

Чтобы продемонстрировать это, в следующем листинге `level` превращается в свойство `var`, чтобы его можно было изменить.

Листинг 3.18. level теперь является изменяемым

```
struct LearningPlan {
   // ... Пропускаем часть кода.
   var level: Int
}
```

Вы создаете копию, обращаясь к структуре. Структуры обладают так называемой *семантикой значений*. Это означает, что когда вы обращаетесь к структуре, она копируется.

Например, вы можете создать интенсивный план обучения, скопировать его и понизить уровень, оставив исходный план обучения без изменений, как показано в следующем листинге.

Листинг 3.19. Копирование структуры

```
var intensePlan = LearningPlan(level: 138, description: "A special plan for
   today!")
intensePlan.contents ❶
var easyPlan = intensePlan ❷
easyPlan.level = 0 ❸
// Вопрос: Что будет выведено в результате?
print(easyPlan.contents)
```

❶ Заполняем ленивое свойство

❷ Делаем копию

❸ Понижаем уровень копии.

Теперь у вас есть копия, а вот вопрос: что вы получаете при выводе `easyPlan.contents`?

Ответ – интенсивный план: *«Прочитайте две научные статьи и переведите их на свой родной язык»*.

После создания `easyPlan` contents было загружено еще до того, как вы сделали копию, поэтому `easyPlan` копирует интенсивный план (см. рис. 3.1).

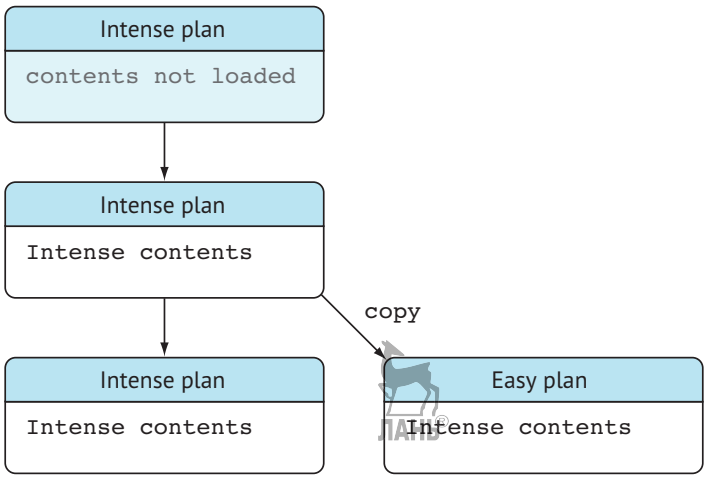


Рис. 3.1. Превращение структуры в перечисление

Кроме того, можно вызвать contents после создания копии, и в этом случае оба плана могут по отдельности выполнить ленивую загрузку своего содержимого (см. рис. 3.2).

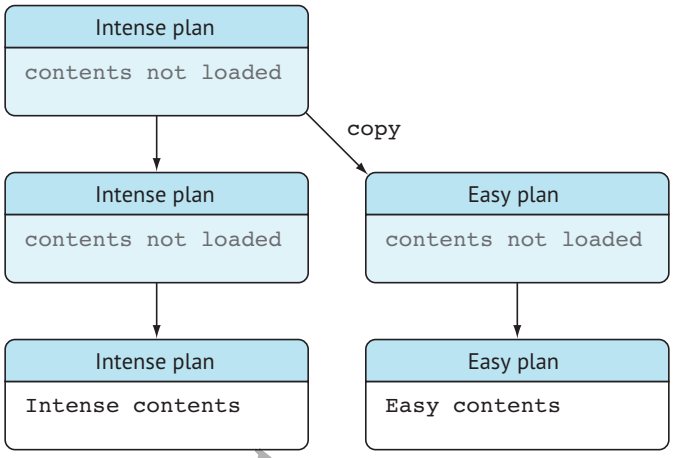


Рис. 3.2. Копирование перед инициализацией описания ленивого свойства

В следующем листинге видно, как оба плана рассчитывают свои планы, потому что копирование происходит до инициализации ленивых свойств.

Листинг 3.20. Копирование перед ленивой загрузкой

```
var intensePlan = LearningPlan(level: 138, description: "A special plan for
   today!")
var easyPlan = intensePlan ❶
easyPlan.level = 0 ❷

// Теперь оба плана имеют соответствующее содержимое.
print(intensePlan.contents) // Прочитайте две научные статьи и переведите
   их на свой родной язык.
print(easyPlan.contents) // Просмотр документального фильма на английском
   языке
```

❶ Делаем копию

❷ easyPlan получает более низкий уровень.

В предыдущих примерах подчеркивается, что сложность возрастает, как только вы начинаете изменять переменные, используемые лениво загружаемыми свойствами. Как только свойство изменяется, оно не синхронизируется с лениво загружаемым свойством.

Поэтому убедитесь, что вы сохраняете свойства неизменяемыми, если от них зависит ленивое свойство. Поскольку структуры копируются, когда вы ссылаетесь на них, становится еще более важным соблюдать особую осторожность при смешивании структур и ленивых свойств.

3.2.6. Упражнения

В этом упражнении вы будете моделировать музыкальную библиотеку (например, Apple Music или Spotify):

```
//: Decodable позволяет превращать необработанные данные (такие как файлы
   plist) в песни
struct Song: Decodable {
   let duration: Int
   let track: String
   let year: Int
}

struct Artist {
   var name: String
   var birthDate: Date
   var songsFileName: String

   init(name: String, birthDate: Date, songsFileName: String) {
      self.name = name
      self.birthDate = birthDate
      self.songsFileName = songsFileName
   }

   func getAge() -> Int? {
      let years = Calendar.current
         .dateComponents([.year], from: Date(), to: birthDate)
         .day
      return years
   }

   func loadSongs() -> [Song] {
      guard
         let fileURL = Bundle.main.url(forResource: songsFileName,
            withExtension: "plist"),
         let data = try? Data(contentsOf: fileURL),
         let songs = try? PropertyListDecoder().decode([Song].self, from:
            data) else {
            return []
         }
      return songs
   }

   mutating func songsReleasedAfter(year: Int) -> [Song] {
      return loadSongs().filter { (song: Song) -> Bool in
         return song.year > year
      }
   }
}
```

1

Можно ли очистить тип `Artist`, используя ленивые и/или вычисляемые свойства?

2

Предположим, что `loadSongs` превращается в ленивое свойство с именем songs. Убедитесь, что приведенный ниже код не нарушает его, пытаясь переопределить данные свойства:

```
// billWithers.songs = []
```

3

Предположим, что `loadSongs` превращается в ленивое свойство с именем songs. Как убедиться, что следующие строки не нарушат лениво загружаемое свойство? Укажите два способа предотвратить неисправность ленивого свойства:

```
billWithers.songs // загружаем песни
billWithers.songsFileName = "oldsongs" // меняем имя файла
billWithers.songs.count // после переименования songsFileName должен быть 0,
   а мы видим 2!!!
```

3.3. Наблюдатели свойств

Иногда вам нужно лучшее из обоих миров: вы хотите сохранить свойство, но вам по-прежнему хочется настраивать его поведение. В этом разделе вы познакомитесь с тем, как сочетать хранение и поведение с помощью *наблюдателей свойств*. Это был бы не Swift, если бы у него не было собственных уникальных правил, поэтому давайте посмотрим, как лучше ориентироваться в мире наблюдателей.

Наблюдатели свойств – это действия, инициализируемые, когда хранимое свойство меняет значение. Это идеальный кандидат на случай, если вам нужно выполнить работу по очистке после установки свойства или когда вы хотите уведомить другие фрагменты вашего приложения об изменениях свойств.

3.3.1. Обрезка пробелов

Представьте себе сценарий, в котором игрок может присоединиться к многопользовательской онлайн-игре. Все, что ему нужно ввести, – это имя, содержащее минимальное количество символов. Однако люди, которым нужны короткие имена, могут заполнить оставшиеся символы пробелами в соответствии с требованиями.

Вы увидите, как можно автоматически очистить имя после его установки. Наблюдатель свойств удаляет ненужные пробелы из имени.

В следующем примере видно, как имя автоматически избавляется от конечных пробелов после обновления свойства. Обратите внимание, что инициализатор не запускает наблюдателя.

Листинг 3.21. Обрезка пробелов

```
let jeff = Player(id: "1", name: "SuperJeff ")
print(jeff.name) // "SuperJeff " ❶
print(jeff.name.count) // 13

jeff.name = "SuperJeff "
print(jeff.name) // "SuperJeff" ❷
print(jeff.name.count) // 9
```

❶ Пробел в имени игрока не обрезается при инициализации типа Player.

❷ Пробел обрезается, когда вы снова обновляете имя игрока.

Свойство name автоматически удаляет пробелы, но *только* при обновлении, а не во время первоначальной настройки. Прежде чем решить эту проблему, посмотрите на класс Player, чтобы увидеть, как работает наблюдатель свойства, как показано в приведенном ниже листинге.

Листинг 3.22. Класс Player

```
import Foundation

class Player {

   let id: String

   var name: String { ❶
      didSet { ❷
         print("My previous name was \(oldValue)") ❸
         name = name.trimmingCharacters(in: .whitespaces) ❹
      }
   }

   init(id: String, name: String) {
      self.id = id
      self.name = name
   }
}
```

❶ Убеждаемся, что name – это переменная, потому что оно изменяется.

❷ Наблюдатель свойства didSet, который инициализируется после установки свойства.

❸ Обращаемся к предыдущему значению через константу oldValue.

❹ Свойство name обрезается после установки. Это не приведет к бесконечному циклу – вы изменяете хранимое имя в этой области.

didSet willSet

Помимо `didSet`, вы также можете использовать наблюдателей `willSet`, которые запускаются непосредственно перед изменением свойства. Если вы используете `willSet`, то можете использовать константу newValue.

3.3.2. Запуск наблюдателей свойств из инициализаторов

Свойство `name` было очищено после того, как вы обновили свойство, но первоначально `name` все еще содержало пробелы, как показано в приведенном ниже листинге.

Листинг 3.23. Наблюдатель свойства не запускается из инициализатора

```
let jeff = Player(id: "1", name: "SuperJeff ")
print(jeff.name) // "SuperJeff " ❶
print(jeff.name.count) // 13
```

❶ Создаем замыкание, которое будет вызываться сразу после инициализации.

Наблюдатели свойств, к сожалению, *не* вызываются из инициализаторов, что является преднамеренным, но вам следует знать об этой ловушке. К счастью, есть обходной путь.

Рекомендуемый метод официально состоит в том, чтобы разделить замыкание `didSet` на функцию, после чего вы сможете вызвать эту функцию из инициализатора. Однако есть еще один трюк, который заключается в том, чтобы добавить оператор `defer` в метод инициализатора.

Когда функция завершается, в действие вступает оператор `defer`. Можно поместить его в любом месте инициализатора, но он будет вызываться только после того, как функция достигнет конца. Это удобно, когда вам нужно запустить код очистки в конце функции.

Добавьте оператор `defer` в инициализатор, который устанавливает заголовок в следующем листинге.

Листинг 3.24. Добавление оператора defer в инициализатор


```
class Player {

   // ... Пропускаем часть кода

   init(id: String, name: String) {
      defer { self.name = name } ❶
      self.id = id
      self.name = name ❷
   }
}
```

❶ Создаем замыкание, которое будет вызываться сразу после инициализации.

❷ Поскольку заголовок в конечном итоге устанавливается с помощью defer, не имеет значения, что здесь задано.

Замыкание вызывается сразу после инициализации Player, инициализируя наблюдателя свойства, как показано в приведенном ниже листинге.

Листинг 3.25. Обрезка пробелов

```
let jeff = Player(id: "1", name: "SuperJeff ")
print(jeff.name) // "SuperJeff"
print(jeff.name.count) // 9

jeff.name = "SuperJeff "
print(jeff.name) // "SuperJeff"
print(jeff.name.count) // 9
```

Вы получаете лучшее из обоих миров. Вы можете хранить свойство, можете инициировать действия с ним, и нет разницы между установкой свойства из инициализатора или аксессора.

Предостережение: трюк с defer официально не предназначен для использования таким образом. Это означает, что данный метод в будущем может не сработать. Однако до тех пор это решение – изящный прием, который вы можете применить.

3.3.3. Упражнения

4

Если вам нужно свойство с поведением и хранением, какое свойство вы бы использовали?

5

Если вам нужно свойство только с поведением и без хранения, какое свойство вы бы использовали?

6

Можете найти ошибку в этом коде?

```
struct Tweet {
   let date: Date
   let author: String
   var message: String {
      didSet {
         message = message.trimmingCharacters(in: .whitespaces)
      }
   }
}

let tweet = Tweet(date: Date(),
```

```
    author: "@tjeerdintveen",
    message: "This has a lot of unnecessary whitespace ")
```

7

Как исправить ошибку?

3.4. В заключение

Даже если вы уже использовали свойства, я надеюсь, что стоит потратить немного времени, чтобы понять их на более глубоком уровне.

Вы видели, как сделать выбор между вычисляемым и ленивым свойством и хранимыми свойствами с поведением. Правильный выбор делает ваш код более предсказуемым, а также очищает интерфейс и поведение ваших классов и структур.

Резюме

- Можно использовать вычисляемые свойства для свойств с определенным поведением, но без хранения.
- Вычисляемые свойства – это функции, маскирующиеся под свойства.
- Можно использовать вычисляемые свойства, когда значение может отличаться при каждом вызове.
- Только легковесные функции должны становиться вычисляемыми свойствами.
- Ленивые свойства прекрасно подходят для затратных или трудоемких вычислений.
- Используйте ленивые свойства, чтобы отложить вычисление или вообще не запускать его.
- Ленивые свойства позволяют обращаться к другим свойствам внутри классов и структур.
- Можно использовать аннотацию `private(set)`, чтобы сделать свойства доступными только для чтения для тех, кто не имеет отношения к классу или структуре.
- Если ленивое свойство обращается к другому свойству, убедитесь, что другое свойство является неизменяемым, чтобы сложность была низкой.
- Можно использовать наблюдателей свойств, таких как `willSet` и `didSet`, чтобы добавить поведение к хранимым свойствам.
- Можно использовать оператор `defer` для запуска наблюдателей свойств из инициализатора.

Ответы

1

Можно ли очистить тип Artist, используя ленивые и/или вычисляемые свойства?

```
struct Artist {
   var name: String
   var birthDate: Date
   let songsFileName: String

   init(name: String, birthDate: Date, songsFileName: String) {
      self.name = name
      self.birthDate = birthDate
      self.songsFileName = songsFileName
   }

   // Возраст теперь посчитан (рассчитывается каждый раз)
   var age: Int? {
      let years = Calendar.current
         .dateComponents([.year], from: Date(), to: birthDate)
         .day
      return years
   }

   // loadSongs () теперь ленивое свойство, потому что загружать файл для
      каждого вызова затратно.
   lazy private(set) var songs: [Song] = {
      guard
         let fileURL = Bundle.main.url(forResource: songsFileName,
            withExtension: "plist"),
         let data = try? Data(contentsOf: fileURL),
         let songs = try? PropertyListDecoder().decode([Song].self,
            from: data) else {
            return []
         }
      return songs
   }()

   mutating func songsReleasedAfter(year: Int) -> [Song] {
      return songs.filter { (song: Song) -> Bool in
         return song.year > year
      }
   }
```

```
}
```

2

Предположим, что `loadSongs` превращается в ленивое свойство с именем `songs`. Убедитесь, что приведенный ниже код не нарушает его, пытаясь переопределить данные свойства:

```
// billWithers.songs = []
```

Этого можно добиться, сделав `songs` свойством `private(set)`.

3

Предположим, что `loadSongs` превращается в ленивое свойство с именем `songs`. Как убедиться, что следующие строки не нарушат лениво загружаемое свойство? Укажите два способа:

```
billWithers.songs
billWithers.songsFileName = "oldsongs"
billWithers.songs.count // Should be 0 after renaming songsFileName,
but is 2
```

Ленивое свойство `song` указывает на переменную `songFileName`. Чтобы предотвратить изменение после лениво загружаемых песен, можно сделать `songFileName` константой с помощью ключевого слова `let`. Кроме того, можно сделать `Artist` классом, чтобы предотвратить эту ошибку.

4

Если вам нужно свойство с поведением и хранением, какое свойство вы бы использовали?

Хранимое свойство с наблюдателем свойств.

5

Если вам нужно свойство только с поведением и без хранения, какое свойство вы бы использовали?

Вычисляемое свойство или ленивое свойство, если вычисление затратно.

6

Можете обнаружить ошибку в коде?

Пробелы не удаляются из инициализатора.

7

Как исправить ошибку?

Добавить инициализатор в структуру с оператором `defer`.

```
struct Tweet {
   let date: Date
   let author: String
   var message: String {
```

```
        didSet {
            message = message.trimmingCharacters(in: .whitespaces)
        }
    }

    init(date: Date, author: String, message: String) {
        defer { self.message = message }
        self.date = date
        self.author = author
        self.message = message
    }
}

let tweet = Tweet(date: Date(),
    author: "@tjeerdintveen",
    message: "This has a lot of unnecessary whitespace ")

tweet.message.count
```

Глава 4. Делаем опционалы второй натурой

В этой главе:

- передовые методы, связанные с опционалами;
- обработка нескольких опционалов с помощью операторов guard;
- правильная работа с опциональными строками по сравнению с пустыми строками;
- одновременное манипулирование несколькими опционалами;
- возврат к значениям по умолчанию с помощью оператора объединения по нулевому указателю;
- упрощение опциональных перечислений;
- работа с опциональными логическими типами различными способами;
- погружение вглубь значений с помощью опциональной цепочки;
- рекомендации по принудительному извлечению;
- приручение неявно извлекаемых опционалов.

В ходе прочтения этой главы у вас появится множество инструментов, которые помогут вам перейти от разочарования при работе с опционалами к *нирване*, применяя при этом передовые методы. Опционалы настолько распространены в Swift, что мы потратили на них лишние страницы, чтобы не оставить камня на камне. Даже если вы знаете, как работать с опционалами, просмотрите эту главу, чтобы заполнить пробелы в знаниях.

Глава начинается с описания того, что такое опционалы и каким образом Swift помогает вам, добавляя синтаксический сахар. Затем мы рассмотрим советы по стилю в сочетании с маленькими примерами, которые вы регулярно встречаете в Swift. После этого вы увидите, как не дать опционалам распространяться внутри метода с помощью оператора guard. Мы также рассмотрим, какой вариант выбрать при возврате строк: пустые или опциональные. Далее мы покажем, как получить более детальный контроль над несколькими опционалами, используя сопоставление с образцом. Затем вы увидите, что можно вернуться к значениям по умолчанию с помощью *оператора объединения по нулевому указателю*.

Когда вы столкнетесь с опционалами, содержащими другие опционалы, то увидите, как можно использовать опциональную цепочку, чтобы копать дальше и добраться до вложенных значений. Затем мы покажем, как использовать сопоставление с образцом для опциональных перечислений. Прочитав этот раздел, вы сможете сбросить вес со своих операторов switch, а также увидите, что опциональные логические типы могут быть несколько неуклюжими; мы покажем, как правильно обращаться с ними в зависимости от потребностей.

Принудительное извлечение – способ обойти безопасность, которую предлагают опционалы, – требует личного внимания. В этом разделе мы расскажем вам, когда принудительное извлечение опционалов допустимо. Вы также увидите, что

можно предоставить больше информации, если, к сожалению, ваше приложение дает сбой. В этом разделе много эвристики, которую можно использовать, чтобы улучшить свой код. После принудительного извлечения мы рассмотрим неявно извлекаемые опционалы, которые по-разному ведут себя в зависимости от контекста. Вы четко увидите, как правильно обращаться с ними.

Эта глава определенно немаленькая. Кроме того, мы будем возвращаться к опционалам в других главах, используя изящные советы и хитрости, такие как применение `map` и `flatMap` для опционалов и `compactMap` для массивов с опционалами. Цель состоит в том, чтобы в конце этой главы вы почувствовали себя экспертом и были уверены в том, что сможете справиться с любым сценарием, где задействованы опционалы.

4.1. Назначение опционалов

Говоря простым языком, опционал – это «коробка», у которой *есть* значение или его *нет*. Возможно, вы слышали об опционалах, таких как тип `Option` в Rust или Scala или тип `Maybe` в Haskell.

Опционалы помогают предотвратить сбои, когда значение пусто; они делают это, прося вас явно обрабатывать каждый случай, когда переменная или константа может быть равна nil. Опциональное значение нужно извлечь, чтобы получить его. Если есть значение, может выполняться определенный фрагмент кода. Если значение отсутствует, код выбирает другой путь.

Благодаря опционалам вы всегда будете знать на этапе компиляции, может ли значение быть равно nil, а это роскошь, которой нет в других языках. Например, если вы получаете значение nil в Ruby и не проверяете его, то можете получить ошибку во время выполнения. Недостатком опционалов является то, что это может замедлить вас и разочаровать, поскольку Swift заставляет вас явно обрабатывать каждый опционал, когда вам нужно его значение. Но в качестве награды взамен вы получаете более безопасный код.

С помощью этой книги вы настолько привыкнете к опционалам, что сможете быстро и без труда с ними справиться. С достаточным количеством инструментов

под вашим поясом вы можете найти, что с опциями приятно работать, так что давайте начнем!

4.2. Чистое извлечение значений

В этом разделе вы увидите, как опционалы представлены в коде и как извлечь их несколькими способами.

Мы начнем с моделирования модели покупателя для бэкэнда несуществующего интернет-магазина под названием The Mayonnaise Depot, который удовлетворит все ваши потребности в майонезе.

Присоединяйтесь!

Будет более познавательно и веселее, если вы будете знакомиться с кодом по мере прохождения этой главы. Исходный код можно скачать по адресу http://mng.bz/6jO5.

Структура `Customer` содержит идентификатор клиента, адрес электронной почты, имя и текущий баланс, которые интернет-магазин может использовать, чтобы вести бизнес с клиентом. В демонстрационных целях мы пропустили ряд других важных свойств, таких как адрес и информация об оплате.

Интернет-магазин довольно снисходителен по отношению к клиентам и позволяет им не указывать свои имя и фамилию при заказе вкусного майонеза. Таким образом клиентам проще заказывать майонез. Мы представим эти значения как опционалы `firstName` и `lastName` внутри структуры, как показано ниже.

Листинг 4.1. Структура Customer

```
struct Customer {
   let id: String ❶
   let email: String
   let balance: Int // Количество в центах.
   let firstName: String? ❷
   let lastName: String?
}
```

❶ Свойства `id`, `email` и `balance` являются обязательными.

❷ У клиента два опциональных свойства: `firstName` и `lastName`.

Опционалы – это переменные или константы, обозначаемые как `?`. Но если присмотреться, то можно увидеть, что опционал – это перечисление, которое может содержать или не содержать значение. Давайте посмотрим на тип `Optional` внутри исходного кода Swift.

Листинг 4.2. Опционалы – это перечисление

```
// Детали опущены.
public enum Optional<Wrapped> {
```

```
    case none
    case some(Wrapped)
}
```

Тип `Wrapped` – это *обобщенный тип*, который представляет значение внутри опционала. Если у него есть значение, оно помещается в блок `some`; если значение отсутствует, у опционала на этот случай есть блок `none`.

Swift добавляет немного синтаксического сахара в синтаксис опционалов, потому что без него вы бы писали структуру `Customer` так:

Листинг 4.3. Опционалы без синтаксического сахара

```
struct Customer {
    let id: String
    let email: String
    let balance: Int
    let firstName: Optional<String> ❶
    let lastName: Optional<String> ❷
}
```

❶ Нотация опционала без синтаксического сахара.

❷ Еще один опционал без синтаксического сахара.

Примечание

Явное написание опционала по-прежнему допустимо в Swift и прекрасно компилируется, но это не характерно для Swift (поэтому используйте эти знания, только чтобы выиграть в викторине в пабе).

4.2.1. Сопоставление для опционалов

Как правило, опционал извлекают, используя выражение `if let`, как в случае с извлечением свойства клиента `firstNameproperty`.

Листинг 4.4. Извлечение с помощью выражения if let

```
let customer = Customer(id: "30", email: "mayolover@gmail.com",
➥ firstName: "Jake", lastName: "Freemason", balance: 300)

print(customer.firstName) // Optional("Jake") ❶
if let firstName = customer.firstName {
    print(firstName) // "Jake" ❷
}
```

❶ Опциональная строка.

❷ Извлекаемая строка.

Но опять же, без синтаксических возможностей, которые предоставляет Swift, мы бы повсюду использовали сопоставление с помощью операторов `switch`. Со-

поставление для опционала показано на рис. 4.1. Оно пригодится позже, например когда вам нужно будет объединить извлечение и сопоставление с образцом.

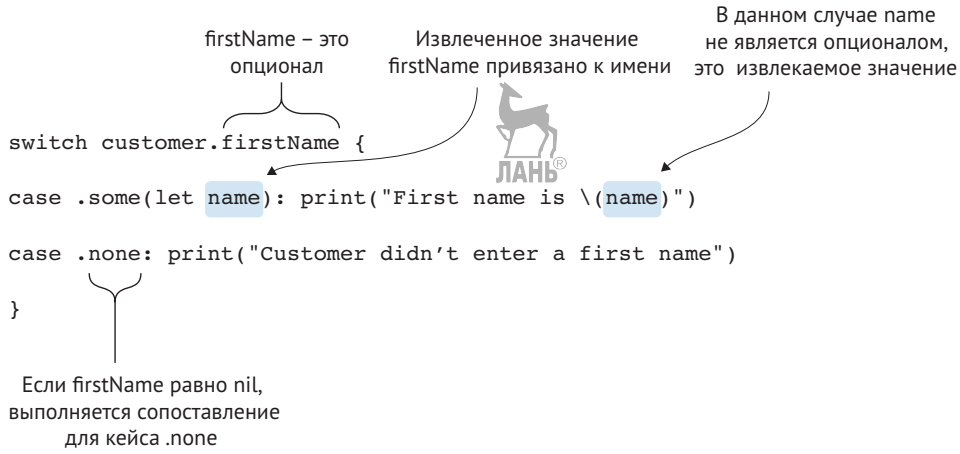


Рис. 4.1. Сопоставление для опционала

Swift согласен с тем, что можно использовать сопоставление с образцом для опционалов, и даже предлагает для этого синтаксические возможности.

Можно опустить кейсы `.some` и `.none` из приведенного ниже примера и заменить их на `let name?` и `nil`, как показано в листинге 4.5.

Листинг 4.5. Сопоставление с использованием синтаксических возможностей

```
switch customer.firstName {
   case let name?: print("First name is \(name)") ❶
   case nil: print("Customer didn't enter a first name") ❷
}
```

❶ Когда у `firstName` есть значение, свяжите его с именем и выведите надпись на экран.

❷ Можно выполнить сопоставление явно, когда `firstName` равно `nil`.

Этот код делает то же, что и раньше, просто он написан немного иначе.

К счастью, Swift предлагает много синтаксического сахара, чтобы сэкономить нам все время при наборе кода. Тем не менее полезно знать, что в глубине души опционалы – это перечисления, когда вы столкнетесь с более сложными или эзотерическими методами извлечения.

4.2.2. Методы извлечения

Извлечение нескольких опционалов можно комбинировать. Комбинирование их, вместо того чтобы создавать отступы, помогает делать отступы ниже. Например, можно обрабатывать два опционала одновременно, извлекая `firstName` и `lastName`, как показано здесь.

Листинг 4.6. Комбинируем извлечение опционалов

```
if let firstName = customer.firstName, let lastName = customer.lastName {
   print("Customer's full name is \(firstName) \(lastName)")
}
```

Извлечение значений опционала не обязательно должно быть единственным действием внутри оператора `if`. Можно комбинировать операторы `if let` с логическими типами данных.

Например, можно извлечь `firstName` клиента и объединить его с логическим типом, таким как свойство `balance`, чтобы написать специальное сообщение.

Листинг 4.7. Объединение логического типа и опционала

```
if let firstName = customer.firstName, customer.balance > 0 {
   let welcomeMessage = "Dear \(firstName), you have money on your account,
   want to spend it on mayonnaise?"
}
```

Но не обязательно останавливаться на логических типах. Также можно использовать сопоставление с образцом, пока вы извлекаете значение опционала, например в случае с `balance`.

В следующем примере мы создадим уведомление для клиента, когда его баланс (указанный в центах) имеет значение, которое находится в диапазоне от 4500 до 5000 центов, что составляет от 45 до 50 долларов.

Листинг 4.8. Сочетание сопоставления с образцом и извлечения опционала

```
if let firstName = customer.firstName, 4500..<5000 ~= customer.balance {
   let notification = "Dear \(firstName), you are getting close to afford
      our $50 tub!"
}
```

Украсив операторы `if` другими действиями, вы тем самым можете сделать свой код чище. Можно отделить проверки логических типов и сопоставление с образцом от извлечения опционалов, но тогда у вас, как правило, будут получаться вложенные операторы `if`, а иногда и несколько блоков с `else`.

4.2.3. Когда значение вас не интересует

Использование выражения `if let` для извлечения опционалов отлично подходит для тех случаев, когда значение вас не интересует. Но если вы хотите выполнять действия, когда значение равно nil или нерелевантно, можно использовать традиционную проверку на nil, как это делают в других языках.

В приведенном ниже примере вам нужно знать, заполнил ли клиент имя и фамилию, но вас не интересует, что это за имя и фамилия. Вы снова можете использовать подстановочный оператор «мне все равно», чтобы привязать извлекаемые значения к нулю.

Листинг 4.9. Использование символа подчеркивания

```
if
   let _ = customer.firstName,
   let _ = customer.lastName {
      print("The customer entered his full name")
}
```

Кроме того, можно использовать проверку на nil, чтобы добиться того же эффекта.

Листинг 4.10. Проверка на nil

```
if
   customer.firstName != nil,
   customer.lastName != nil {
      print("The customer entered his full name")
}
```

Вы также можете выполнять действия, когда значение пусто.

Затем вы проверяете, есть ли у клиента имя, не интересуясь, каким оно может быть.

Листинг 4.11. Если опционал равен нулю

```
if customer.firstName == nil,
   customer.lastName == nil {
      print("The customer has not supplied a name.")
}
```

Некоторые могут утверждать, что проверка на nil не соответствует духу Swift. Но эти проверки можно найти внутри исходного кода Swift. Не стесняйтесь использовать их, если вы думаете, что это поможет сделать код более удобным для чтения. Ясность важнее стиля.

4.3. Сокрытие переменной

Поначалу у вас может возникнуть соблазн придумать уникальные имена при извлечении (например, имея константу `firstName`, вы даете извлекаемой константе имя `unwrappedFirstName`), но это не обязательно.

Можно присвоить извлекаемому значению то же имя, что и у опционала, в другой области видимости. Этот метод носит название *сокрытие переменной*. Давайте посмотрим, как это выглядит.

4.3.1. Реализация протокола CustomStringConvertible

Чтобы продемонстрировать сокрытие переменной, мы создадим метод, который использует опциональные свойства `Customer`. Давайте обеспечим соответствие `Customer` протоколу `CustomStringConvertible`.

Заставляя Customer соответствовать протоколу CustomStringConvertible, вы указываете, что он выводит пользовательское представление с помощью операторов print. Соответствие этому протоколу заставляет вас реализовать свойство description.

Листинг 4.12. Соответствие протоколу CustomStringConvertible

```
extension Customer: CustomStringConvertible { ❶
   var description: String {
      var customDescription: String = "\(id), \(email)"

      if let firstName = firstName { ❷
         customDescription += ", \(firstName)" ❸
      }

      if let lastName = lastName { ❹
         customDescription += " \(lastName)" ❺
      }

      return customDescription
   }
}

let customer = Customer(id: "30", email: "mayolover@gmail.com",
➥ firstName: "Jake", lastName: "Freemason", balance: 300)

print(customer) // 30, mayolover@gmail.com, Jake Freemason ❻
```

❶ Customer реализует протокол CustomStringConvertible.

❷ Извлекаем свойство firstName, привязав его к константе с тем же именем.

❸ Внутри извлекаемой области if let извлекается firstName, и оно не является опциональным.

❹ Извлекаем свойство lastName, привязав его к константе с тем же именем.

❺ Константа lastName извлекается в области действия if let.

❻ Теперь при выводе на экран клиента отображается пользовательское описание.

В этом примере показано, как извлечь firstName и lastName с использованием сокрытия переменной, что не дает нам применять специальные имена. Поначалу сокрытие переменной может показаться непонятным, но если вы какое-то время уже работали с опционалами, то вы убедитесь, что такой код приятно читать.

4.4. Когда опционалы запрещены

Давайте посмотрим, как обработать несколько опционалов, когда нельзя использовать значения nil. Когда интернет-магазин The Mayonnaise Depot получает за-

каз, клиенты получают по электронной почте сообщение с подтверждением, которое должно быть создано бэкэндом. Вот как выглядит это сообщение.

Листинг 4.13. Сообщение с подтверждением

```
Dear Jeff,
Thank you for ordering the economy size party tub!
Your order will be delivered tomorrow.

Kind regards,
The Mayonnaise Depot
```


Эта функция помогает создать сообщение с подтверждением, принимая имя клиента и продукт, который он заказал.

Листинг 4.14. Сообщение о подтверждении заказа

```
func createConfirmationMessage(name: String, product: String) -> String {
   return """ ❶
   Dear \(name), ❷
   Thank you for ordering the \(product)!
   Your order will be delivered tomorrow.
   Kind regards,
   The Mayonnaise Depot
   """
}

let confirmationMessage = createConfirmationMessage(name: "Jeff", product:
   "economy size party tub") ❸
```

❶ Используем многострочный строковый оператор Swift, – """–, для создания многострочной строки.

❷ Меняем имя и продукт внутри строки на то, что вы передаете в функцию.

❸ Можно вызвать эту функцию, используя имя и продукт.

Чтобы получить имя клиента, сначала выберите имя для отображения. Поскольку `firstName` и `lastName` – опциональные, нужно убедиться, что у вас есть эти значения для отображения их в электронном письме.

4.4.1. Добавление вычисляемого свойства

Мы будем использовать вычисляемое свойство, чтобы указать действительное имя клиента. Назовем его `displayName`. Цель этого свойства – помочь отобразить имя клиента, чтобы использовать его в электронных письмах, веб-страницах и т. д.

Можно использовать оператор `guard`, чтобы убедиться, что у всех свойств есть значение; если нет, вы возвращаете пустую строку. Но вы быстро увидите, что есть решение получше, нежели возврат пустой строки, как показано в этом листинге.

Листинг 4.15. Структура Customer

```
struct Customer {
   let id: String
   let email: String
   let firstName: String?
   let lastName: String?

   var displayName: String {
      guard let firstName = firstName, let lastName = lastName else { ❶
         return "" ❷
      }

      return "\(firstName) \(lastName)" ❸
   }
}
```

❶ guard связывает извлекаемые значения firstName и lastName с одинаково именованными константами.

❷ Если firstName, либо lastName, либо оба они равны нулю, guard возвращает пустое имя. Скоро мы это усовершенствуем.

❸ Далее извлекаются опционалы и возвращается полное имя.

Операторы guard отлично подходят при использовании подхода «никто не пройдет», когда опционалы не нужны. Через мгновение вы увидите, как получить более детальный контроль с несколькими опционалами.

Операторы guard и отступы

Благодаря операторам guard отступы получаются небольшими, что делает их хорошим кандидатом для извлечения.

Теперь можно воспользоваться вычисляемым свойством displayName, чтобы использовать его при любом общении с клиентами.

Листинг 4.16. displayName в действии

```
let customer = Customer(id: "30", email: "mayolover@gmail.com",
➥ firstName: "Jake", lastName: "Freemason", balance: 300)

customer.displayName // Jake Freemason
```

4.5. Возврат опциональных строк

В реальных приложениях у вас может возникнуть желание вернуть пустую строку, потому что это избавляет от необходимости извлечения опциональной строки. Пустые строки часто тоже имеют смысл, но в нашем сценарии они не выгодны. Давайте рассмотрим, почему.

Вычисляемое свойство displayName служит своей цели, но проблема возникает, когда свойства firstName и lastName имеют значение nil: displayName возвращает пустую строку, например "". Если у вас нет надежного коллеги, чтобы добавить проверку displayName.isEmpty для всего приложения, вы можете пропустить один случай, и некоторые клиенты получат электронное письмо, начинающееся со слова «Уважаемый», где имя отсутствует.

Ожидается, что строки будут пустыми там, где они имеют смысл, например при загрузке текстового файла, который может быть пустым, но пустая строка имеет меньший смысл в displayName, потому что разработчики этого кода ожидают отображения какого-либо имени.

При таком сценарии лучший вариант – не быть двусмысленным и сообщить разработчикам, реализующим метод, что строка может быть опциональной. Для этого нужно заставить свойство displayName вернуть опциональную строку.

Преимущество возврата опциональной строки состоит в том, что на этапе компиляции вы знаете, что у displayName может не быть значения, тогда как при проверке isEmpty вы узнаете об этом во время выполнения. Безопасность на этапе компиляции оказывается полезной, когда вы рассылаете информационный бюллетень полумиллиону человек и не хотите, чтобы он начинался со слова «Уважаемый».

Возврат опциональной строки может показаться бессмыслицей, но такое часто происходит внутри проектов, особенно когда разработчики не слишком заинтересованы в опционалах. Не возвращая опционал, вы меняете безопасность на этапе компиляции на потенциальную ошибку во время выполнения.

В качестве подходящего примера displayName вернет опциональную строку для displayName.

Листинг 4.17. Заставляем displayName вернуть опциональную строку

```
struct Customer {
   // ... Пропускаем часть кода

   var displayName: String? { ❶
      guard let firstName = firstName, let lastName = lastName else {
         return nil ❷
      }
      return "\(firstName) \(lastName)"
   }
}
```


❶ displayName теперь возвращает опциональную строку.

❷ Оператор guard возвращает ноль, когда имена пустые.

Теперь, когда displayName возвращает опциональную строку, код, вызывающий метод, должен явно извлечь опционал. Необходимость извлечения displayName, как показано ниже, может быть хлопотным делом, но взамен вы получите больше безопасности.

Листинг 4.18. Извлечение displayName

```
if let displayName = customer.displayName {
   createConfirmationMessage(name: displayName, product: "Economy size
      party tub")
} else {
   createConfirmationMessage(name: "customer", product: "Economy size
      party tub")
}
```

Ради душевного спокойствия можно добавить проверку `isEmpty` для обеспечения дополнительной безопасности во время выполнения, чтобы сделать критические места своего приложения сверхбезопасными – например, при отправке информационных бюллетеней, – но, по крайней мере, теперь вы получаете какую-то помощь от компилятора.

4.6. Детальный контроль над опционалами

В настоящее время `displayName` должно иметь значения `firstName` и `lastName`, прежде чем он вернет правильную строку. Давайте немного ослабим `displayName`, чтобы он мог возвращать любое найденное имя, делая свойство более мягким. Если известно только имя или фамилия, функция `displayName` может возвращать одно или оба этих значения, в зависимости от того, какие имена указаны.

Можно сделать `displayName` более гибким, заменив оператор guard оператором switch. По сути, это означает, что извлечение двух опционалов становится частью логики `displayName`, тогда как с помощью guard вы блокируете любые значения nil, прежде чем метод свойства продолжит работу.

В качестве улучшения мы поместим оба опционала в кортеж, например (`firstName`, `lastName`), а затем выполним сопоставление для них одновременно. Таким образом можно вернуть значение в зависимости от того, какие опционалы его содержат.

Используя оператор «?» в операторах `case`, вы связываете *и* извлекаете опционалы. Вот почему вы получаете неопциональное свойство в строках внутри этих операторов.

Теперь, когда у клиента нет полного имени, вы все равно можете использовать часть имени, например только фамилию.

Листинг 4.19. displayName работает с частично заполненным именем

```
let customer = Customer(id: "30", email: "famthompson@gmail.com",
➥ firstName: nil, lastName: "Thompson", balance: 300)

print(customer.displayName) // "Thompson"
```

Просто будьте осторожны, добавляя слишком много опционалов в кортеж. Обычно при добавлении трех или более опционалов можно использовать другую

абстракцию, чтобы код было удобнее читать, например комбинацию операторов `if let`.

4.6.1. Упражнения

1

Если опционалам в функции запрещено иметь значение, какая тактика подойдет для того, чтобы убедиться, что все опционалы заполнены?

2

Если функции выбирают разные пути в зависимости от опционалов внутри них, какой подход следует выбрать для обработки этих путей?

4.7. Откат назад, если опционал равен nil

Ранее вы видели, как мы проверяли пустые и опциональные строки для свойства `displayName`.

Когда вы получили непригодное к использованию свойство `displayName`, вы вернулись к «клиенту», поэтому при переписке The Mayonnaise Depot начинает свои письма со слов «Уважаемый клиент».

Когда опционал равен нулю, вполне типично будет прибегнуть к исходному значению. Поэтому Swift предлагает синтаксический сахар в виде оператора ?? под названием оператор объединения по нулевому указателю.

Можно использовать оператор ??, чтобы вернуться к «`customer`», когда у клиента нет доступного имени.

Листинг 4.20. Возврат к значению по умолчанию с помощью оператора объединения по нулевому указателю

```
let title: String = customer.displayName ?? "customer"
createConfirmationMessage(name: title, product: "Economy size party tub")
```

Как и прежде, всякий раз, когда имя клиента не заполнено, электронное письмо начинается со слов «Уважаемый клиент». Однако на этот раз мы убрали явное извлечение с использованием выражения `if let` из нашего кода.

Оператор объединения по нулевому указателю не только возвращается к значению по умолчанию, он также извлекает опционал, когда у него есть значение.

Обратите внимание, что `title` – это строка, но вы указываете здесь опционал `customer.displayName`. Это означает, что `title` будет иметь извлекаемое имя клиента *или* будет возвращаться к неопциональному значению «`customer`».

4.8. Упрощение опциональных перечислений

Ранее мы видели, что опционалы – это перечисления и что можно использовать для них сопоставление с образцом, например когда создавали значение

displayName в структуре Customer. Даже если опционалы являются перечислениями, вы можете столкнуться с перечислением, которое является опционалом. Опциональное перечисление – это перечисление внутри перечисления. В этом разделе вы увидите, как обрабатывать опциональные перечисления.

Наш любимый интернет-магазин The Mayonnaise Depot предлагает два типа членства, silver и gold, со скидками пять и десять процентов соответственно на свои вкусные продукты. Клиенты могут оплатить эти членства, чтобы получать скидки на все свои заказы.

Перечисление Membership, как показано в этом примере, представляет эти типы членства.

Листинг 4.21. Перечисление Membership

```
enum Membership {
   /// Скидка 10 %
   case gold
   /// Скидка 5 %
   case silver
}
```

Не все клиенты стали членами, что приводит к появлению опционального свойства membership в структуре Customer.

Листинг 4.22. Добавление свойства membership в структуру Customer

```
struct Customer {
   // ... Пропускаем часть кода.
   let membership: Membership?
}
```

Когда вам нужно прочитать это значение, первая тактика реализации – вначале извлечь перечисление и действовать соответственно.

Листинг 4.23. Извлечение опционала перед сопоставлением с образцом

```
if let membership = customer.membership {
   switch membership {
      case .gold: print("Customer gets 10% discount")
      case .silver: print("Customer gets 5% discount")
   }
} else {
   print("Customer pays regular price")
}
```

Более того, можно выбрать путь покороче и использовать сопоставление для опционального перечисления с помощью оператора ?. Оператор ? указывает на то, что вы извлекаете и читаете свойство membership одновременно.

Листинг 4.24. Сопоставление с образцом для опционала

```
switch customer.membership {
   case .gold?: print("Customer gets 10% discount")
   case .silver?: print("Customer gets 5% discount")
   case nil: print("Customer pays regular price")
}
```

Примечание

В случае сопоставления для `nil` компилятор сообщит вам, как только вы добавите в перечисление новый оператор `case`. В случае с `default` такой роскоши у вас не будет.

Одним махом мы выполнили сопоставление с образцом для перечисления и извлекли его, тем самым исключая из своего кода лишний шаг, связанный с извлечением и использованием выражения `if let`.

4.8.1. Упражнение

3

У вас есть два перечисления. Одно перечисление представляет содержимое визитки (некие данные, которые пользователь вырезал или скопировал в визитную карточку):

```
enum PasteBoardContents {
   case url(url: String)
   case emailAddress(emailAddress: String)
   case other(contents: String)
}
```

`PasteBoardEvent` обозначает событие, связанное с `PasteBoardContents`. Возможно, содержимое было добавлено в визитку, стерто из визитки или вставлено из нее:

```
enum PasteBoardEvent {
   case added
   case erased
   case pasted
}
```

Функция `descriptionAction` принимает два перечисления и возвращает строку, описывающую событие, например: «Пользователь добавил адрес электронной почты в визитную карточку». Цель этого упражнения – заполнить тело функции:

```
func describeAction(event: PasteBoardEvent?, contents:
   PasteBoardContents?) -> String {
```

```
        // Что здесь?
}
```

Имеются исходные данные:

```
describeAction(event: .added, contents: .url(url: "www.manning.com"))
describeAction(event: .added, contents: .emailAddress(emailAddress:
   "info@manning.com"))
describeAction(event: .erased, contents: .emailAddress(emailAddress:
   "info@manning.com"))
describeAction(event: .erased, contents: nil)
describeAction(event: nil, contents: .other(contents: "Swift in Depth"))
```

Убедитесь, что вывод выглядит следующим образом:

```
"User added an url to pasteboard: www.manning.com."
"User added something to pasteboard."
"User erased an email address from the pasteboard."
"The pasteboard is updated."
"The pasteboard is updated."
```

4.9. Цепочки опционалов

Иногда вам нужно значение из опционального свойства, которое также может содержать еще одно такое свойство.

Давайте продемонстрируем это, создав тип продукта, который The Mayonnaise Depot предлагает в своем магазине, например гигантский бочонок с майонезом. Приведенная ниже структура обозначает продукт.

Листинг 4.25. Представляем Product

```
struct Product {
   let id: String
   let name: String
   let image: UIImage?
}
```

У клиента может быть любимый продукт для специальных быстрых заказов. Добавим его в структуру `Customer`.

Листинг 4.26. Добавление favoriteProduct в структуру Customer

```
struct Customer {
   // ... Пропускаем часть кода.
   let favoriteProduct: Product?
```

Если вы хотите получить изображение из `favoriteProduct`, вам придется немного покопаться, чтобы добраться до него. Можно делать это внутри опционалов с помощью оператора ? для выполнения *опциональной цепочки*.

Кроме того, если вам нужно установить изображение для UIImageView из фреймворка UIKit, вы можете дать ему изображение продукта клиента с помощью цепочки.

Листинг 4.27. Применение цепочки опциональных вызовов

```
let imageView = UIImageView()
imageView.image = customer.favoriteProduct?.image
```

Обратите внимание, что мы использовали оператор ?, чтобы добраться до изображения внутри favoriteProduct, которое является опционалом.

Вы по-прежнему можете выполнять обычные извлечения для опционалов в цепочке, используя выражение if let, что особенно удобно, когда вы хотите выполнить какое-либо действие, когда опционал в цепочке равен nil.

В следующем листинге делается попытка отобразить изображение из продукта и вернуться к отсутствующему изображению по умолчанию, если favProduct или его свойства image равны нулю.

Листинг 4.28. Извлечение опционала в цепочке

```
if let image = customer.favoriteProduct?.image {
   imageView.image = image
} else {
   imageView.image = UIImage(named: "missing_image")
}
```

Чтобы добиться того же эффекта, можно использовать комбинацию цепочки опциональных вызовов и объединения по нулевому указателю.

Листинг 4.29. Сочетание объединения по нулевому указателю и цепочки опциональных вызовов

```
imageView.image = customer.favoriteProduct?.image ?? UIImage(named:
➥ "missing_image")
```

Цепочки опциональных вызовов не являются обязательным методом, но они помогают при сокращенном извлечении опционалов.

4.10. Ограничение опциональных логических типов

Логические типы данных – это значение, которое может быть истинным или ложным, что делает ваш код красивым и предсказуемым. Теперь приходит Swift и говорит: «Вот логический тип с тремя состояниями: false, true и nil».

Какой-то квантовый логический тип данных, который может содержать три состояния, может смутить вас. Nil – в данном случае то же самое, что и false? Зависит от контекста. Можно иметь дело с логическими типами данных в трех сценариях:

один сценарий, когда нулевое значение обозначает ложь, второй, когда оно обозначает истинное значение, и третий, когда вам явно нужно три состояния.

Какой бы подход вы ни выбрали, мы приводим здесь эти методы, чтобы убедиться, что нулевые значения не распространяются слишком глубоко в ваш код и не вызывают сумятицу.

4.10.1. Сокращение логического типа до двух состояний

Можно получить опциональный логический тип данных, например когда вы проводите синтаксический анализ данных из API, где пытаетесь прочитать логический тип, или когда извлекаете ключ из словаря.

К примеру, сервер может вернуть предпочтения, которые пользователь установил для приложения, допустим, желание автоматически войти в систему или использовать либо не использовать Apple Face ID для входа в систему, как показано в приведенном ниже примере.

Листинг 4.30. Получение опционального логического типа данных

```
let preferences = ["autoLogin": true, "faceIdEnabled": true] ❶

let isFaceIdEnabled = preferences["faceIdEnabled"] ❷
print(isFaceIdEnabled) // Опционал(истинно)
```

❶ Словарь получен от сервера.

❷ Узнаем, включен ли Face ID.

Если вы хотите трактовать ноль как `false`, может быть полезно сделать его обычным логическим типом данных, и работа с опциональным логическим типом не будет распространяться дальше по всему коду.

Это можно сделать, используя исходное значение, с помощью оператора объединения по нулевому указателю ??.

Листинг 4.31. Откат назад с помощью оператора объединения по нулевому указателю

```
let preferences = ["autoLogin": true, "faceIdEnabled": true]

let isFaceIdEnabled = preferences["faceIdEnabled"] ?? false
print(isFaceIdEnabled) // истинно, это уже не опционал.
```

При использовании этого оператора логический тип данных снова возвращается к обычному типу.

4.10.2. Откат к значению true

Контрапункт: не рекомендуется слепо возвращаться к значению `false`. В зависимости от сценария вместо этого можно использовать значение `true`.

Рассмотрим сценарий, при котором нам нужно убедиться, включен ли у клиента Face ID, чтобы мы могли направить пользователя на экран настроек Face ID. В этом случае можно вернуться к значению `true`.

Листинг 4.32. Откат к значению true

```
if preferences["faceIdEnabled"] ?? true {
   // Переход к экрану с настройками to Face ID.
} else {
   // клиент отключил Face ID
}
```

Это незначительный момент, но он показывает, что рассматривать опциональный логический тип данных, думая: «Давайте сделаем его ложным», – не всегда хорошая идея.

4.10.3. Логический тип данных с тремя состояниями

Можно добавить в опциональный логический тип данных больше контекста, если вам все же нужно три состояния. Вместо этого рассмотрим перечисление, чтобы сделать эти состояния явными.

Следуя примеру с пользовательскими предпочтениями из предыдущего раздела, мы преобразуем логический тип данных в перечисление `UserPreference` с тремя кесайми: `.enabled`, `.disabled` и `.notSet`. Это нужно для того, чтобы быть более явным в своем коде и получать преимущества на этапе компиляции.

Листинг 4.33. Преобразование логического типа данных в перечисление

```
let isFaceIdEnabled = preferences["faceIdEnabled"]
print(isFaceIdEnabled) // Optional(true)
// В данном случае мы конвертируем опциональный логический тип в перечисление.
let faceIdPreference = UserPreference(rawValue: isFaceIdEnabled) ❶

// Можно выполнить сопоставление для перечисления UserPreference.
switch faceIdPreference { ❷
   case .enabled: print("Face ID is enabled")
   case .disabled: print("Face ID is disabled")
   case .notSet: print("Face ID preference is not set")
}
```

❶ Передается логический тип данных для создания перечисления `UserPreference`.

❷ Можно выполнить сопоставление для перечисления с тремя кейсами: `.enabled`, `.disabled` или `.notSet`.

Приятным преимуществом является то, что теперь другие разработчики, реализующее это перечисление, должны явно обрабатывать все три кейса, в отличие от опционального логического типа данных.

4.10.4. Реализация протокола RawRepresentable

Можно добавить обычный инициализатор в перечисление, чтобы преобразовать его из логического типа. Тем не менее вы можете действовать в стиле Swift и реализовать протокол RawRepresentable (https://developer.apple.com/documentation/swift/rawrepresentable) в качестве соглашения.

Соответствие протоколу RawRepresentable является идиоматическим способом преобразования типа в необработанное значение и обратно и упрощает соответствие другим протоколам, таким как Equatable, Hashable и Comparable, – подробнее об этом в главе 7.

После реализации протокола RawRepresentable тип должен реализовать инициализатор rawValue, так же как и свойство rawValue, для преобразования типа в перечисление и обратно.

Вот как выглядит перечисление UserPreference:

Листинг 4.34. Перечисление UserPreference

```
enum UserPreference: RawRepresentable { ❶
   case enabled
   case disabled
   case notSet

   init(rawValue: Bool?) { ❷
      switch rawValue { ❸
         case true?: self = .enabled ❹
         case false?: self = .disabled ❹
         default: self = .notSet
      }
   }

   var rawValue: Bool? { ❺
      switch self {
         case .enabled: return true
         case .disabled: return false
         case .notSet: return nil
      }
   }
}
```

❶ Перечисление соответствует RawRepresentable.

❷ Можно инициализировать UserPreference с опциональным логическим типом данных.

❸ Причина, по которой можно использовать оператор switch в rawValue, заключается в том, что он является опционалом, а опционалы – это перечисления.

❹ Вы используете вопросительный знак для выполнения сопоставления для опционального значения.

❺ Чтобы соответствовать протоколу `RawRepresentable`, `UserPreference` также должно возвращать исходное значение `rawValue`.

Внутри инициализатора мы используем сопоставление с образцом для опционального логического типа. С помощью знака вопроса мы напрямую выполняем сопоставление для значения внутри опционала; затем следуют соответствующие кейсы.

В качестве последнего шага по умолчанию идет кейс `.notSet`, что происходит, если мы возвращаем nil.

Теперь вы ограничили логическое значение перечислением и добавили ему больше контекста. Но это обходится дорого: вы вводите новый тип, который может испортить кодовую базу. Если вы хотите, чтобы все было явно, и хотите получать преимущества на этапе компиляции, перечисление может стоить этих затрат.

4.10.5. Упражнение

4

Имеется опциональный логический тип данных:

```
let configuration = ["audioEnabled": true]
```

Создайте перечисление `AudioSetting`, которое может обрабатывать все три кейса:

```
let audioSetting = AudioSetting(rawValue: configuration["audioEnabled"])

switch audioSetting {
   case .enabled: print("Turn up the jam!")
   case .disabled: print("sshh")
   case .unknown: print("Ask the user for the audio setting")
}
```

Кроме того, убедитесь, что вы снова можете получить значение из перечисления:

```
let isEnabled = audioSetting.rawValue
```

4.11. Рекомендации по принудительному извлечению значения

Принудительное извлечение значения означает, что вы извлекаете опционал, не проверяя, существует ли значение. Извлекая опционал принудительно, вы добираетесь до извлекаемого значения, чтобы использовать его. Если у опционала есть значение, это здорово. Однако если он пустой, приложение рушится, а это уже не так здорово.

Возьмем, к примеру, тип URL из Foundation. Он принимает параметр String в своем инициализаторе. Затем URL либо создается, либо нет, в зависимости от того, является ли переданная строка правильным путем – следовательно, инициализатор URL может вернуть nil.

Листинг 4.35. Создание URL

```
let optionalUrl = URL(string: "https://www.themayonnaisedepot.com")
➥ // Опционал (http://www.themayonnaisedepot.com)
```

Вы можете принудительно извлечь опционал, используя восклицательный знак, минуя любые безопасные методы.

Листинг 4.36. Принудительное извлечение URL

```
let forceUnwrappedUrl = URL(string: "https://www.themayonnaisedepot.com")!
➥ // http://www.themayonnaisedepot.com. Обратите внимание, что мы
   используем восклицательный знак для принудительного извлечения.
```

Теперь вам больше не нужно извлекать опционал. Но принудительное извлечение приводит к сбою приложения на неверном пути.

Листинг 4.37. Сбой

```
let faultyUrl = URL(string: «mmm mayonnaise»)! ❶
```

❶ Сбой – URL нельзя создать, и он извлечен принудительно.

4.11.1. Когда принудительное извлечение значения является «приемлемым»

В идеале вы никогда бы не стали использовать принудительное извлечение. Но иногда этого нельзя избежать, потому что ваше приложение может оказаться в плохом состоянии. Тем не менее подумайте об этом: неужели нет другого способа предотвратить принудительное извлечение? Возможно, вместо этого можно вернуть nil или выбросить ошибку.

Используйте принудительное извлечение значения только в качестве крайней меры и рассмотрите следующие исключения.

Отсрочка обработки ошибок

Наличие обработки ошибок поначалу может замедлить вас. Когда ваши функции могут выбросить ошибку, код, вызывающий функцию, должен будет иметь дело с ошибкой, которая является дополнительной работой и логикой, для реализации которой требуется время.

Одной из причин применения принудительного извлечения является быстрое создание рабочего фрагмента кода, например создание прототипа или созданного наспех сценария на Swift. Тогда вы можете беспокоиться об обработке ошибок позже.

Можно использовать принудительное извлечение, чтобы запустить приложение, а затем рассматривать его как маркер, чтобы заменить его надлежащей обработкой ошибок. Но мы с вами знаем, что в программировании фраза «я сделаю это позже» означает «я никогда этого не сделаю», поэтому примите этот совет с недоверием.

Когда вы знаете лучше, чем компилятор

Если вы имеете дело со значением, которое зафиксировано на этапе компиляции, принудительное извлечение опционала может иметь смысл.

В приведенном ниже листинге вы знаете, что здесь есть переданный URL и что будет выполнен синтаксический анализ, хотя компилятор еще этого не знает. Принудительное извлечение в данном случае безопасно.

Листинг 4.38. Принудительное извлечение действительного URL

```
let url = URL(string: "http://www.themayonnaisedepot.com")!
// http://www.themayonnaisedepot.com
```

Но если этот URL является загружаемой во время выполнения переменной, например пользовательским вводом, нельзя гарантировать безопасное значение, и в этом случае если вы принудительно извлечете его, то рискуете аварийно завершить работу.

4.11.2. Аварийный сбой со стилем

Иногда нельзя избежать сбоя. Тогда вы, возможно, захотите предоставить больше информации, вместо того чтобы выполнять принудительное извлечение.

Например, представьте, что вы создаете URL-адрес из пути, который получаете во время выполнения, – это означает, что переданный путь не гарантированно пригоден для использования – и приложение по какой-то причине не может продолжить работу, если этот URL-адрес недействителен, а означает, что тип URL возвращает nil.

Вместо принудительного извлечения этого типа вы можете выбрать сбой вручную и добавить дополнительную информацию, которая поможет вам при отладке в журналах.

Можно сделать это, вручную вызвав сбой в работе приложения с помощью функции `fatalError`. Затем вы можете предоставить дополнительную информацию, например `#line` и `#function`, которая предоставляет точную информацию о сбое приложения.

В приведенном ниже листинге сперва вы пытаетесь извлечь URL, используя оператор guard. Но если извлечение завершается неудачно, вы вручную вызываете `fatalError` с дополнительной информацией, которая может помочь вам во время отладки.

Листинг 4.39. Вызов аварийного сбоя вручную

```
guard let url = URL(string: path) else {
   fatalError("Could not create url on \(#line) in \(#function)")
}
```

Важное предостережение: если, например, вы создаете приложение под iOS, ваши пользователи могут увидеть конфиденциальную информацию, которую вы указали в журнале сбоев. Что вы добавляете в сообщение `fatalError`, то должны решать сами в каждом конкретном случае.

4.12. Приручаем неявно извлекаемые опционалы

Неявно извлекаемые опционалы коварны, потому что это единственные опционалы, которые извлекаются автоматически в зависимости от их контекста. Но если вы не будете осторожны, они могут привести к аварийному сбою вашего приложения!

В этом разделе рассказывается о том, как подчинить их своей воле и убедиться, что они не причинят вам вреда.

4.12.1. Как распознать неявно извлекаемый опционал

Когда идет речь о неявно извлекаемом опционале, мы знаем, что речь идет о неявном извлечении типа, а не экземпляра.

Листинг 4.40. Знакомство с неявно извлекаемым опционалом

```
let lastName: String! = "Smith" ❶
let name: String? = 3
let firstName = name! ❷
```

❶ Неявно извлекаемый опционал.

❷ Принудительно извлекаемый экземпляр.

Можно распознать неявно извлекаемый опционал по восклицательному знаку (!) после типа, например `String!`. Рассматривайте их как *предварительно извлекаемые* опционалы.

Как и принудительное извлечение, знак ! указывает на опасность в Swift. Неявно извлекаемые опционалы также могут привести к аварийному сбою вашего приложения, в чем вы вскоре убедитесь.

4.12.2. Неявно извлекаемые опционалы на практике

Неявно извлекаемые опционалы похожи на дрель. Вы можете редко использовать их, но они пригодятся, когда вам понадобятся. Однако если вы совершите ошибку, они могут испортить ваш фундамент.

При создании неявно извлекаемого опционала вы обещаете, что переменная или константа заполняется вскоре *после* инициализации, но *до* того, как они вам понадобятся. Когда это обещание нарушается, люди получают травмы (ну, технически приложение может аварийно завершиться). Но катастрофа случается, когда ваше приложение управляет атомными электростанциями или помогает стоматологам подавать пациентам веселящий газ.

Вернемся к The Mayonnaise Depot. Интернет-магазин решил добавить в свой бэкэнд чат, чтобы клиенты могли обращаться за помощью при заказе продуктов.

При запуске сервера базы данных сервер в первую очередь запускает монитор процессов. Этот монитор гарантирует, что система готова, прежде чем другие службы будут инициализированы и запущены. Это означает, что вам нужно будет запустить сервер чата после монитора процессов. После того как монитор готов, ему передается сервер чата (см. рис. 4.2).

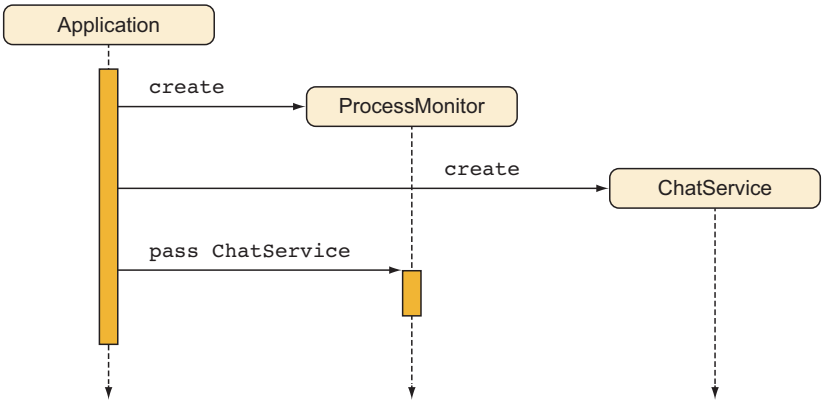


Рис. 4.2. Запуск монитора процессов

Поскольку сервер в первую очередь инициирует ProcessMonitor, у ProcessMonitor может быть опциональная ссылка на службу чата. В результате ChatService может быть передан ProcessMonitor позже. Но делать сервер чата опциональным в мониторе процессов неудобно, потому что для доступа к этому серверу вам придется каждый раз извлекать службу чата, зная, что у монитора процессов есть действительная ссылка.

Вы также можете сделать службу чата ленивым свойством в ProcessMonitor, но тогда ProcessMonitor будет отвечать за инициализацию ChatService. В этом случае ProcessMonitor не захочет обрабатывать возможные зависимости ChatService.

Неплохой сценарий для неявно извлекаемого опционала. Делая службу чата неявно извлекаемым опционалом, вам не нужно передавать ее инициализатору монитора процессов и также не нужно делать ее опционалом.

Создание неявно извлекаемого опционала

В приведенном далее листинге содержится код для ChatService и ProcessMonitor. У ProcessMonitor есть метод start() для создания монитора и метод status(), чтобы проверить, все ли еще работает.

Листинг 4.41. Знакомство с ChatService и ProcessMonitor

```
class ChatService {
   var isHealthy = true
   // Оставим его пустым в демонстрационных целях
}

class ProcessMonitor {
   var chatService: ChatService! ❶

   class func start() -> ProcessMonitor {
      // В реальном приложении: проводим детальную диагностику.
      return ProcessMonitor()
   }
```

```
    func status() -> String {
        if chatService.isHealthy {
            return "Everything is up and running"
        } else {
            return "Chatservice is down!" }
        }
    }
}
```


❶ chatService – неявно извлекаемый опционал, который можно узнать по знаку !.

Процесс инициализации запускает монитор, затем службу чата, а после этого, наконец, передает службу монитору.

Листинг 4.42. Процесс инициализации

```
let processMonitor = ProcessMonitor.start()
// Здесь processMonitor проводит важную диагностику.
// processMonitor готов.

let chat = ChatService() // Запускаем Chatservice.

processMonitor.chatService = chat
processMonitor.status() // "Все работает"
```

Таким образом, сначала можно запустить processMonitor, но у вас есть преимущество, заключающееся в том, что chatService доступен для processMonitor как раз перед тем, как он вам понадобится.

Но chatService – это неявно извлекаемый опционал, а эти опционалы могут быть опасными. Если по какой-либо причине вы получили доступ к свойству chatService до того, как оно перешло к processMonitor, это приведет к аварийному сбою.

Листинг 4.43. Аварийный сбой

```
let processMonitor = ProcessMonitor.start()
processMonitor.status() // Фатальная ошибка: неожиданно обнаружено nil.
```

Когда вы делаете chatService неявно извлекаемым опционалом, вам не нужно инициализировать его с помощью инициализатора и также не нужно извлекать его каждый раз, когда нужно прочитать значение. Это беспроигрышный вариант с добавленной опасностью. По мере того как вы будете больше работать со Swift, вы найдете другие способы избавиться от неявно извлекаемых опционалов, потому что они представляют собой обоюдоострый меч. Например, можно передать ChatServiceFactory в ProcessMonitor, который может создать сервер чата для ProcessMonitor. При этом ProcessMonitor не нужно будет знать о зависимостях.

4.12.3. Упражнение

5

Назовите подходящие альтернативы неявно извлекаемым опционалам.

4.13. В заключение

В этой главе было рассмотрено множество сценариев, включающих в себя использование опционалов. Прохождение всех этих методов было нелегким делом – так что можете гордиться собой!

Чувствовать себя комфортно при работе с опционалами очень важно для программиста, работающего с Swift. Овладение опционалами помогает сделать правильный выбор среди изобилия ситуаций, с которыми вы сталкиваетесь при повседневном программировании на языке Swift.

Последующие главы раскрывают тему опционалов, где рассматривается применение `map`, `flatMap` и `compactMap`. После того как вы поработаете над этими сложными темами, вы постигните истину и будете обрабатывать любую уловку, связанную с опционалами.

Резюме

- Опционалы – это перечисления с синтаксическим сахаром.
- Можно использовать сопоставление с образцом для опционалов.
- Используйте сопоставление с образцом для нескольких опционалов одновременно, помещая их внутрь кортежа.
- Можно использовать объединение по нулевому указателю, чтобы вернуться к значениям по умолчанию.
- Используйте цепочки опционалов, чтобы углубиться в опциональные значения.
- Вы можете использовать объединение по нулевому указателю для преобразования опционального логического типа данных в обычный.
- Вы можете преобразовывать опциональный логический тип данных в перечисление для трех явных состояний.
- Возвращайте опциональные строки вместо пустых строк, когда ожидается значение.
- Используйте принудительное извлечение, только если ваша программа не может вернуться в исходное состояние после того, как вы наблюдаете значение nil.
- Используйте принудительное извлечение, если хотите отложить обработку ошибок, например при создании прототипов.
- Безопаснее принудительно извлекать опционалы, если вы знаете лучше, чем компилятор.
- Используйте неявно извлекаемые опционалы для свойств, которые создаются сразу после инициализации.

Ответы

1

Если опционалам в функции запрещено иметь значение, какая тактика подойдет для того, чтобы убедиться, что все опционалы заполнены?

Используйте оператор guard – это может заблокировать дополнительные функции в верхней части функции.

2

Если функции выбирают разные пути в зависимости от опционалов внутри них, какой подход следует выбрать для обработки этих путей?

Помещение нескольких опционалов в кортеж позволяет выполнить сопоставление с образцом и использовать разные пути в функции.

3

Код выглядит так:

```
// Используем единичный оператор switch внутри descriptionAction.
func describeAction(event: PasteBoardEvent?, contents:
   PasteBoardContents?) -> String {
      switch (event, contents) {
         case let (.added?, .url(url)?): return "User added a url to
            pasteboard: \(url)"
         case (.added?, _): return "User added something to pasteboard"
         case (.erased?, .emailAddress?): return "User erased an email
            address from the pasteboard"
         default: return "The pasteboard is updated"
      }
   }
```

4

Код выглядит так:

```
enum AudioSetting: RawRepresentable {
   case enabled
   case disabled
   case unknown

   init(rawValue: Bool?) {
      switch rawValue {
         case let isEnabled? where isEnabled: self = .enabled
         case let isEnabled? where !isEnabled: self = .disabled
         default: self = .unknown
      }
   }
```

```
    var rawValue: Bool? {
       switch self {
          case .enabled: return true case .disabled: return false
          case .unknown: return nil
       }
    }
}
```

5

Назовите подходящие альтернативы неявно извлекаемым опционалам.

Ленивые свойства или фабрики, которые передаются через инициализатор.

Глава 5. Разбираемся с инициализаторами

В этой главе:

- разбор правил инициализатора в Swift;
- странности структурных инициализаторов;
- сложные правила инициализатора при создании подклассов;
- как поддерживать низкое число инициализаторов при создании подклассов;
- когда и как работать с требуемыми инициализаторами.

Поскольку вы являетесь разработчиком на Swift, инициализация ваших классов и структур – один из основных принципов, которые вы используете.

Однако инициализаторы в Swift не интуитивны. Swift предлагает почленные, пользовательские, назначенные, вспомогательные, требуемые, не говоря уже об опциональных, проваливающихся и генерирующих инициализаторах. Честно говоря, иногда это может сбивать с толку.

Данная глава проливает свет на ситуацию, когда, вместо того чтобы устраивать поединок с компилятором, можно извлечь максимальную пользу из инициализации структур, классов и подклассов.

В этой главе мы смоделируем иерархию настольной игры, которую составим из структур и классов. При построении этой иерархии вы испытаете радость от странных правил инициализации Swift и от того, как работать с ними. Поскольку создание игровой механики – это тема для отдельной книги, в этой главе мы сосредоточимся только на основах инициализаторов.

Вначале вы поработаете с инициализаторами структуры и узнаете о странностях, которые с ними связаны. После этого вы перейдете к инициализаторам классов и сопровождающим их правилам создания подклассов; обычно это то место, где появляются сложности при работе в Swift. Затем увидите, как уменьшить количество инициализаторов во время создания подклассов, чтобы свести это количество к минимуму. Наконец, увидите, какую роль играют требуемые инициализаторы, а также когда и как их использовать.

Цель этой главы – научить вас писать инициализаторы за один раз, вместо того чтобы совершать неловкий танец, состоящий из проб и ошибок, с целью угодить богам компилятора.

5.1. Правила инициализаторов структуры

Структуры можно инициализировать относительно простым способом, потому что их нельзя делить на подклассы. Тем не менее существуют специальные правила, применимые к структурам, которые мы рассмотрим в этом разделе. В следующем разделе мы будем моделировать настольную игру, но сначала смоделируем игроков, которые могут играть в эту игру. Мы создадим структуру `Player`,

содержащую имя и тип пешки каждого игрока. Мы решили смоделировать Player как структуру, потому что структуры хорошо подходят для небольших моделей данных (среди прочего). Кроме того, структуру нельзя разделить на подклассы, что прекрасно подходит для моделирования Player.

Присоединяйтесь!

Будет более познавательно и веселее, если вы будете знакомиться с кодом по мере прохождения этой главы. Исходный код можно скачать по адресу http://mng.bz/nQE5.

Player (игрок), которого мы будем моделировать, напоминает пешку в настольной игре. Можно создать игрока, передав ему имя и тип пешки, например машина (car), ботинок (shoe) или шляпа (hat).

Листинг 5.1. Создание игрока

```
let player = Player(name: «SuperJeff», pawn: .shoe)
```

Видно, что у игрока есть два свойства: name и pawn. Обратите внимание, что в структуре не определен инициализатор. Внутри мы получаем так называемый *почленный инициализатор*. Это инициализатор, который компилятор генерирует за нас, как показано здесь.

Листинг 5.2. Структура Player

```
enum Pawn {
   case dog, car, ketchupBottle, iron, shoe, hat
}

struct Player {
   let name: String
   let pawn: Pawn
}

let player = Player(name: "SuperJeff", pawn: .shoe) ❶
```

❶ Swift предлагает почленный инициализатор.

5.1.1. Пользовательские инициализаторы

Swift – очень строгий язык, когда речь заходит о желании заполнить все свойства структурами и классами, что не секрет. Если до этого вы работали с языками, где это не так (например, Ruby и Objective-C), борьба с компилятором Swift поначалу может разочаровать вас.

Например, нельзя инициализировать Player, используя только имя, без пешки. Приведенный ниже код не будет компилироваться.

Листинг 5.3. Опускаем свойство

```
let player = Player(name: "SuperJeff") error: missing argument for
```

```
parameter 'pawn' in call
let player = Player(name: "SuperJeff")
^
, pawn: Pawn
```

Чтобы упростить инициализацию, можно убрать pawn из параметров инициализатора. Структуре нужны все свойства, распространяемые со значением. Если структура инициализирует свои свойства, вам не нужно передавать значения. В листинге 5.4 мы добавим пользовательский инициализатор, где можно передать только имя игрока. Затем можно случайным образом выбрать пешку для игроков, что упрощает инициализацию вашей структуры. Пользовательский инициализатор принимает имя и случайным образом выбирает пешку.

Сперва убедитесь, что перечисление Pawn соответствует протоколу CaseIterable; это позволяет вам получить массив всех кейсов посредством свойства allCases. Затем в инициализаторе Player вы можете использовать метод randomElement() в allCases, чтобы выбрать случайный элемент.

Примечание

CaseIterable работает только для перечислений без ассоциированных значений, потому что с ними перечисление теоретически может иметь бесконечное число вариаций.

Листинг 5.4. Создание инициализатора

```
enum Pawn: CaseIterable { ❶
case dog, car, ketchupBottle, iron, shoe, hat
}

struct Player {
   let name: String
   let pawn: Pawn

   init(name: String) { ❷
      self.name = name
      self.pawn = Pawn.allCases.randomElement()! ❸
   }
}

// Пользовательский инициализатор в действии.
let player = Player(name: "SuperJeff")
print(player.pawn) // shoe
```

❶ Делаем так, чтобы Pawn соответствовал CaseIterable, что дает вам свойство allCases в Pawn, возвращая массив всех кейсов.

❷ Вручную создаем инициализатор, принимая имя String.

❸ Выполняем рандомизацию пешки с помощью метода randomElement() в Pawn.allCases.

Примечание

Метод `randomElement()` возвращает опционал, который необходимо извлечь. Вы принудительно извлекаете его с помощью `!`. Обычно принудительное извлечение считается плохой практикой. Но в данном случае на этапе компиляции вы знаете, что извлечение безопасно.

Теперь нерешительные игроки могут выбрать себе пешку.

5.1.2. Странность инициализатора структуры

Вот интересная странность. Нельзя использовать почленный инициализатор из предыдущего примера; он не сработает.

Листинг 5.5. Инициализация игрока с использованием пользовательского инициализатора

```
let secondPlayer = Player(name: "Carl", pawn: .dog)
error: extra argument 'pawn' in call
```

Причина, по которой почленный инициализатор больше не работает, состоит в том, что нужно убедиться, что разработчики не могут обойти логику в пользовательском инициализаторе. Это полезный механизм защиты! В вашем случае было бы полезно предложить оба инициализатора: пользовательский и почленный. Можно предложить оба инициализатора, расширив структуру и поместив туда свой пользовательский инициализатор.

Вначале мы восстанавливаем структуру `Player`, чтобы в ней не было никаких пользовательских инициализаторов, возвращая себе почленный инициализатор обратно. Затем мы расширяем структуру и помещаем туда свой пользовательский инициализатор.

Листинг 5.6. Восстановление структуры Player

```
struct Player {
   let name: String
   let pawn: Pawn
}

extension Player { ❶
   init(name: String) { ❷
      self.name = name
      self.pawn = Pawn.allCases.randomElement()!
   }
}
```

❶ Расширяем структуру `Player`, открывая ее для большей функциональности.

❷ Добавляем пользовательский инициализатор в расширение.

Это сработало, потому что теперь мы можем инициализировать игрока с помощью обоих инициализаторов.

Листинг 5.7. Инициализация игрока с обоими инициализаторами

```
// Теперь работают оба инициализатора.
let player = Player(name: "SuperJeff")
let anotherPlayer = Player(name: "Mary", pawn: .dog)
```

Используя расширение, можно сохранить лучшее из обоих миров. Вы можете использовать почленный инициализатор и пользовательский инициализатор с определенной логикой.

5.1.3. Упражнения

1

Имеется следующая структура:

```
struct Pancakes {

   enum SyrupType {
      case corn
      case molasses
      case maple
   }

   let syrupType: SyrupType
   let stackSize: Int

   init(syrupType: SyrupType) {
      self.stackSize = 10
      self.syrupType = syrupType
   }
}
```

Будут ли работать эти инициализаторы:

```
let pancakes = Pancakes(syrupType: .corn, stackSize: 8)
let morePancakes = Pancakes(syrupType: .maple)
```

2

Если эти инициализаторы не работают, можно ли заставить их работать, не добавляя еще один инициализатор?

5.2. Инициализаторы и подклассы

Создание подклассов – способ достижения полиморфизма. С помощью полиморфизма вы можете предложить один интерфейс, например функцию, которая работает с несколькими типами.

Но создание подклассов не слишком популярно в сообществе Swift, особенно потому, что Swift часто рекламируется как протокольно-ориентированный язык, который является одной из альтернатив подклассов.

Вы видели в главе 2, что у подклассов есть тенденция становиться жесткой структурой данных, а также видели, что перечисления являются гибкой альтернативой подклассам. Вы встретитесь с более гибкими подходами, когда начнете работать с протоколами и обобщениями.

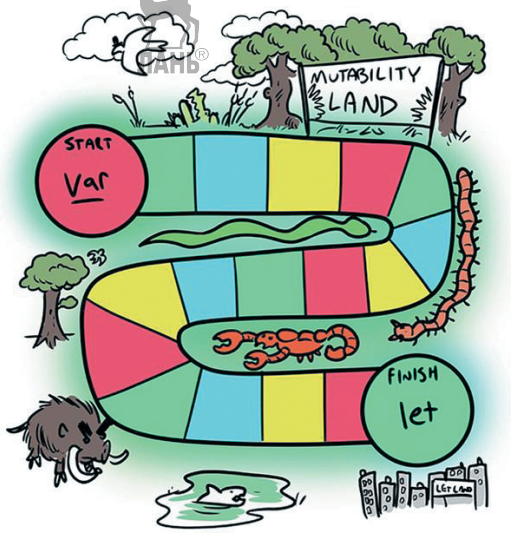
Тем не менее создание подклассов – все еще действенный инструмент, который предлагает Swift. Вы можете рассмотреть переопределение или расширение поведения из классов, что включает в себя код из фреймворков, которых у вас даже нет. UIKit от компании Apple – типичный тому пример. Здесь вы можете создать подкласс `UIView` для создания новых элементов экрана.

Этот раздел поможет вам понять, как инициализаторы работают с классами и подклассами, – что также известно как наследование, – поэтому, в случае если вы не переходите на сторону протокольно-ориентированного программирования, можете предложить чистые конструкции с использованием подклассов.

5.2.1. Создание суперкласса настольной игры

Модель `Player` полностью настроена. Теперь пришло время приступить к моделированию иерархии настольной игры.

Сперва мы создадим класс `BoardGame`, который будет служить суперклассом. Затем вы создадите подкласс `BoardGame`, чтобы создать собственную настольную игру под названием Mutability Land. Это увлекательная игра, которая учит разработчиков на Swift писать неизменяемый код.



После настройки иерархии вы познакомитесь со странностями Swift, которые возникают при создании подклассов, и подходами, которые используются для борьбы с ними.

5.2.2. Инициализаторы BoardGame

У суперкласса BoardGame есть три инициализатора: один назначенный инициализатор и два вспомогательных для упрощения инициализации (см. рис. 5.1).

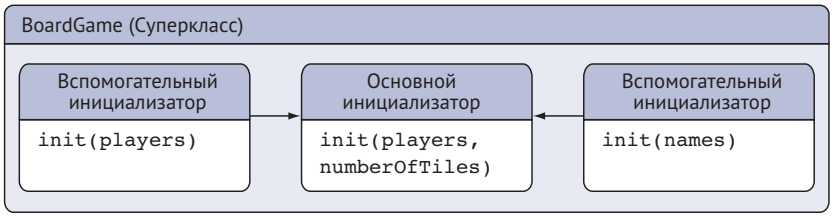


Рис. 5.1. Инициализаторы BoardGame

Прежде чем продолжить, давайте познакомимся с ними.

Первый – это *назначенный инициализатор* заурядного типа. Назначенные инициализаторы предназначены для обеспечения инициализации всех свойств. Классы, как правило, имеют очень мало назначенных инициализаторов, обычно только один, но возможно и большее количество. Назначенные инициализаторы указывают на суперкласс, если он есть.

Второй – это *вспомогательный инициализатор*, который может упростить инициализацию путем предоставления значений по умолчанию или создания более простого синтаксиса инициализации. Вспомогательные инициализаторы могут вызывать другие вспомогательные инициализаторы, но в конечном итоге они вызывают назначенный инициализатор из одного и того же класса.

Заглянув внутрь BoardGame, вы убедитесь в использовании одного основного и двух вспомогательных инициализаторов, как показано здесь.

Листинг 5.8. Суперкласс BoardGame

```
class BoardGame {
   let players: [Player]
   let numberOfTiles: Int

   init(players: [Player], numberOfTiles: Int) { ❶
      self.players = players
      self.numberOfTiles = numberOfTiles
   }

   convenience init(players: [Player]) { ❷
      self.init(players: players, numberOfTiles: 32)
   }

   convenience init(names: [String]) { ❸
      var players = [Player]()
      for name in names {
         players.append(Player(name: name))
      }
```

```
        self.init(players: players, numberOfTiles: 32)
    }
}
```

❶ Назначенный инициализатор.

❷ Вспомогательный инициализатор принимает игроков.

❸ Вспомогательный инициализатор преобразует строки в игроков.

Примечание

В качестве альтернативы также можно добавить к назначенному инициализатору значение по умолчанию numberOfTiles, например `init(players: [Player], numberOfTiles: Int = 32)`. Тем самым вы можете избавиться от одного из вспомогательных инициализаторов. В качестве примера мы продолжим работать с двумя отдельными вспомогательными инициализаторами.

У BoardGame есть два свойства: массив players, в котором содержатся все игроки, и целочисленное значение numberOfTiles, указывающее размер настольной игры.

Вот несколько способов инициализации суперкласса BoardGame.

Листинг 5.9. Инициализация BoardGame

```
// Вспомогательный инициализатор
let boardGame = BoardGame(names: ["Melissa", "SuperJeff", "Dave"])

let players = [
    Player(name: "Melissa"),
    Player(name: "SuperJeff"),
    Player(name: "Dave")
]

// Вспомогательный инициализатор
let boardGame = BoardGame(players: players)
// Назначенный инициализатор
let boardGame = BoardGame(players: players, numberOfTiles: 32)
```

Вспомогательные инициализаторы принимают только массив игроков или массив имен, которые BoardGame превращает в игроков. Эти вспомогательные инициализаторы указывают на основной инициализатор.

Отсутствие почленных инициализаторов

К сожалению, классы не получают почленных инициализаторов, в отличие от структур. В случае с классами вам придется вводить их вручную.

5.2.3. Создание подкласса

Теперь, когда вы создали BoardGame, можете приступить к созданию подклассов для создания настольных игр. В этой главе мы создадим только одну игру, которая точно не гарантирует установку суперкласса. Теоретически вы можете создать множество настольных игр, которые будут делить BoardGame на подклассы.

Подкласс носит название MutabilityLand и наследует от класса BoardGame (см. рис. 5.2). MutabilityLand наследует все инициализаторы от BoardGame.

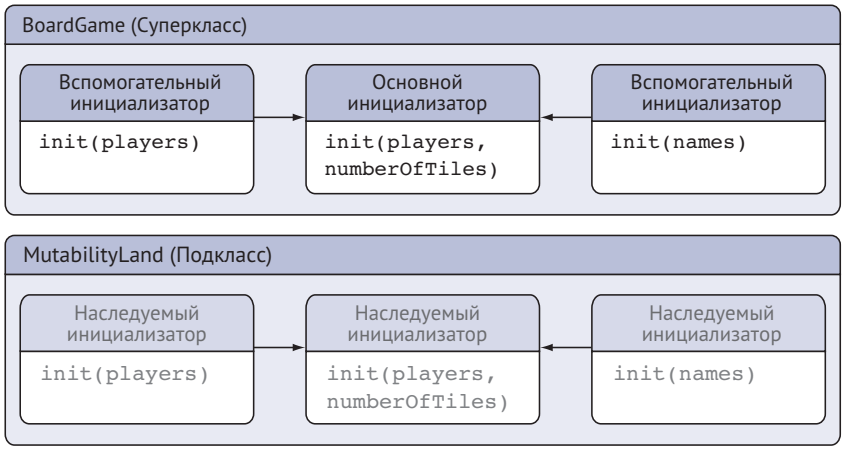


Рис. 5.2. Деление BoardGame на подклассы

Как показано в листинге 5.10, вы можете инициализировать MutableLand так же, как BoardGame, поскольку он наследует все инициализаторы, которые BoardGame может предложить.

Листинг 5.10. MutabilityLand наследует все инициализаторы BoardGame

```
// Вспомогательный инициализатор
let mutabilityLand = MutabilityLand(names: ["Melissa", "SuperJeff", "Dave"])
// Вспомогательный инициализатор
let mutabilityLand = MutabilityLand(players: players)
// Назначенный инициализатор
let mutabilityLand = MutabilityLand(players: players, numberOfTiles: 32)
```

Заглянув внутрь MutabilityLand, вы увидите, что у него есть два собственных свойства: scoreBoard и winner. scoreBoard отслеживает имя каждого игрока и его счет. Свойство winner запоминает последнего победителя в игре.

Листинг 5.11. Класс MutabilityLand

```
class MutabilityLand: BoardGame {
   // ScoreBoard инициализируется с пустым словарем
   var scoreBoard = [String: Int]()
   var winner: Player?
}
```

Возможно, вы удивитесь, но этим свойствам не нужен инициализатор, потому что scoreBoard уже инициализировано за пределами инициализатора, а winner – это опционал, который может равняться nil.

5.2.4. Потеря вспомогательных инициализаторов

Именно здесь процесс усложняется: *как только подкласс добавляет незаполненные свойства, потребители подкласса теряют ВСЕ инициализаторы суперкласса*.

Давайте посмотрим, как это работает. Сперва мы добавим в MutabilityLand свойство instructions, которое сообщает игрокам правила игры.

На рис. 5.3 показана текущая иерархия с новой настройкой. Обратите внимание, что наследуемые инициализаторы исчезли, когда вы добавили новое свойство (instructions).

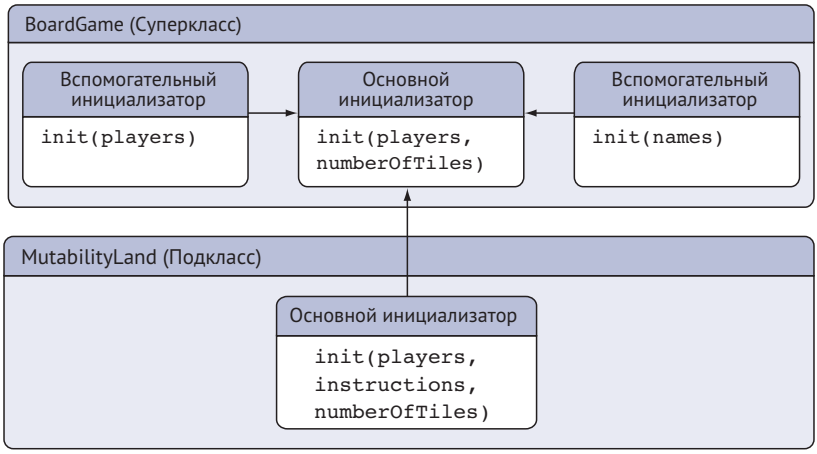


Рис. 5.3. Исчезновение инициализаторов

Чтобы заполнить новое свойство instructions, мы создадим для этого назначенный инициализатор. Давайте посмотрим, как это выглядит в коде.

Листинг 5.12. Создание назначенного инициализатора для MutabilityLand

```
class MutabilityLand: BoardGame {
   var scoreBoard = [String: Int]()
   var winner: Player?

   let instructions: String ❶

   init(players: [Player], instructions: String, numberOfTiles: Int) { ❷
      self.instructions = instructions
      super.init(players: players, numberOfTiles: numberOfTiles) ❸
   }
}
```

❶ Новое свойство instructions.

❷ Новый назначенный инициализатор для инстанциирования инструкций.

❸ Основной инициализатор вызывает инициализатор настольной игры.

На этом этапе MutableLand потерял три наследуемых инициализатора из своего суперкласса Boardgame.

Вы больше не можете инициализировать MutableLand с наследуемыми инициализаторами.

Листинг 5.13. Наследуемые инициализаторы больше не работают

```
// Они больше не работают
let mutabilityLand = MutabilityLand(names: ["Melissa", "SuperJeff", "Dave"])
let mutabilityLand = MutabilityLand(players: players)
let mutabilityLand = MutabilityLand(players: players, numberOfTiles: 32)
```

Чтобы убедиться, что мы потеряли наследуемые инициализаторы, попробуем создать экземпляр MutableLand с инициализатором из BoardGame – только на этот раз мы получим ошибку.

Листинг 5.14. Потеря инициализаторов суперкласса

```
error: missing argument for parameter 'instructions' in call
let mutabilityLand = MutabilityLand(names: ["Melissa", "SuperJeff", "Dave"])
^

,
```

Потеря наследуемых инициализаторов может показаться странной, но есть законная причина, по которой MutabilityLand их теряет. Наследуемые инициализаторы пропали, потому что BoardGame не может заполнить новое свойство своего подкласса instructions. Более того, поскольку Swift хочет, чтобы все свойства были заполнены, теперь он может полагаться только на вновь назначенный инициализатор в MutabilityLand.

5.2.5. Возвращение инициализаторов суперкласса

Существует способ вернуть все инициализаторы суперкласса, чтобы MutabilityLand мог пользоваться инициализаторами из BoardGame.

Переопределив назначенный инициализатор из суперкласса, подкласс возвращает инициализаторы суперкласса обратно. Другими словами, MutabilityLand переопределяет назначенный инициализатор из BoardGame, чтобы вернуть вспомогательные инициализаторы обратно. Назначенный инициализатор из MutabilityLand по-прежнему указывает на назначенный инициализатор из BoardGame (см. рис. 5.4).

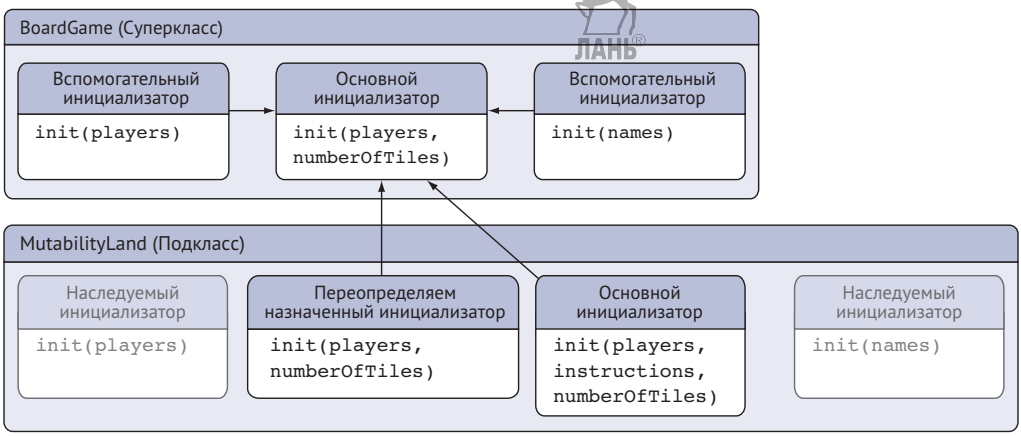


Рис. 5.4. Исчезновение инициализаторов

Переопределяя назначенный инициализатор суперкласса, MutabilityLand возвращает все вспомогательные инициализаторы обратно из BoardGame. Переопределение инициализатора достигается путем добавления ключевого слова override в назначенный инициализатор в MutabilityLand. Код можно найти в листинге 5.15.

Воронки назначенного инициализатора

Видно, что назначенные инициализаторы похожи на воронки. В иерархии классов вспомогательные инициализаторы идут горизонтально, а назначенные инициализаторы – вертикально.

Листинг 5.15. MutabilityLand переопределяет назначенный инициализатор

```
class MutabilityLand: BoardGame {
   // ... Пропускаем часть кода.
   override init(players: [Player], numberOfTiles: Int) {
      self.instructions = "Read the manual" ❶
      super.init(players: players, numberOfTiles: numberOfTiles)
   }
}
```

❶ Необходимо инициализировать свойства MutableLand перед вызовом super.init. Обратите внимание, что здесь вы даете instructions значение по умолчанию.

Поскольку вы переопределяете инициализатор суперкласса, MutabilityLand должен придумать свои инструкции. Теперь, когда назначенный инициализатор из BoardGame переопределен, вам есть из чего выбрать при инициализации MutabilityLand.

Листинг 5.16. Все доступные инициализаторы для MutabilityLand

```
//Инициализатор MutabilityLand
```

```
let mutabilityLand = MutabilityLand(players: players, instructions: "Just
➥ read the manual", numberOfTiles: 40)

//Все инициализаторы BoardGame снова работают
let mutabilityLand = MutabilityLand(names: ["Melissa", "SuperJeff", "Dave"])
let mutabilityLand = MutabilityLand(players: players)
let mutabilityLand = MutabilityLand(players: players, numberOfTiles: 32)
```

Благодаря одному переопределению мы получаем все инициализаторы, которые может предложить суперкласс.

5.2.6. Упражнение

3

Приведенный ниже суперкласс под названием `Device` регистрирует устройства в офисе и отслеживает помещения, в которых эти устройства можно найти. Его подкласс `Television` – одно из таких устройств, которое делит `Device` на подклассы.

Задача состоит в том, чтобы инициализировать подкласс `Television` с помощью инициализатора `Device`. Другими словами, заставьте следующую строку кода работать, добавив где-нибудь один инициализатор:

```
let firstTelevision = Television(room: "Lobby")
let secondTelevision = Television(serialNumber: "abc")
```

Вот как выглядят классы:

```
class Device {
   var serialNumber: String
   var room: String

   init(serialNumber: String, room: String) {
      self.serialNumber = serialNumber
      self.room = room
   }

   convenience init() {
      self.init(serialNumber: "Unknown", room: "Unknown")
   }

   convenience init(serialNumber: String) {
      self.init(serialNumber: serialNumber, room: "Unknown")
   }

   convenience init(room: String) {
      self.init(serialNumber: "Unknown", room: room)
   }
}
```

```
class Television: Device {
    enum ScreenType {
        case led
        case oled
        case lcd
        case unknown
    }

    enum Resolution {
        case ultraHd
        case fullHd
        case hd
        case sd
        case unknown
    }

    let resolution: Resolution
    let screenType: ScreenType

    init(resolution: Resolution, screenType: ScreenType, serialNumber:
        String, room: String) {
        self.resolution = resolution
        self.screenType = screenType
        super.init(serialNumber: serialNumber, room: room)
    }
}
```

5.3. Минимизация инициализаторов класса

Вы видели, что у BoardGame есть один назначенный инициализатор; у его подкласса MutabilityLand их два. Если вы снова захотите разделить MutabilityLand на подклассы и добавить хранимое свойство, у этого подкласса будет три инициализатора, и т. д. При такой скорости чем больше у вас будет подклассов, тем больше инициализаторов вам придется переопределять, что усложняет иерархию. К счастью, существует решение, позволяющее сохранять небольшое количество назначенных инициализаторов, чтобы каждый подкласс содержал только один назначенный инициализатор.

5.3.1. Реализация назначенного инициализатора в качестве вспомогательного с использованием ключевого слова override

В предыдущем разделе MutabilityLand переопределял назначенный инициализатор из класса BoardGame. Но есть одна хитрость, которая состоит в том, чтобы

превратить переопределяемый инициализатор из MutabilityLand во вспомогательный, используя ключевое слова override (см. рис. 5.5).

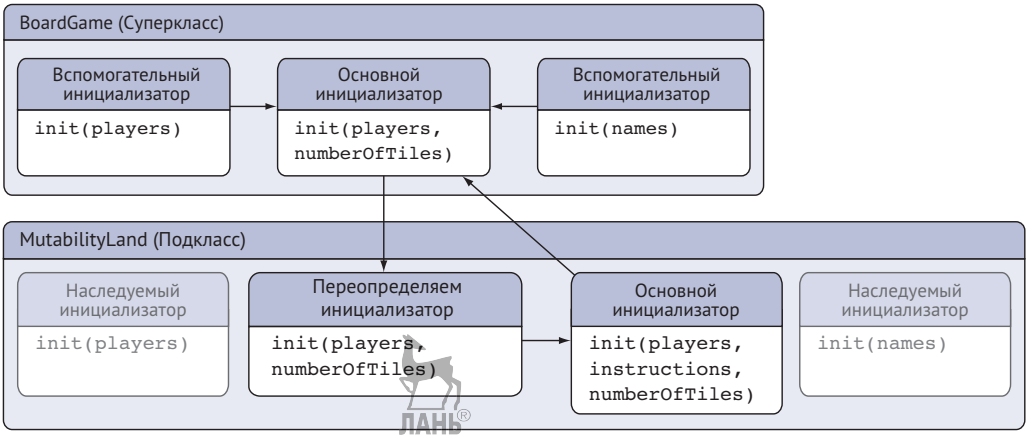


Рис. 5.5. MutabilityLand выполняет переопределение

Теперь переопределяющий инициализатор в MutabilityLand – это инициализатор, который косвенно указывает на назначенный инициализатор внутри MutabilityLand. Этот назначенный инициализатор от MutabilityLand по-прежнему указывает вверх на назначенный инициализатор внутри BoardGame.

В нашем классе это можно реализовать, используя ключевые слова convenience override.

Листинг 5.17. Переопределение с использованием ключевых слов convenience override

```
class MutabilityLand: BoardGame {
   var scoreBoard = [String: Int]()
   var winner: Player?

   let instructions: String

   convenience override init(players: [Player], numberOfTiles: Int) { ❶
      self.init(players: players, instructions: «Read the manual»,
      numberOfTiles: numberOfTiles) ❷
   }

   init(players: [Player], instructions: String, numberOfTiles: Int) { ❸
      self.instructions = instructions
      super.init(players: players, numberOfTiles: numberOfTiles)
   }
}
```

❶ Теперь это переопределяющий вспомогательный инициализатор.

❷ Инициализатор теперь указывает в сторону (self.init), а не вверх (super.init).

❸ Оставляем назначенный инициализатор как есть.

Поскольку переопределяющий инициализатор теперь является вспомогательным, он горизонтально указывает на назначенный инициализатор в том же классе. Таким образом, `MutabilityLand` идет от двух назначенных инициализаторов к вспомогательному инициализатору и одному назначенному. Любому подклассу теперь нужно переопределять только один назначенный инициализатор, чтобы получить все инициализаторы из суперкласса.

Недостатком является то, что этот подход не так гибок. Например, вспомогательный инициализатор теперь должен выяснить, как заполнить свойство instructions. Но если в вашем коде работает переопределение, о котором идет речь в этом разделе, количество назначенных инициализаторов сокращается.

5.3.2. Деление подкласса на подклассы

Чтобы убедиться, что вы можете сохранить небольшое количество назначенных инициализаторов, мы введем из `MutabilityLand` подкласс для детей с именем `MutabilityLandJunior`. Эта игра немного проще, и в ней есть возможность воспроизводить звуки, обозначенные новым свойством `soundsEnabled`.

Благодаря трюку, описанному в предыдущем разделе, этот новый подкласс должен переопределять только один назначенный инициализатор. Иерархия показана на рис. 5.6.

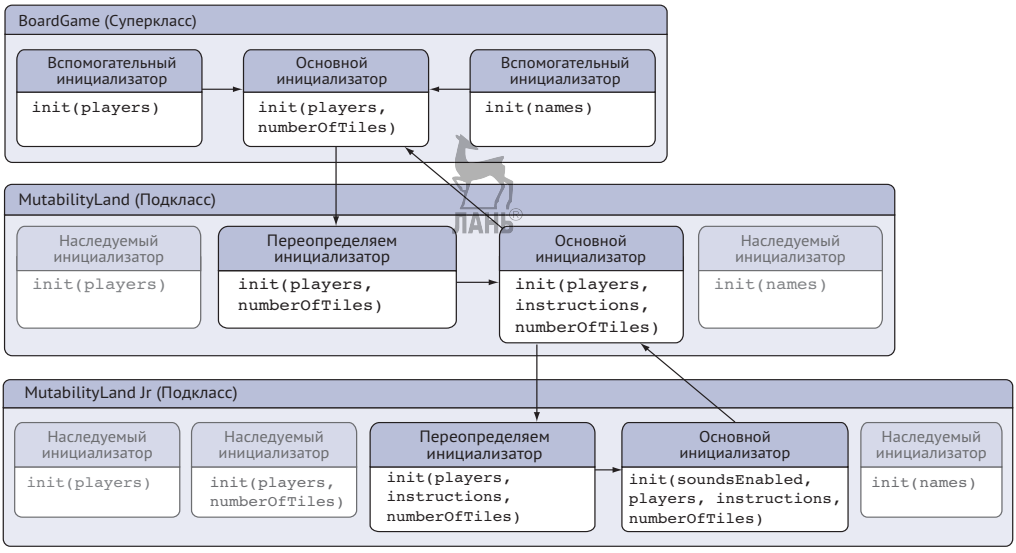


Рис. 5.6. MutabilityLandJunior нужно переопределить только один инициализатор

Видно, что этому подподклассу нужно переопределить только один инициализатор, чтобы наследовать все инициализаторы. По привычке этот инициализатор также относится к типу, о котором идет речь в разделе 5.3.1, в случае если `MutabilityLandJunior` снова будет поделен на подклассы, как показано в приведенном ниже листинге.

Листинг 5.18. MutabilityLandJunior

```
class MutabilityLandJunior: MutabilityLand {
   let soundsEnabled: Bool

   init(soundsEnabled: Bool, players: [Player], instructions: String,
      numberOfTiles: Int) { ❶
      self.soundsEnabled = soundsEnabled
      super.init(players: players, instructions: instructions,
         numberOfTiles: numberOfTiles)
   }

   convenience override init(players: [Player], instructions: String,
      numberOfTiles: Int) { ❷
      self.init(soundsEnabled: false, players: players, instructions:
      ➥ instructions, numberOfTiles: numberOfTiles)
   }
}
```

❶ MutabilityLandJunior получает собственный назначенный инициализатор.

❷ Добавлен единственный переопределяющий инициализатор.

Теперь можно инициализировать эту игру пятью способами.

Листинг 5.19. Инициализация MutabilityLandJunior со всеми инициализаторами

```
let mutabilityLandJr =
➥ MutabilityLandJunior(players: players, instructions: "Kids don't read
➥ manuals", numberOfTiles: 8)

let mutabilityLandJr = MutabilityLandJunior(soundsEnabled: true, players:
➥ players, instructions: "Kids don't read manuals", numberOfTiles: 8)

let mutabilityLandJr = MutabilityLandJunior(names: ["Philippe", "Alex"])

let mutabilityLandJr = MutabilityLandJunior(players: players)

let mutabilityLandJr = MutabilityLandJunior(players: players,
➥ numberOfTiles: 8)
```

Теперь этот подкласс получает множество инициализаторов бесплатно.

Помимо этого, независимо от того, сколько у вас подклассов (надеюсь, не слишком много), подклассы должны переопределять только один инициализатор!

5.3.3. Упражнение

4

Имеется класс, который делит на подклассы Television из предыдущего упражнения:

```
class HandHeldTelevision: Television {
let weight: Int

init(weight: Int, resolution: Resolution, screenType: ScreenType,
➥ serialNumber: String, room: String) {
      self.weight = weight
      super.init(resolution: resolution, screenType: screenType,
      ➥ serialNumber: serialNumber, room: room)
   }
}
```

Добавьте два вспомогательных инициализатора в иерархию подклассов, чтобы этот инициализатор работал из самого верхнего суперкласса:

```
let handheldTelevision = HandHeldTelevision(serialNumber: "293nr30znNdjW")
```

5.4. Требуемые инициализаторы

Вы, возможно, видели, как требуемые инициализаторы иногда всплывали, например при работе с `UIViewController` из UIKit. Требуемые инициализаторы играют решающую роль при создании подклассов. Можно оформить эти инициализаторы ключевым словом `required`. Добавление ключевого слова `required` гарантирует, что подклассы реализуют требуемый инициализатор. Это ключевое слово необходимо по двум причинам – фабричные методы и протоколы, – которые мы рассмотрим в этом разделе.

5.4.1. Фабричные методы

Фабричные методы – это первая причина, по которой нам нужно ключевое слово `required`. Фабричные методы – типичный шаблон проектирования, который облегчает создание экземпляров класса. Можно вызывать эти методы для типа, такого как класс или структура (в отличие от экземпляра), для простого создания экземпляров предварительно сконфигурированных экземпляров. Вот пример, где мы создаем экземпляр `BoardGameinstance` или `MutabilityLand` с помощью метода фабрики `makeGame`.

Листинг 5.20. Фабричные методы в действии

```
let boardGame = BoardGame.makeGame(players: players)

let mutabilityLand = MutabilityLand.makeGame(players: players)
```

Теперь вернемся к суперклассу `BoardGame`, где добавим функцию класса `makeGame`.

Метод `makeGame` принимает только игроков и возвращает экземпляр `Self`. `Self` ссылается на текущий тип, для которого вызывается `makeGame`; это может быть `BoardGame` или один из его подклассов.

В реальном сценарии настольная игра может иметь всевозможные настройки, такие как ограничение по времени и локаль, как показано в следующем примере, что добавляет дополнительные преимущества при создании фабричного метода.

Листинг 5.21. Знакомство с фабричным методом makeGame

```
class BoardGame {
   // ... Пропускаем часть кода.

   class func makeGame(players: [Player]) -> Self {
      let boardGame = self.init(players: players, numberOfTiles: 32)
      // Здесь идет конфигурация.
      // E.g.
      // boardGame.locale = Locale.current
      // boardGame.timeLimit = 900
      return boardGame
   }
}
```

Причина, по которой makeGame возвращает Self, заключается в том, что Self для каждого подкласса разный. Если бы методу нужно было вернуть экземпляр BoardGame, makeGame не смог бы вернуть, например, экземпляр MutabilityLand. Но мы пока еще туда не добрались; это дает следующую ошибку.

Листинг 5.22. Ошибка required

```
constructing an object of class type 'Self' with a metatype value must use a
'required' initializer
return self.init(players: players, numberOfTiles: 32)
~~~~ ^
```

Инициализатор выдает ошибку, потому что makeGame может вернуть BoardGame или любой экземпляр подкласса. Поскольку makeGame обращается к self.init, нужна гарантия того, что подклассы реализуют этот метод. Добавление ключевого слова required в назначенный инициализатор заставляет подклассы реализовывать инициализатор, который удовлетворяет этому требованию.

Сначала мы добавляем ключевое слово required в инициализатор, к которому обращаются в makeGame.

Листинг 5.23. Добавление ключевого слова required в инициализаторы

```
class BoardGame {
   // ... Пропускаем часть кода.
   required init(players: [Player], numberOfTiles: Int) {
      self.players = players
      self.numberOfTiles = numberOfTiles
   }
}
```

Подклассы теперь могут заменить слово override на слово required в соответствующем инициализаторе.

Листинг 5.24. Подкласс required

```
class MutabilityLand: BoardGame {

   // ... Пропускаем часть кода.

   convenience required init(players: [Player], numberOfTiles: Int) {
      self.init(players: players, instructions: «Read the manual»,
      numberOfTiles: numberOfTiles)
   }
}
```

Теперь компилятор доволен, и вы можете воспользоваться преимуществами фабричных методов при создании суперклассов и подклассов.

5.4.2. Протоколы

Протоколы – это вторая причина, по которой существует ключевое слово required.

Протоколы

Мы будем подробно рассматривать протоколы в главах 7, 8, 12 и 13.

Когда у протокола есть инициализатор, класс, принимающий этот протокол, *должен* оформить данный инициализатор ключевым словом required. Давайте посмотрим, почему и как это работает.

Сначала мы вводим протокол BoardGameType, который содержит инициализатор.

Листинг 5.25. Протокол BoardGameType

```
protocol BoardGameType {
   init(players: [Player], numberOfTiles: Int)
}
```

Затем мы реализуем этот протокол в классе BoardGame, чтобы BoardGame реализовывал метод init из протокола.

Листинг 5.26. Реализация протокола BoardGameType

```
class BoardGame: BoardGameType {
   // ... Пропускаем часть кода
```

На данный момент компилятор по-прежнему не доволен. Поскольку BoardGame соответствует протоколу BoardGameType, его подклассы также должны соответствовать этому протоколу и реализовывать init(players: [Player], numberOfTiles: Int).

Можно снова использовать ключевое слово required, чтобы заставить наши подклассы реализовать этот инициализатор, точно так же, как в предыдущем примере.

5.4.3. Когда классы являются финальными

Один из способов избежать необходимости использовать ключевое слово required – добавление к классу ключевого слова final. Когда вы делаете класс финальным, это означает, что вы не можете делить его на подклассы. Это может быть хорошим началом до тех пор, пока вам явно не понадобится поведение подклассов, поскольку добавление ключевого слова final повышает производительность[1].

Если класс является финальным, можно удалить из инициализаторов все ключевые слова required. Например, допустим, что никто не любит играть в игры, которые разделены на подклассы, кроме самого BoardGame. Теперь вы можете сделать BoardGame финальным и удалить любые подклассы. Обратите внимание, что мы убираем ключевое слово required из назначенного инициализатора.

Листинг 5.27. BoardGame – это теперь финальный класс

```
protocol BoardGameType {
   init(players: [Player], numberOfTiles: Int)
}

final class BoardGame: BoardGameType { ❶
   let players: [Player] let numberOfTiles: Int

   // Использование ключевого слова required не требуется
   init(players: [Player], numberOfTiles: Int) { ❷
      self.players = players
      self.numberOfTiles = numberOfTiles
   }

   class func makeGame(players: [Player]) -> Self {
      return self.init(players: players, numberOfTiles: 32)
   }

   // ... Пропускаем часть кода
}
```


❶ Теперь это финальный класс.

❷ Назначенный инициализатор больше не требуется.

Несмотря на реализацию протокола BoardGameType и наличие фабричного метода makeGame, BoardGame не нужно никаких требуемых инициализаторов, потому что это *финальный* класс.

[1] Increasing Performance by Reducing Dynamic Dispatch // https://developer.apple.com/swift/blog/?id=27.

5.4.4. Упражнения

5

Будут ли требуемые инициализаторы полезны для структур? Почему да или почему нет?

6

Назовите два варианта использования требуемых инициализаторов.

5.5. В заключение

Вы стали свидетелями того, что у Swift есть много типов инициализаторов, каждый из которых имеет своим правила и странности. Вы обратили внимание, что инициализаторы еще сложнее, когда вы создаете подклассы. Возможно, создание подклассов непопулярно, но это может быть жизнеспособной альтернативой в зависимости от фона, стиля кодирования и кода, который вы можете получить в наследство, когда начнете работать в интересной компании.

Прочитав эту главу, я надеюсь, что вы почувствуете себя достаточно уверенно, чтобы написать без проблем свои инициализаторы, и сможете предложить чистые интерфейсы для инициализации своих типов.

Резюме

- Структурам и классам нужно, чтобы все их неопциональные свойства были инициализированы.
- Структуры генерируют «бесплатные» почленные инициализаторы.
- Структуры лишаются почленных инициализаторов, если вы добавляете пользовательский инициализатор.
- Если вы расширяете структуры с помощью пользовательских инициализаторов, у вас могут быть как почленные, так и пользовательские инициализаторы.
- У классов должен быть один или несколько назначенных инициализаторов.
- Вспомогательные инициализаторы указывают на назначенные инициализаторы.
- Если у подкласса есть свои хранимые свойства, он не наследует напрямую инициализаторы суперкласса.
- Если подкласс переопределяет назначенные инициализаторы, он получает вспомогательные инициализаторы из суперкласса.
- При переопределении инициализатора суперкласса с помощью вспомогательного инициализатора подкласс уменьшает количество назначенных инициализаторов.
- Ключевое слово `required` гарантирует, что подклассы реализуют инициализатор и что фабричные методы работают в подклассах.

- Как только у протокола появляется инициализатор, ключевое слово `required` гарантирует, что подклассы соответствуют протоколу.
- Делая класс финальным, инициализаторы могут удалить ключевое слово `required`.

Ответы

1

Будут ли работать эти инициализаторы:

```
let pancakes = Pancakes(syrupType: .corn, stackSize: 8)
let morePancakes = Pancakes(syrupType: .maple)
```

Нет. Когда вы используете пользовательский инициализатор, почленный инициализатор будет недоступен.

2

Если эти инициализаторы не работают, можно ли заставить их работать, не добавляя еще один инициализатор?

```
struct Pancakes {
   enum SyrupType {
      case corn
      case molasses
      case maple
   }

   let syrupType: SyrupType
   let stackSize: Int
}

extension Pancakes { ❶
   init(syrupType: SyrupType) { ❷
      self.stackSize = 10
      self.syrupType = syrupType
   }
}

let pancakes = Pancakes(syrupType: .corn, stackSize: 8)
let morePancakes = Pancakes(syrupType: .maple)
```


❶ Расширяем `Pancakes`.

❷ Помещаем пользовательский инициализатор в расширение.

3

Приведенный ниже суперкласс под названием `Device` регистрирует устройства в офисе и отслеживает помещения, в которых эти устройства можно найти. Его

подкласс Television – одно из таких устройств, которое делит Device на подклассы.

Задача состоит в том, чтобы инициализировать подкласс Television с помощью инициализатора Device. Другими словами, заставьте следующую строку кода работать, добавив где-нибудь один инициализатор:

```
class Television: Device {

    override init(serialNumber: String, room: String) { ❶
        self.resolution = .unknown
        self.screenType = .unknown
        super.init(serialNumber: serialNumber, room: room)
    }
    // ... Пропускаем остальной код.
}
```

❶ Переопределяем назначенный инициализатор из Device.

4

Имеется класс, который наследует от класса Television из предыдущего упражнения. Добавьте два вспомогательных инициализатора в иерархию подклассов, чтобы этот инициализатор работал из самого верхнего суперкласса:

```
class Television {

    convenience override init(serialNumber: String, room: String) { ❶
        self.init(resolution: .unknown, screenType: .unknown,
        serialNumber: serialNumber, room: room)
    }
    // ... Пропускаем часть кода.
}

class HandHeldTelevision: Television {

    convenience override init(resolution: Resolution, screenType:
    ➥ ScreenType, serialNumber: String, room: String) { ❷
        self.init(weight: 0, resolution: resolution, screenType:
        ➥ screenType, serialNumber: «Unknown», room: «UnKnown»)
    }
    // ... Пропускаем остальной код.
}
```

❶ Добавляем вспомогательный инициализатор к Television, в результате чего назначенный инициализатор из Device будет переопределен.

❷ Добавляем вспомогательный инициализатор в HandHeldTelevision, в результате чего назначенный инициализатор из Television будет переопределен.

5

Будут ли требуемые инициализаторы полезны для структур? Почему да или почему нет?

Нет, требуемые инициализаторы принудительно применяют инициализаторы для подклассов, а структуры нельзя делить на подклассы.

6

Назовите два варианта использования требуемых инициализаторов.

Принудительное использование фабричных методов для подклассов и соответствие протоколу, определяющему инициализатор.

Глава 6. Непринужденная обработка ошибок

В этой главе:

- передовые методы обработки ошибок (и недостатки);
- поддержание вашего приложения в надлежащем состоянии при генерации ошибки;
- как распространяются ошибки;
- добавление информации для клиентских приложений (и для устранения неполадок);
- преобразование в NSError;
- как сделать использование API проще, не нарушая целостности приложения.

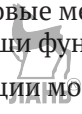
Обработка ошибок является неотъемлемой частью при разработке любого программного обеспечения и не зациклена на Swift. Но то, как Swift относится к обработке ошибок, влияет на то, как вы доставляете код, который приятно использовать, и учитывает проблемы, которые могут возникнуть в системе. В этой главе вы будете изящно генерировать и перехватывать ошибки, проходя тонкую грань между созданием полезного API вместо утомительного.

Обработка ошибок в теории звучит просто: сгенерируйте несколько ошибок и перехватите их, и ваша программа продолжит работу. Но на самом деле сделать это правильно может быть довольно сложно. Swift также добавляет уникальный вид сверху, где он устанавливает правила, предлагает синтаксический сахар и проверки на этапе компиляции, чтобы убедиться, что вы обрабатываете сгенерированные ошибки.

Несмотря на то что когда что-то идет не так, пора сгенерировать ошибку, есть много тонкостей. Более того, как только вы столкнетесь с ошибкой, вам нужно будет знать, на ком лежит ответственность за обработку (или игнорирование) ошибки. Вы распространяете ошибку вплоть до пользователя, или ваша программа может вообще предотвратить их? Кроме того, когда именно генерировать ошибки?

В этой главе рассматривается, как Swift обрабатывает ошибки, но это не будет сухим повторением документации от компании Apple. Мы включили передовые методы по поддержанию приложения в хорошем состоянии, более внимательный взгляд на недостатки обработки ошибок и методы правильной обработки ошибок.

В первом разделе рассматриваются ошибки Swift и способы их генерации. В нем также приводятся передовые методы по поддержанию вашего кода в предсказуемом состоянии, когда ваши функции начинают генерировать ошибки.



Потом вы увидите, как функции могут распространять ошибки через приложение. Вы узнаете, как добавить техническую информацию для устранения неполадок и локализованную информацию для конечных пользователей, узнаете о пре-

образовании в NSError и подробнее познакомитесь с централизацией обработки ошибок при реализации полезных протоколов.

В конце главы вы познакомитесь с недостатками, возникающими при генерации ошибок, и как свести их на нет. Вы также узнаете, как сделать ваши API более приятными в использовании, при этом сохраняя целостность системы.

6.1. Ошибки в Swift

Ошибки могут быть всевозможных форм и размеров. Если вы спросите трех разработчиков, что такое ошибка, то можете получить три разных ответа и, скорее всего, услышите разные точки зрения относительно исключений, ошибок, проблем во время выполнения или даже обвинения в адрес пользователя из-за того, что он столкнулся с проблемой.

Чтобы начать эту главу правильно, давайте разделим ошибки на три категории, чтобы мы находились на одной странице.

- *Ошибки программирования* – эти ошибки могли бы предотвратить программисты, которые хорошо выспались. К ним относятся, например, вышедшие за предел массивы, деление на ноль и целочисленные переполнения. По сути, это проблемы, которые можно исправить на уровне кода. Именно здесь модульные тесты и обеспечение качества могут спасти вас, когда вы допускаете ошибку. Обычно в Swift можно использовать проверки, например `assert`, чтобы убедиться, что ваш код работает, как и было задумано, и `precondition`, чтобы другие знали, что ваш API вызывается правильно. Однако `assert` и `precondition` не то, о чем идет речь в этой главе.
- *Ошибки пользователя* – это когда пользователь взаимодействует с системой и не может правильно выполнить задачу, например случайно отправляя селфи в пьяном виде вашему боссу. Ошибки пользователя могут быть вызваны неполным пониманием системы, когда человек отвлекся, или неуклюжим пользовательским интерфейсом. Даже если оскорбление интеллекта клиента может быть забавным времяпрепровождением, вы можете обвинить пользовательскую ошибку в самом приложении, и можете предотвратить эти проблемы с помощью хорошего дизайна, четкого взаимодействия и формирования программного обеспечения таким образом, чтобы это помогло пользователям достичь того, чего они хотят.
- *Ошибки, обнаруженные во время выполнения* – эти ошибки могут быть связаны с невозможностью создания файла приложением из-за переполнения жесткого диска, сбоем сетевого запроса, сертификатами с истекшим сроком действия, JSON-парсерами, которые выражают недовольство после того, как им предоставили неверные данные, и многим другим, что может пойти не так, когда приложение работает. Последняя категория ошибок относится к категории исправимых (вообще говоря), и именно об этом идет речь в данной главе.

В этом разделе вы увидите, как Swift определяет ошибки, каковы их слабые стороны и как их перехватывать. Помимо генерирования ошибок, функции могут выполнять дополнительную служебную работу, чтобы гарантировать, что прило-

жение остается в предсказуемом состоянии. Это еще одна тема, которую мы исследуем в данном разделе.

6.1.1. Протокол Error

Присоединяйтесь!

Будет более познавательно и веселее, если вы будете знакомиться с кодом по мере прохождения этой главы. Исходный код можно скачать по адресу http://mng.bz/oN4j.

Swift предлагает протокол Error, который вы можете использовать, чтобы указать, что что-то пошло не так в вашем приложении. Перечисления хорошо подходят для ошибок, поскольку каждый случай является взаимоисключающим. Например, у вас есть перечисление ParseLocationError с тремя вариантами сбоя.

Листинг 6.1. Перечисление Error

```
enum ParseLocationError: Error {
   case invalidData
   case locationDoesNotExist
   case middleOfTheOcean
}
```

У протокола Error нет требований, и, следовательно, он не требует реализации, что также означает, что вам не нужно делать каждую ошибку перечислением. Например, вы также можете использовать другие типы, такие как структуры, чтобы указать, что что-то пошло не так. Структуры менее подходят, но могут быть полезны, когда нужно добавить более обширные данные к ошибке.

Например, вы можете иметь структуру Error, которая содержит несколько ошибок и другие свойства.

Листинг 6.2. Структура Error


```
struct MultipleParseLocationErrors: Error {
   let parsingErrors: [ParseLocationError]
   let isShownToUser: Bool
}
```

6.1.2. Генерация ошибок

Ошибки существуют, чтобы их можно было генерировать и обрабатывать. Например, если функция не может сохранить файл, она может сгенерировать ошибку с указанием причины, например переполнение жесткого диска или отсутствие прав на запись на диск. Когда функция или метод может генерировать ошибку, Swift требует наличия ключевого слова throws в сигнатуре функции за закрывающей скобкой.

Например, можно превратить две строки в тип Location, содержащий константы latitude и longitude, как показано в следующем коде. Функция parseLocation может затем преобразовать строки. В случае неудачи функция parseLocation генерирует ошибку ParseLocationError.invalidData.

Листинг 6.3. Преобразование строк

```
struct Location { ❶
   let latitude: Double
   let longitude: Double
}

func parseLocation(_ latitude: String, _ longitude: String) throws ->
   Location { ❷
      guard let latitude = Double(latitude), let longitude =
      Double(longitude) else {
         throw ParseLocationError.invalidData
      }
      return Location(latitude: latitude, longitude: longitude)
   }
   do { ❸
      try parseLocation("I am not a double", "4.899431") ❹
   } catch {
      print(error) // invalidData ❺
   }
```


❶ Определяем тип Location, который возвращает parseLocation.

❷ Функция parseLocation либо возвращает Location, либо выдает ошибку, указанную ключевым словом throws.

❸ Перехватываем ошибку с помощью ключевых слов do catch.

❹ Вызываем функцию, генерирующую ошибку, с помощью ключевого слова try.

❺ Swift автоматически выдает константу error для сопоставления в операторе catch.

Поскольку parseLocation – функция, генерирующая ошибку, на что указывает ключевое слово throws, ее необходимо вызывать, используя ключевое слово try. Компилятор также заставляет код, вызывающий функции, генерирующие ошибку, каким-то образом справляться с ошибкой. Позже вы увидите методы, которые сделают ваши API более приятными для других разработчиков.

6.1.3. Swift не показывает ошибки

Еще один специфический аспект обработки ошибок в Swift заключается в том, что функции не показывают, какие ошибки они могут генерировать. Функция, поме-

ченная как throws, теоретически не может генерировать ошибки или пять миллионов различных ошибок, и это нельзя узнать, посмотрев на сигнатуру функции. Отсутствие явного перечисления и обработки каждой ошибки дает вам гибкость, но существенным недостатком является то, что нельзя быстро узнать, какие ошибки может вызывать или распространять функция.

Функции не показывают своих ошибок, поэтому рекомендуется предоставлять *какую-нибудь* информацию там, где это возможно. К счастью, ваш друг Quick Help может помочь вам предоставить дополнительную информацию об ошибках, которые вы можете сгенерировать. Вы также можете использовать его, чтобы указать, когда могут возникать ошибки, как показано в листинге 6.4.

Можно сгенерировать документацию по Quick Help в Xcode, поместив курсор на функцию и нажав сочетание клавиш **Cmd-Alt-/**, чтобы сгенерировать шаблон Quick Help, включая возможные ошибки (см. рис. 6.1).

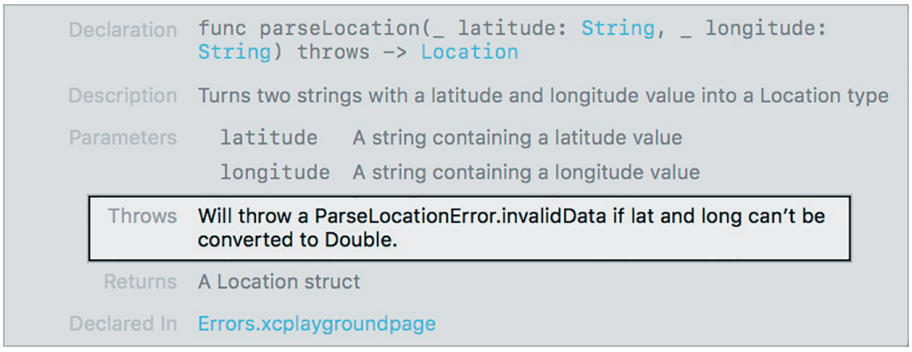


Рис. 6.1. Quick Help сообщает об ошибках

Листинг 6.4. Добавление информации об ошибке в функцию

```
/// Превращает две строки со значениями latitude и longitude в тип Location.
///
/// - Параметры:
/// - latitude: строка, содержащая значение широты
/// - longitude: строка, содержащая значение долготы
/// - Возвращает: структуру Location
/// - Генерирует: сгенерирует ParseLocationError.invalidData, если широту
➥ и долготу нельзя преобразовать в Double. ❶

func parseLocation(_ latitude: String, _ longitude: String) throws ->
   Location {
      guard let latitude = Double(latitude), let longitude = Double(longitude)
   else {
      throw ParseLocationError.invalidData
   }
   return Location(latitude: latitude, longitude: longitude)
}
```

❶ Добавляем ключевое слово `throws`: комментарий к документации Quick Help.

Это своего рода повязка, но добавление быстрой справки дает разработчику хоть какую-то информацию относительно ожидаемых ошибок.

6.1.4. Сохранение среды в предсказуемом состоянии

Вы видели, как код, вызывающий функцию, обрабатывает всевозможные ошибки, которые могут генерировать ваши функции. Но выбросить ошибку может быть недостаточно. Иногда функция, генерирующая ошибку, может пройти лишнюю милю и убедиться, что состояние приложения остается тем же самым, когда происходит ошибка.

Подсказка

Вообще говоря, поддержание генерирующей функции в предсказуемом состоянии после выдачи ошибки – хорошая привычка.

Предсказуемое состояние препятствует тому, чтобы среда находилась в подвешенном состоянии между состоянием ошибки и нормальным состоянием. Поддержание приложения в предсказуемом состоянии означает, что когда функция или метод выдает ошибку, оно должно предотвращать или отменять любые изменения, которые были внесены в среду или экземпляр.

Допустим, у вас есть кеш памяти, и вы хотите сохранить значение в этом кеше с помощью метода. Если этот метод выдает ошибку, вы, вероятно, ожидаете, что ваше значение *не* будет кешировано. Однако если функция сохраняет значение в памяти при ошибке, внешний механизм повтора может даже привести к нехватке памяти в системе. Цель состоит в том, чтобы при возникновении ошибок вернуть среду в нормальное состояние, дабы вызывающая программа могла повторить попытку или продолжить работу другими способами.

Самый простой способ не дать функциям, генерирующим ошибку, модифицировать среду – когда функции прежде всего вообще не меняют среду. Создание неизменяемой функции – один из способов добиться этого. Неизменяемые функции и методы, в общем, имеют свои преимущества, но тем более когда функция генерирует ошибку.

Если вернуться к функции `parseLocation`, то можно увидеть, что она касается только тех значений, которые она передала, и не вносит никаких изменений во внешние значения, что означает отсутствие скрытых побочных эффектов. Поскольку функция `parseLocation` является неизменяемой, она работает предсказуемо.

Давайте рассмотрим еще два метода для достижения предсказуемого состояния.

Изменение временных значений

Второй способ сохранить свою среду в предсказуемом состоянии – это изменить копию или временное значение, а затем сохранить новое состояние после завершения изменения без ошибок.

Рассмотрим приведенный ниже тип TodoList, который может хранить массив строк. Однако если строка после обрезки пустая, метод append генерирует ошибку.

Листинг 6.5. Структура TodoList, которая изменяет состояние при наличии ошибок

```
enum ListError: Error {
   case invalidValue
}

struct TodoList {

   private var values = [String]()

   mutating func append(strings: [String]) throws {
      for string in strings {
         let trimmedString = string.trimmingCharacters(in: .whitespacesAnd
         ➥ Newlines)
         if trimmedString.isEmpty {
            throw ListError.invalidValue ❶
         } else {
            values.append(trimmedString) ❷
         }
      }
   }
}
```

❶ Если строка пустая, генерируем ошибку.

❷ Если строка не пустая, добавляем значение в массив значений.

Проблема состоит в том, что после того, как метод append выдает ошибку, тип теперь имеет наполовину заполненное состояние. Вызывающая программа может предположить, что все вернулось к тому, что было, и повторить попытку позже. Но в текущем состоянии TodoList оставляет конечную информацию в своих значениях.

Вместо этого вы можете рассмотреть возможность изменения временного значения и добавления окончательного результата к фактическому свойству values только после каждой итерации. Однако если метод append выдает ошибку во время итерации, новое состояние не сохраняется, а временное значение исчезает, оставляя TodoList в том же состоянии, что и до появления ошибки.

Листинг 6.6. TodoList работает с временными значениями

```
struct TodoList {
   private var values = [String]()

   mutating func append(strings: [String]) throws {
      var trimmedStrings = [String]() ❶
```

```
        for string in strings {
            let trimmedString = string.trimmingCharacters(in: .whitespacesAnd
            ➥ Newlines)
            if trimmedString.isEmpty {
                throw ListError.invalidValue
            } else {
                trimmedStrings.append(trimmedString) ❷
            }
        }
        values.append(contentsOf: trimmedStrings) ❸
    }
}
```

❶ Временной массив создан.

❷ Временной массив изменен.

❸ Если ошибка не генерируется, свойство values обновляется.

Корректирующий код и оператор defer

Один из способов вернуться в исходное состояние, после того как функция сгенерировала ошибку, – *отменить* изменения, находясь в середине операции. Отмена изменений на полпути, как правило, случается реже, но может быть единственным вариантом для вас, когда записываете данные, например файлы на жесткий диск.

В качестве примера рассмотрим функцию `writeToFiles`, которая может записывать несколько файлов в несколько локальных URL-адресов. Предостережение, однако, заключается в том, что у этой функции есть требование «все или ничего». Если запись в один файл не удалась, не записывайте файлы на диск. Чтобы сохранить функцию в предсказуемом состоянии в случае возникновения ошибки, вам нужно написать код очистки, который удаляет записанные файлы после того, как функция начинает выдавать ошибки.

Вы можете использовать оператор `defer` для операции очистки. Оператор `defer` запускается *после* того, как функция завершит работу, независимо от того, завершается ли функция нормально или потому, что произошла ошибка. Можно отслеживать все сохраненные файлы в функции, а затем удалить все сохраненные файлы в замыкании с `defer`, но только если количество сохраненных файлов не соответствует количеству путей, которые вы задаете для функции.

Листинг 6.7. Использование оператора defer

```
import Foundation

func writeToFiles (data: [URL: String]) throws {
    var storedUrls = [URL]() ❶
    defer { ❷
```

```
        if storedUrls.count != data.count { ❸
            for url in storedUrls {
                try! FileManager.default.removeItem(at: url) ❹
            }
        }
    }

    for (url, contents) in data {
        try contents.write(to: url, atomically: true, encoding:
        ➥ String.Encoding.utf8) ❺
        storedUrls.append(url) ❻
    }
}
```

❶ Для хранения успешно сохраненных URL-адресов создается массив, на случай если вам нужно удалить их в ходе очистки.

❷ Хотя он и объявлен вверху, оператор `defer` вызывается в конце функции.

❸ Если файл нельзя сохранить, удалите все успешно сохраненные файлы.

❹ Используйте `try!`, чтобы заявить, что эта операция не закончится неудачей (подробнее об этом позже).

❺ Файл записан, но в случае сбоя генерируется ошибка, на что указывает ключевое слово `try`.

❻ Если эта запись файла не генерируется, вы добавляете URL в массив `selectedUrls`.

Очистка, после того как произошло изменение, может быть сложной, потому что вы в основном перематываете время. Функция `writeToFiles`, например, удаляет все файлы в случае ошибки, но что, если бы там были файлы до того, как были записаны новые файлы? Блок `defer` в `writeToFiles` должен был быть более продвинутым, чтобы вести более тщательную запись о том, что именно происходило до того, как была сгенерирована ошибка. При написании корректирующего кода помните, что сложность отслеживания нескольких сценариев может возрастать.

6.1.5. Упражнения

1

Назовите один или более недостатков того, как Swift обрабатывает ошибки и как их компенсировать.

2

Назовите три способа убедиться, что функции, генерирующие ошибку, возвращаются в исходное состояние после выдачи ошибок.

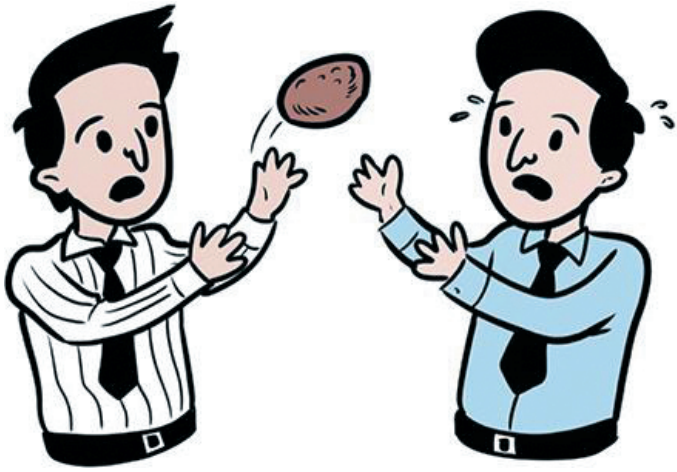
6.2. Распространение ошибок и перехват

Swift предлагает четыре способа обработки ошибок: их можно перехватывать, можно генерировать выше по стеку (это называется *распространением*, или «бурлением»), можно превратить их в опционалы с помощью ключевого слова `try?`, утверждая, что ошибка не возникнет, с помощью ключевого слова `try!`.

В этом разделе вы узнаете о распространении и методах чистого перехвата. Сразу после этого мы подробно рассмотрим ключевые слова `try?` и `try!`.

6.2.1. Распространение ошибок

Мой любимый способ решения проблем – передать их кому-то еще. К счастью, вы можете сделать то же самое в Swift с ошибками, которые получаете: вы распространяете их, генерируя их выше по стеку, как в односторонней игре с горячим картофелем.



Давайте используем этот раздел, чтобы создать последовательность функций, вызывающих друг друга, чтобы увидеть, как функция низшего уровня может распространять ошибку вплоть до функции высшего уровня.

Тенденция при поиске кулинарных рецептов заключается в том, что вам нужно просеять чью-то личную историю, просто чтобы добраться до рецепта и начать готовить. Длинное вступление полезно для поисковых систем, но было бы неплохо отбросить все лишнее, чтобы можно было сразу извлечь рецепт и приступить к приготовлению пищи, прежде чем в желудке начнет урчать.

В качестве примера мы создадим структуру `RecipeExtractor` (см. листинг 6.8), которая извлекает рецепт из HTML-страницы, которую она передает. RecipeExtractor использует функции поменьше для выполнения этой задачи. Мы сосредоточимся на распространении ошибок, а не на реализации (см. рис. 6.2).

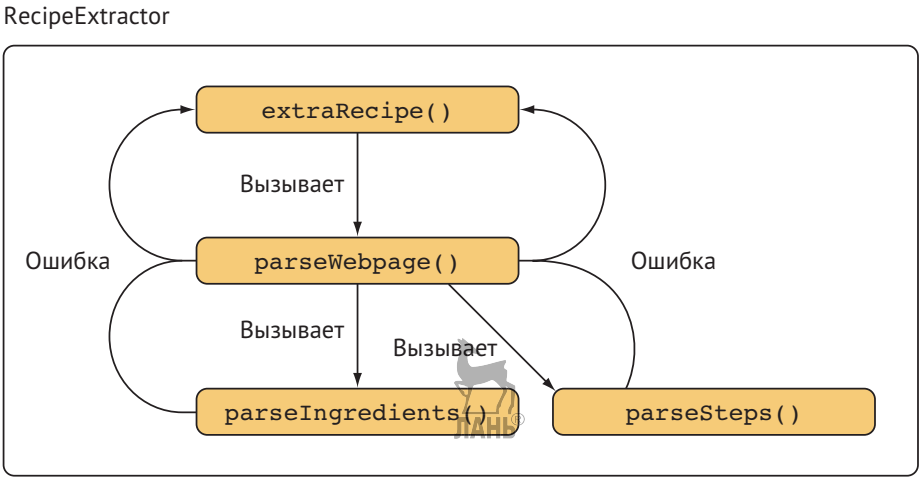


Рис. 6.2. Распространение ошибки

Распространение работает через метод, который может вызывать метод низшего уровня, который, в свою очередь, может также вызывать методы низшего уровня. Но ошибка может распространиться снова до самого высокого уровня, если вы ей это позволите.

Когда для `RecipeExtractor` вызывается функция `extractRecipe`, она вызывает функцию низшего уровня, `parseWebpage`, которая, в свою очередь, вызывает `parseIngredients` и `parseSteps`. И `parseIngredients`, и `parseSteps` могут выдать ошибку, которую `parseWebpage` получит и распространит обратно в функцию `extractRecipe`, как показано в этом коде.

Листинг 6.8. RecipeExtractor

```
struct Recipe { ❶
   let ingredients: [String]
   let steps: [String]
}

enum ParseRecipeError: Error { ❷
   case parseError
   case noRecipeDetected
   case noIngredientsDetected
}

struct RecipeExtractor {
   let html: String

   func extractRecipe() -> Recipe? { ❸
      do { ❹
         return try parseWebpage(html)
      } catch {
```

```
            print(«Could not parse recipe»)
            return nil
        }
    }

    private func parseWebpage(_ html: String) throws -> Recipe {
        let ingredients = try parseIngredients(html) ❺
        let steps = try parseSteps(html) ❺
        return Recipe(ingredients: ingredients, steps: steps)
    }

    private func parseIngredients(_ html: String) throws -> [String] {
        // ...Здесь идет парсинг
        // .. Если не генерируется ошибка
        throw ParseRecipeError.noIngredientsDetected
    }

    private func parseSteps(_ html: String) throws -> [String] {
        // ... Здесь идет парсинг
        // .. Если не генерируется ошибка
        throw ParseRecipeError.noRecipeDetected
    }
}
```

❶ Структура Recipe, которую возвращает RecipeExtractor.

❷ Ошибка, которая может быть выброшена внутри RecipeExtractor.

❸ Функция extractRecipe запускает извлечение.

❹ Распространение ошибок здесь останавливается из-за оператора do catch.

❺ И parseIngredients, и parseSteps вызываются с ключевым словом try; любые возникающие ошибки будут распространяться вверх.

Получение рецепта может быть неудачным по нескольким причинам; возможно, HTML-файл вообще не содержит рецепт, или экстрактор не может получить ингредиенты. Поэтому функции низшего уровня parseIngredients и parseSteps могут выдавать ошибку. Поскольку их родительская функция parseWebpage не знает, как обрабатывать эти ошибки, она снова передает ошибку обратно в функцию структуры extractRecipe с помощью ключевого слова try. Так как parseWebpage распространяет ошибку снова, в сигнатуре функции также содержится ключевое слово throws.

Обратите внимание, что у метода extractRecipe есть оператор catch без сопоставления; таким образом, extractRecipe перехватывает все возможные ошибки. Если бы оператор catch проводил сопоставление для конкретных ошибок, теоретически некоторые ошибки не были бы перехвачены и должны были бы распространяться еще выше, также превращая функцию extractRecipe в генерирующую.

6.2.2. Добавление технических деталей для устранения неполадок

Вблизи того места, где возникает ошибка, ее окружает большое количество контекста. У вас есть среда в конкретный момент времени. Вы точно знаете, какое действие завершилось неудачей и каково состояние каждой переменной в непосредственной близости от ошибки. Когда ошибка распространяется, это точное состояние может быть потеряно, при этом теряется полезная информация для обработки ошибки.

При регистрации ошибки, чтобы устранить неполадку, может быть полезно добавить полезную информацию для разработчиков, например в случае неудачного парсинга рецептов. После добавления дополнительной информации можно выполнить для ошибки сопоставление с образцом и извлечь информацию при устранении неполадок.

Например, как показано ниже, можно добавить свойства `symbol` и `line` в кейс `parseError` в перечислении `ParseRecipeError`, чтобы предоставить немного больше информации в случае неудачи.

Листинг 6.9. Добавляем дополнительную информацию в ParseRecipeError

```
enum ParseRecipeError: Error {
   case parseError(line: Int, symbol: String) ❶
   case noRecipeDetected
   case noIngredientsDetected
}
```

❶ Добавление дополнительной информации.

Таким образом, можно более явно выполнить сопоставление с образцом при устранении неполадок. Обратите внимание, что мы по-прежнему используем оператор `catch`, чтобы не дать `extractRecipes` превратиться в генерирующую функцию.

Листинг 6.10. Сопоставление для определенной ошибки

```
struct RecipeExtractor {
   let html: String

   func extractRecipe() -> Recipe? {
      do {
         return try parseWebpage(html)
      } catch let ParseRecipeError.parseError(line, symbol) { ❶
         print("Parsing failed at line: \(line) and symbol: \(symbol)")
         return nil
      } catch {
         print(«Could not parse recipe»)
         return nil
```

```
        }
    }

    // ... Пропускаем остальной код
}
```

❶ Сопоставление с образцом и извлечение конкретной информации.

Добавление удобочитаемой информации

Теперь, когда есть техническая информация, вы можете использовать ее для преобразования данных в ошибку в удобочитаемом для пользователя формате. Причина, по которой вы не передаете ошибке удобочитаемую строку, заключается в том, что с помощью технических деталей можно сделать различие между удобочитаемой ошибкой и подробной технической информацией для разработчика.

Один из подходов к получению удобочитаемой информации – внедрение протокола `LocalizedError`. Придерживаясь этого протокола, вы указываете, что ошибка следует определенным соглашениям и содержит информацию в удобочитаемом для пользователя формате. Соответствие `LocalizedError` сообщает обработчику ошибок, что он может с уверенностью показывать пользователю имеющуюся в наличии информацию без необходимости выполнять какое-либо преобразование.

Чтобы внедрить протокол `LocalizedError`, можно реализовать ряд свойств, но все они имеют значение по умолчанию nil, чтобы вы могли приспособить ошибку к тем свойствам, которые вы хотели бы реализовать. Пример приведен в листинге 6.11. В этом сценарии вы решаете внедрить свойство `errorDescription`, которое может дать больше информации о самой ошибке. Вы также добавляете свойство `failureReason`, которое помогает объяснить, почему произошла ошибка, и `recoverySuggestion`, чтобы помочь пользователям предпринять нужные действия, в этом случае – попробовать другую страницу с рецептами. При работе в OS X также можно включить свойство `helpAnchor`, которое можно использовать для ссылки на средство просмотра справки Help Viewer от компании Apple, но в данном примере это свойство не обязательно.

Поскольку строки ориентированы на пользователя, рассмотрите возможность возврата локализованных строк вместо обычных, чтобы сообщения соответствовали локали пользователя.

Листинг 6.11. Реализация протокола LocalizedError

```
extension ParseRecipeError: LocalizedError { ❶
    var errorDescription: String? { ❷
        switch self {
            case .parseError:
                return NSLocalizedString("The HTML file had unexpected symbols.",
                    comment: "Parsing error reason unexpected symbols")
```

```
            case .noIngredientsDetected:
               return NSLocalizedString("No ingredients were detected.",
                  comment: "Parsing error no ingredients.")
            case .noRecipeDetected:
               return NSLocalizedString("No recipe was detected.",
                  comment: «Parsing error no recipe.») }
         }
      }

      var failureReason: String? { ❸
         switch self {
            case let .parseError(line: line, symbol: symbol):
               return String(format: NSLocalizedString("Parsing data failed at
               ➥ line: %i and symbol: %@",
                  comment: "Parsing error line symbol"), line, symbol)
            case .noIngredientsDetected:
               return NSLocalizedString("The recipe seems to be missing its
               ➥ ingredients.",
                  comment: "Parsing error reason missing ingredients.")
            case .noRecipeDetected:
               return NSLocalizedString("The recipe seems to be missing a
               ➥ recipe.",
                  comment: "Parsing error reason missing recipe.")
         }
      }

      var recoverySuggestion: String? { ❹
         return "Please try a different recipe."
      }
   }
```

❶ Можно отдельно соответствовать протоколу с помощью расширения, чтобы разделить код.

❷ Свойство `errorDescription` помогает объяснить, что пошло не так.

❸ Мы можем реализовать свойство `failReason`.

❹ С помощью `recoverySuggestion` можно предложить, как вернуться в исходное состояние после ошибки.

Все эти свойства не являются обязательными. Как правило, реализации `errorDescription` и `recoverySuggestion` должно быть достаточно.

Как только обнаруживается удобочитаемая ошибка, ее можно безопасно передать, например `UIAlert` в iOS, вывести в командной строке или использовать уведомление в OS X.

Преобразование в NSError

Приложив немного усилий, можно реализовать протокол CustomNSError, который помогает преобразовать Swift.Error с NSError, в случае если речь идет о Objective-C. CustomNSError ожидает три свойства: статическое errorDomain, целое число errorCode и словарь errorUserInfo.

errorDomain и errorCode – это то, где вам нужно сделать выбор. Для удобства можно заполнить errorUserInfo значениями, которые вы предварительно определили (и использовать пустые значения, если они равны nil).

Листинг 6.12. Реализация NSError

```
extension ParseRecipeError: CustomNSError {
   static var errorDomain: String { return "com.recipeextractor" } ❶

   var errorCode: Int { return 300 } ❷

   var errorUserInfo: [String: Any] {
      return [
         NSLocalizedDescriptionKey: errorDescription ?? "", ❸
         NSLocalizedFailureReasonErrorKey: failureReason ?? "", ❸
         NSLocalizedRecoverySuggestionErrorKey: recoverySuggestion ?? "" ❸
      ]
   }
}

let nsError: NSError = ParseRecipeError.parseError(line: 3, symbol: "#") as
   NSError ❹
print(nsError) // Error Domain=com.recipeextractor Code=300 "Parsing data
➥ failed at line: 3 and symbol: #" UserInfo={NSLocalizedFailureReason=The
➥ HTML file had unexpected symbols., NSLocalizedRecoverySuggestion=Please
➥ try a different recipe., NSLocalizedDescription=Parsing data failed at
➥ line: 3 and symbol: #}
```

❶ Придумываем домен, который относится к ошибке.

❷ Создаем некий код, уникальный для этой ошибки.

❸ В случае с errorUserInfo вы используете три ключа по умолчанию, но можете повторно использовать свойства, которые у вас были раньше. Также обратите внимание, что вы возвращаетесь обратно к пустым строкам (nil будет выдавать вам предупреждения).

❹ Легко преобразовываем ошибку в NSError с помощью действия as NSError.

Без предоставления этой информации преобразование Error в NSError означает, что у ошибки нет соответствующей информации о коде и домене. Использование CustomNSError дает вам жесткий контроль над этим преобразованием.

6.2.3. Централизация обработки ошибок

Функция низшего уровня иногда может сама разрешать ошибку – как, например, механизм повторного запуска при передаче данных, – но обычно эта функция распространяет ошибку по стеку обратно в точку вызова, поскольку в ней отсутствует контекст относительно того, как обрабатывать ошибку. Например, если у встраиваемого фреймворка не получается сохранить файл и он выдает ошибку, она не будет знать, что приложение iOS, реализующее этот фреймворк, захочет показать диалоговое окно `UIAlertController`, или что инструмент командной строки Linux захочет записать журнал в stderr.

При обработке распространяемых ошибок полезной практикой является централизация обработки ошибок. Представьте, что при перехвате ошибки вам нужно отобразить диалоговое окно с сообщением об ошибке в приложении для iOS или OS X. Если код обработки ошибок у вас находится в десятках разных мест в приложении, вносить изменения сложно, что делает ваше приложение устойчивым к изменениям. Чтобы исправить жесткую настройку обработки ошибок, можно использовать центральное место для представления ошибок. При перехвате кода можно передавать ошибку обработчику, который знает, что с ней делать, например открыть пользователю диалоговое окно, отправить ошибку в системы диагностики, записать ее в `stderr` и т. д.

Например, у вас есть один обработчик ошибок, где одна и та же функция `handleError` повторяется несколько раз посредством перегрузок функций. Благодаря перегрузкам `ErrorHandler` может получить детальный контроль над тем, какая ошибка готова к представлению пользователю и какие ошибки должны откатываться назад при универсальном сообщении.

Листинг 6.13. ErrorHandler с перегрузками функций

```
struct ErrorHandler {

   static let 'default' = ErrorHandler() ❶

   let genericMessage = "Sorry! Something went wrong" ❷

   func handleError(_error: Error) { ❸
      presentToUser(message: genericMessage)
   }

   func handleError(_error: LocalizedError) { ❹
      if let errorDescription = error.errorDescription {
         presentToUser(message: errorDescription)
      } else {
         presentToUser(message: genericMessage)
      }
   }

   func presentToUser(message: String) { ❺
      // Не показано в коде: Открываем окно с предупреждением в iOS
```

```
        или OS X или выводим в stderror.
        print(message) //Теперь пишем ошибку в консоль.
    }
}
```

❶ Используем синглтон, чтобы любой код мог добраться до обработчика ошибок (синглтоны часто не очень хорошая практика, но подходят для этого примера).

❷ Сообщение, если у вас нет больше информации, которую можно показать пользователю.

❸ ErrorHandler показывает ошибки без пользовательской информации в виде универсального сообщения.

❹ ErrorHandler передает ошибку напрямую пользователю, если она соответствует LocalizedErrors.

❺ Не показано в коде: можно вывести предупреждение в iOS или OS X. Вы также пишете ошибку в консоль или файл.

Реализация централизованного обработчика ошибок

Давайте посмотрим, как лучше всего вызвать централизованный обработчик ошибок. Поскольку вы централизуете обработку ошибок, `RecipeExtractor` не должен возвращать опционал и обрабатывать ошибки. Если вызывающая функция также рассматривает опционал как ошибку, это может привести к двойной обработке ошибок. Вместо этого `RecipeExtractor` может вернуть обычный `Recipe` (не опционал) и передать ошибку вызывающей программе, как показано в приведенном ниже коде. Затем вызывающая программа может передать любую ошибку центральному обработчику ошибок.

Листинг 6.14. RecipeExtractor и ключевое слово throws

```
struct RecipeExtractor {

    let html: String

    func extractRecipe() throws -> Recipe { ❶
        return try parseHTML(html) ❷
    }

    private func parseHTML(_html: String) throws -> Recipe {
        let ingredients = try extractIngredients(html)
        let steps = try extractSteps(html)
        return Recipe(ingredients: ingredients, steps: steps)
    }

    // ...Пропускаем часть кода
}
let html = ... // Можно получить html из исходника
```

```
let recipeExtractor = RecipeExtractor(html: html)

do {
   let recipe = try recipeExtractor.extractRecipe() ❸
} catch {
   ErrorHandler.default.handleError(error) ❸
}
```

❶ Теперь `extractRecipe` не обрабатывает ошибки и имеет ключевое слово `throws`, позволяя вызывающей программе справляться с любыми ошибками. Он может прекратить возвращать опциональный рецепт (Recipe). Вместо этого он может вернуть обычный рецепт.

❷ Любая ошибка передается вызывающей программе.

❸ Теперь вызывающая программа может перехватить ошибку и передать ее центральному обработчику ошибок, который знает, как с ней работать. Обратите внимание, что вам не нужно определять ошибку в операторе `catch`.

Если вы централизуете обработку ошибок, то отделяете ее от кода, ориентированного на подход «счастливый путь» (happy path), и поддерживаете приложение в хорошем состоянии. Вы не только предотвращаете дублирование. Кроме того, вам проще менять способ обработки ошибок. Например, вы можете решить отображать ошибки другим способом – например, в виде уведомления вместо диалогового окна, – и вам нужно только изменить это в одном месте.

Сложность состоит в том, что теперь у вас может быть один большой тип для обработки ошибок, который рискует стать гигантским и сложным. В зависимости от потребностей и размера вашего приложения можно разделить этот тип на обработчики поменьше, которые будут подключаться к большому обработчику и в дальнейшем могут специализироваться на обработке определенных ошибок.

6.2.4. Упражнения

3

В чем состоит недостаток передачи сообщений для пользователя внутри ошибки?

4

Приведенный ниже код не компилируется. Какие два изменения в `loadFile` можно сделать, чтобы скомпилировать код (не прибегая к использованию `try?` и `try!`)?

```
enum LoadError {
   case couldntLoadFile
}

func loadFile(name: String) -> Data? {
   let url = playgroundSharedDataDirectory.appendingPathComponent(name)
   do {
```

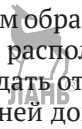
```
        return try Data(contentsOf: url)
    } catch let error as LoadError {
        print("Can't load file named \(name)")
        return nil
    }
}
```

6.3. Создание симпатичных API

Работать в приложении с API, которые выдают ошибки чаще, чем подающий высшей лиги бросает свой мяч, не очень весело. Когда API с упорством безумца генерируют ошибки, их реализация может стать помехой. Бремя обработки этих ошибок ложится на разработчика. Разработчики могут начать перехватывать все ошибки в одной большой сети и обрабатывать их одинаково или дать низкоуровневым ошибкам возможность распространяться на клиента, который не всегда знает, что с ними делать. Иногда ошибки доставляют неудобства, потому что разработчику может быть непонятно, что делать с каждой ошибкой. Кроме того, разработчики могут перехватывать ошибки с помощью комментария `// TODO: Implement`, который живет вечно, поглощая одновременно как небольшие, так и серьезные ошибки, оставляя критические проблемы незамеченными.

В идеале каждой ошибке следует уделять самое пристальное внимание. Но в реальном мире нужно соблюдать сроки, запускать услуги и успокаивать руководителей проектов. Обработка ошибок может ощущаться как препятствие, которое тормозит вас, что иногда приводит к тому, что разработчики выбирают легкий путь.

Кроме того, Swift обрабатывает ошибки таким образом, что нельзя знать наверняка, какие ошибки выдает функция. Конечно, располагая некоторой документацией, вы можете сообщить, каких ошибок ожидать от функции или метода. Но по моему опыту, наличие на 100 % самой последней документации может быть таким же распространенным явлением, как и лох-несское чудовище. Функции начинают выдавать новые ошибки или прекращают выдавать несколько ошибок разом, и есть вероятность, что с течением времени вы можете пропустить пару-другую ошибок.



API быстрее и проще в реализации, если они редко выдают ошибки. Но вы должны убедиться, что не ставите под угрозу качество приложения. Учитывая недостатки при обработке ошибок, давайте рассмотрим методы, которые сделают ваши API более удобными и простыми в реализации, при этом обращая внимание на проблемы, которые могут возникнуть в вашем коде.

6.3.1. Сбор достоверных данных в типе

Можно уменьшить объем работы по обработке ошибок, который вам нужно выполнить, собирая валидные данные внутри типа.

Например, первая попытка подтвердить номер телефона – применить функцию `validatePhoneNumber`, а затем использовать ее постоянно, когда это необ-

ходимо. Хотя наличие функции validatePhoneNumber не является ошибкой, вы быстро узнаете, как ее улучшить, в следующем листинге.

Листинг 6.15. Подтверждение номера телефона

```
enum ValidationError: Error {
   case noEmptyValueAllowed
   case invalidPhoneNumber
}

func validatePhoneNumber (_ text: String) throws {
   guard !text.isEmpty else {
      throw ValidationError.noEmptyValueAllowed ❶
   }

   let pattern = «^(\\([0-9]{3}\\) |[0-9]{3}-)[0-9]{3}-[0-9]{4}$»
   if text.range(of: pattern, options: .regularExpression, range:
   ➥ nil, locale: nil) == nil {
      throw ValidationError.invalidPhoneNumber ❶
   }
}

do {
   try validatePhoneNumber ("(123) 123-1234") ❷
   print("Phonenumber is valid")
} catch {
   print(error)
}
```

❶ Функция validatePhoneNumber выдает ошибку, если номер недействителен.

❷ Ошибку нужно перехватить; например, с помощью оператора do catch. Но это должно происходить каждый раз для одного и того же номера телефона.

При таком подходе вы можете в конечном итоге проверять одну и ту же строку несколько раз: например, один раз при вводе данных в форму, еще раз перед выполнением API-вызова и снова при обновлении профиля. В ходе всего этого процесса вы возлагаете на разработчика бремя обработки ошибки.

Вместо этого можно зафиксировать подлинность номера телефона в типе, создав новый тип, даже если номер телефона представляет собой только одну строку, как показано ниже. Вы создаете тип PhoneNumber и даете ему инициализатор, который может сообщать об ошибках, и он будет проверять номер телефона. Этот инициализатор либо выдаст ошибку, либо вернет правильный тип PhoneNumber, поэтому вы сможете перехватить любые ошибки прямо при создании типа.

Листинг 6.16. Тип PhoneNumber

```
struct PhoneNumber {
```

```
    let contents: String

    init(_ text: String) throws { ❶
       guard !text.isEmpty else {
          throw ValidationError.noEmptyValueAllowed
       }

       let pattern = "^(\\([0-9]{3}\\) |[0-9]{3}-)[0-9]{3}-[0-9]{4}$"
       if text.range(of: pattern, options: .regularExpression, range: nil,
          locale: nil) == nil {
          throw ValidationError.invalidPhoneNumber
       }
       self.contents = text ❷
    }
}

do {
    let phoneNumber = try PhoneNumber("(123) 123-1234") ❸
    print(phoneNumber.contents) // (123) 123-1234 ❹
} catch {
    print(error)
}
```

❶ Создаем проваливающийся инициализатор.

❷ Если номер телефона подтвержден, значение сохраняется.

❸ Создаем тип PhoneNumber. Вы должны использовать ключевое слово try, потому что у Phone Number есть инициализатор, генерирующий ошибку.

❹ Вы можете прочитать номера телефона в приложении.

После того как вы получите PhoneNumber, можете безопасно передавать его по своему приложению, будучи уверенными в том, что он является действительным и вам не нужно перехватывать ошибки всякий раз, когда вам нужно получить значение номера телефона. С этого момента ваши методы могут принимать тип PhoneNumber, и, просто взглянув на сигнатуры методов, вы будете знать, что имеете дело с действительным номером телефона.

6.3.2. Ключевое слово try?

Можно предотвратить распространение и другими способами. При создании типа PhoneNumber можно рассматривать его как опционал, чтобы ошибка не пошла выше. Если функция является функцией, генерирующей ошибку, но вас не интересуют причины сбоя, можно рассмотреть возможность преобразования результата такой функции в опционал с помощью ключевого слова try?, как показано здесь.

Листинг 6.17. Применение ключевого слова try?

```
let phoneNumber = try? PhoneNumber("(123) 123-1234")
```

```
print(phoneNumber) // Опционал (PhoneNumber(contents: "(123) 123-1234"))
```

Используя ключевое слово try?, вы останавливаете распространение ошибок. Можно использовать это ключевое слово, чтобы свести различные причины ошибок в один опционал. В этом случае PhoneNumber нельзя создать, а с помощью try? вы указываете на то, что вас не интересует причина или ошибка, просто успешно прошел процесс создания или нет.

6.3.3. Ключевое слово try!

Вы можете утверждать, что ошибки не произойдет. В этом случае, например, когда вы выполняете принудительное извлечение, вы либо правы, либо у вас происходит сбой.

Если вы должны использовать try! для создания PhoneNumber, вы утверждаете, что создание не будет неудачным.

Листинг 6.18. Применение ключевого слова try!

```
let phoneNumber = try! PhoneNumber("(123) 123-1234")
print(phoneNumber) // PhoneNumber(содержимое: "(123) 123-1234")
```

Ключевое слово try! избавляет вас от извлечения опционала. Но если вы ошиблись, происходит сбой приложения:

```
let phoneNumber = try! PhoneNumber("Not a phone number") // Аварийный сбой
```

Как и при принудительном извлечении, используйте try!, только когда вы знаете лучше, чем компилятор. В противном случае это игра в русскую рулетку.

6.3.4. Возвращение опционалов

Опционалы – это способ обработки ошибок: либо есть значение, либо его нет. Можно использовать опционалы, чтобы сообщить, что что-то не так, а это – элегантная альтернатива генерации ошибок.

Допустим, вы хотите загрузить файл с игровых площадок Swift, который может дать сбой, но причина сбоя не имеет значения. Чтобы снять бремя обработки ошибок с вызывающих программ, можно сделать так, чтобы ваша функция возвращала опциональное значение Data в случае сбоя.

Листинг 6.19. Возвращаем опционал

```
func loadFile(name: String) -> Data? { ❶
   let url = playgroundSharedDataDirectory.appendingPathComponent(name)
   return try? Data(contentsOf: url) ❷
}
```

❶ Функция возвращает опциональное значение Data

❷ Вы перехватываете любые ошибки, которые приходят из Data, и превращаете их в опционал.

Если у функции имеется единственная причина для сбоя и она возвращает значение, практическое правило – возвращать опционал, вместо того чтобы генерировать ошибку. Если причина сбоя имеет значение, можно сгенерировать ошибку.

Если вы не знаете, что интересует вызывающую программу, и не возражаете против введения типов ошибок, вы по-прежнему можете выдать ошибку. Вызывающая программа всегда может решить превратить ошибку в опционал при необходимости, используя ключевое слово try?.

6.3.5. Упражнение

5

Назовите по крайней мере три способа, чтобы разработчикам было проще работать с API, которые генерируют ошибки.

6.4. В заключение

Как вы убедились, обработка ошибок может казаться простой на бумаге, но важно применять передовые методы при работе с ошибками. Одним из худших случаев является то, что ошибки проглатываются или игнорируются. Используя передовые методики, приведенные в этой главе, вы, я надеюсь, приобрели хороший арсенал приемов для борьбы с этими ошибками и их адекватной обработки. Если вам нравятся другие способы обработки ошибок, вам повезло – в главе 11 рассказывается об ошибках в асинхронной среде.

Резюме

- Хотя ошибки обычно являются перечислениями, любой тип может реализовать протокол `Error`.
- Определить, какие ошибки генерирует функция, невозможно, но можно использовать Quick Help, чтобы смягчить удар.
- Продолжайте генерировать код в предсказуемом состоянии, когда возникает ошибка. Можно добиться предсказуемого состояния с помощью неизменяемых функций, работая с копиями или временными значениями и используя оператор `defer` для отмены любых изменений, которые могут произойти до того, как будет сгенерирована ошибка.
- Можно обрабатывать ошибки четырьмя способами: `do catch`, `try?` и `try!` – и распространять их выше по стеку.
- Если функция не перехватывает все ошибки, любая возникающая ошибка распространяется выше по стеку.
- Ошибка может содержать техническую информацию для устранения неполадок. Пользовательские сообщения могут быть выведены из технической информации путем реализации протокола `LocalizedError`.
- Реализуя `CustomNSError`, можно преобразовать ошибку с `NSError`.

- Полезно использовать централизованную обработку ошибок. Благодаря этому можно легко изменить способ обработки ошибок.
- Можно предотвратить генерацию ошибок, превратив их в опционалы с помощью ключевого слова `try?`.
- Если вы уверены, что ошибки не произойдет, можно получить значение из функции, генерирующей ошибку, с помощью ключевого слова `try!`, рискуя вызвать сбой приложения.
- Если причина сбоя одна, рассмотрите возможность возврата опционала, вместо того чтобы создавать функции, генерирующие ошибки.
- Полезно использовать сбор достоверных данных в типе. Вместо частого использования функции, генерирующей ошибки, создайте тип с инициализатором, который генерирует ошибки, и передавайте этот тип, будучи уверенными, что он проверен.

Ответы

1

Назовите один или более недостатков того, как Swift обрабатывает ошибки и как их компенсировать.

Функции помечены как генерирующие ошибки, поэтому они перекладывают бремя проблемы на разработчика. Но функции не показывают, какие ошибки выдают.

Можно добавить аннотацию Quick Help к функциям, чтобы поделиться, какие ошибки могут быть выданы.

2

Назовите три способа убедиться, что функции, генерирующие ошибки, возвращаются в исходное состояние после выдачи ошибок.

- Используйте неизменяемые функции;
- работайте с копиями или временными значениями;
- применяйте оператор defer, чтобы отменить изменение, которое произошло до того, как была сгенерирована ошибка.

3

В чем состоит недостаток передачи сообщений для пользователя внутри ошибки?

В этом случае будет сложнее разграничить техническую информацию для отладки и информацию, которая отображается пользователю.

4

Приведенный ниже код не компилируется. Какие два изменения в `loadFile` можно сделать, чтобы скомпилировать код (не прибегая к использованию `try?` и `try!`)?

Заставьте `loadFile` перехватывать все ошибки, а не только одну конкретную. Или заставьте `loadFile` генерировать ошибку, чтобы снова распространить ее.

5

Назовите по крайней мере три способа, чтобы разработчикам было проще работать с API, которые генерируют ошибки.

- Перехват ошибки при создании типа, поэтому ошибка будет обрабатываться только в момент создания типа, а не при передаче значения;
- возврат опционала вместо выдачи ошибки, когда причина сбоя одна;
- преобразование ошибки в опционал с помощью ключевого слова `try?` и возврат опционала;
- предотвращение распространения с помощью ключевого слова `try!`.

Глава 7. Обобщения

В этой главе:

- как и когда писать обобщения;
- как рассматривать обобщения;
- ограничение обобщений с помощью одного или более протоколов;
- использование протоколов `Equatable`, `Comparable` и `Hashable`;
- создание повторно используемых типов;
- как подклассы работают с обобщениями.

Обобщения – это ключевой компонент Swift, и поначалу их сложно понять. Нет ничего постыдного, если вы держитесь подальше от обобщений. Возможно, это связано с тем, что иногда они пугают или сбивают с толку. Подобно тому, как плотник может выполнять работу без молотка, так и вы можете разрабатывать программное обеспечение, не используя обобщения. Но включение обобщений в процесс разработки программного обеспечения, безусловно, полезно, потому что с помощью обобщений вы можете создавать код, который будет работать при текущих и последующих требованиях, и это помогает избежать повторений. Благодаря использованию обобщений ваш код становится более лаконичным, мощным, современным, и в нем становится меньше шаблонов.

Эта глава начинается с рассмотрения преимуществ обобщений, а также того, когда и как их применять. Мы начнем с малого, но затем увеличим сложность, рассматривая более сложные варианты использования. Обобщения – краеугольный камень Swift, поэтому их стоит усвоить, потому что они будут часто появляться как в книге, так и во время чтения и написания кода в реальных условиях.

После достаточного количества разоблачений и моментов типа «ага!» вы начнете чувствовать себя комфортно, используя обобщения для написания жесткого, многократно используемого кода.

Вы узнаете о цели и преимуществах обобщений и о том, как они могут спасти вас от написания дублирующего кода. Затем вы подробнее познакомитесь с тем, что обобщения делают за кулисами и как их рассматривать. На этом этапе обобщения должны быть не такими страшными.

В Swift обобщения и протоколы жизненно важны для создания полиморфного кода. Вы узнаете, как использовать протоколы для ограничения обобщений, что позволяет создавать обобщенные функции с определенным поведением. Попутно в этой главе представлены два важных протокола: `Equatable` и `Comparable`.

Далее вы узнаете, как ограничить обобщения несколькими протоколами и как улучшить удобочитаемость с помощью оператора `where`. Вы познакомитесь с протоколом `Hashable`. Это еще один важный протокол, который часто встречается при работе с обобщениями.

Все соединяется воедино, когда вы будете создавать гибкую структуру. Эта структура будет содержать несколько ограниченных обобщений и реализовывать

протокол Hashable. Попутно вы получите представление о функции Swift, которая носит название *условное соответствие*.

Чтобы еще больше укрепить свои знания в области обобщений, вы узнаете, что они не всегда смешиваются с подклассами. При одновременном использовании обоих методов понадобится знание нескольких специальных правил.

Когда обобщения станут частью вашего инструментария, вы сможете уменьшить свою кодовую базу с помощью элегантного и многократно используемого кода. Давайте посмотрим.

7.1. Преимущества обобщений

Присоединяйтесь!

Будет более познавательно и веселее, если вы будете знакомиться с кодом по мере прохождения этой главы. Исходный код можно скачать по адресу http://mng.bz/nQE8.

Давайте начнем с малого, чтобы прочувствовать обобщения. Представьте, что у вас есть функция firstLast, которая извлекает первый и последний элементы из массива целых чисел.

Листинг 7.1. Первый и последний элементы

```
let (first, last) = firstLast(array: [1,2,3,4,5])

print(first) // 1
print(last) // 5

func firstLast(array: [Int]) -> (Int, Int) {
   return (array[0], array[array.count-1])
}
```

Теперь вам нужно проделать то же самое для массивов, содержащих String. В приведенном ниже листинге вы определяете аналогичную функцию, за исключением того, что она указана для типа String. Swift знает, какую функцию с таким же именем выбрать.

Листинг 7.2. firstLast с массивом строк

```
func firstLast(array: [String]) -> (String, String) {
   return (array[0], array[array.count-1])
}

let (first, last) = firstLast(array: ["pineapple", "cherry",
➥ "steam locomotive"])

print(first) // "pineapple"
print(last) // "steam locomotive"
```

Необходимость создания новой функции для каждого типа не вполне масштабируется. Если вы хотите использовать этот метод для массива типов Double, UIImage или пользовательского типа Waffle, вам придется каждый раз создавать новые функции.

Кроме того, вы можете написать одну функцию, которая работает с Any, но тогда возвращаемое значение кортежа будет (Any, Any), а не (String, String) или (Int, Int). Вы должны были бы произвести нисходящее приведение (Any, Any) до (String, String) или (Int, Int) во время выполнения.

Уйти от стереотипов и избежать использования Any – вот где могут помочь обобщения. С их помощью можно создать полиморфную функцию на этапе компиляции. Полиморфный код означает, что он может работать для нескольких типов. В случае с обобщенной функцией нужно будет определить функцию только один раз, и она будет работать с Int, String и любым другим типом, включая пользовательские типы, которые вы ввели или еще не написали. При использовании обобщений вам не придется иметь дело с Any, что спасет вас от нисходящего преобразования типов во время выполнения.

7.1.1. Создание обобщенной функции

Давайте сравним необобщенную и обобщенную версии функции firstLast.

Мы добавим обобщенный параметр типа <T> в сигнатуру функции. Это поможет нам ввести обобщение в нашу функцию, к которому можно обращаться в остальной части функции. Обратите внимание, что все, что вы делаете, – это определяете <T> и заменяете все вхождения Int на T.

Листинг 7.3. Сравнение сигнатуры обобщенной и необобщенной функций

```
// Версия без использования обобщения.
func firstLast(array: [Int]) -> (Int, Int) {
   return (array[0], array[array.count-1])
}л

// Версия с обобщением.
func firstLast<T>(array: [T]) -> (T, T) {
   return (array[0], array[array.count-1])
}
```

Чашку T²?

Обычно обобщение часто определяется как тип T, что означает *Type* (тип). Обобщения принято называть как-нибудь абстрактно, например T, U или V. Обобщения также могут быть словами, например Wrapped, когда они используются в Optional.

Объявив обобщение с помощью синтаксиса <T>, вы можете обращаться к нему в остальной части функции, например array:[T] и его возвращаемый тип (T, T).

[2] В английском языке буква T и слово tea (чай) звучат одинаково. – *Прим. перев.*

Можно обращаться к T в теле функции путем его расширения, как показано здесь.

Листинг 7.4. Обращение к обобщению из тела функции

```
func firstLast<T>(array: [T]) -> (T, T) {
   let first: T = array[0]
   let last: T = array[array.count-1]
   return (first, last)
}
```

Можно увидеть свою функцию в действии. Обратите внимание, что она работает с несколькими типами. Можно сказать, что ваша функция не зависит от типа.

Листинг 7.5. Обобщенная функция в действии

```
let (firstString, lastString) = firstLast(array: ["pineapple", "cherry",
   "steam locomotive"])
print(firstString) // "pineapple"
print(lastString) // "steam locomotive"
```

Если бы вы проверили значения `firstString` или `lastString`, то увидели бы, что они имеют тип `String`, а не `Any`. Вы также можете передавать пользовательские типы, что демонстрирует далее структура `Waffle`:

Листинг 7.6. Пользовательские типы

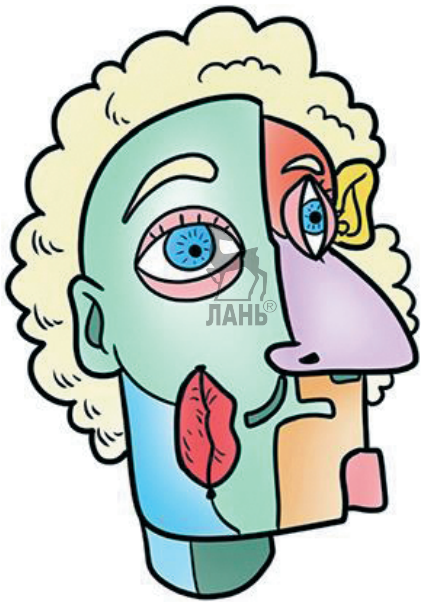
```
// Пользовательские типы тоже работают.
struct Waffle {
   let size: String
}

let (firstWaffle: Waffle, lastWaffle: Waffle) = firstLast(array: [
   Waffle(size: «large»),
   Waffle(size: «extra-large»),
   Waffle(size: «snack-size»)
])

print(firstWaffle) // Waffle(size: «large»)
print(lastWaffle) // Waffle(size: «snack-size»)
```

Это все, что нужно. Благодаря обобщениям можно использовать одну функцию для массива, содержащего `Int`, `String`, `Waffle` или что-либо еще, и получать обратно конкретные типы. Вы не манипулируете `Any` или не осуществляете нисходящее преобразование типа во время выполнения; компилятор объявляет все на этапе компиляции.

Почему бы сразу не писать обобщения?

Начать с функции, в которой нет обобщений, а затем заменить все типы на обобщения проще, чем начать писать обобщенную функцию с самого начала. Но если вы чувствуете себя уверенно, тогда вперед!

7.1.2. Рассмотрение обобщений



Рассмотрение обобщений может быть сложным и абстрактным. Иногда `T` – это строка (`String`), а иногда – целое число (`Int`) и т. д. Можно поместить и то, и другое внутрь массива с помощью обобщенной функции, как показано здесь.

Листинг 7.7. Помещение значения внутрь массива

```
func wrapValue<T>(value: T) -> [T] {
   return [value]
}

wrapValue (value: 30) // [30]
wrapValue (value: "Howdy!") // ["Howdy!"]
```

Но нельзя специализировать тип из тела функции; это означает, что нельзя передать `T` и превратить его в значение `Int` из тела функции.

Листинг 7.8. Неисправная функция

```
func illegalWrap<T>(value: T) -> [Int] {
   return [value]
}
```

В листинге 7.8 выдается ошибка компилятора Swift:

```
Cannot convert value of type 'T' to expected element type 'Int'.
```

При работе с обобщениями Swift генерирует специализированный код на этапе компиляции. Можно рассматривать это как обобщенную функцию `wrapValue`, превращающуюся в специализированные функции, которые вы можете использовать. Это напоминает призму, через которую проходит белый свет, а на выходе появляются разные цвета (см. рис. 7.1 и приведенный ниже листинг):

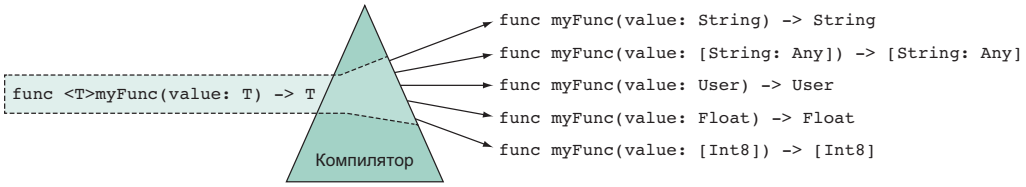


Рис 7.1. Обобщенный код превращается в специализированный во время компиляции

Листинг 7.9. Функция wrapValue

```
// Имеется обобщенная функция...
func wrapValue<T>(value: T) -> [T] { ... }

// ... вы получаете доступ к специализированным функциям.
func wrapValue (value: Int) -> [Int] { ... }
func wrapValue (value: String) -> [String] { ... }
func wrapValue (value: YourOwnType) -> [YourOwnType] { ... }
```

Swift создает за кулисами несколько функций `wrapValue` – этот процесс носит название *мономорфизация*, когда компилятор превращает полиморфный код в конкретный единый код. Swift достаточно умен, чтобы предотвратить генерацию кода тоннами, во избежание больших двоичных файлов. Он использует различные приемы, включающие метаданные, чтобы ограничить объем генерируемого кода. В этом случае компилятор создает низкоуровневую функцию `wrapValue`. Затем для соответствующих типов Swift генерирует метаданные под названием *value witness tables (VWTs)*. Во время выполнения Swift передает соответствующие метаданные низкоуровневому представлению `wrapValue`, когда это необходимо.

Компилятор достаточно умен, чтобы минимизировать количество генерируемых метаданных. Поскольку Swift может принимать разумные решения касательно того, когда и как генерировать код, вам не нужно беспокоиться о больших двоичных файлах, что также известно как *раздувание кода*, или слишком продолжительном времени компиляции!

Еще одно существенное преимущество обобщений в Swift заключается в том, что вы знаете, с какими значениями имеете дело на этапе компиляции. Если бы вы проверили возвращаемое значение `wrapValue`, используя клавишу **Alt** в Xcode, вы бы уже увидели, что оно возвращает [`String`] или [`Int`], или то, что вы там по-

местили. Вы можете проверять типы, прежде чем даже подумать о запуске приложения, что упрощает рассмотрение полиморфных типов.

7.1.3. Упражнение

1

Какие из этих функций компилируются? Подтвердите это, выполнив код:

```
func wrap<T>(value: Int, secondValue: T) -> ([Int], U) {
   return ([value], secondValue)
}

func wrap<T>(value: Int, secondValue: T) -> ([Int], T) {
   return ([value], secondValue)
}

func wrap(value: Int, secondValue: T) -> ([Int], T) {
   return ([value], secondValue)
}

func wrap<T>(value: Int, secondValue: T) -> ([Int], Int) {
   return ([value], secondValue)
}

func wrap<T>(value: Int, secondValue: T) -> ([Int], Int)? {
   if let secondValue = secondValue as? Int {
      return ([value], secondValue)
   } else {
      return nil
   }
}
```

2

В чем состоит преимущество использования обобщений по сравнению с типом Any (например, `func <T> (process: [T])` вместо `func (process: [Any])`)?

7.2. Ограничение обобщений

Ранее вы видели, как работать с обобщенным типом T, который может быть чем угодно. Данный тип в предыдущих примерах является неограниченным обобщением. Но когда тип может быть чем угодно, с ним тоже много не сделать.

Можно ограничить его с помощью протокола; давайте посмотрим, как это работает.

7.2.1. Нужна функция ограничения

Представьте, что вы хотите написать обобщенную функцию, которая дает вам самое низкое значение внутри массива. Эта функция настраивается так, чтобы она работала для массива с любым типом, как показано здесь.

Листинг 7.10. Запуск функции lowest

```
lowest([3,1,2]) // Опционал(1)
lowest([40.2, 12.3, 99.9]) // Опционал (12.3)
lowest(["a", "b", "c"]) // Опционал ("a")
```

Функция `lowest` может возвращать nil, когда переданный массив пуст, поэтому значения обозначены как `Optional`.

Ваша первая попытка – создать функцию с обобщенным параметром типа `T`. Но в следующем коде вы быстро обнаружите, что в сигнатуре функции чего-то не хватает.

Листинг 7.11. Функция lowest (будет скомпилирована в ближайшее время, но не сейчас)

```
// Сигнатура функции еще не закончена!
func lowest<T>(_ array: [T]) -> T? {
   let sortedArray = array.sorted { (lhs, rhs) -> Bool in ❶
      return lhs < rhs ❷
   }
   return sortedArray.first ❸
}

lowest([1,2,3])
```

❶ Сортируем массив, передавая ему замыкание.

❷ В этой строке кода сравниваются два значения внутри массива. Если `lhs` ниже, чем `rhs`, вы получите восходящий массив.

❸ Функция lowest возвращает самое низкое сравниваемое значение.

lhs и rhs

`lhs` означает «левая сторона» (left hand side), а `rhs` означает «правая сторона» (right hand side). Это соглашение, используемое при сравнении двух одинаковых типов.

К сожалению, код в листинге 7.11 не сработает. Swift выдает следующую ошибку:

```
error: binary operator '<' cannot be applied to two 'T' operands
if lowest < value {
~~~~~~ ^ ~~~~~
```

Ошибка возникает из-за того, что `T` может быть чем угодно. Тем не менее вы выполняете над ним действия, например сравниваете его с другим `T` с помощью

оператора <. Но так как T обозначает все, что угодно, функция lowest не знает, что она может сравнивать значения T.

Давайте выясним, как исправить ситуацию с помощью протокола. Сперва нам нужно будет совершить небольшую экскурсию, чтобы познакомиться с двумя ключевыми протоколами, которые нам понадобятся, чтобы завершить функцию lowest.

7.2.2. Протоколы Equatable и Comparable

Протоколы определяют интерфейс с требованиями, такими как функции или переменные для реализации. Типы, которые соответствуют протоколу, реализуют необходимые функции и переменные. Вы уже наблюдали это в предыдущих главах, когда настраивали типы в соответствии с протоколами CustomStringConvertible и RawRepresentable.

Преобладающий протокол Equatable позволяет проверить, являются ли два типа одинаковыми. Такими типами могут быть целые числа, строки и многие другие, включая ваши пользовательские типы:

```
5 == 5 // Истинно.
30231 == 2 // Ложно.
"Generics are hard!" == "Generics are easy!" // Ложно.
```

Когда тип соответствует протоколу Equatable, этот тип должен реализовать статическую функцию ==, как показано здесь.

Листинг 7.12. Equatable

```
public protocol Equatable {
   static func == (lhs: Self, rhs: Self) -> Bool
}
```

Примечание

В случае со структурами и перечислениями Swift может синтезировать реализацию Equatable, что избавит вас от ручной реализации метода ==.

Еще один распространенный протокол, который предлагает Swift, – Comparable. Типы, соответствующие Comparable, можно сравнивать друг с другом, чтобы увидеть, какое значение больше или меньше другого значения:

```
5 > 2 // Истинно.
3.2 <= 1.3 // Ложно.
"b" > "a" // Истинно.
```

Интересно, что Comparable также состоит из статических функций, но это не является обязательным требованием для протоколов.

Листинг 7.13. Протокол Comparable

```
public protocol Comparable : Equatable { ❶
static func < (lhs: Self, rhs: Self) -> Bool ❷
static func <= (lhs: Self, rhs: Self) -> Bool ❷
static func >= (lhs: Self, rhs: Self) -> Bool ❷
static func > (lhs: Self, rhs: Self) -> Bool ❷
}
```

❶ Типы, реализующие протокол Comparable, также должны будут реализовать Equatable.

❷ Когда тип соответствует протоколу Comparable, <, <=,> = и > plus == становятся доступными для использования с типом.

Оба протокола широко распространены, и оба находятся в базовой библиотеке Swift.

7.2.3. Ограничивать значит специализировать

Вернемся к проблеме. Наша функция lowest сравнивала два типа T, но T по-прежнему не соответствует протоколу Comparable. Можно специализировать эту функцию, указав, что T соответствует Comparable, как показано здесь.

Листинг 7.14. Ограничение обобщения

```
// Прежде. Не компилировалось.
func lowest<T>(_ array: [T]) -> T? {

// После. Приведенная ниже сигнатура верна.
func lowest<T: Comparable>(_ array: [T]) -> T? {
```

Внутри области видимости функции lowest T обозначает все, что соответствует Comparable. Код снова компилируется и работает с несколькими типами Comparable, такими как целые числа, числа с плавающей точкой, строки и все остальное, что соответствует протоколу Comparable. Вот полный вариант функции:

Листинг 7.15. Функция lowest

```
func lowest<T: Comparable>(_ array: [T]) -> T? {
   let sortedArray = array.sorted { (lhs, rhs) -> Bool in
      return lhs < rhs
   }

   return sortedArray.first
}
```

Ранее вы узнали, что sorted принимает два значения и возвращает Bool. Но sorted может использовать возможности протоколов, если все его элементы со-

ответствуют протоколу Comparable. Нужно только вызвать метод sorted без аргументов, что делает тело функции намного короче.

Листинг 7.16. Функция lowest (сокращенный вариант)

```
func lowest<T: Comparable>(_ array: [T]) -> T? {
   return array.sorted().first
}
```

7.2.4. Реализация протокола Comparable

Мы также можем применять функцию lowest к своим типам. Сначала создадим перечисление, соответствующее Comparable. Это перечисление представляет три королевских ранга, которые вы можете сравнивать. Затем вы передадите массив этого перечисления функции lowest.

Листинг 7.17. Перечисление RoyalRank, соответствующее протоколу Comparable

```
enum RoyalRank: Comparable { ❶
   case emperor
   case king
   case duke

   static func <(lhs: RoyalRank, rhs: RoyalRank) -> Bool { ❷
      switch (lhs, rhs) { ❸
         case (king, emperor): return true
         case (duke, emperor): return true
         case (duke, king): return true
         default: return false
      }
   }
}
```

❶ Перечисление RoyalRank реализует протокол Comparable.

❶ Реализуем метод < для соответствия протоколу.

❸ Используем сопоставление с образцом, чтобы проверить, является ли один кейс ниже, чем другой.

Чтобы это перечисление соответствовало протоколу Comparable, обычно нужно реализовать метод == из протокола Equatable. К счастью, Swift синтезирует этот метод за вас, избавляя нас от написания реализации. Кроме того, нам также необходимо реализовать четыре метода из Comparable. Но речь идет только о методе <, потому что в случае с реализациями методов < и == Swift может вывести все другие реализации для Comparable. В результате нам нужно всего лишь реализовать метод <, избавляя себя от написания шаблонов.

Мы заставили перечисление соответствовать протоколу Comparable и косвенно Equatable – поэтому теперь мы можем сравнивать ранги друг с другом или передавать их массив функции lowest, как показано ниже.

Листинг 7.18. Протокол Comparable в действии

```
let king = RoyalRank.king
let duke = RoyalRank.duke

duke < king // Истинно.
duke > king // Ложно.
duke == king // Ложно.

let ranks: [RoyalRank] = [.emperor, .king, .duke]
lowest(ranks) // .duke
```

Одним из преимуществ обобщений является то, что вы можете писать функции для типов, которых еще даже не существует. Если в будущем вы будете вводить новые типы Comparable, то можете передать их функции lowest без лишних хлопот!

7.2.5. Ограничение в сравнении с гибкостью

Не все типы соответствуют протоколу Comparable. Например, если вы передали массив логических типов данных, то получите ошибку:

```
lowest([true, false])
error: in argument type '[Bool]', 'Bool' does not conform to expected type
'Comparable'
```

Логические типы данных не соответствуют протоколу Comparable. В результате функция lowest не будет работать с таким массивом.

Ограничение обобщения означает гибкость для функциональности. Ограничиваемое обобщение становится более специализированным, но менее гибким.

7.3. Ряд ограничений

Зачастую один протокол не решит все ваши проблемы, если его ограничить.

Представьте себе сценарий, в котором вы хотите отслеживать не только самые низкие значения внутри массива, но и их вхождения. Вам нужно будет сравнить значения и, вероятно, сохранить их в словаре, чтобы отслеживать их вхождения. Если вам нужно обобщение, которое можно сравнивать и хранить в словаре, вам необходимо, чтобы оно соответствовало протоколам Comparable и Hashable.

Мы еще не рассматривали протокол Hashable. Давайте сделаем это сейчас, прежде чем увидеть, как ограничить обобщение несколькими протоколами.

7.3.1. Протокол Hashable

Типы, соответствующие протоколу Hashable, могут быть сведены к одному целому числу, *хеш-значению*. Хеширование выполняется с помощью функции, которая превращает тип в целое число (см. рис. 7.2).

Примечание

Функции хеширования – сложная тема. Их подробное изучение выходит за рамки этой книги.

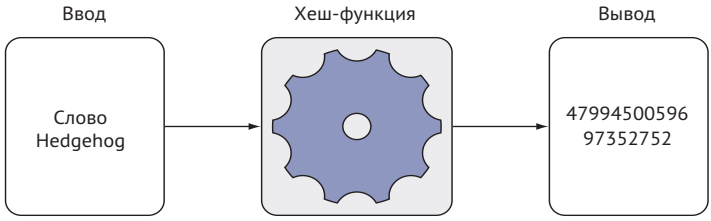


Рис. 7.2. В Swift хеш-функция превращает значение в целое число

Типы, соответствующие протоколу Hashable, можно использовать в качестве ключей словаря или как часть набора, помимо других вариантов использования. Одним из таких распространенных типов является String.

Листинг 7.19. Строка в качестве ключа словаря

```
let dictionary = [
   "I am a key": "I am a value",
   "I am another key": "I am another value",
]
```

То же относится и к целым числам. Они также могут служить словарными ключами или храниться внутри набора (Set):

```
let integers: Set = [1, 2, 3, 4]
```

Многие встроенные типы, которые соответствуют протоколу Equatable, также соответствуют протоколу Hashable, например Int, String, Character и Double.

Присмотритесь к протоколу Hashable

Протокол Hashable определяет метод, который принимает хешер, универсальную хеш-функцию, используемую такими типами, как Set и Dictionary; тип реализации может затем передавать значения в хешер. Обратите внимание, что в этом примере протокол Hashable расширяет протокол Equatable.

Листинг 7.20. Протокол Hashable

```
public protocol Hashable : Equatable { ❶
   func hash(into hasher: inout Hasher) ❷
// ... Детали опущены.
```

❶ Типы, соответствующие Hashable, также должны соответствовать Equatable.

❷ Типы, соответствующие Hashable, должны использовать метод func hash (into hasher: inout Hasher).

Типы, придерживающиеся Hashable, должны использовать статический метод == из протокола Equatable и метод func hash (into hasher: inout Hasher) из протокола Hashable.

Примечание

Как и в случае с Equatable, Swift может бесплатно синтезировать реализации для Hashable для структур и перечислений, что продемонстрировано в разделе 7.4.

7.3.2. Комбинируем ограничения

Чтобы создать функцию lowestOccurences, как упоминалось ранее, нам нужен универсальный тип, который соответствует протоколам Hashable и Comparable. Соответствие двум протоколам возможно, когда вы ограничиваете обобщение несколькими типами, как показано на рис. 7.3 и в листинге 7.21.

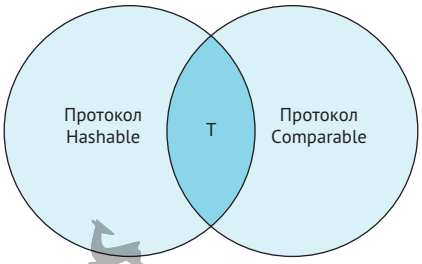


Рис. 7.3. Обобщение, соответствующее двум протоколам

У функции lowerOccurrence есть универсальный тип T, который ограничен протоколами Comparable и Hashable с помощью оператора &.

Листинг 7.21. Объединение ограничений

```
func lowestOccurrences<T: Comparable & Hashable>(values: [T]) -> [T: Int] {
    // ... Пропускаем остальную часть кода.
}
```

Теперь T можно сравнить и поместить в словарь внутри тела функции.

Если обобщенную сигнатуру становится трудновато читать, можно использовать оператор where в качестве альтернативы, которое идет в конце функции, как показано здесь.

Листинг 7.22. Оператор where

```
func lowestOccurrences<T>(values: [T]) -> [T: Int] ❶
    where T: Comparable & Hashable { ❷
    // ... Пропускаем остальную часть кода.
}
```

❶ Вы по-прежнему пишете обобщение, как и раньше.

❷ Ограничения идут в конце функции внутри оператора where.

Это два разных стиля написания обобщенных ограничений, но оба они работают одинаково.

7.3.3. Упражнения

3

Напишите функцию, которая при наличии массива возвращает словарь вхождений каждого элемента в массиве.

4

Создайте регистратор, который выводит описание универсального типа и описание отладки при его прохождении.

Подсказка: помимо протокола CustomStringConvertible, который гарантирует, что типы реализуют свойство описания, Swift также предлагает протокол CustomDebugStringConvertible, который заставляет тип реализовывать свойство debugDescription.

7.4. Создание обобщенного типа

До сих пор мы применяли обобщения к своим функциям. Но мы также можем делать типы обобщенными.

В главе 4 мы узнали, что тип Optional использует обобщенный тип Wrapped для хранения своего значения, как показано в этом листинге.

Листинг 7.23. Обобщенный тип Wrapped

```
public enum Optional<Wrapped> {
   case none
   case some(Wrapped)
}
```

Еще один обобщенный тип – Array. Мы записываем их как [Int] или [String] или что-то в этом роде. Это синтаксический сахар. Скажем по секрету: речь идет о синтаксисе Array<Element>, например Array <Int>, что также компилируется.

Давайте используем этот раздел для создания обобщенной структуры, которая поможет вам комбинировать типы, поддерживающие протокол Hashable. Цель состоит в том, чтобы прояснить, как манипулировать несколькими обобщениями одновременно во время работы с этим протоколом.

7.4.1. Желание совместить два типа, соответствующих протоколу Hashable

К сожалению, использование двух типов, которые соответствуют протоколу Hashable, в качестве ключа для словаря невозможно, даже если этот ключ состоит из двух таких типов. В частности, можно объединить две строки, которые соответствуют протоколу Hashable, в кортеж и попытаться передать его как ключ словаря, как показано ниже.

Листинг 7.24. Использование кортежа в качестве ключа для словаря

```
let stringsTuple = ("I want to be part of a key", "Me too!"»)
let anotherDictionary = [stringsTuple: "I am a value"]
```

Но Swift быстро пресекает это.

Листинг 7.25. Ошибка при использовании кортежа в качестве ключа

```
error: type of expression is ambiguous without more context
let anotherDictionary = [stringsTuple: "I am a value"]
^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Как только вы поместите два типа в кортеж, этот кортеж больше не будет соответствовать протоколу Hashable. В Swift нет способа объединения двух хеш-значений из кортежа. Мы решим эту проблему, создав тип Pair, который содержит два свойства, соответствующих протоколу Hashable.

7.4.2. Создание типа Pair

Можно объединить два типа Hashable, введя новую обобщенную структуру, которую мы назовем Pair. Тип Pair принимает два типа Hashable и сам станет соответствовать этому протоколу, как показано в листинге 7.26.

Листинг 7.26. Тип Pair в действии

```
let keyPair = Pair("I want to be part of a key", "Me too!")
let anotherDictionary = [keyPair: "I am a value"] // Это работает.
```

Pair может хранить два типа, которые соответствуют протоколу Hashable. Поначалу по наивности можно объявить одно обобщение T, как показано в следующем листинге.

Листинг 7.27. Знакомство с Pair

```
struct Pair<T: Hashable> { ❶
   let left: T ❷
   let right: T ❸

   init(_ left: T, _ right: T) {
      self.left = left
      self.right = right
```

```
    }
}
```

❶ Структура Pair содержит пару типов T, которые ограничены Hashable.

❷ Первое свойство Hashable в структуре.

❸ Второе свойство Hashable в структуре.

Pair пока еше не соответствует протоколу Hashable, но скоро мы к этому вернемся. Сейчас у нас другая проблема – догадайтесь, какая?

7.4.3. Несколько обобщений

Поскольку T используется для обоих значений, Pair ограничивается типами, как показано ниже:

Листинг 7.28. Структура Pair ограничивается типами: свойства left и right одного типа

```
struct Pair {
   let left: Int
   let right: Int
}

struct Pair {
   let left: String
   let right: String
}
```

В настоящее время у нас не может быть структуры Pair, в которой свойство left – это один тип, например String, а свойство right – другой, например Int.

Вопрос

Прежде чем продолжить, угадайте, как исправить структуру, чтобы она принимала два отдельных типа?

Можно убедиться, что Pair принимает два отдельных (или одинаковых) типа, определив для нее два разных обобщения.

Листинг 7.29. Структура принимает два обобщения

```
struct Pair<T: Hashable, U: Hashable> { ❶
   let left: T
   let right: U ❷

   init(_ left: T, _ right: U) { ❸
      self.left = left
      self.right = right
   }
}
```

❶ Pair теперь принимает два универсальных типа, T и U. Оба ограничены Hashable.

❷ Свойство right – теперь тип U.

❸ Инициализатор также обновился, чтобы принять тип U.

Теперь Pair может принимать два разных типа, таких как String и Int, а также два похожих типа, как показано здесь.

Листинг 7.30. Pair принимает смешанные типы

```
// Pair принимает смешанные типы
let pair = Pair("Tom", 20)
// Такие же типы, как и две строки
let pair = Pair("Tom", "Jerry")
```

Благодаря введению нескольких универсальных типов Pair становится более гибкой, поскольку компилятор по отдельности специализирует типы T и U.

7.4.4. Соответствие протоколу Hashable

В настоящее время структура Pair пока не соответствует протоколу Hashable – сейчас мы это исправим.

Чтобы создать хеш-значение для Pair, у вас есть два варианта: вы можете позволить Swift синтезировать реализацию за вас или можете создать свою собственную. Сперва сделаем это простым способом, когда Swift синтезирует реализацию Hashable за вас, чтобы избавить вас от написания шаблонного кода.

Начиная с версии 4.1 Swift использует причудливый метод под названием *условное соответствие протоколу*. Оно позволяет автоматически согласовывать определенные типы с протоколом Hashable или Equatable. Если все свойства соответствуют этим протоколам, Swift синтезирует все необходимые методы. Например, если все свойства соответствуют протоколу Hashable, Pair также может автоматически соответствовать ему.

В этом случае все, что вам нужно сделать, – это настроить Pair на соответствие Hashable, и вам не нужно давать реализацию; это работает до тех пор, пока и left и right соответствуют протоколу Hashable, как показано в этом примере.

Листинг 7.31. Структура принимает два обобщения

```
struct Pair<T: Hashable, U: Hashable>: Hashable { ❶
   let left: T ❷
   let right: U ❷
   // ... Пропускаем остальную часть кода.
}
```

❶ Структура теперь также соответствует протоколу Hashable.

❷ Не нужно делать ничего особенного, потому что left и right тоже соответствуют протоколу.

Теперь Swift создает хеш-значение для Pair. Без особых усилий вы можете использовать Pair в качестве типа Hashable, например добавить его в набор (Set).

Листинг 7.32. Добавление Pair в Set

```
let pair = Pair<Int, Int>(10, 20)
print(pair.hashValue) // 5280472796840031924

let set: Set = [
   Pair("Laurel", "Hardy"),
   Pair("Harry", "Lloyd")
]
```

Обратите внимание, что вы можете явно указать типы внутри Pair, используя синтаксис Pair <Int, Int>.

Поскольку Pair соответствует протоколу Hashable, вы можете передать ему хешер, который Pair обновляет, используя значения, как показано ниже.

Листинг 7.33. Передаем хешер

```
let pair = Pair("Madonna", "Cher")

var hasher = Hasher() ❶
hasher.combine(pair) ❶
// Альтернативный способ: pair.hash(into: &hasher) ❷
let hash = hasher.finalize() ❸
print(hash) // 4922525492756211419
```

❶ Вы также можете создать хешер вручную и передать его Pair.

❷ Также можно передать хешер другим способом.

❸ Как только мы вызываем метод finalize(), то получаем хеш-значение.

Победителя среди хешеров нет. Одни из них быстрые, другие медлительны, но более безопасны, а третьи преуспели в криптографии. Поскольку мы можем передавать пользовательские хешеры, мы контролируем, как и когда хешировать типы. Затем тип, соответствующий протоколу Hashable, такой как Pair, контролирует, что нужно хешировать.

Ручная реализация протокола Hashable

Swift может синтезировать реализацию Hashable для структур и перечислений. Но синтез реализаций протоколов Equatable и Hashable не будет работать для классов. Кроме того, возможно, вам бы хотелось больше контроля над ними.

В этих случаях ручная реализация Hashable имеет больше смысла. Давайте посмотрим, как это можно сделать.

Можно объединить хеш-значения из двух свойств внутри Pair с помощью реализации метода func hash (into hasher: inout Hasher). В этом методе мы вызываем combine для каждого значения, которое вы хотите включить в операцию

хеширования, а также реализуем статический метод == из протокола Equatable, в котором вы сравниваете оба значения из двух пар.

Листинг 7.34. Реализация протокола Hashable вручную

```
struct Pair<T: Hashable, U: Hashable>: Hashable {
   // ... Пропускаем часть кода.

   func hash(into hasher: inout Hasher) { ❶
      hasher.combine(left) ❷
      hasher.combine(right) ❷
   }

   static func ==(lhs: Pair<T, U>, rhs: Pair<T, U>) -> Bool { ❸
      return lhs.left == rhs.left && lhs.right == rhs.right ❹
   }
}
```

❶ Чтобы соответствовать протоколу Hashable, структуре нужно реализовать метод func hash (into hasher: inout Hasher).

❷ Вызываем метод combine для хешера для каждого значения, которое мы хотим хешировать.

❸ Чтобы соответствовать протоколу Hashable, необходимо также соответствовать протоколу Equatable. Это можно сделать, реализовав метод static ==, в котором мы сравниваем два типа Pair.

❹ Сравниваем два типа Pair, сравнив их свойства left и right.

Для написания Pair был предпринят ряд шагов, но теперь у вас есть гибкий тип, который можно использовать в разных проектах. Существуют ли другие обобщенные структуры, которые могут облегчить вам жизнь? Возможно, синтаксический анализатор, который превращает словарь в конкретный тип или структуру, которая может записать любой тип в файл.

7.4.5. Упражнение

5

Напишите обобщенный кеш, который позволяет хранить значения с помощью ключей Hashable.

7.5. Обобщения и подтипы

В этом разделе рассматриваются подклассы вместе с обобщениями.

Звучит несколько теоретически, но это проливает немного света на хитрые ситуации, если вы хотите глубже изучить обобщения.

Создание подклассов – один из способов достижения полиморфизма. Обобщения – это еще один способ. Если вы начнете смешивать их, то вам следует знать

ряд правил, потому что обобщения становятся сложными в использовании, когда эти два механизма полиморфизма переплетаются. Swift скрывает за полиморфизмом множество сложностей, поэтому обычно не стоит беспокоиться по поводу теории. Но рано или поздно вы зайдете в тупик, когда будете использовать деление на подклассы в сочетании с обобщениями, и в этом случае немного теории может пригодиться. Чтобы понять деление на подклассы и обобщения, необходимо подробнее рассмотреть такие понятия, как *полиморфизм подтипов* и *вариантность*.

7.5.1. Подтипы и инвариантность

Представьте, что вы моделируете данные для образовательного сайта, где подписчик может начать посещать определенные курсы в режиме онлайн. Рассмотрим следующую структуру классов: класс `OnlineCourse` – это суперкласс для курсов, например `SwiftOnTheServer`, который наследует от `OnlineCourse`. Мы опускаем детали, чтобы сосредоточиться на обобщениях, как показано ниже.

Листинг 7.35. Два класса

```
class OnlineCourse {

   func start() {
      print("Starting online course.")
   }
}

class SwiftOnTheServer: OnlineCourse {
   override func start() {
      print("Starting Swift course.")
   }
}
```

Небольшое напоминание о том, как работает деление на подклассы: всякий раз при определении `OnlineCourse`, например для переменной, можно присвоить его `SwiftOnTheServer`, как показано в следующем листинге, поскольку оба они – типа `OnlineCourse`.

Листинг 7.36. Присваиваем подкласс суперклассу

```
var swiftCourse: SwiftOnTheServer = SwiftOnTheServer()
var course: OnlineCourse = swiftCourse // Разрешено
course.start() // "Starting Swift course".
```

Можно было бы заявить, что `SwiftOnTheServer` – это *подтип* `OnlineCourse`. Обычно подтипы относятся к подклассам. Но иногда это не так. Например, `Int` является подтипом опционального типа `Int?`, потому что всякий раз, когда ожидается `Int?`, вы можете передать обычный `Int`.

7.5.2. Инвариантность в Swift

Передача подкласса, когда ваш код ожидает, что это будет суперкласс, – отличный и модный способ. Но как только обобщенный тип оборачивает суперкласс, мы теряем возможности для создания подтипов. Например, мы вводим обобщенный тип Container, содержащий значение типа T. Затем мы пытаемся присвоить Container<SwiftOnTheServer> Container<OnlineCourse>, как и до этого, когда мы присвоили SwiftOnTheServer OnlineCourse. К сожалению, сделать это нельзя, как показано на рис. 7.4.

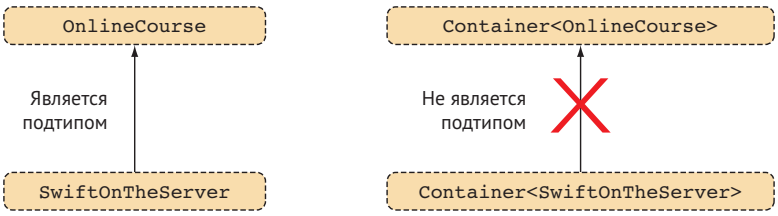


Рис. 7.4. Деление на подтипы не применимо к Container

Даже если SwiftOnTheServer является подтипом OnlineCourse, Container <SwiftOnTheServer> – это не подтип Container<OnlineCourse>, как показано в этом листинге.

Листинг 7.37. Структура Container

```
struct Container<T> {}

var containerSwiftCourse: Container<SwiftOnTheServer> =
➥ Container<SwiftOnTheServer>()
var containerOnlineCourse: Container<OnlineCourse> = containerSwiftCourse

error: cannot convert value of type 'Container<SwiftOnTheServer>' to
➥ specified type 'Container<OnlineCourse>'
```

Давайте рассмотрим этот недостаток в сценарии, немного ближе к реальной ситуации. Представьте себе универсальный тип cache, в котором хранятся данные. Вы хотите обновить кеш, содержащий онлайн-курсы, с помощью метода refreshCache.

Листинг 7.38. Структура Cache

```
struct Cache<T> {
   // Метод опущен.
}

func refreshCache(_ cache: Cache<OnlineCourse>) {
   // ... Пропускаем часть кода.
}
```

Но здесь снова видно, что вы можете передавать только типы Cache <OnlineCourse>, но не Cache<SwiftOnTheServer>.

Листинг 7.39. Инвариантность в действии

```
refreshCache(Cache<OnlineCourse>()) // Разрешено.
refreshCache(Cache<SwiftOnTheServer>()) // Ошибка: преобразование невозможно
➥ value of type 'Cache<SwiftOnTheServer>' to expected argument type
➥ 'Cache<OnlineCourse>'
```


Обобщения в Swift *инвариантны*, а это говорит о том, что даже если универсальный тип обертывает подкласс, он не делает его подтипом обобщения, обертывая его суперкласс. Почему? Поскольку Swift – относительно молодой язык, инвариантность – это безопасный способ справиться с полиморфизмом, пока язык не станет более понятным.

7.5.3. Универсальные типы Swift получают особые привилегии

Чтобы еще больше вас запутать, обобщенные типы Swift, такие как `Array` или `Optional`, допускают создание подтипов с обобщениями. Другими словами, типы Swift из стандартной библиотеки не имеют ограничения, которое вы только что видели. Ограничения имеют только те обобщения, которые вы определяете сами.

Для лучшего сравнения запишите опционалы как их истинный обобщенный аналог; например, `Optional<OnlineCourse>` вместо `OnlineCourse?`. После этого мы передадим `Optional<SwiftOnTheServer>` функции, принимающей `Optional<OnlineCourse>`. Помните, что в случае с обобщением `Container` это было неправильно, но теперь все нормально.

Листинг 7.40. Типы Swift являются ковариантными

```
func readOptionalCourse(_ value: Optional<OnlineCourse>) {
   // ... Пропускаем часть кода.
}

readOptionalCourse(OnlineCourse()) // Разрешено.
readOptionalCourse(SwiftOnTheServer()) // Разрешено, тип Optional является
   ковариантным.
```


Встроенные обобщенные типы Swift являются ковариантными, а это значит, что обобщенные типы могут быть подтипами других обобщенных типов. Ковариантность объясняет, почему можно передать `Int` в метод, ожидающий `Int?`.

Вы летите эконом-классом, в то время как типы Swift наслаждаются дополнительным пространством для ног в бизнес-классе. Надеемся, что скоро ваши обобщенные типы также смогут стать ковариантными.

На первый взгляд, ситуация, когда вам нужно смешать обобщения и подклассы, может быть неприятной. Честно говоря, можно многого добиться, и не используя подклассы. На самом деле если вы не используете конкретные фреймворки, которые зависят от подклассов, такие как UIKit, то можете разработать законченное приложение, вообще не прибегая к созданию подклассов. Если вы сделаете свои

классы финальными (`final`) по умолчанию, это также может помочь отказаться от подклассов и стимулировать протоколы и расширения добавить классам функциональности. В этой книге придается большое значение альтернативам подклассам, которые предлагает Swift.

7.6. В заключение

Прочитав эту главу, я надеюсь, что вы будете чувствовать себя уверенно, создавая обобщенные компоненты в своих проектах.

За абстракции нужно платить. Код становится немного сложнее, и его сложнее интерпретировать с помощью обобщений. Но взамен вы получаете много гибкости. Если немного попрактиковаться, может показаться, что это похоже на фильм *«Матрица»* (если вы знакомы с ним): когда вы смотрите на типы `T`, `U` и `V`, они превращаются в блондинок, брюнеток и рыжеволосых, или, возможно, в `String`, `Int` и `Float`.

Чем удобнее вам будет пользоваться обобщениями, тем легче вам будет уменьшить размер кодовой базы и написать больше повторно используемых компонентов. Я не могу выразить, насколько важно понимание обобщений на фундаментальном уровне, потому что они будут появляться в других главах и во многих типах Swift в реальных примерах.

Резюме

- Добавление неограниченного обобщения в функцию позволяет ей работать со всеми типами.
- Обобщения нельзя специализировать из области видимости функции или типа.
- Код обобщений преобразуется в специализированный код, который работает с несколькими типами.
- Обобщения могут быть ограничены для более специализированного поведения, что может исключать некоторые типы.
- Тип может быть ограничен несколькими обобщениями, чтобы разблокировать больше функциональных возможностей обобщенного типа.
- Swift может синтезировать реализации протоколов `Equatable` и `Hashable` для структур и перечислений.
- Синтез реализаций по умолчанию не работает с классами.
- Обобщения, которые вы пишете, инвариантны, следовательно, нельзя использовать их в качестве подтипов.
- Обобщенные типы в стандартной библиотеке ковариантны, и можно использовать их в качестве подтипов.

Ответы

1

Какие из этих функций компилируются? Подтвердите это, выполнив код.

Эта функция будет работать:

```
func wrap<T>(value: Int, secondValue: T) -> ([Int], T) {
   return ([value], secondValue)
}
```

И эта тоже:

```
func wrap<T>(value: Int, secondValue: T) -> ([Int], Int)? {
   if let secondValue = secondValue as? Int {
      return ([value], secondValue)
   } else {
      return nil
   }
}
```

2

В чем состоит преимущество использования обобщений по сравнению с типом Any (например, `func <T>(process:[T])` вместо `func(process:[Any])`)?

При использовании обобщения код становится полиморфным на этапе компиляции. В случае с Any нужно использовать нисходящее приведение типа во время выполнения.

3

Напишите функцию, которая при наличии массива возвращает словарь вхождений каждого элемента в массиве.

```
func occurrences<T: Hashable>(values: [T]) -> [T: Int] {
   var groupedValues = [T: Int]()
   for element in values {
      groupedValues[element, default: 0] += 1
   }
   return groupedValues
}

print(occurrences(values: ["A", "A", "B", "C", "A"])) // ["C": 1, "B": 1, "A": 3]
```

4

Создайте регистратор, который выводит описание универсального типа и описание отладки при его прохождении.

```
struct CustomType: CustomDebugStringConvertible, CustomStringConvertible {
```

```
    var description: String {
        return "This is my description"
    }
    var debugDescription: String {
        return "This is my debugDescription"
    }
}

struct Logger {
    func log<T>(type: T)
        where T: CustomStringConvertible & CustomDebugStringConvertible {
            print(type.debugDescription)
            print(type.description)
    }
}

let logger = Logger()
logger.log(type: CustomType())
```

5

Напишите обобщенный кеш, который позволяет хранить значения с помощью ключей Hashable.

```
class MiniCache<T: Hashable, U> {

    var cache = [T: U]()

    init() {}

    func insert(key: T, value: U) {
        cache[key] = value
    }

    func read(key: T) -> U? {
        return cache[key]
    }
}

let cache = MiniCache<Int, String>()
cache.insert(key: 100, value: "Jeff")
cache.insert(key: 200, value: "Miriam")
cache.read(key: 200) // Опционал ("Miriam")
cache.read(key: 99) // Опционал ("Miriam")
```

Глава 8. Становимся профессионалами в протокольно-ориентированном программировании

В этой главе:

- взаимосвязь и компромисс между обобщениями и использованием протоколов в качестве типов;
- ассоциированные типы;
- обход протоколов с ассоциированными типами;
- хранение и ограничение протоколов с ассоциированными типами;
- упрощение API с помощью наследования протоколов.

Протоколы дают вашему коду много мощи и гибкости. Некоторые могут сказать, что это своего рода флагман Swift, тем более что компания Apple рекламирует Swift как язык протокольно-ориентированного программирования. Но как бы хороши ни были протоколы, они быстро могут стать сложными. Здесь задействовано множество тонкостей, таких как использование протоколов во время выполнения или на этапе компиляции и ограничение протоколов с ассоциированными типами.

Цель этой главы – заложить прочную основу в отношении протоколов; она покажет, как использовать протоколы в качестве интерфейса по сравнению с их использованием для ограничения обобщений. Эта глава также направлена на то, чтобы рассказать, что можно считать продвинутыми протоколами, то есть протоколами с ассоциированными типами. Конечная цель – убедиться, что вы понимаете, почему, когда и как применять протоколы (и обобщения) в нескольких сценариях. Единственное требование – вы должны быть хотя бы немного знакомы с протоколами и должны были прочитать главу 7 «Обобщения». После этой главы мы еще не раз будем встречаться с протоколами и ассоциированными типами, поэтому я не рекомендую пропускать ее!

Сперва вы увидите, как протоколы живут сами по себе по сравнению с их использованием для ограничения обобщений. Вы посмотрите на обе стороны медали и выберете два подхода. Один подход использует обобщения, а другой нет. Цель этой главы – помочь вам найти компромисс и выбрать правильный подход, который будете использовать в повседневной работе.

Во втором разделе мы перейдем к более сложному аспекту протоколов, когда приступим к использованию ассоциированных типов. Протоколы с ассоциированными типами можно рассматривать как обобщенные протоколы, и вы поймете, зачем они нужны и когда можно применять их в своих приложениях.

Как только вы начинаете передавать протоколы с ассоциированными типами, это значит, что вы работаете с очень гибким кодом. Но будет непросто. Вы увидите, как передавать протоколы с ассоциированными типами и как создавать типы, которые хранят их с помощью ограничений. Кроме того, мы применим отличный трюк для очистки ваших API, используя метод под названием наследование протоколов.

8.1. Время выполнения в сравнении со временем компиляции

До сих пор эта книга широко освещала обобщения и их связь с протоколами. С помощью обобщений мы создаем полиморфные функции, определенные на этапе компиляции. Но протоколы не всегда нужно использовать с обобщениями, если вы хотите получить определенные преимущества на этапе выполнения (также известные как *динамическая диспетчеризация*). В этом разделе мы создадим протокол, а затем вы увидите, как найти компромисс между обобщениями, ограниченными протоколами, и как использовать протоколы в качестве типов, не прибегая к помощи обобщений.

8.1.1. Создание протокола

Присоединяйтесь!

Будет более познавательно и веселее, если вы будете знакомиться с кодом по мере прохождения этой главы. Исходный код можно скачать по адресу `http://mng.bz/qJZ2`.

Начнем с создания портфеля криптовалют, в котором могут быть монеты, такие как Bitcoin, Ethereum, Litecoin, Neo, Dogecoin и масса других.

Не нужно писать портфель или функцию для каждой валюты, особенно при наличии сотен других валют, которые могут поддерживаться вашим приложением. Ранее вы видели, как использовать перечисления для работы с полиморфизмом, что часто является подходящим подходом. Но перечисления в этом случае были бы слишком ограничительными. Вам пришлось бы объявлять кейс для каждой валюты (а их могут быть сотни!). Или же если мы используем фреймворк, а разработчики тогда не смогут добавить дополнительные кейсы для типов валют? В этих случаях можно использовать протоколы, которые являются более гибким способом достижения полиморфизма. Если вы вводите протокол, можно придерживаться этого протокола в случае с каждой валютой.

Чтобы продемонстрировать данный метод, мы введем протокол, как показано в следующем листинге, под названием `CryptoCurrency`, с четырьмя свойствами: `name`, `symbol`, `price` (для обозначения текущей цены в настройке валюты пользователя) и `holdings` (для обозначения количества криптовалют, которое есть у пользователя).

Листинг 8.1. Протокол CryptoCurrency

```
import Foundation

protocol CryptoCurrency {
   var name: String { get }
   var symbol: String { get }
   var holdings: Double { get set }
   var price: NSDecimalNumber? { get set }
}
```

Вы используете общности, когда определяете минимальные свойства, которые должны быть у каждой криптовалюты. В этом протоколы похожи на копии чертежей. Вы можете писать функции и свойства, которые работают только для протокола `CryptoCurrency`, и вам не нужно знать о валюте, которую ему передали.

В качестве следующего шага мы объявляем несколько криптовалют, которые соответствуют этому протоколу.

Листинг 8.2. Объявление констант для обозначения криптовалют

```
struct Bitcoin: CryptoCurrency {
   let name = "Bitcoin"
   let symbol = "BTC"
   var holdings: Double
   var price: NSDecimalNumber?
}

struct Ethereum: CryptoCurrency {
   let name = "Ethereum"
   let symbol = "ETH"
   var holdings: Double
   var price: NSDecimalNumber?
}
```

var или let

Каждый раз, когда вы объявляете свойство для протокола, оно всегда является переменной (`var`). Затем разработчик может выбрать, использовать ли ключевое слово `let`, чтобы объявить его как константу, или ключевое слово `var`. Кроме того, если у протокола есть модификатор `get set` для свойства, разработчик должен использовать переменную, допускающую изменение.

8.1.2. Обобщения в сравнении с протоколами

Давайте создадим портфель, в котором содержится несколько криптовалют для клиента. Вы уже видели, как можно использовать обобщения для работы с прото-

колом. Начнем с приведенного ниже списка, где используются обобщения, и вы быстро увидите проблему, которая его сопровождает.

Листинг 8.3. Представляем портфель (данный вариант пока еще не полностью работоспособен!)

```
final class Portfolio<Coin: CryptoCurrency> { ❶
   var coins: [Coin] ❷

   init(coins: [Coin]) {
      self.coins = coins
   }

   func addCoin(_ newCoin: Coin) {
      coins.append(newCoin)
   }

   // ... Пропускаем остальную часть кода и исключаем удаление криптовалют,
   // подсчет общего значения и другие функциональные возможности.
}
```

❶ Объявляем обобщение.

❷ Сохраняем криптовалюты, используя обобщенный тип Coin.

Предыдущий код представляет собой небольшой сегмент более крупного класса Portfolio, который может обладать большей функциональностью, включая удаление криптовалют, отслеживание прибылей и убытков и другие варианты использования.

Вопрос

У класса Portfolio есть проблема. Можете определить, какая?

8.1.3. Находим компромисс

Тут есть один изъян. Обобщение Coin обозначает один тип, поэтому в портфель можно добавить только один тип криптовалют.

Листинг 8.4. Попытка добавить разные криптовалюты в портфель

```
let coins = [
   Ethereum(holdings: 4, price: NSDecimalNumber(value: 500)), ❶
   // Если мы смешиваем валюты, то не можем передать их в портфель
   // Bitcoin(holdings: 4, price: NSDecimalNumber(value: 6000)) ❷
]

let portfolio = Portfolio(coins: coins) ❷
```

❶ Создаем криптовалюту Ethereum со значением 500. (Обычно это будет получено из реальных данных.)

❷ Портфель не может принять массив с разными криптовалютами.

В настоящее время в портфеле содержится криптовалюта Ethereum. Поскольку мы использовали обобщение внутри портфеля, оно теперь прикреплено к Ethereum. Из-за обобщений, если добавим другую криптовалюту, например Bitcoin, компилятор нас остановит, как показано в этом листинге.

Листинг 8.5. Нельзя смешивать протоколы с обобщениями

```
let btc = Bitcoin(holdings: 3, price: nil)
portfolio.addCoin(btc)

error: cannot convert value of type ‘Bitcoin’ to expected argument type
'Ethereum'
```

Компилятор выдает ошибку. На этапе компиляции инициализатор `Portfolio` преобразует обобщение в Ethereum, а это означает, что нельзя добавлять в портфель разные типы криптовалют. Можно убедиться в этом, проверив тип криптовалют, которые хранятся в портфеле, как показано в этом примере.

Листинг 8.6. Проверка типа криптовалют

```
print(type(of: portfolio)) // Portfolio<Ethereum>
print(type(of: portfolio.coins)) // Array<Ethereum>
```

type(of:)

Используя `type(of :)`, можно проверить тип.

Обобщения дают преимущества. Например, вы знаете типы, с которыми имеете дело на этапе компиляции. Компилятор также может применить оптимизацию производительности благодаря этой дополнительной информации.

Но в этом случае нам не нужно закреплять тип `Coin` на этапе компиляции. Вам нужно смешать и сопоставить криптовалюты и даже добавить новые во время выполнения. Чтобы выполнить это требование, мы перейдем от времени компиляции ко времени выполнения, поэтому пришло время отказаться от обобщений. Не волнуйтесь. Мы по-прежнему будем использовать протокол `CryptoCurrency`.

8.1.4. Переход ко времени выполнения

Удалив обобщение, мы переходим ко времени выполнения. В следующем примере мы реорганизуем портфель таким образом, чтобы в нем содержались криптовалюты, которые соответствуют протоколу `CryptoCurrency`.

Листинг 8.7. Динамический портфель

```
// До
final class Portfolio<Coin: CryptoCurrency> {
   var coins: [Coin]
   // ... Пропускаем остальной код
}
```

```
// После
final class Portfolio { ❶
   var coins: [CryptoCurrency] ❷
   // ... Пропускаем остальной код
}
```

❶ Удаляем определение обобщения.

❷ Массив криптовалют теперь имеет тип CryptoCurrency вместо обобщения Coin.

В портфеле нет обобщений. В нем содержатся только типы CryptoCurrency. Можно смешивать и сопоставлять типы внутри массива CryptoCurrency, как указано далее, добавив несколько типов криптовалют.

Листинг 8.8. Смешивание и совпадение криптовалют

```
// Не нужно указывать, что находится внутри портфеля.
let portfolio = Portfolio(coins: []) ❶

// Теперь криптовалюты можно смешивать.
let coins: [CryptoCurrency] = [
   Ethereum(holdings: 4, price: NSDecimalNumber(value: 500)),
   Bitcoin(holdings: 4, price: NSDecimalNumber(value: 6000))
]
portfolio.coins = coins ❷
```

❶ Не нужно указывать какие-либо параметры обобщений.

❷ Можно добавлять различные типы валют.

Отказавшись от обобщений, мы снова обрели гибкость.

8.1.5. Выбор между временем компиляции и временем выполнения

Это тонкое различие, однако эти примеры демонстрируют, что обобщения определяются в мире времени компиляции, в то время как протоколы в качестве типов живут в мире времени выполнения.

При использовании протокола во время выполнения можно рассматривать их как интерфейсы или типы. Как показано в следующем примере, при проверке типов внутри массива coins мы получим массив CryptoCurrency, тогда как ранее массив преобразовывался в один тип, а именно [Ethereum] и, возможно, другие.

Листинг 8.9. Проверка массива

```
print(type(of: portfolio)) // Портфель

let retrievedCoins = portfolio.coins
print(type(of: retrievedCoins)) // Array<CryptoCurrency>
```

Использование протокола во время выполнения означает, что можно смешивать и сопоставлять всевозможные типы, а это фантастическое преимущество. Но мы работаем с протоколом CryptoCurrency. Если у Bitcoin есть специальный метод bitcoinStores(), мы не сможем получить к нему доступ из портфеля, только если в протоколе также не определен метод. Это означает, что все криптовалюты теперь должны реализовать данный метод. Кроме того, во время выполнения можно проверить, имеет ли криптовалюта определенный тип, но это может рассматриваться как антишаблон и не масштабируется с сотнями возможных криптовалют.

8.1.6. Когда обобщение – лучший вариант

Рассмотрим другой сценарий, который демонстрирует еще одно различие между протоколами в качестве типов и использованием протоколов для ограничения обобщений. На этот раз ограничение обобщения с протоколом – лучший вариант.

Допустим, у нас есть функция с именем retrievePrice. Мы передаем этой функции криптовалюту, например структуру Bitcoin или Ethereum, и получаем ту же криптовалюту, но уже с указанием последней стоимости. Далее мы видим две похожие функции: одну с обобщенной реализацией для использования на этапе компиляции и другую с протоколом в качестве типа для использования во время выполнения.

Листинг 8.10. Обобщенный протокол в сравнении с протоколом времени выполнения

```
func retrievePriceRunTime(coin: CryptoCurrency, completion: ((CryptoCurrency)
➥ -> Void) ) { ❶
// ... Пропускаем часть кода. Сервер возвращает валюту с самой актуальной
ценой.
   var copy = coin
   copy.price = 6000
   completion(copy)
}

func retrievePriceCompileTime<Coin: CryptoCurrency>(coin: Coin,
➥ completion: ((Coin) -> Void)) { ❷
   // ... Пропускаем часть кода. Сервер возвращает валюту с самой
   // актуальной ценой.
   var copy = coin
   copy.price = 6000
   completion(copy)
}

let btc = Bitcoin(holdings: 3, price: nil)
retrievePriceRunTime(coin: btc) { (updatedCoin: CryptoCurrency) in ❸
   print("Updated value runtime is \(updatedCoin.price?.doubleValue ?? 0)")
}
```

```
retrievePriceCompileTime(coin: btc) { (updatedCoin: Bitcoin) in ❹
print("Updated value compile time is \(updatedCoin.price?.doubleValue
?? 0)")
}
```

❶ Функция retrievePriceRuntime использует протокол времени выполнения. Мы возвращаем протокол CryptoCurrency внутри замыкания.

❷ Функция retrievePriceCompileTime на этот раз использует обобщение, ограниченное протоколом CryptoCurrency. Обработчик завершения также использовал обобщение Coin.

❸ Обратите внимание, что обработчик завершения передал значение типа CryptoCurrency. Внутри замыкания мы не знаем, какой это конкретно тип до времени выполнения.

❹ Но мы знаем, чего ожидать от версии с обобщением. Функция передает тип Bitcoin. Это тот же тип, который мы передали в функцию.

Благодаря обобщениям мы точно знаем, с каким типом работаем внутри замыкания. При использовании протокола в качестве типа это преимущество утрачивается.

Появление обобщений внутри нашего приложения, возможно, выглядит несколько громоздко, но я надеюсь, что вы будете использовать их благодаря преимуществам, которые у них есть, по сравнению с протоколами времени выполнения.

Вообще говоря, использование протокола в качестве типа ускоряет процесс программирования и облегчает процесс смешивания и замены. Обобщения более ограничены и многословны, но они дают вам преимущества в производительности и знание типов на этапе компиляции, которые вы реализуете. Часто обобщения являются лучшим (но более сложным) вариантом, а в некоторых случаях это единственный выбор, к которому вы придете позже при реализации протоколов с ассоциированными типами.

8.1.7. Упражнения

1

Имеется протокол

```
protocol AudioProtocol {}
```

В чем разница между приведенными ниже утверждениями?

```
func loadAudio(audio: AudioProtocol) {}
func loadAudio<T: AudioProtocol>(audio: T) {}
```

2

Как бы вы использовали обобщенный или необобщенный протокол в этом примере:

```
protocol Ingredient {}
struct Recipe<I: Ingredient> {
   let ingredients: [I]
   let instructions: String
}
```

3

Как бы вы использовали обобщенный или необобщенный протокол в этой структуре:

```
protocol APIDelegate {
   func load(completion:(Data) -> Void)
}

struct ApiLoadHandler: APIDelegate {
   func load(completion: (Data) -> Void) {
      print("I am loading")
   }
}

class API {
   private let delegate: APIDelegate?
   init(delegate: APIDelegate) {
      self.delegate = delegate
   }
}

let dataModel = API(delegate: ApiLoadHandler())
```


8.2. Зачем нужны ассоциированные типы

Давайте углубимся в детали и поработаем с так называемыми продвинутыми протоколами, также известными как протоколы с ассоциированными типами.

Протоколы являются абстрактными, но при использовании протоколов с ассоциированными типами вы делаете их обобщенными, что делает ваш код еще более абстрактным и экспоненциально сложным.

Владение протоколами с ассоциированными типами – полезный навык. Одно дело – слышать о них, кивать и думать, что однажды они могут вам пригодиться. Но я хочу, чтобы вы знали, как их использовать, и чувствовали себя комфортно при работе с ними, чтобы в конце этого раздела они не пугали вас и вы чувствовали, что готовы начать их реализацию (когда это имеет смысл).

В этом разделе мы приступим к моделированию протокола и столкнемся с недостатками, которые в конечном итоге решим с помощью ассоциированных типов. Попутно вы получите опыт рассмотрения и принятия решения о том, для чего нужно использовать протоколы с ассоциированными типами.

8.2.1. Недостатки протоколов

Представьте, что вам нужно создать протокол, напоминающий часть работы, которую он должен выполнить. Это может быть задание или задача, например отправка электронных писем всем клиентам, миграция базы данных, изменение размера изображений в фоновом режиме или обновление статистики путем сбора информации. Можно смоделировать этот фрагмент кода через протокол, который мы назовем `Worker`.

Первый тип, который мы создадим и который соответствует протоколу `Worker`, – это `MailJob`. `MailJob` требует адреса электронной почты, используя строковый тип для ввода, и возвращает логическое значение в качестве выходных данных, указывая, успешно ли завершилось задание. Мы начнем с того, что покажем ввод и вывод `MailJob` в протоколе `Worker`. У этого протокола единственный метод, start, который принимает `String` в качестве ввода и `Bool` в качестве вывода, как показано в приведенном ниже листинге.

Листинг 8.11. Протокол Worker

```
protocol Worker {
   @discardableResult ❶
   func start(input: String) -> Bool ❷
}

class MailJob: Worker { ❸
   func start(input: String) -> Bool {
      // Отправка почты на электронный адрес (input может обозначать
      адрес электронной почты)
      // По окончании возвращаем значение, прошло все успешно или нет
      return true
   }
}
```

❶ Игнорируем вывод метода `start`. Можно подавить предупреждения компилятора с помощью ключевого слова `@discardableResult`.

❷ У протокола `Worker` есть один-единственный метод.

❸ Класс `MailJob` соответствует протоколу `Worker`.

Примечание

Обычно можно запускать реализацию этого протокола в фоновом режиме, но для простоты мы оставляем ее в том же потоке.

Однако наш протокол не покрывает всех требований. `Worker` соответствует потребностям `MailJob`, но не масштабируется на другие типы, у которых может и не быть `String` и `Bool` в качестве входных и выходных данных.

Давайте познакомимся с еще одним типом, который соответствует протоколу `Worker` и специализируется на удалении файлов. Назовем его `FileRemover`.

FileRemover принимает тип URL в качестве входных данных для своего каталога, удаляет файлы и возвращает массив строк, представляющих удаленные файлы. Поскольку FileRemover принимает тип URL и возвращает [String], он не может соответствовать протоколу Worker (см. рис. 8.1).

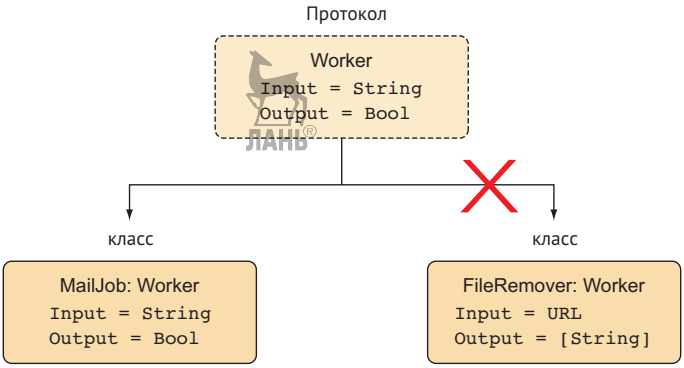


Рис. 8.1. Попытка соответствовать протоколу Worker

К сожалению, протокол плохо *масштабируется*. Попробуйте два разных подхода, прежде чем перейти к решению, включающему использование протоколов с ассоциированными типами.

8.2.2. Попытка превратить все в протокол

Прежде чем решить проблему с ассоциированными типами, рассмотрим решение, которое включает в себя другие протоколы. Как насчет того, чтобы сделать ввод (Input) и вывод (Output) протоколами? Похоже, предложение правильное – это позволит нам полностью избегать использования протоколов с ассоциированными типами. Но у этого подхода есть проблемы, которые мы сейчас увидим.

Листинг 8.12. Протокол Worker без ассоциированных типов

```
protocol Input {} ❶
protocol Output {} ❶

protocol Worker {
   @discardableResult
   func start(input: Input) -> Output ❷
}
```

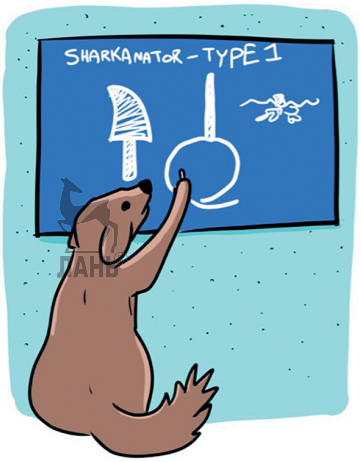
❶ Объявляем два протокола: один для Input и другой для Output.

❷ Метод start теперь принимает Input и Output в качестве протоколов вместо ассоциированных типов.

Данный подход работает. Но для *каждого* типа, который вы хотите использовать для ввода и вывода, нужно сделать так, чтобы эти типы соответствовали протоколам Input и Output. Такой подход – верный способ в конечном итоге получить шаблонный код, например заставить String, URL и [URL] соответствовать протоколу Input и снова протоколу Output. Еще один недостаток заключается в том, что

вы вводите новый протокол для каждого параметра и возвращаемого типа. Кроме того, если бы вы вводили новый метод для `Input` или `Output`, вам пришлось бы реализовать его для всех типов, придерживающихся этих протоколов. Если вы будете превращать все в протокол, то это подойдет для проекта небольшого размера, но будет плохо масштабироваться, приведет к созданию шаблонного кода и ляжет бременем на разработчиков.

8.2.3. Разработка обобщенного протокола



Давайте рассмотрим еще один подход, где нам нужно, чтобы и `MailJob`, и `FileRemover` соответствовали протоколу `Worker`. В приведенном ниже листинге мы сначала наивно подойдем к решению (которое не будет компилироваться), но оно будет подчеркивать мотивацию в использовании ассоциированных типов. После этого мы воспользуемся ими, чтобы найти решение.

Протокол `Worker` хочет убедиться, что каждая реализация может решить для себя, что обозначают ввод и вывод. Можно попытаться сделать это, определив в протоколе два обобщения, `Input` и `Output`. К сожалению, наш подход пока не работает – давайте посмотрим, почему.

Листинг 8.13. Протокол Worker (пока еще не будет компилироваться!)

```
protocol Worker<Input, Output> { ❶
   @discardableResult
   func start(input: Input) -> Output ❷
}
```

❶ Нельзя объявлять обобщения в протоколе.

❷ Протокол запускает задание с вводом и возвращает вывод.

Swift быстро остановит нас:

```
error: protocols do not allow generic parameters; use associated types
instead
```

Swift не поддерживает протоколы с обобщенными параметрами. При поддержке обобщенного протокола нужно будет определить конкретный тип протокола для реализации. Например, предположим, что вы моделируете класс `MailJob` с обобщением `Input` типа `String` и обобщением `Output` типа `Bool`. Тогда его реализация будет выглядеть так: `class MailJob: Worker<String, Bool>`. Поскольку протокол `Worker` будет обобщенным, теоретически можно реализовать его несколько раз. Однако несколько реализаций одного и того же протокола не компилируется, как показано в листинге 8.14.

Листинг 8.14. MailJob реализует протокол Worker несколько раз

```
// Не поддерживается: Реализация обобщенного протокола Worker.
class MailJob: Worker<String, Bool> { ❶
   // Реализация опущена.
}

class MailJob: Worker<Int, [String]> { ❶
   // Реализация опущена.
}
// и т. д.
```

❶ Теоретически можно реализовать протокол `Worker` для разных типов, таких как `<String, Bool>` или `<Int, [String]>`.

На момент написания этих строк Swift не поддерживает несколько реализаций одного и того же протокола, однако можно реализовать протокол только один раз для каждого типа. Другими словами, класс `MailJob` может реализовать протокол `Worker` один раз. Ассоциированные типы обеспечивают равновесие, гарантируя, что вы сможете реализовать протокол один раз при работе с обобщенными значениями. Посмотрим, как это работает.

8.2.4. Моделирование протокола с ассоциированными типами

Мы видели два альтернативных варианта, которые не решают нашу проблему. Давайте создадим жизнеспособное решение. Мы будем следовать советам компилятора и использовать ассоциированные типы. Можно заново написать протокол `Worker` и использовать ключевое слово `associatedtype`, с помощью которого мы объявляем обобщения `Input` и `Output` как ассоциированные типы.

Листинг 8.15. Протокол Worker с ассоциированными типами

```
protocol Worker { ❶
   associatedtype Input ❷
   associatedtype Output ❷
   @discardableResult
   func start(input: Input) -> Output ❸
}
```

❶ Протокол Worker не объявляет обобщения.

❷ Объявляем ассоциированные типы с помощью ключевого слова associatedtype.

❸ В оставшейся части сигнатуры протокола можно ссылаться на ассоциированные типы.

Теперь обобщения Input и Output объявлены как ассоциированные типы. Ассоциированные типы напоминают обобщения, но они определены *внутри* протокола. Обратите внимание, что у протокола Worker отсутствует нотация <Input, Output>. Теперь можно начать говорить о соответствии протоколу относительно типов MailJob и FileRemover.

8.2.5. Реализация протокола с ассоциированными типами

Протокол Worker готов к реализации. Благодаря ассоциированным типам классы Mailjob и FileRemover могут успешно соответствовать протоколу. MailJob устанавливает для Input и Output значения String и Bool, в то время как FileRemover устанавливает для них значения URL и [String] (см. рис. 8.2).

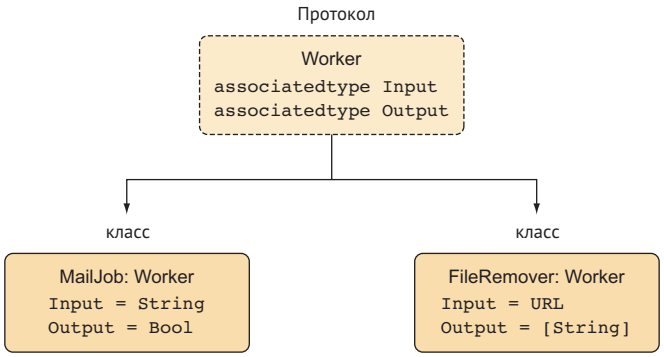


Рис. 8.2. Протокол Worker реализован

Присмотревшись к классу MailJob в приведенном ниже листинге, можно увидеть, что он устанавливает для Input и Output конкретные типы.

Листинг 8.16. Класс Mailjob (реализация опущена)

```
class MailJob: Worker {
   typealias Input = String ❶
   typealias Output = Bool ❷
   func start(input: String) -> Bool {
      // Отправляем письмо на адрес электронной почты (input может
         представлять собой адрес электронной почты)
      // По окончании возвращаем значение, чтобы убедиться, что все
         прошло успешно
      return true
   }
}
```

❶ Ассоциированный тип Input определяется как String.

❷ Ассоциированный тип Output определяется как Bool.

Теперь класс MailJob все время использует String и Bool для Input и Output.

Примечание

Каждый тип, соответствующий протоколу, может иметь только одну реализацию протокола. Но вы все равно получаете обобщенные значения с помощью ассоциированных типов. Преимущество заключается в том, что каждый тип может решать, что представляют собой эти ассоциированные значения.

Реализация класса FileRemover отличается от реализации MailJob, и его ассоциированные типы также другие. Обратите внимание, как показано ниже, что можно опустить нотацию typealias, если Swift может вывести ассоциированные типы.

Листинг 8.17. Класс FileRemover

```
class FileRemover: Worker {
   // Псевдоним типа Input = URL ❶
   // Псевдоним типа Output = [String] ❶

   func start(input: URL) -> [String] {
      do {
         var results = [String]()
         let fileManager = FileManager.default
         let fileURLs = try fileManager.contentsOfDirectory(at: input,
         ➥ includingPropertiesForKeys: nil) ❷
         for fileURL in fileURLs {
            try fileManager.removeItem(at: fileURL) ❷
            results.append(fileURL.absoluteString)
         }
         return results
      } catch {
         print("Clearing directory failed.")
         return []
      }
   }
}
```

❶ Можно опустить типы, если компилятор может вывести типы из сигнатур метода и свойства.

❷ FileRemover находит каталог и перебирает файлы, чтобы удалить их.

При использовании протоколов с ассоциированными типами несколько типов может соответствовать одному и тому же протоколу; тем не менее каждый тип может определить, что представляет собой ассоциированный тип.

Подсказка

Еще один способ – это рассматривать ассоциированный тип как обобщение, за исключением того, что он является обобщением, которое обитает внутри протокола.

8.2.6. Протоколы с ассоциированными типами в стандартной библиотеке

Swift использует ассоциированные типы в стандартной библиотеке, и мы уже использовали несколько таких типов!

Наиболее распространенные случаи использования протокола с ассоциированными типами – протоколы `IteratorProtocol`, `Sequence` и `Collection`. Этим протоколам соответствуют `Array`, `String`, `Set`, `Dictionary` и др., которые используют ассоциированный тип Element, представляющий элемент внутри коллекции. Но вы также видели другие протоколы, например `RawRepresentable` для перечислений, в котором ассоциированный тип `RawValue` позволяет преобразовывать любой тип в перечисление, и наоборот.

Ключевое слово Self

Еще одна особенность ассоциированного типа – ключевое слово `Self`. Типичный пример – это протокол `Equatable`, который вы видели в главе 7. В случае с этим протоколом сравниваются два одинаковых типа, представленных `Self`. Как показано в этом листинге, `Self` преобразуется в тип, соответствующий протоколу `Equatable`.

Листинг 8.18. Протокол Equatable

```
public protocol Equatable {
   static func == (lhs: Self, rhs: Self) -> Bool ❶
}
```

❶ У протокола `Equatable` есть `Self`.

8.2.7. Другие случаи использования ассоциированных типов

Сосредоточить свое внимание на протоколе с ассоциированным типом может быть сложно. Введение ассоциированных типов начинает обретать смысл, когда те, кто соответствуют протоколу, используют разные типы в своей реализации. Как правило, такие протоколы обычно чаще встречаются во фреймворках из-за более высокой вероятности повторного использования.

Вот несколько вариантов использования ассоциированных типов:

- протокол *Recording* (запись) – у каждой записи есть продолжительность, а также возможность поддержки скраббинга с помощью метода `seek()`, но фактические данные могут быть разными для каждой реализации, например аудиофайл, видеофайл или потоковое видео в YouTube;

- протокол *Service* – загружает данные. Один тип может возвращать данные в формате JSON из API, а другой – локально искать и возвращать необработанные строковые данные;
- протокол *Message* – отслеживает посты в социальных сетях. В одной реализации это твит; в другой – личное сообщение в Facebook, а в третьем варианте это может быть сообщение в WhatsApp;
- протокол *SearchQuery* – напоминает запросы к базе данных, где результат для каждой реализации разный;
- протокол *Paginator* – ему можно дать страницу и смещение для просмотра базы данных. Каждая страница может представлять собой какие-либо данные. У него может быть несколько пользователей в таблице пользователей в базе данных либо список файлов или продуктов в представлении.

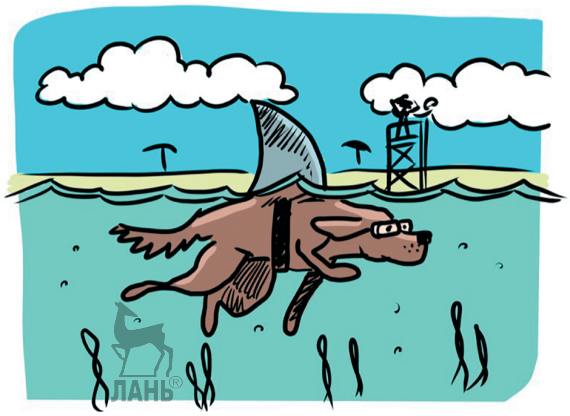
8.2.8. Упражнение

4

Рассмотрим приведенную ниже иерархию подклассов для игры, где у вас есть враги, которые могут атаковать вас, причинив определенный урон. Можно ли заменить эту иерархию решением на основе протокола?

```
class AbstractDamage {}

class AbstractEnemy {
   func attack() -> AbstractDamage {
      fatalError("This method must be implemented by subclass")
   }
}

class Fire: AbstractDamage {}

class Imp: AbstractEnemy {
   override func attack() -> Fire {
      return Fire()
   }
}

class BluntDamage: AbstractDamage {}

class Centaur: AbstractEnemy {
   override func attack() -> BluntDamage {
      return BluntDamage()
   }
}
```

8.3. Передача протокола с ассоциированными типами



Давайте рассмотрим способы передачи протокола с ассоциированными типами. Мы будем использовать протокол `Worker` из предыдущего раздела с двумя ассоциированными типами, `Input` и `Output`.

Представьте, что вам нужно написать обобщенную функцию или метод, который принимает одного работника и массив элементов, которые этот работник должен обработать. Передавая массив типа `[W.Input]`, где `W` – это `Worker` (Работник), вы убеждены, что ассоциированный тип Input – именно тот самый тип, который `Worker` может обрабатывать (см. рис. 8.5). Протоколы с ассоциированными типами могут быть реализованы только в виде обобщенных ограничений, за исключением некоторых сложных исключений, поэтому мы будем использовать обобщения, чтобы оставаться в мире кода на этапе компиляции.

Примечание

Можно спокойно опускать любые ссылки на `W.Output` в функции `runWorker`, потому что вы с ней ничего не делаете.

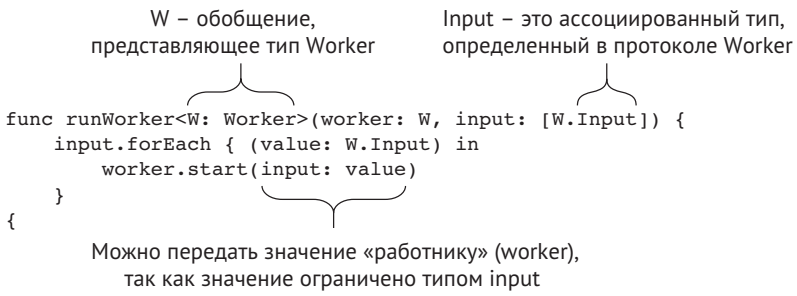


Рис. 8.3

При наличии функции `runWorker` можно передать ее нескольким типам `Worker`, например `MailJob` или `FileRemover`, как показано в приведенном ниже листинге.

Убедитесь, что вы передаете соответствующие типы Input для каждого работника: строки для MailJob и URL для FileRemover.

Листинг 8.19. Процесс передачи

```
let mailJob = MailJob()
runWorker(worker: mailJob, input: [«grover@sesamestreetcom», «bigbird@
sesamestreet.com»]) ❶

let fileRemover = FileRemover()
runWorker(worker: fileRemover, input: [ ❷
   URL(fileURLWithPath: "./cache", isDirectory: true),
   URL(fileURLWithPath: "./tmp", isDirectory: true),
])
```

❶ Передаем MailJob в runWorker со списком адресов электронной почты.

❷ Передаем экземпляр FileRemover в runWorker со списком URL-адресов.

Примечание

Как и обобщения, ассоциированные типы также преобразуются на этапе компиляции.

8.3.1. Использование оператора where с ассоциированными типами

Можно ограничить ассоциированные типы в функциях с помощью оператора *where*, что становится полезным, если вам нужно несколько ограничить функциональность. Ограничение ассоциированных типов очень напоминает ограничение обобщения, но синтаксис немного отличается.

Например, допустим, вам нужно обработать массив пользователей; возможно, вам нужно убрать пробелы из их имен или обновить другие значения. Можно передать массив пользователей одному работнику и убедиться, что ассоциированный тип Input относится к типу User, с помощью оператора where, чтобы иметь возможность выводить имена пользователей, которые обрабатывает работник. При ограничении ассоциированного типа функция работает только с пользователями в качестве входных данных.

Листинг 8.20. Ограничение ассоциированного типа Input

```
final class User { ❶
   let firstName: String
   let lastName: String
   init(firstName: String, lastName: String) {
      self.firstName = firstName
      self.lastName = lastName
   }
```

```
}

func runWorker<W>(worker: W, input: [W.Input])
where W: Worker, W.Input == User { ❷
   input.forEach { (user: W.Input) in
      worker.start(input: user)
      print("Finished processing user \(user.firstName) \(user.lastName)") ❸
   }
}
```

❶ Определяем класс User.

❷ Внутри функции runWorker мы ограничиваем ассоциированный тип Input для класса User. Обратите внимание, что в данном случае также можно ограничить W.

❸ Теперь можно ссылаться на пользователей по их свойствам внутри тела функции.

8.3.2. Типы, ограничивающие ассоциированные типы

Вы только что видели, как ассоциированные типы передаются через функции. Теперь сосредоточимся на том, как эти типы работают со структурами, классами или перечислениями.

Например, у нас есть класс `ImageProcessor`, который может хранить тип `Worker` (см. рис. 8.6). Рабочими в этом контексте могут быть типы, которые обрезают изображение, изменяют его размер или тонируют его в коричневый цвет. Что именно делает класс `ImageProcessor`, зависит от `Worker`. Дополнительным преимуществом `ImageProcessor` является то, что он может пакетно обрабатывать большое количество изображений, извлекая их из хранилища, такого как база данных.

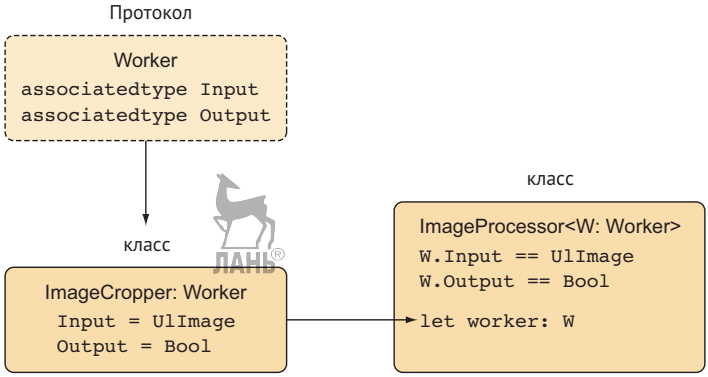


Рис. 8.4. Класс ImageProcessor

`ImageProcessor` принимает `ImageCropper` типа `Worker`.

Листинг 8.21. Вызов ImageProcessor

```
let cropper = ImageCropper(size: CGSize(width: 200, height: 200))
```

```
let imageProcessor: ImageProcessor<ImageCropper> = ImageProcessor(worker:
➥ cropper) ❶
```

❶ Здесь мы явно определяем обобщенный тип внутри ImageProcessor через ImageProcessor <ImageCropper>.

Сначала мы познакомимся с Worker, в данном случае это ImageCropper. Реализация опущена, чтобы сосредоточиться на соответствии протоколу.

Листинг 8.22. ImageCropper

```
final class ImageCropper: Worker {
   let size: CGSize
   init(size: CGSize) {
      self.size = size
   }

   func start(input: UIImage) -> Bool {
      // Опущено: Изменяем размер изображения до self.size
      // возвращаем булево значение, чтобы показать, что процесс прошел
      // успешно
      return true
   }
}
```

Здесь мы создадим тип ImageProcessor, который принимает обобщение Worker. Но у этого обобщения есть два ограничения: первое ограничение устанавливает для Input тип UIImage, и ожидается, что Output будет логическим типом, который отражает, было задание Worker выполнено успешно или нет.

Можно ограничить ассоциированные типы Worker с помощью оператора *where*, поставив его перед открывающими скобками, как показано здесь:

Листинг 8.23. Тип ImageProcessor

```
final class ImageProcessor<W: Worker> ❶
where W.Input == UIImage, W.Output == Bool { ❷

   let worker: W

   init(worker: W) {
      self.worker = worker
   }

   private func process() { ❸
      // start batches
      var results = [Bool]()
      let amount = 50
      var offset = 0
      var images = fetchImages(amount: amount, offset: offset)
```

```
        var failedCount = 0
        while !images.isEmpty { ❹
            for image in images {
                if !worker.start(input: image) { ❺
                    failedCount += 1
                }
            }
            offset += amount
            images = fetchImages(amount: amount, offset: offset)
        }
        print(«\(failedCount) images failed»)
    }

    private func fetchImages(amount: Int, offset: Int) -> [UIImage] {
        // Не отображено: Возвращаем изображения из базы данных
        // или жесткого диска
        return [UIImage(), UIImage()] ❻
    }
}
```

❶ ImageProcessor определяет обобщение `W` типа `Worker`.

❷ Ограничиваем обобщение `W` определенными типами `Input` и `Output`.

❸ Метод `process` запускает `ImageProcessor`.

❹ Перебираем изображения в хранилище и обрабатываем их, пока они не закончатся.

❺ Для каждого изображения вызывается метод `start`, чтобы выполнить его обработку. В случае неудачи мы увеличиваем `failCount` на единицу.

❻ В этом примере мы имитируем возврат изображений, но в реальном сценарии это будет настоящее хранилище данных.

Принимая обобщение `Worker`, класс `ImageProcessor` может принимать различные типы, например обрезчики изображений, типы, изменяющие размеры, или типы, которые делают изображение черно-белым.

8.3.3. Очистка API и наследование протокола

В зависимости от того, насколько обобщенным оказывается это приложение, можно в конечном итоге передать обобщение `Worker`. Хотя повторное объявление одних и тех же ограничений, например where `W.Input == UIImage`, `W.Output == Bool`, может утомить.

Для удобства можно использовать наследование протокола для дальнейшего его ограничения. Наследование протоколов означает, что вы создаете новый протокол, который наследует определение другого протокола. Рассматривайте это как деление протокола на подклассы.

Можно создать протокол ImageWorker, который наследует все свойства и функции из протокола Worker, но с одним большим отличием: протокол ImageWorker ограничивает ассоциированные типы Input и Output с помощью оператора *where*, как показано здесь:

Листинг 8.24. ImageWorker

```
protocol ImageWorker: Worker where Input == UIImage, Output == Bool {
   //Если хотите, тут могут идти дополнительные методы.
}
```


Расширение протокола

В данном случае протокол ImageWorker пустой, но можно добавить к нему дополнительные определения, если хотите. Затем типы, придерживающиеся ImageWorker, должны реализовать их поверх протокола Worker.

В этом протоколе подразумевается использование оператора *where*, а передача ImageWorker означает, что типы больше не нужно вручную ограничивать типами Image и Bool. Протокол ImageWorker может сделать API ImageProcessor более понятным.

Листинг 8.25. Ограничения больше не нужны

```
// Прежде:
final class ImageProcessor<W: Worker>
where W.Input == UIImage, W.Output == Bool { ... }

// После:
final class ImageProcessor<W: ImageWorker> { ... }
```

8.3.4. Упражнения

5

Имеются следующие типы:

```
// Протокол, представляющий нечто, может воспроизводить файл в каком-то месте.
protocol Playable {
   var contents: URL { get }
   func play()
}

// Структура, которая наследует этот протокол.
final class Movie: Playable {
   let contents: URL

   init(contents: URL) {
      self.contents = contents
```

```
    }

    func play() { print("Playing video at \(contents)") }
}
```

Вы вводите новый тип song, но вместо воспроизведения файла по URL-адресу он использует тип AudioFile. Что можно предпринять? Посмотрите, можно ли сделать так, чтобы протокол отражал это изменение:

```
struct AudioFile {}

final class Song: Playable {
    let contents: AudioFile

    init(contents: AudioFile) {
        self.contents = contents
    }

    func play() { print("Playing song") }
}
```

6

Посмотрите на приведенный ниже плей-лист. Вначале он мог воспроизводить только фильмы. Как убедиться, что он может воспроизводить фильмы или песни?

```
final class Playlist {

    private var queue: [Movie] = []

    func addToQueue(playable: Movie) {
        queue.append(playable)
    }

    func start() {
        queue.first?.play()
    }
}
```

8.4. В заключение

Протоколы с ассоциированными типами и обобщениями открывают доступ к абстрактному коду, но заставляют вас рассматривать типы на этапе компиляции. Хотя в процессе работы иногда могут возникать сложности, компиляция многократно используемого абстрактного кода может быть полезной. Однако не всегда нужно усложнять ситуацию. Иногда одного обобщения или конкретного кода достаточно, чтобы получить то, что вам нужно. Теперь, когда вы увидели, как работают ассоциированные типы, вы готовы использовать их, когда они появятся снова в последующих главах.

Резюме

- Можно использовать протоколы в качестве обобщенных ограничений, но так же, как типы во время выполнения (динамическая диспетчеризация), когда вы отказываетесь от обобщений.
- Использование протоколов в качестве общего ограничения – обычно правильный путь, пока вам не понадобится динамическая диспетчеризация.
- Ассоциированные типы – это обобщения, которые привязаны к протоколу.
- Протоколы с ассоциированными типами позволяют конкретному типу определять ассоциированный тип. Каждый конкретный тип может специализировать ассоциированный тип в другой.
- Протоколы с ключевым словом `Self` – это уникальная разновидность ассоциированных типов, ссылающихся на текущий тип.
- Протоколы с ассоциированными типами или ключевое слово Self заставляют вас рассматривать типы на этапе компиляции.
- Можно заставить протокол наследовать другой протокол для дальнейшего ограничения его ассоциированных типов.

Ответы

1

В чем разница между приведенными ниже утверждениями?

- Необобщенная функция использует динамическую диспетчеризацию (время выполнения).
- Обобщенная функция преобразуется на этапе компиляции.

2

Как бы вы использовали обобщенный или необобщенный протокол в этом примере?

Рецепт требует несколько разных ингредиентов. Используя обобщение, можно использовать только один тип ингредиента. Повсеместное использование яиц может наскучить, поэтому в этом случае следует отказаться от обобщений.

3

Как бы вы использовали обобщенный или необобщенный протокол в этой структуре?

Делегат – единственный тип. В данном случае можно безопасно использовать обобщение. Вы получаете преимущества на этапе компиляции, такие как повышенная производительность и возможность видеть, какой тип вы будете использовать. Код может выглядеть так:

```
protocol APIDelegate {
   func load(completion:(Data) -> Void)
}
```

```
struct ApiLoadHandler: APIDelegate {
   func load(completion: (Data) -> Void) {
      print("I am loading")
   }
}

class API<Delegate: APIDelegate> {
   private let delegate: Delegate?
   init(delegate: Delegate) {
      self.delegate = delegate
   }
}

let dataModel = API(delegate: ApiLoadHandler())
```

4

Рассмотрим приведенную ниже иерархию подклассов для игры, где у вас есть враги, которые могут атаковать вас, причинив определенный урон. Можно ли заменить эту иерархию решением на основе протокола?

```
protocol Enemy {
   associatedtype DamageType
   func attack() -> DamageType
}

struct Fire {}

class Imp: Enemy {
   func attack() -> Fire {
      return Fire()
   }
}

struct BluntDamage {}

class Centaur: Enemy {
   func attack() -> BluntDamage {
      return BluntDamage()
   }
}
```

5

Вы вводите новый тип song, но вместо воспроизведения файла по URL-адресу он использует тип AudioFile. Что можно предпринять? Посмотрите, можно ли сделать так, чтобы протокол отражал это изменение.

Введите ассоциированный тип, например Media. Свойство contents теперь имеет тип Media, который преобразуется во что-то другое для каждой реализации:

```
protocol Playable {
   associatedtype Media
   var contents: Media { get }
   func play()
}

final class Movie: Playable {
   let contents: URL
   init(contents: URL) {
      self.contents = contents
   }
   func play() { print("Playing video at \(contents)") }
}

struct AudioFile {}
final class Song: Playable {
   let contents: AudioFile
   init(contents: AudioFile) {
      self.contents = contents
   }
   func play() { print(«Playing song») }
}
```


6

Посмотрите на приведенный ниже плей-лист. Вначале он мог воспроизводить только фильмы. Как убедиться, что он может воспроизводить фильмы или песни?

```
final class Playlist<P: Playable> {

   private var queue: [P] = []

   func addToQueue(playable: P) {
      queue.append(playable)
   }

   func start() {
      queue.first?.play()
   }
}
```

Примечание

Нельзя смешивать фильмы и песни, но можно создать список воспроизведения песен (или список воспроизведения фильмов).

Глава 9. Итераторы, последовательности и коллекции

В этой главе:

- более пристальный взгляд на итерацию в Swift;
- как Sequence связан с `IteratorProtocol`;
- изучаем полезные методы, которые предоставляет `Sequence`;
- различные протоколы коллекций;
- создание структур данных с помощью протоколов `Sequence` и `Collection`.

При программировании на языке Swift мы постоянно используем итераторы, последовательности и коллекции. Всякий раз при использовании `Array`, `String`, `stride`, `Dictionary` и других типов вы работаете с чем-то, что можно итерировать. Итераторы позволяют использовать циклы *for*, а также большое количество методов, включая `filter`, `map`, `sorted` и `reduce`.

В этой главе вы увидите, как работают эти итераторы, узнаете о полезных методах (таких как `reduce` и `lazy`), как создавать типы, соответствующие протоколу `Sequence`. Мы также рассмотрим протокол `Collection` и его многочисленные подпротоколы, такие как `MutableCollection`, `RandomAccessCollection` и др. Вы узнаете, как реализовать протокол `Collection`, чтобы получить множество бесплатных методов для своих типов. Познакомившись с `Sequence` и `Collection`, вы сможете глубже понять, как работает итерация и как создавать собственные типы, работающие на итераторах.

Мы начнем с самого низа. Вы узнаете, как работают циклы и каковы их синтаксические возможности, которые используются в методах `IteratorProtocol` и `Sequence`.

Затем мы поближе познакомимся с `Sequence` и узнаем, как он создает итераторы и зачем это нужно.

После этого мы изучим несколько полезных методов в `Sequence`, которые помогут расширить наш словарь итераторов. Вы познакомитесь с методами `lazy`, `reduce`, `zip` и др.

Чтобы убедиться, что вы освоили `Sequence`, мы создадим пользовательский тип, соответствующий этому протоколу. Эта последовательность будет представлять собой структуру данных под названием `Bag` или `MultiSet`, которая похожа на Set, но предназначена для нескольких значений.

Затем мы перейдем к протоколу `Collection` и посмотрим, чем он отличается от `Sequence`. Вы увидите различные типы протоколов `Collection`, которые предлагает Swift, и узнаете об их уникальных особенностях.

Напоследок мы интегрируем `Collection` в пользовательскую структуру данных. Не нужно быть мастером алгоритма, чтобы воспользоваться преимуществами этого протокола. Вместо этого мы рассмотрим практический подход, используя обычную структуру данных.

Я позаботился о том, чтобы вы не заснули во время чтения этой главы. Помимо теории, в ней содержится множество примеров практического использования.

9.1. Итерация

При программировании на языке Swift вы все время перебираете (итерируете) данные, например извлекаете элементы внутри массива, получаете отдельные символы внутри строки, обрабатываете целые числа внутри диапазона. Давайте начнем с теории, чтобы вы представляли себе внутреннее устройство итерации в Swift. После этого вы будете готовы применить полученные знания на практике.

Присоединяйтесь!

Будет более познавательно и веселее, если вы будете знакомиться с кодом по мере прохождения этой главы. Исходный код можно скачать по адресу http://mng.bz/7JPy.

9.1.1. Циклы и метод makeIterator

Каждый раз, когда вы используете цикл *for*, вы используете итератор. Например, можно регулярно перебирать элементы массива с помощью *for in*.

Листинг 9.1. Использование for in

```
let cheeses = [«Gouda», «Camembert», «Brie»]

for cheese in cheeses {
   print(cheese)
}

// Вывод:
"Gouda"
"Camembert"
"Brie"
```

Но *for in* – это синтаксический сахар. На самом деле внутри происходит следующее: итератор создается с помощью метода `makeIterator()`. Swift проходит по элементам с помощью цикла *while*, как показано здесь.

Листинг 9.2. Использование метода makeIterator ()

```
var cheeseIterator = cheeses.makeIterator() ❶
while let cheese = cheeseIterator.next() { ❷
   print(cheese)
```

```
}

// Вывод:
"Gouda"
"Camembert"
"Brie"
```

❶ Итератор создан; обратите внимание, что он изменяемый и требует переменную.

❷ За кулисами Swift непрерывно вызывает `next()` для итератора, пока тот не устанет и не завершит цикл.

Хотя цикл *for* вызывает `makeIterator()`, можно передать итератор непосредственно в цикл *for*:

```
var cheeseIterator = cheeses.makeIterator()
for element in cheeseIterator {
   print(cheese)
}
```

Метод `makeIterator()` определен в протоколе `Sequence`, который тесно связан с `IteratorProtocol`. Прежде чем перейти к `Sequence`, давайте вначале подробнее рассмотрим `IteratorProtocol`.

9.1.2. IteratorProtocol

Итератор реализует `IteratorProtocol`. Это небольшой, но мощный компонент Swift. В `IteratorProtocol` есть ассоциированный тип `Element` и метод `next()`, который возвращает опционал `Element` (см. рис. 9.1 и приведенный ниже листинг). Итераторы генерируют значения, таким образом вы можете перебирать несколько элементов.

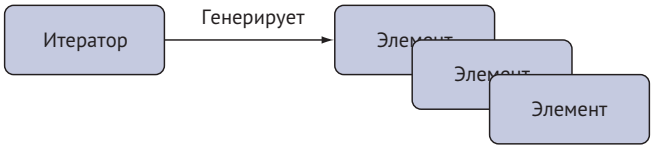


Рис. 9.1. IteratorProtocol создает элементы

Листинг 9.3. IteratorProtocol в Swift

```
public protocol IteratorProtocol {
   // Тип элемента, пройденный итератором.
   associatedtype Element ❶
   mutating func next() -> Element?
}
```

❶ Определяем ассоциированный тип `Element`, представляющий элемент, который является результатом работы итератора.

Примечание

Если вы когда-либо видели расширения Array, это тот же Element, который использует данное расширение.

Каждый раз, когда мы вызываем next для итератора, мы получаем следующее значение, которое производит итератор, до тех пор, пока он не устанет и вы не получите значение nil.

Итератор похож на пакет с продуктами – можно вытаскивать из него элементы один за другим. Когда сумка пуста, на этом все. Соглашение состоит в том, что после того, как итератор опустошается, он возвращает nil, и любой последующий вызов next() также должен возвращать nil, как показано в этом примере.

Листинг 9.4. Прохождение через итератор

```
let groceries = ["Flour", "Eggs", "Sugar"]
var groceriesIterator: IndexingIterator<[String]> = groceries.makeIterator()
print(groceriesIterator.next()) // Опционал ("Flour")
print(groceriesIterator.next()) // Опционал ("Eggs")
print(groceriesIterator.next()) // Опционал ("Sugar")
print(groceriesIterator.next()) // nil
print(groceriesIterator.next()) // nil
```

Примечание

Массив возвращает IndexingIterator, но это может быть и что-то другое в зависимости от типа.

9.1.3. Протокол Sequence

Тесно связанный с IteratorProtocol, протокол Sequence реализуется в Swift повсеместно. На самом деле мы постоянно его используем. Sequence – это основа любого типа, который можно итерировать. Sequence также является суперпротоколом Collection, который наследуют Array, Set, String, Dictionary и другие типы, а это означает, что эти типы также соответствуют протоколу Sequence.

9.1.4. Посмотрим поближе

Sequence может создавать итераторы. В случае с протоколом IteratorProtocol, после того как элементы внутри итератора заканчиваются, итератор опустошается. Но для Sequence это не проблема, потому что он может создать новый итератор для нового цикла. Таким образом, типы, соответствующие этому протоколу, могут итерироваться многократно (см. рис. 9.2).

Обратите внимание, что в листинге 9.5 у протокола Sequence есть ассоциированный тип Iterator, который ограничен IteratorProtocol, а также метод makeIterator, который создает итератор.

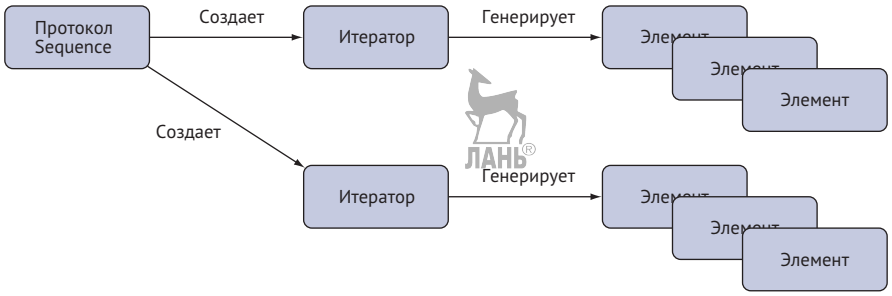


Рис. 9.2. Протокол Sequence создает итераторы

Sequence – это большой протокол с большим количеством методов по умолчанию, но данная глава охватывает только важнейшие элементы.

Листинг 9.5. Протокол Sequence (сокращенная версия)

```
public protocol Sequence {
   associatedtype Element ❶
   associatedtype Iterator: IteratorProtocol where Iterator.Element ==
   ➥ Element ❷ ❸
   func makeIterator() -> Iterator ❹
   func filter( ❺
      _isIncluded: (Element) throws -> Bool
   ) rethrows -> [Element]
   func forEach(_body: (Element) throws -> Void) rethrows ❺
   // ... Пропускаем остальную часть кода
}
```

❶ Определяем ассоциированный тип `Element`, представляющий элемент, который создает итератор.

❷ Sequence гарантирует, что ассоциированный тип `Element` совпадает с типом из `IteratorProtocol`.

❸ Ассоциированный тип определяется и ограничивается `IteratorProtocol`.

❹ Sequence может продолжить создавать итераторы.

❺ В Sequence определено множество методов, например `filter` и `forEach`.

Это не SequenceProtocol?

Sequence не является SequenceProtocol. Тем не менее Sequence ограничивает Iterator протоколом IteratorProtocol, что, возможно, объясняет, почему IteratorProtocol дали такое имя. Странно, но пусть будет так.

Чтобы реализовать Sequence, нужно всего лишь реализовать метод make Iterator(). Способность создавать итераторы – секретный способ того, как можно многократно итерировать Sequence, например перебирать элементы массива несколько раз. Sequence может напоминать фабрику по производству итерато-

ров, но не позволяйте этому фрагменту кода обмануть себя. Sequence – довольно мощный протокол, потому что он предлагает множество методов по умолчанию, таких как filter, map, reduce, flatMap, forEach, dropFirst, contains, регулярное зацикливание с for in и многое другое. Наличие типа, соответствующего протоколу Sequence, означает, что он получает множество функциональных возможностей бесплатно. Sequence не зарезервирован для типов Swift. Вы также можете создавать пользовательские типы, которые соответствуют Sequence.

Давайте рассмотрим несколько важных методов протокола Sequence.

9.2. Сила Sequence

Типы, которые реализуют Sequence, получают много полезных методов бесплатно. Мы не будем пересказывать их все, потому что вы, вероятно, знакомы с некоторыми из них (и мне бы не хотелось, чтобы количество страниц в этой книге перевалило за 1000). Давайте воспользуемся этой возможностью, чтобы пролить свет на некоторые полезные или хитрые методы для создания нашего словаря итераторов.

9.2.1. Метод filter

Swift не стесняется заимствовать идеи из концепций *функционального программирования*, такие как, например, методы map, filter или reduce. filter – распространенный метод, который предлагает Sequence.

Например, с его помощью фильтруют данные в зависимости от того, какое замыкание вы передаете. filter передает каждый элемент в функцию замыкания и ожидает логический тип в ответ.

Метод filter возвращает новую коллекцию, за исключением того, что он сохраняет элементы, из которых вы возвращаете значение true, в переданном замыкании.

Чтобы проиллюстрировать это, обратите внимание, как этот метод используется для фильтрации значений массива. Он возвращает массив со всеми значениями больше 1.

Листинг 9.6. filter является функцией высшего порядка

```
let result = [1,2,3].filter { (value) -> Bool in
   return value > 1
}
print(result) // [2, 3]
```

9.2.2. Метод forEach

Ежедневное употребление в пищу макаронных изделий может наскучить – то же самое происходит, если изо дня в день использовать цикл *for*. Если вас одолела скука, можете использовать forEach; это хорошая альтернатива обычному циклу

for, чтобы показать, что вам нужен так называемый побочный эффект. Побочный эффект возникает, когда изменяется какое-либо внешнее состояние, например сохранение элемента в базе данных, вывод, рендеринг представления, как указано `forEach`, не возвращая значения.

Например, у нас есть массив строк, где для каждого элемента вызывается функция `deleteFile`.

Листинг 9.7. Использование forEach

```
["file_one.txt", "file_two.txt"].forEach { path in
deleteFile(path: path)
}
func deleteFile(path: String) {
   // Удаляем файл ....
}
```

С `forEach` у вас есть прекрасный способ для вызова функции. Фактически, если функция принимает только аргумент и ничего не возвращает, можно вместо этого напрямую передать ее `forEach`. Обратите внимание, что в этом листинге фигурные скобки `{}` заменены круглыми `()`, потому что вы передаете функцию напрямую.

Листинг 9.8. Использование forEach путем передачи функции

```
["file_one.txt", "file_two.txt"].forEach(deleteFile)
```

9.23. Метод enumerated

Если вам нужно отслеживать количество зацикливаний, можно использовать для этого метод `enumerated`, как показано в следующем листинге. Он возвращает специальную последовательность под названием `EnumeratedSequence`, которая ведет учет итераций. Итерация начинается с нуля.

Листинг 9.9. Метод enumerated

```
[«First line», «Second line», «Third line»]
   .enumerated() ❶
```

```
    .forEach { (index: Int, element: String) in
       print("\(index+1): \(element)") ❷
    }

// Вывод:
1: First line
2: Second line
3: Third line
```

❶ Метод enumerated() возвращает EnumeratedSequence, благодаря чему создается элемент со смещением (index).

❷ Используем index, чтобы поставить перед текстом номер строки. Мы добавляем к индексу цифру 1, чтобы список начинался с 1, потому что индекс начинается с 0.

Обратите внимание, что forEach хорошо подходит, когда мы объединяем методы.

9.2.4. Ленивая итерация

Всякий раз, когда вы вызываете такие методы, как forEach, filter или другие, элементы *интенсивно* итерируются. Интенсивная итерация означает, что доступ к элементам внутри Sequence можно получить сразу после итерации. В большинстве случаев это то, что нужно. Но в некоторых ситуациях данный подход не является идеальным. Например, если вы располагаете чрезвычайно большими (или даже бесконечными) ресурсами, вам вряд ли захочется проходить всю последовательность, а только некоторые элементы за раз.

Для такого случая Swift предлагает определенную последовательность под названием LazySequence, которую можно получить с помощью ключевого слова lazy (ленивый).

Например, у нас есть диапазон чисел от 0 до огромного числа Int.max. Нам нужно отфильтровать этот диапазон, чтобы сохранить все четные числа, а затем получить последние три числа. Мы будем использовать ключевое слово lazy для предотвращения полной итерации гигантской коллекции. Благодаря этому слову данная итерация обходится довольно дешево, потому что ленивый итератор вычисляет только несколько выбранных элементов.

Обратите внимание, что в этом листинге LazySequence ничего, по сути, не делает, пока вы не начнете читать элементы.

Листинг 9.10. Использование ключевого слова lazy

```
let bigRange = 0..<Int.max ❶

let filtered = bigRange.lazy.filter { (int) -> Bool in
   return int % 2 == 0
} ❷
```

```
let lastThree = filtered.suffix(3) ❸

for value in lastThree {
   print(value)
}

// Вывод:
9223372036854775802
9223372036854775804
9223372036854775806
```


❶ Определяем диапазон.

❷ Определяем отфильтрованные элементы; обратите внимание, что ни один элемент еще не был оценен.

❸ Получаем последние три элемента; обратите внимание, что (опять же) ни один элемент не был оценен.

Использование `lazy` может стать отличным инструментом для больших коллекций, когда вас не интересуют все значения. Один из недостатков состоит в том, что замыкания с `lazy` – это ключевое слово `@escaping`, и хранят замыкания временно. Еще одним недостатком является то, что каждый раз, когда вы получаете доступ к элементам, результаты пересчитываются на лету. При интенсивной итерации, когда мы получаем результаты, на этом все.

9.2.5. Метод reduce

Метод reduce, возможно, один из самых хитрых методов `Sequence`. Он основывается на функциональном программировании, и его можно встретить в других языках. Это может быть `fold` в Kotlin или Rust либо `inject` в Ruby. Чтобы еще больше все усложнить, Swift предлагает два варианта метода reduce, поэтому вам нужно будет решить, какой из них выбрать для каждого случая. Давайте рассмотрим оба варианта.

С помощью этого метода мы перебираем каждый элемент в `Sequence`, например `Array`, и кумулятивно создаем объект. Когда метод завершит работу, он вернет завершенный объект. Другими словами, с помощью reduce мы превращаем несколько элементов в новый объект. Чаще всего мы делаем это с массивом, но метод reduce можно вызывать и для других типов.

reduce в действии

Здесь мы используем метод `reduce`, чтобы увидеть его в действии. Тип `String` соответствует протоколу `Sequence`, поэтому можно вызывать для него наш метод и перебрать его символы. Например, представим, что нам нужно узнать количество разрывов строк в строке. Мы сделаем это, перебирая каждый символ и увеличивая количество, если символ равен "\ n".

Для начала нужно дать `reduce` начальное значение. Затем при каждой итерации мы будем это значение обновлять. В приведенном ниже сценарии нам нужно по-

считать количество разрывов строк, поэтому мы передаем целое число со значением 0, обозначающее счет. Тогда наш окончательный результат – это количество измеренных разрывов строк, сохраненных в numberOfLineBreaks.

Листинг 9.11. Подготовка

```
let text = "It's hard to come up with fresh exercises.\nOver and over again.\
   nAnd again."
let startValue = 0
let numberOfLineBreaks = text.reduce(startValue) {
   // ... Пропускаем часть кода
   // ... Проделываем здесь ряд действий
}

print(numberOfLineBreaks) // 2
```

При каждой итерации reduce передает два значения с помощью замыкания: одно значение – это элемент из итерации, а другое – целое число (startValue), которое вы передали reduce.

Затем для каждой итерации мы проверяем наличие символа новой строки, «\ n», и увеличиваем целое число на единицу, при необходимости возвращая новое целое число (см. рис. 9.3).

```
let text = "This is some text.\nAnother line.\nYet, another line again."
let startValue = 0

let numberOfLineBreaks = text.reduce(startValue) { (accumulation: Int, char: Character) in
    if char == "\n" {
        return accumulation + 1
    } else {
        return accumulation
    }
}
```

Значение, которое мы приводим здесь... ...возвращается в следующей итерации

Рис. 9.3. Как метод reduce передает значения

Но reduce делает нечто хитрое: при каждой итерации мы получаем следующий символ и новое целое число, которое только что обновили в предыдущей итерации.

Внутри замыкания мы возвращаем значение acculumation, а в следующей итерации reduce отдает его обратно, что позволяет нам и дальше обновлять acculumation. Таким образом, мы перебираем все элементы, одновременно обновляя одно значение. После того как reduce завершит итерацию, он вернет итоговое значение, которое мы создали, – в этом случае numberOfLineBreaks.

9.2.6. Метод reduce into

Представьте, что вы используете метод reduce для определенного количества оценок учеников и словарь, который подсчитывает количество оценок, преобра-

зованных в A, B, C или D. Вы будете обновлять этот словарь при каждой итерации. Проблема, однако, заключается в том, что вам придется создавать изменяемую копию этого словаря для каждой итерации. Копирование происходит всякий раз, когда вы ссылаетесь на словарь, потому что `Dictionary` – это *тип значения*. Давайте посмотрим в приведенном ниже листинге, как это работает с `reduce`, прежде чем улучшить код с помощью `reduce(into:)`.

Листинг 9.12. Метод reduce с меньшей производительностью

```
let grades = [3.2, 4.2, 2.6, 4.1]
let results = grades.reduce([:]) { (results: [Character: Int],
➥ grade: Double) in ❶
   var copy = results ❷
   switch grade {
      case 1..<2: copy["D", default: 0] += 1 ❸
      case 2..<3: copy["C", default: 0] += 1 ❸
      case 3..<4: copy["B", default: 0] += 1 ❸
      case 4...: copy["A", default: 0] += 1 ❸
      default: break }
   return copy ❹
}

print(results) // ["C": 1, "B": 1, "A": 2] ❺
```

❶ Мы используем метод reduce с начальным значением [:], которое представляет собой пустой словарь. Swift может определить, какой это тип, избавляя нас от необходимости писать код [`Character: Int`] ().

❷ Создаем изменяемую копию с помощью ключевого слова `var`, чтобы иметь возможность изменять словарь.

❸ Увеличиваем счетчик, выполняя сопоставление для оценок.

❹ Возвращаем копию для каждой итерации.

❺ В конце мы получаем словарь с оценками.

Поскольку мы копируем словарь для каждой итерации с помощью `var copy = results`, то получаем снижение производительности. Тут и вступает в действие `reduce(into:)`. С помощью этого варианта метода `reduce` можно и дальше изменять тот же самый словарь, как показано здесь.

Листинг 9.13. reduce (into:)

```
let results = grades.reduce(into: [:]) { (results: inout [Character: Int],
   grade: Double) in ❶
   switch grade {
      case 1..<2: results[«D», default: 0] += 1
      case 2..<3: results[«C», default: 0] += 1
      case 3..<4: results[«B», default: 0] += 1
```

```
        case 4...: results[«A», default: 0] += 1
        default: break
    }
}
```

❶ Используем `into:`.

Примечание

При использовании метода `reduce(into:)` мы ничего не возвращаем из замыкания.

Когда вы работаете с более крупными типами значений, такими как словари или массивы, рассмотрите возможность применения `reduce(into:)` для повышения производительности.

Использование `reduce` может показаться несколько непривычным, потому что в качестве альтернативы можно было бы применять цикл *for*. Но этот метод может стать очень кратким способом кумулятивного создания значения. Также можно утверждать, что `reduce` демонстрирует намерение. В случае с циклом *for* нужно запустить компилятор, чтобы увидеть, фильтруете вы или преобразуете данные, или делаете что-то еще. Если вы видите метод `reduce`, то можно быстро сделать вывод, что несколько значений превращается в нечто новое.

Reduce достаточно абстрактен, чтобы с ним можно было делать множество разных вещей. Но этот метод выглядит наиболее элегантно, когда преобразует список элементов в простое целое число или сложный класс либо структуру. Вне зависимости от этого `reduce` – ваш друг.

9.2.7. Метод zip

Чтобы закрыть этот раздел, давайте взглянем на метод `zip`, который позволяет сжать (соединить) два итератора. Если один итератор опустошается, `zip` прекращает итерацию.

В следующем листинге обратите внимание на то, что мы выполняем итерацию от a до c и от 1 до 10. Поскольку массив со строками является самым коротким, zip завершается после трех итераций.

Листинг 9.14. Метод zip

```
for (integer, string) in zip(0..<10, [«a», «b», «c»]) {
   print(«\(integer): \(string)»)
}

// Вывод:
// 0: a
// 1: b
// 2: c
```

Примечание

`zip` является автономной функцией и не вызывается для типа.

Существует много других методов, включая `map`, `flatMap` и `compactMap`, которым отведена отдельная глава (глава 10). Давайте продолжим и посмотрим, как создать тип, соответствующий `Sequence`, который получает все эти методы бесплатно.

9.2.8. Упражнения

1

В чем разница между reduce и reduce(into:)?

2

Какой из них вы бы выбрали?

9.3. Создание обобщенной структуры данных с помощью Sequence

Ранее мы прошли теоретический урок, в котором было показано внутреннее устройство `Sequence` и `IteratorProtocol`. Теперь мы попробуем создать что-нибудь, используя полученные знания.

В этом разделе мы создадим структуру данных под названием `Bag` (сумка), также известную как мультимножество. Мы будем использовать `Sequence` и `IteratorProtocol`, чтобы получать имеющиеся в свободном доступе методы для `Bag`, такие как `contains` или `filter`.

Наша структура данных будет похожа на `Set`: он хранит элементы в неупорядоченном виде и имеет возможность быстрой вставки и поиска. Но, в отличие от `Set` `Bag`, может хранить один и тот же элемент несколько раз.

В фреймворке Foundation существует класс `NSCountingSet`, который обладает похожими свойствами, но не является нативным типом Swift с Swift-подобным интерфейсом. То, что мы создаем здесь, обладает меньшими функциональными возможностями, но абсолютно в духе Swift. Кроме того, это хорошее упражнение.

9.3.1. Bag в действии

Для начала давайте посмотрим, как работает `Bag` (см. рис. 9.4). Можно вставлять в него объекты точно так же, как в `Set`, но с одной большой разницей: можно сохранять один и тот же объект несколько раз. Однако у `Bag` есть одна хитрость, потому что он отслеживает количество раз, которое объект сохраняется, и физически не сохраняет объект по нескольку раз. При одноразовом сохранении элемента потребление памяти снижается.

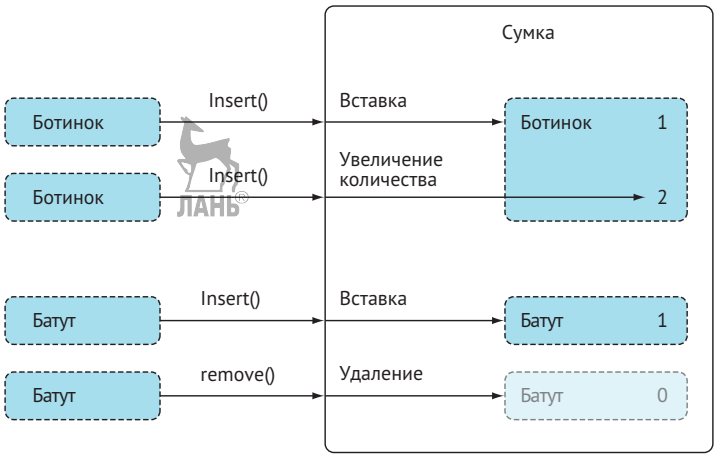


Рис. 9.4. Как данные хранятся в «сумке»

Приведенный ниже листинг показывает Bag в действии. Интерфейс почти такой же, как и у Set, но обратите внимание, что можно добавлять одну и ту же строку несколько раз. Также обратите внимание, что Bag является обобщением, как и Set: Bag хранит строки, целые числа или все, что соответствует протоколу Hashable.

Листинг 9.15. Использование bag

```
var bag = Bag<String>()
bag.insert("Huey") ❶
bag.insert("Huey") ❶
bag.insert("Huey") ❶
bag.insert("Mickey") ❶
bag.remove("Huey") ❷
bag.count // 3

print(bag)
// Вывод:
// Huey встречается 2 раза
// Mickey встречается 1 раз

let anotherBag: Bag = [1.0, 2.0, 2.0, 3.0, 3.0, 3.0] ❸
print(anotherBag)

// Вывод:
// 2.0 встречается 2 раза
// 1.0 встречается 1 раз
// 3.0 встречается 3 раза
```

❶ В Bag можно вставлять элементы.

❷ А также можно удалять их.

❸ Можно создать сумку так же, как и набор (Set). Вы увидите, как это сделать с помощью протокола ExpressibleByArrayLiteral.

Примечание

В сумке можно хранить все, что соответствует протоколу Hashable. Несмотря на то что в ней можно хранить что угодно, она не может просто смешивать и сопоставлять типы – такие как строки и целые числа – так же, как в случае с Array или Set.

Прежде чем реализовывать какие-либо протоколы, давайте настроим базовую структуру данных, чтобы было проще работать.

Как и Set, в Bag хранятся типы Hashable, так что мы можем быстро искать, помещая элемент в словарь. Для этого мы определяем обобщение Element, ограниченное Hashable, представляющее элементы, которые хранятся в Bag. Bag хранит каждый элемент в свойстве store и увеличивает счетчик элемента, если мы добавляем тот же элемент снова. И наоборот, уменьшает его, если мы удалим элемент. Когда счетчик достигает нуля, Bag удаляет элемент полностью, тем самым освобождая память, как показано в этом листинге.

Листинг 9.16. Внутри Bag

```
struct Bag<Element: Hashable> { ❶

   private var store = [Element: Int]() ❷

   mutating func insert(_ element: Element) {
      store[element, default: 0] += 1 ❸
   }

   mutating func remove(_ element: Element) {
      store[element]? -= 1 ❹
      if store[element] == 0 { ❺
         store[element] = nil ❺
      }
   }

   var count: Int {
      return store.values.reduce(0, +) ❻
   }
}
```

❶ Bag хранит элементы Hashable.

❷ Данные хранятся в словаре с элементами и их значениями.

❸ Когда мы вставляем элемент, то увеличиваем количество на 1. Если в хранилище элемента еще нет, он добавляется со значением по умолчанию 0, которое мы также незамедлительно увеличиваем.

❹ При удалении элементов вы уменьшаете их число.

❺ Если количество элементов равно 0, вы удаляете элемент из сумки.

❻ Получаем общее количество с помощью метода `reduce()`

Теперь, разобравшись с базовыми функциями, давайте приступим к реализации полезных протоколов. Чтобы нам было проще заглянуть внутрь сумки, реализуем `CustomStringConvertible`. Всякий раз во время вывода свойство `description` предоставляет пользовательскую строку элементов и их вхождения, как показано в этом листинге.

```
extension Bag: CustomStringConvertible {
   var description: String {
      var summary = String()
         for (key, value) in store {
            let times = value == 1 ? "time" : "times"
            summary.append(«\(key) occurs \(value) \(times)\n»)
         }
      return summary
   }
}
```

Вот и вывод:

```
let anotherBag: Bag = [1.0, 2.0, 2.0, 3.0, 3.0, 3.0]
print(anotherBag)
// Вывод:
// 2.0 встречается 2 раза
// 1.0 встречается 1 раз
// 3.0 встречается 3 раза
```

9.3.2. Создаем BagIterator

Мы уже можем использовать `Bag` как есть. Но у нас нет возможности итерировать его до тех пор, пока мы не реализуем `Sequence`. Чтобы это сделать, нам нужен итератор, который мы назовем `BagIterator`.

Внутри `Bag` свойство store (*хранилище*) содержит данные. `Bag` передает это свойство `IteratorProtocol`, чтобы он мог выдавать значения одно за другим. Отправка копии в `IteratorProtocol` означает, что он может изменить свою копию store, не затрагивая `Bag`.

Поскольку `store` является типом значения, оно копируется в `IteratorProtocol`, когда `Bag` передает хранилище `IteratorProtocol`.

Нужно применить небольшую хитрость, потому что `Bag` лжет. Он не содержит то количество элементов, которые, как утверждает, у него есть; в нем есть всего лишь элемент со счетчиком. А вот и наша хитрость: `BagIterator` возвращает один и тот же элемент несколько раз в зависимости от количества элементов. Таким

образом, постороннему не нужно знать о приемах, которые использует Bag, а количество получаемых элементов при этом будет правильным.

Каждый раз, когда вы вызываете next в BagIterator, итератор возвращает элемент и уменьшает его количество. Как только счет достигнет нуля, BagIterator удалит элемент. Если элементов больше нет, итератор опустошается и возвращает nil, давая сигнал, что BagIterator завершил итерацию. В приведенном ниже листинге есть пример этого.

Листинг 9.18. Создаем BagIterator

```
struct BagIterator<Element: Hashable>: IteratorProtocol { ❶

   var store = [Element: Int]() ❷

   mutating func next() -> Element? { ❸
      guard let (key, value) = store.first else { ❹
         return nil ❹
      }
      if value > 1 { ❺
         store[key]? -= 1 ❺
      } else {
         store[key] = nil ❻
      }
      return key ❼
   }
}
```

❶ BagIterator соответствует IteratorProtocol, и у него также есть обобщение Element, как Bag, для своих элементов.

❷ В BagIterator есть своя копия хранилища из Bag.

❸ Метод next из IteratorProtocol вызывается для предоставления элементов.

❹ Если в хранилище ничего нет, итератор опустошается и возвращает nil.

❺ Если у значения осталось несколько подсчетов, мы уменьшаем подсчет.

❻ Если у значения больше не осталось подсчетов, удаляем его из словаря.

❼ В конце мы возвращаем элемент.

Почти все. После этого можно расширить Bag и привести его в соответствие с Sequence, как показано в следующем листинге. Все, что нам нужно сделать, – это реализовать makeIterator и вернуть только что созданный итератор BagIterator, который получает свежую копию store.

Листинг 9.19. Реализация Sequence

```
extension Bag: Sequence { ❶
   func makeIterator() -> BagIterator<Element> { ❷
```

```
        return BagIterator(store: store) ❸
    }
}
```

❶ Bag теперь соответствует Sequence.

❷ Нам нужно только реализовать метод makeIterator.

❸ Мы создаем и возвращаем новый BagIterator, но не раньше, чем дадим ему копию своего хранилища.

Вот и все. Теперь у нас есть возможности Sequence, и Bag обладает множеством бесплатных функций. Используя экземпляр Bag, мы можем вызывать filter, lazy, reduce, contains и множество других методов.

Листинг 9.20. Сила Sequence

```
bag.filter { $0.count > 2}
bag.lazy.filter { $0.count > 2}
bag.contains("Huey") // Истинно.
bag.contains("Mickey") // Ложно.
```

У нас все готово. Тем не менее мы можем реализовать как минимум еще две оптимизации, связанные с AnyIterator и ExpressibleByArrayLiteral. Давайте рассмотрим их.

9.3.3. Реализация AnyIterator

В случае с Bag нам не пришлось особо трудиться при расширении Sequence. Все, что мы сделали, – это создали экземпляр BagIterator и вернули его. В подобном случае можно вернуть итератор AnyIterator, чтобы полностью исключить создание BagIterator.

AnyIterator – это итератор *стирания типов*, который можно рассматривать как универсальный итератор. При инициализации он принимает замыкание, которое вызывается всякий раз при вызове next. Другими словами, можно поместить функциональность next из BagIterator в замыкание, которое мы передаем AnyIterator.

В результате можно расширить Bag с помощью соответствия протоколу Sequence, вернуть новый AnyIterator, после чего можно удалить BagIterator.

```
extension Bag: Sequence {
    func makeIterator() -> AnyIterator<Element> { ❶
        var exhaustiveStore = store // создаем копию

        return AnyIterator<Element> { ❷
            guard let (key, value) = exhaustiveStore.first else {
                return nil
            }
            if value > 1 {
```

```
            exhaustiveStore[key]? -= 1
        } else {
            exhaustiveStore[key] = nil
        }
        return key
    }
  }
}
```

❶ Метод makeIterator теперь возвращает AnyIterator.

❷ Мы создаем AnyIterator, передавая ему замыкание, которое опустошает exhaustiveStore.

Хотите ли вы использовать AnyIterator, зависит от ситуации, но он может быть отличной альтернативой пользовательскому итератору, чтобы избавиться от шаблонного кода.

9.3.4. Реализация ExpressibleByArrayLiteral

Ранее мы видели, как создать сумку (bag), используя синтаксис литералов массивов. Обратите внимание на то, как мы ее создаем, даже при наличии массивоподобной нотации:

```
let colors: Bag = ["Green", "Green", "Blue", "Yellow", "Yellow", "Yellow"]
```

Чтобы получить такие синтаксические возможности, можно реализовать протокол ExpressibleByArrayLiteral. Тем самым мы реализуем инициализатор, принимающий массив элементов. Затем мы можем использовать эти элементы для распространения свойства store. Мы будем использовать метод reduce в массиве, чтобы свести все элементы в один словарь store.

Листинг 9.22. Реализация протокола ExpressibleByArrayLiteral

```
extension Bag: ExpressibleByArrayLiteral { ❶
    typealias ArrayLiteralElement = Element
    init(arrayLiteral elements: Element...) { ❷
        store = elements.reduce(into: [Element: Int]()) {
            (updatingStore, element) in ❸
            updatingStore[element, default: 0] += 1 ❹
        }
    }
}

let colors: Bag = ["Green", "Green", "Blue", "Yellow", "Yellow", "Yellow"] ❺
print(colors)
// Вывод:
// Green встречается 2 раза
```

```
// Blue встречается 1 раз
// Yellow встречается 3 раза
```

❶ Делаем так, чтобы Bag соответствовал протоколу ExpressibleByArrayLiteral.

❷ Чтобы соответствовать протоколу, нужно реализовать инициализатор для принятия массива элементов.

❸ Используем метод reduce для преобразования массива элементов в словарь.

❹ Увеличиваем значение каждого элемента в словаре на единицу со значением по умолчанию, равным нулю.

❺ Теперь можно использовать синтаксис массива для создания сумки. Обратите внимание, что цвета имеют тип Bag, поэтому Swift знает, что цвета не являются массивом.

Тип Bag имеет интерфейс Swift-friendly. В своем текущем состоянии он готов к использованию, и можно продолжить вносить в него добавления. Например, можно добавить методы intersection и union, чтобы сделать его более ценным. Как правило, не нужно создавать пользовательскую структуру данных каждый день, но время от времени она используется. Знание Sequence закладывает важную основу, потому что это основа для протокола Collection, который немного выше уровнем. О нем пойдет речь в следующем разделе.

9.3.5. Упражнение

3

Создайте бесконечную последовательность. Она итерирует последовательность, которую вы передаете. Бесконечная последовательность удобна для генерации данных, например при использовании метода zip. Бесконечная последовательность продолжает работать, но другая последовательность может опустошиться, тем самым остановив итерацию.

Например, код

```
let infiniteSequence = InfiniteSequence(["a", "b", "c"])
for (index, letter) in zip(0..<100, infiniteSequence) {
   print("\(index): \(letter)")
}
```

выводит следующее:

```
0: a
1: b
2: c
3: a
4: b
... Пропускаем часть кода
```

```
95: c
96: a
97: b
98: c
99: a
```

9.4. Протокол Collection

Настало время перейти к следующему уровню итерации: помимо `IteratorProtocol` и `Sequence`, существует протокол `Collection`.

Протокол `Collection` представляет собой подпротокол `Sequence`. Это означает, что `Collection` наследует все функции от `Sequence`. Одно из основных различий между ними заключается в том, что типы `Collection` индексируются. Другими словами, с их помощью можно получить прямой доступ к элементу в определенной позиции, например через нижний индекс. К примеру, можно использовать `myarray[2]` или `myset["monkeywrench"]`.

Еще одно отличие состоит в том, что `Sequence` не определяет, является ли он деструктивным, то есть когда две итерации могут не давать одинаковых результатов, что может стать проблемой, если вам нужно итерировать одну и ту же последовательность несколько раз.

Например, если вы прервете итерацию и продолжите с того места, где остановились, вы не узнаете, продолжила ли последовательность работу с того места, где остановилась, или перезапустилась с самого начала.

Листинг 9.23. Возобновление итерации

```
let numbers = // Допустим, что numbers - это последовательность,
              // а не коллекция
for number in numbers {
   if number == 10 {
      break
   }
}

for number in numbers {
   // Возобновится ли итерация или начнется с самого начала?
}
```

`Collection` гарантирует, что он не является деструктивным, и позволяет повторно итерировать тип каждый раз с одинаковыми результатами.

Наиболее распространенные типы `Collection`, которые мы используем ежедневно: `String`, `Array`, `Dictionary` и `Set`. Начнем с более внимательного изучения всех доступных протоколов `Collection`.

9.4.1. Ландшафт Collection

Используя протоколы Collection, мы получаем возможности индексирования. Например, в String можно использовать индексы для получения элементов, таких как получение слов до и после пробела, как показано в этом листинге.

Листинг 9.24. Индексирование типа String

```
let strayanAnimals = "Kangaroo Koala"
if let middleIndex = strayanAnimals.index(of: " ") { ❶
   strayanAnimals.prefix(upTo: middleIndex) // Kangaroo
   strayanAnimals.suffix(from: strayanAnimals.index(after: middleIndex))
   // Koala
}
```

❶ Получаем индекс символа внутри строки.

Помимо возможностей индексирования, у Collection есть несколько подпротоколов, каждый из которых предлагает ограничения и оптимизации поверх основного протокола. Например, MutableCollection допускает изменение коллекции, но не позволяет менять ее длину. В то же время RangeReplaceableCollection позволяет это. BidirectionalCollection дает возможность проходить коллекцию в обратном направлении. RandomAccessCollection обещает улучшение производительности по сравнению с BidirectionalCollection (см. рис. 9.5).

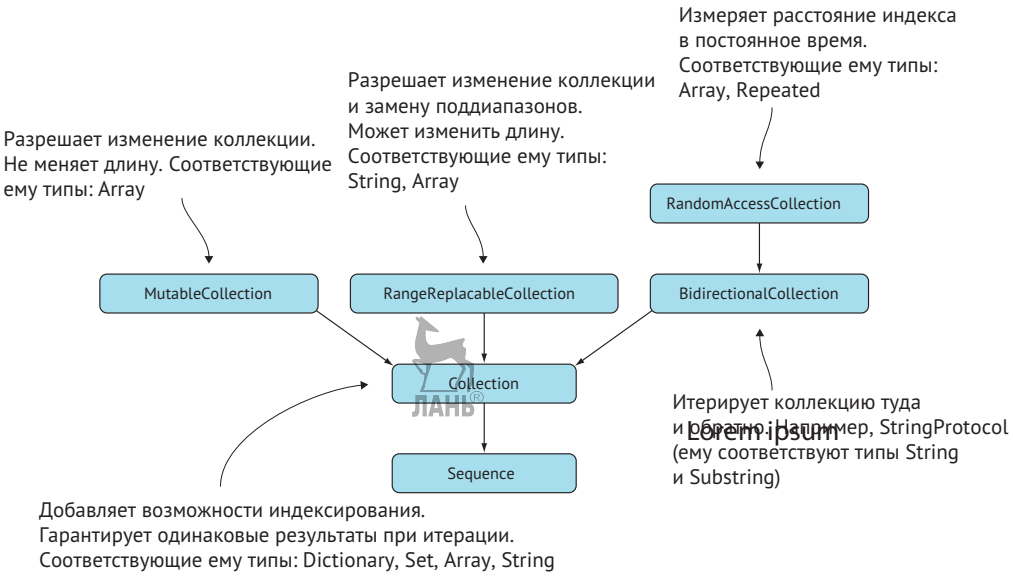


Рис. 9.5. Обзор протоколов Collection

9.4.2. MutableCollection

MutableCollection предлагает методы, которые изменяют элементы на месте без изменения длины коллекции. Поскольку методы MutableCollection не изменяют

длину, протокол может предложить гарантии и улучшения производительности. Наиболее распространенный тип, который соответствует MutableCollection, – это Array.

MutableCollection добавляет несколько специальных методов. Чтобы получить эти методы, нужна переменная Array – в противовес константе. Ее можно объявить, используя ключевое слово var.

С помощью MutableCollection можно выполнить сортировку на месте:

```
var mutableArray = [4, 3, 1, 2]
mutableArray.sort() // [1, 2, 3, 4]
```

Еще один интригующий метод, который предлагает MutableCollection, – это partition. Этот метод перегруппировывает массив на две части, которые вы определяете, и, кроме того, возвращает индекс того места, где массив разделен. Можно, как показано в этом примере, разбить массив целых чисел на нечетные и четные числа.

Листинг 9.25. Использование метода partition

```
var arr = [1,2,3,4,5]
let index = arr.partition { (int) -> Bool in
   return int % 2 == 0 ❶
}

print(arr) // [1, 5, 3, 4, 2] ❷
print(index) // 3 ❸

arr[..<index] // [1, 5, 3] ❹
arr[index...] // [4, 2] ❺
```

❶ Разбиваем массив на четные числа.

❷ Массив перегруппирован; четные числа теперь идут в конце.

❸ Это индекс точки отсечения между нечетными и четными числами.

❹ Первые числа до разделенного индекса являются нечетными.

❺ Числа после разделенного индекса являются четными.

Другие методы, такие как reverse() и swapAt(), также пригодятся. Я рекомендовал бы поэкспериментировать с ними и сделать их частью вашего словаря итераторов.

String не соответствует протоколу MutableCollection

Возможно, вы удивитесь, но тип String не соответствует протоколу MutableCollection, потому что его длина может измениться, если вы измените порядок символов. Изменение длины коллекции – это то, чего MutableCollection не допускает.

Длина String меняется при изменении порядка символов из-за его базовой структуры; символ может состоять из скалярных значений Юникода. Например,

символ é может существовать вне скаляров e и знака ударения ‘; в результате замены é может превратиться в ‘e. Поскольку движущиеся скаляры могут потенциально создавать разные символы, тип String может изменять длину, поэтому он не соответствует MutableCollection.

9.4.3. RangeReplaceableCollection

Следующим по списку идет протокол RangeReplaceableCollection, который позволяет выгружать диапазоны и изменять их длину. Ему соответствуют типы Array и String. Более того, он предоставляет несколько удобных методов, таких как конкатенация с помощью метода +.

Если массив определяется изменчиво с помощью ключевого слова var (не путать с MutableCollection), можно использовать ключевое слово + = для присоединения. Обратите внимание в приведенном ниже листинге, что можно изменить длину массива с помощью методов, предлагаемых RangeReplaceableCollection.

Листинг 9.26. Изменение длины массива

```
var muppets = ["Kermit", "Miss Piggy", "Fozzie bear"]

muppets += ["Statler", "Waldorf"]
print(muppets) // ["Kermit", "Miss Piggy", "Fozzie bear", "Statler", "Waldorf"]

muppets.removeFirst() // "Kermit"
print(muppets) // ["Miss Piggy", "Fozzie bear", "Statler", "Waldorf"]

muppets.removeSubrange(0..<2)
print(muppets) // ["Statler", "Waldorf"]
```

Поскольку тип String соответствует протоколу RangeReplaceableCollection, можно изменять строки на месте и менять их длину.

Листинг 9.27. Изменение типа String

```
var matrix = "The Matrix"
matrix += "Reloaded"
print(matrix) // The Matrix Reloaded
removeAll
```

В RangeReplaceableCollection также существует полезный метод под названием removeAll(where:). С помощью этого метода можно быстро удалить элементы из коллекции, к примеру Array (например, когда вам нужно удалить «Donut» из массива продуктов здорового питания, как показано в этом коде).

Листинг 9.28. removeAll в действии

```
var healthyFood = ["Donut", "Lettuce", "Kiwi", "Grapes"]
healthyFood.removeAll(where:{ $0 == "Donut" })
print(healthyFood) // ["Lettuce", "Kiwi", "Grapes"]
```

У вас может возникнуть соблазн использовать вместо этого метод `filter`, но при удалении значений можно применять специальные оптимизации, тем самым ускоряя использование метода `removeAll` для коллекций переменных.

Примечание

Метод `removeAll` доступен только в том случае, если ваша коллекция является переменной, на что указывает ключевое слово var.

9.4.4. BidirectionalCollection

С помощью `BidirectionalCollection` можно итерировать коллекцию в обратном направлении от индекса, как показано далее. Можно получить индекс перед еще одним индексом, который позволяет получить доступ к предыдущему элементу (например, получение предыдущих символов в строке).

Листинг 9.29. Итерация в обратном направлении

```
var letters = "abcd"
var lastIndex = letters.endIndex
while lastIndex > letters.startIndex {
   lastIndex = letters.index(before: lastIndex)
   print(letters[lastIndex])
}

//Вывод:
// d
// c
// b
// a
```


Использование `index(before:)` – немного низкий уровень для обычного использования. Идиоматически можно применять ключевое слово `reversed()`, чтобы повернуть коллекцию.

```
var letters = "abcd"
for value in letters.reversed() {
   print(value)
}
```

Однако `index(before:)` действительно помогает, если вам нужно выполнить итерацию в обратном направлении только определенное количество раз. При использовании `reversed()` итератор продолжит перебор, пока вы не разорвете цикл.

9.4.5. RandomAccessCollection

`RandomAccessCollection` наследует от `BidirectionalCollection` и предлагает некоторые улучшения производительности для его методов. Основное отличие

состоит в том, что он может измерять расстояния между индексами в постоянное время. Другими словами, RandomAccessCollection накладывает больше ограничений на разработчика, поскольку он должен иметь возможность измерять расстояния между индексами без итерации. Потенциально итерация коллекции может быть затратной, когда речь идет об индексах, например, с помощью index(_ offsetBy:), что не относится к RandomAccessCollection.

Массив соответствует всем протоколам Collection, включая RandomAccess Collection, но более эзотерический тип, который соответствует RandomAccess Collection, – это тип Repeated. Данный тип удобен для многократного перечисления значения. Его можно получить с помощью функции repeatElement, как показано здесь.

Листинг 9.31. Тип repeated

```
for element in repeatElement("Broken record", count: 3) {
   print(element)
}
// Вывод:
// Broken record
// Broken record
// Broken record
```


Можно использовать repeatElement для быстрой генерации значений, что полезно, например, при генерации тестовых данных. Можно даже использовать метод zip, как показано в следующем примере, для более расширенной итерации.

Листинг 9.32. Использование метода zip

```
zip(repeatElement("Mr. Sniffles", count: 3), repeatElement(100,
➥ count: 3)).forEach { name, index in
   print("Generated \(name) \(index)")
}
Generated Mr. Sniffles 100
Generated Mr. Sniffles 100
Generated Mr. Sniffles 100
```

9.5. Создание коллекции

Посмотрим правде в глаза, в большинстве случаев мы не изобретаем новый тип коллекции каждый день. Обычно можно обойтись Set, Array и другими типами, которые предлагает Swift. Это не значит, что изучение Collection – пустая трата времени. Чаще всего у нас есть какая-то структура данных, которая может извлечь выгоду при использовании Collection. Немного кода – и можно привести свои типы в соответствие с Collection и воспользоваться всеми преимуществами, не зная, как сбалансировать двоичные деревья поиска или реализовать другие причудливые алгоритмы.

9.5.1. Создание плана поездки

Мы начнем с создания структуры данных под названием `TravelPlan` (план поездки). План поездки – это последовательность дней, состоящая из одного или нескольких действий. Это может быть поездка во Флориду, которая включает в себя ряд мероприятий, таких как посещение города Ки-Уэст, преследование аллигаторами на поле для гольфа или завтрак на пляже.

Сначала мы создадим структуры данных, а затем сделаем так, чтобы `TravelPlan` соответствовала `Collection`. Таким образом мы сможем итерировать структуру и получать индексирование. Мы начнем со структур `Activity` и `Day`, прежде чем перейти к `TravelPlan`.

Как видно из этого листинга, `Activity` – это не более чем временная отметка и описание.

Листинг 9.33. Activity

```
struct Activity: Equatable {
   let date: Date
   let description: String
}
```

`Day` несколько сложнее, чем `Activity`. У нее есть дата, но поскольку она охватывает целый день, можно сократить время, как показано в следующем листинге. Затем можно сделать так, чтобы `Day` соответствовала протоколу `Hashable`, дабы позже сохранить ее в словаре в качестве ключа.

Listing 9.34. Day

```
struct Day: Hashable { ❶
   let date: Date

   init(date: Date) {
      // Убираем время из Date.
      let unitFlags: Set<Calendar.Component> = [.day, .month, .year] ❷
      let components = Calendar.current.dateComponents(unitFlags, from: date)
      guard let convertedDate = Calendar.current.date(from: components) else { ❸
         self.date = date
         return
      }
      self.date = convertedDate
   }
}
```

❶ `Day` соответствует протоколу `Hashable`.

❷ Нас интересуют только компоненты `day`, `month` и `year`, а не время.

❸ Создаем новую дату без времени.

TravelPlan хранит дни в качестве ключей и мероприятия – в качестве значений. Таким образом, в плане поездки может быть несколько дней, где каждый день может включать в себя несколько мероприятий. Наличие нескольких мероприятий в день отражено в словаре внутри TravelPlan. Поскольку мы обращаемся к этому словарю несколько раз, для удобства вы вводим объявление типа DataType с помощью ключевого слова typealias.

Кроме того, мы добавляем инициализатор, который принимает мероприятия (activities). После этого можно сгруппировать эти мероприятия по дням и сохранить их в словаре. В следующем листинге мы будем использовать метод grouping, чтобы превратить [Activity] в [Day:[Activity]].

Листинг 9.35. TravelPlan

```
struct TravelPlan {
   typealias DataType = [Day: [Activity]] ❶
   private var trips = DataType()

   init(activities: [Activity]) {
      self.trips = Dictionary(grouping: activities) { activity -> Day in ❷
         Day(date: activity.date)
      }
   }
}
```

❶ DataType – это псевдоним типа, чтобы нам не нужно было все время набирать [Day: [Activity]].

❷ Можно создать словарь из массива мероприятий, используя инициализатор grouping в словаре.

grouping работает путем передачи ему последовательности – такой как [Activity] – и замыкания. После этого для каждого элемента вызывается замыкание. Мы возвращаем тип Hashable внутри замыкания, а затем каждое значение массива добавляется к соответствующему ключу. В итоге мы получаем ключи, которые возвращаем в замыкании, и массив значений для каждого ключа.

9.5.2. Реализация Collection

Теперь наша структура данных TravelPlan функционирует, но пока мы не можем итерировать ее. Вместо протокола Sequence мы сделаем так, чтобы TravelPlan соответствовала протоколу Collection.

Вы, вероятно, удивитесь, но нам не нужно реализовывать метод makeIterator. Мы могли бы это сделать, но протокол Collection по умолчанию предоставляет итератор IndexingIterator, который является хорошим преимуществом соответствия Collection.

Чтобы соответствовать Collection, нужно реализовать четыре вещи: две переменные (startIndex и endIndex) и два метода (index(after:) и subscript(index:)).

Поскольку у нас нет собственного алгоритма и мы инкапсулируем тип, который соответствует Collection – в данном случае словарь, – вместо этого можно использовать методы базового словаря, как показано в этом листинге. По сути, мы пересылаем основные методы словаря.

Листинг 9.36. Реализация Collection

```
extension TravelPlan: Collection {
   typealias KeysIndex = DataType.Index ❶
   typealias DataElement = DataType.Element ❶

   var startIndex: KeysIndex { return trips.keys.startIndex } ❷
   var endIndex: KeysIndex { return trips.keys.endIndex } ❷

   func index(after i: KeysIndex) -> KeysIndex { ❸
      return trips.index(after: i)
   }

   subscript(index: KeysIndex) -> DataElement { ❸
      return trips[index]
   }
}
```

❶ Объявляем два псевдонима типа для удобства.

❷ Пересылаем свойства startIndex и endIndex из базового словаря.

❸ Пересылаем методы index (after :) и subscript (index :) из базового словаря.

Вот и все, что нужно! Теперь TravelPlan можно итерировать.

Листинг 9.37. Итерация TravelPlan

```
for (day, activities) in travelPlan {
   print(day)
   print(activities)
}
```

Не нужно придумывать пользовательский итератор, потому что у нас есть IndexingIterator.

Листинг 9.38. Итератор по умолчанию

```
let defaultIterator: IndexingIterator<TravelPlan> = travelPlan.makeIterator()
```

9.5.3. Пользовательские сабскрипты

Соответствие протоколу Collection дает нам возможность использовать сабскрипты, которые позволяют получить доступ к коллекции с помощью квадратных скобок [].

Но поскольку мы пересылаем один из словаря, нам придется использовать индексный тип эзотерического словаря. В следующем листинге вместо этого мы

создадим несколько удобных сабскриптов, что позволит нам получить доступ к элементам.

Листинг 9.39. Реализация сабскриптов

```
extension TravelPlan {
   subscript(date: Date) -> [Activity] {
      return trips[Day(date: date)] ?? []
   }

   subscript(day: Day) -> [Activity] {
      return trips[day] ?? []
   }
}

// Теперь можно получить доступ к содержимому через сабскрипты.
travelPlan[Date()] ❶
let day = Day(date: Date())
travelPlan[day] ❷
```

❶ Оформляем в квадратные скобки дату.

❷ Оформляем в квадратные скобки день.

Еще один критический аспект `Collection` заключается в том, что сабскрипт должен мгновенно возвращать результат, если для вашего типа не указано иное. Мгновенный доступ также известен как *постоянное время* или производительность O(1) в нотации «O» большое. Поскольку мы переадресовываем вызовы методов из словаря, `TravelPlan` возвращает значения в постоянном времени. Если бы нам пришлось итерировать всю коллекцию, чтобы получить значение, доступ не был бы мгновенным, и у нас была бы так называемая производительность O(n). При производительности O(n) более длинные коллекции означают, что время поиска увеличивается линейно. Разработчики, которые будут использовать нашу коллекцию, могут не ожидать такого.

9.5.4. ExpressibleByDictionaryLiteral

Как и в случае с типом `Bag` в разделе 9.3.4 «Реализация ExpressibleByArrayLiteral», можно реализовать протокол `ExpressibleByArrayLiteral` для удобства инициализации. Чтобы соответствовать этому протоколу, нужно всего лишь использовать инициализатор, который принимает несколько элементов. Как показано в этом листинге, внутри тела инициализатора можно ретранслировать метод, вызывая существующий инициализатор.

Листинг 9.40. Реализация ExpressibleByArrayLiteral

```
extension TravelPlan: ExpressibleByArrayLiteral {
   init(arrayLiteral elements: Activity...) {
      self.init(activities: elements)
```

```
    }
}
```

Еще один протокол, который можно реализовать, – это ExpressibleByDictionaryLiteral. Он похож на ExpressibleByArrayLiteral, но позволяет создать TravelPlan из словарной нотации. Мы реализуем инициализатор, который предоставляет массив кортежей. Затем можно использовать метод uniquingKeysWith:, чтобы превратить кортежи в словарь. Замыкание, которое мы передаем uniquingKeysWith:, вызывается, когда возникает конфликт двух ключей. В этом случае мы выбираем одно из двух конфликтующих значений, как показано в этом коде.

Листинг 9.41. Реализация ExpressibleByDictionaryLiteral

```
extension TravelPlan: ExpressibleByDictionaryLiteral {
    init(dictionaryLiteral elements: (Day, [Activity])...) {
        self.trips = Dictionary(elements, uniquingKeysWith: { (first: Day,) in ❶
            return first // Выбираем одно из значений. ❷
        })
    }
}

let adrenalineTrip = Day(date: Date())
let adrenalineActivities = [
    Activity(date: Date(), description: "Bungee jumping"),
    Activity(date: Date(), description: "Driving in rush hour LA"),
    Activity(date: Date(), description: "Sky diving")
]
let adrenalinePlan = [adrenalineTrip: activities] // Теперь можно создать
    TravelPlan из словаря.
```

❶ Передаем массив кортежей инициализатору Dictionary.

❷ Если есть два одинаковых дня, у нас конфликт. Мы используем замыкание, чтобы выбрать один из дней.

9.5.5. Упражнение

4

Сделайте так, чтобы приведенный ниже тип соответствовал протоколу Collection:

```
struct Fruits {
    let banana = "Banana"
    let apple = "Apple"
    let tomato = "Tomato"
}
```

9.6. В заключение

Мы успешно – и относительно безболезненно – создали пользовательский тип, который соответствует протоколу `Collection`. Истинная сила заключается в способности распознать, когда можно реализовать эти протоколы итерации. Скорее всего, в ваших проектах могут быть типы, которые могут получить дополнительную функциональность, соответствуя протоколу `Collection`.

Мы также поближе познакомились с `Sequence` и `IteratorProtocol`, чтобы лучше понять, как работает итерация в Swift. Не нужно быть мастером алгоритма, чтобы привести в действие свои типы, а это пригодится в повседневной работе. Мы также изучили несколько распространенных и полезных методов итераторов, которые можно найти в `Sequence`. Если вам нужны дополнительные советы по итерации, ознакомьтесь с главой 10, в которой рассматриваются методы `map`, `flatMap` и `compactMap`.

Резюме

- Итераторы создают элементы.
- Для итерации Swift использует цикл `while` в методе `makeIterator()`.
- Последовательности создают итераторы, разрешая многократную итерацию.
- Последовательности не гарантируют одинаковые значения при многократной итерации.
- `Sequence` является основой для таких методов, как `filter`, `reduce`, `map`, `zip`, `repeat` и многих других.
- `Collection` наследует из `Sequence`.
- `Collection` – это протокол, который предлагает возможности использования сабскриптов и гарантирует наличие одинаковых результатов при итерации.
- У `Collection` есть подпротоколы, представляющие собой его более специализированные версии.
- `MutableCollection` – это протокол, который предлагает методы изменения без изменения длины коллекции.
- `RangeReplaceableCollection` – это протокол, который ограничивает коллекции для легкой модификации части коллекции. В результате длина коллекции может меняться. Он также предлагает такие полезные методы, как `removeAll(where:)`.
- `BidirectionalCollection` – это протокол, который определяет коллекцию, которую можно итерировать как вперед, так и в обратном направлении.
- `RandomAccessCollection` ограничивает коллекции до итерации постоянного времени между индексами.
- Можно реализовать протокол `Collection` для обычных типов, которые используются в повседневном программировании.

Ответы

1

В чем разница между методами reduce и reduce(into:)?

С помощью метода reduce(into:) можно предотвратить копирование для каждой итерации.

2

Какой из них вы бы выбрали?

reduce имеет смысл, когда вы не создаете затратные копии для каждой итерации, например когда речь идет о целом числе. reduce(into:) имеет больше смысла, когда речь идет о структуре, например массиве или словаре.

3

Создайте бесконечную последовательность. Она будет проходить последовательность, которую вы передаете.

```
// Если вы реализуете Sequence и IteratorProtocol, вам нужно реализовать
➥ только этот метод.
struct InfiniteSequence<S: Sequence>: Sequence, IteratorProtocol {
   let sequence: S
   var currentIterator: S.Iterator
   var isFinished: Bool = false

   init(_ sequence: S) {
      self.sequence = sequence
      self.currentIterator = sequence.makeIterator()
   }

   mutating func next() -> S.Element? {
      guard !isFinished else {
         return nil
      }
      if let element = currentIterator.next() {
         return element
      } else {
         self.currentIterator = sequence.makeIterator()
         let element = currentIterator.next()
         if element == nil {
            // Если последовательность остается пустой после создания новой,
            // то последовательность была пустой с самого начала; нужно
            // будет обезопасить себя от этого в случае бесконечного цикла.
            isFinished = true
         }
```

```
            return element
        }
    }
}

let infiniteSequence = InfiniteSequence(["a", "b", "c"])
for (index, letter) in zip(0..<100, infiniteSequence) {
    print("\(index): \(letter)")
}
```

4

Сделайте так, чтобы приведенный ниже тип соответствовал протоколу `Collection`:

```
struct Fruits {
    let banana = "Banana"
    let apple = "Apple"
    let tomato = "Tomato"
}

extension Fruits: Collection {
    var startIndex: Int {
        return 0
    }
    var endIndex: Int {
        return 3 // Да, 3, а не 2. Этого хочет Collection.
    }
    func index(after i: Int) -> Int {
        return i+1
    }
    subscript(index: Int) -> String {
        switch index {
            case 0: return banana
            case 1: return apple
            case 2: return tomato
            default: fatalError("The fruits end here.")
        }
    }
}

let fruits = Fruits()
fruits.forEach { (fruit) in
    print(fruit)
}
```

Глава 10. map, flatMap и compactMap

В этой главе:

- отображение массивов, словарей и других коллекций;
- когда и как использовать метод `map` с опционалами;
- как и зачем использовать метод `flatMap` для коллекций;
- использование метода `flatMap` для опционалов;
- цепочки и их прерывание с помощью метода `flatMap`;
- как смешивать методы `map` и `flatMap` для расширенных преобразований.

Современные языки, такие как Swift, заимствуют многие концепции из мира функционального программирования, и методы `map` и `flatMap` – яркий тому пример. Рано или поздно при работе с Swift вы столкнетесь или напишите код, где эти методы используются в случае с массивами, словарями и даже опционалами. С помощью `map` и `flatMap` можно написать мощный, лаконичный код, который является неизменяемым, а следовательно, и более безопасным. Более того, начиная с версии 4.1 Swift использует метод `compactMap`, который помогает выполнять эффективные преобразования для опционалов внутри коллекций.

На самом деле вы даже можете быть знакомы с тем, как время от времени использовать методы `map` и `flatMap` в своем коде. В этой главе подробно показано, как применять эти методы различными способами. Кроме того, мы сравним их с альтернативами и рассмотрим связанные с ними компромиссы, поэтому вы точно будете знать, какой стиль выбрать: функциональное программирование или императивный стиль.

При чтении этой главы не возникнет проблем, если вы прочитали главу 9. Если вы еще не читали ее, рекомендую вам это сделать, прежде чем двигаться дальше.

Вначале мы увидим, как метод map работает с массивами и как можно легко выполнять эффективные преобразования с помощью конвейера. Кроме того, вы узнаете, насколько итеративны циклы *for* в сравнении с методом map, поэтому у вас будет несколько практических правил, чтобы выбрать стиль.

Затем мы более детально изучим сопоставление словарей и других типов, соответствующих протоколу `Sequence`.

Изучив сопоставление коллекций, вы узнаете больше о сопоставлении опционалов, что позволяет отложить извлечение опционалов и выбрать безошибочный способ программирования. После этого вы увидите, что map – это абстракция, и насколько это полезно для вашего кода.

Возможность применения метода `flatMap` для опционалов – следующая большая тема. Вы увидите, что `map` не всегда подходит, когда имеешь дело с вложенными опционалами, но, к счастью, `flatMap` может помочь в этом.

После этого вы увидите, как `flatMap` может помочь в борьбе с вложенным извлечением опционалов, где используется конструкция *if let*, которую иногда называют пирамидой гибели. Вы узнаете, как создавать мощные преобразования, сохраняя при этом высокую читабельность кода.

Как и map, метод `flatMap` также определяется в коллекции. В этой главе рассматриваются такие типы коллекций, как массивы и строки, чтобы показать, как использовать `flatMap` для коротких операций.

Сделав еще один шаг вперед, вы узнаете о влиянии метода `CompactMap` на опционалы и коллекции. Вы увидите, как можно фильтровать нулевые значения при преобразовании коллекций.

Вдобавок к этому изучите различия и преимущества после того, как приступите к выполнению вложений с помощью `map`, `flatMap` и `compactMap`.

Цель этой главы – убедиться, что вы с уверенностью и часто используете map, `flatMap` и `compactMap` в повседневной работе. Вы узнаете, как написать более сжатый, а также более неизменяемый код, и узнаете ряд интересных советов и приемов, даже если вы уже время от времени используете методы `map` или `flatMap`. Еще одна причина освоить эти методы – это подготовка к асинхронной обработке ошибок, о которой пойдет речь в главе 11. Там мы снова вернемся к `map` и `flatMap`.

10.1. Знакомство с map

Присоединяйтесь!

Будет более познавательно и веселее, если вы будете знакомиться с кодом по мере прохождения этой главы. Исходный код можно скачать по адресу http://mng.bz/mzr2.

Получение массива, перебор его элементов и преобразование его значений – это обычные операции.

Предположим, что у нас есть имена и фиксации участников какого-то проекта в git. Можно создать более читабельный формат, чтобы увидеть, участвуют ли некоторые разработчики в проекте. Чтобы преобразовать имена и фиксации в удобочитаемый формат, мы начнем с цикла *for* для перебора каждого значения и использования его для заполнения нужного массива.

Листинг 10.1. Преобразование массива

```
let commitStats = [ ❶
   (name: "Miranda", count: 30),
   (name: "Elly", count: 650),
   (name: "John", count: 0)
]

let readableStats = resolveCounts(statistics: commitStats) ❷
print(readableStats) // ["Miranda isn't very active on the project", "Elly
```

```
➥ is quite active", "John isn't involved in the project"]

func resolveCounts(statistics: [(String, Int)]) -> [String] {
   var resolvedCommits = [String]() ❸

   for (name, count) in statistics { ❹
      let involvement: String ❺

      switch count { ❻
         case 0: involvement = "\(name) isn't involved in the project"
         case 1..<100: involvement = "\(name) isn't active on the project"
         default: involvement = "\(name) is active on the project"
      }
      resolvedCommits.append(involvement) ❼
   }
   return resolvedCommits ❽
}
```

❶ Массив кортежей в качестве данных, с которыми мы будем работать.

❷ Передаем массив в функцию `resolveCounts`.

❸ Внутри функции `resolveCounts` мы создаем временный массив, который вернем.

❹ Чтобы перебирать кортежи, name и count привязаны к константам.

❺ Создаем строковый тип `involvement`, который будет заполнен значением.

❻ Используем оператор *switch*.

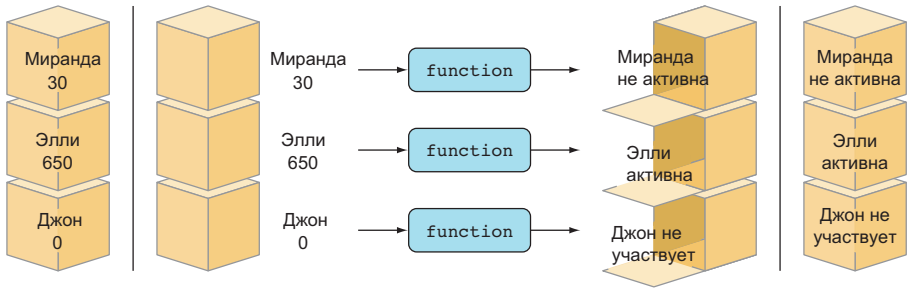
❼ Добавляем строковый тип в конец переменной `resolvedCommits`.

❽ Когда все готово, переменная `resolvedCommits` возвращается в виде преобразованного результата.

Цикл *for* – хорошее начало, но у нас тут шаблонный код. Нам нужен новый массив переменных (`resolvedCommits`), который теоретически можно случайно изменить.

Теперь, когда функция готова, можно легко перепроектировать ее тело, не затрагивая остальную часть программы. Здесь мы используем метод map, чтобы превратить императивный стиль зацикливания в операцию с map.

С помощью этого метода можно перебирать каждый элемент и передавать его замыканию, которое возвращает новое значение. После этого мы получим новый массив с новыми значениями (см. рис. 10.1).



1. Начинаем с массива

2. С помощью метода map картой мы применяем функцию к каждому значению в массиве

3. Метод map создает новый массив с преобразованными значениями внутри

Рис. 10.1. Использование метода map при работе с массивом

В коде это означает, что мы передаем замыкание методу map, который вызывается для каждого элемента в массиве. В замыкании мы возвращаем новую строку, и новый массив создается из этих строк. После того как map завершает работу, метод `resolveCounts` возвращает новый массив.

```
func resolveCounts(statistics: [(String, Int)]) -> [String] {
    return statistics.map { (name: String, count: Int) -> String in ❶
        switch count { ❷
            case 0: return "\(name) isn't involved in the project." ❸
            case 1..<100: return "\(name) isn't very active on the project." ❸
            default: return "\(name) is active on the project." ❸
        }
    }
}
```

❶ Передаем замыкание методу map в массиве `statistics`. После того как map завершает работу, метод `resolveCounts` возвращает новый массив.

❷ Все еще выполняем сопоставление, чтобы определить строку.

❸ Новая строка возвращается в замыкании для каждой итерации.

Обратите внимание, что структура нового массива осталась нетронутой. В итоге мы получаем новый массив такой же длины, за исключением того, что его внутренние значения преобразованы.

И использование map, и цикл *for* дают одинаковые результаты, но вариант с map короче, и такой код неизменяем, что дает сразу два преимущества. Еще одно отличие состоит в том, что при использовании цикла *for* мы ответственны за создание нового массива, в то время как функция map делает это за нас.

10.1.1. Создание конвейера с помощью метода map

Конвейеры – это выразительный способ описания шагов преобразования данных. Давайте узнаем, как он работает.

Представим, что у нас есть функция, которая снова принимает имена и количество фиксаций. Она возвращает только числа, но они отсортированы, при этом пустые отфильтровываются, например:

```
[(name: "Miranda", count:30), (name: "Elly", count:650),
   (name: "John", count:0)] превращается в [650, 30].
```

Можно начать с цикла *for* и преобразовать массив таким образом. Обратите внимание, что в приведенном ниже листинге можно выполнять фильтрацию внутри оператора for вне тела цикла.

Листинг 10.3. Преобразование данных с помощью цикла for

```
func counts(statistics: [(String, Int)]) -> [Int] {
   var counts = [Int]() ❶
   for (name, count) in statistics where count > 0 { ❷
      counts.append(count)
   }
   return counts.sorted(by: >) ❸
}
```

❶ Опять же, мы создаем временный массив.

❷ Выполняем фильтрацию во время итерации с помощью оператора where.

❸ Сортируем counts по убыванию.

Цикл *for* работает прекрасно. В качестве альтернативы можно использовать конвейер, когда вы отдельно описываете каждый шаг. Обратите внимание, что при каждом шаге значение входит и выходит, что позволяет соединять фрагменты конвейера модульным способом. Можно выразить цель операций довольно элегантно, как показано ниже.

Листинг 10.4. Преобразование данных с помощью конвейера

```
func counts(statistics: [(String, Int)]) -> [Int] {
   return statistics ❶
   .map { $0.1 } ❷
   .filter { $0 > 0 } ❸
   .sorted(by: >) ❹
}
```

❶ Когда преобразования сделаны, результат возвращается.

❷ Используем метод map для кортежей внутри массива статистики, возвращая только числа.

❸ Отфильтровываем все пустые числа.

❹ Сортируем.

Метод map является важным компонентом в построении конвейера, потому что он возвращает новый массив. Это помогает преобразовывать данные, сохраняя при этом цепочку, например применяя операции сортировки и фильтрации.

Конвейер определяет четкий и неизменный способ преобразования данных в отдельные этапы. В то время если вы выполняете много операций внутри одного цикла *for*, вы рискуете получить более сложный цикл и изменяемые массивы. Второй недостаток этого цикла заключается в том, что множество действий может запутать цикл, в результате чего вы получите сложный в обслуживании или даже ошибочный код.

Недостатком использования конвейера является то, что вы выполняете как минимум два цикла для этого массива: один для метода `filter`, один для метода map и итерации, – используемых методом `sorted`, который не должен выполнять полную итерацию. С помощью цикла *for* можно повысить эффективность, объединив `map` и `filter` в одном цикле (а также выполнить сортировку, если вы хотите заново изобрести колесо). Как правило, конвейер достаточно эффективен и стоит выразительной и неизменной природы – в конце концов, Swift – быстрый язык. Но когда абсолютная производительность жизненно необходима, лучше использовать цикл *for*.

Еще одним преимуществом цикла *for* является возможность остановить итерацию на полпути с помощью `continue`, `break` или `return`. Но в случае с map или конвейером вам придется выполнять полную итерацию массива для каждого действия.

Как правило, читабельность и неизменность более важны, нежели оптимизация производительности. Используйте то, что лучше всего подходит в вашем конкретном случае.

10.1.2. Использование метода map для словарей

Массив – не единственный тип коллекции, с которым можно использовать метод `map`. Речь идет о словарях. Например, если у вас есть данные фиксаций в форме словаря, вы можете снова преобразовать их в строковые значения.

Прежде чем приступить к работе со словарями, можно легко превратить наш массив кортежей в словарь с помощью инициализатора `uniqueKeysWithValues`, как показано в этом коде.

Листинг 10.5. Превращение кортежей в словарь

```
print(commitStats) // [(name: "Miranda", count: 30), (name: "Elly", count:
   650), (name: "John", count: 0)]
let commitsDict = Dictionary(uniqueKeysWithValues: commitStats)
print(commitsDict) // ["Miranda": 30, "Elly": 650, "John": 0]
```

uniqueKeys

Если два кортежа с одинаковыми ключами передаются в `Dictionary (uniqueKeysWithValues:)`, возникает ошибка времени выполнения. Например, если мы передаем `[(name: "Miranda", count: 30), name: "Miranda", count 40)]`, приложение может аварийно завершить работу. В этом случае можно использовать `Dictionary(_:uniquingKeysWith:)`.

Теперь, когда у нас есть словарь, мы можем приступить к работе.

Листинг 10.6. Использование метода map со словарем

```
print(commitsDict) // ["Miranda": 30, "Elly": 650, "John": 0]
let mappedKeysAndValues = commitsDict.map { (name: String, count: Int) ->
   String in
   switch count {
      case 0: return "\(name) isn't involved in the project."
      case 1..<100: return "\(name) isn't very active on the project."
      default: return "\(name) is active on the project."
   }
}
print(mappedKeysAndValues) // ["Miranda isn't very active on the project",
➥ "Elly is active on the project", "John isn't involved in the project"]
```


От словаря к массиву

Имейте в виду, что переход от словаря к массиву означает, что вы переходите от неупорядоченной коллекции к упорядоченной. Возможно, это не тот порядок, который вам нужен.

Используя метод map, мы превратили ключ и значение в одну строку. Если нужно, можно применять только значения словаря. Например, можно преобразовать числа в строки, при этом сохраняя имена в словаре, как в приведенном ниже примере.

Листинг 10.7. Использование метода map со значениями словаря

```
let mappedValues = commitsDict.mapValues { (count: Int) -> String in
   switch count {
      case 0: return "Not involved in the project."
      case 1..<100: return "Not very active on the project."
      default: return "Is active on the project"
   }
}
print(mappedValues) // ["Miranda": "Very active on the project", "Elly":
➥ "Is active on the project", "John": "Not involved in the project"]
```


Обратите внимание, что, используя mapValues, мы продолжаем владеть словарем, а при применении map получаем массив.

10.1.3. Упражнения

1

Создайте функцию, которая превращает массив во вложенный массив. Убедитесь, что вы используете метод map:

```
makeSubArrays(["a","b","c"]) // [[a], [b], [c]]
makeSubArrays([100,50,1]) // [[100], [50], [1]]
```

2

Создайте функцию, которая преобразует значения в словаре фильмов. Каждую оценку от 1 до 5 необходимо преобразовать в удобочитаемый формат (например, оценка 1,2 – «Очень низкая», оценка 3 – «Средняя», а оценка 4,5 – «Отлично»):

```
let moviesAndRatings: [String : Float] = ["Home Alone 4" : 1.2, "Who
➥ Framed Roger Rabbit?" : 4.6, "Star Wars: The Phantom Menace" : 2.2,
➥ "The Shawshank Redemption" : 4.9]

let moviesHumanRadable = transformRating(moviesAndRatings)

print(moviesHumanRadable) // ["Home Alone 4": "Weak", "Star Wars: The
➥ Phantom Menace": "Average", "Who Framed Roger Rabbit?": "Excellent",
➥ "The Shawshank Redemption": "Excellent"]
```

3

Все еще просматривая фильмы и рейтинги, преобразуйте словарь в описание для каждого фильма с добавлением рейтинга к заголовку. Например:

```
let movies: [String : Float] = ["Home Alone 4" : 1.2, "Who framed Roger
➥ Rabbit?" : 4.6, "Star Wars: The Phantom Menace" : 2.2, "The Shawshank
➥ Redemption" : 4.9]
```

превращается в

```
["Home Alone 4 (Weak)", "Star Wars: The Phantom Menace (Average)",
➥ "Who framed Roger Rabbit? (Excellent)", "The Shawshank Redemption
➥ (Excellent)"]
```

Попробуйте использовать метод map, если это возможно.

10.2. Последовательности

Ранее мы видели, как можно использовать метод map для типов `Array` и `Dictionary`. Эти типы реализуют протоколы `Collection` и `Sequence`.

В качестве примера можно сгенерировать фиктивные данные – например, данные тестов или данные для заполнения экранов в пользовательском интерфейсе – путем определения имен, создания диапазона целых чисел и использования метода map для каждой итерации. В этом случае можно быстро сгенерировать массив имен, как показано в этом примере.

Листинг 10.8. Использование метода map с последовательностью Range

```
let names = [ ❶
   "John",
```

```
    "Mary",
    "Elizabeth"
]

let nameCount = names.count ❷

let generatedNames = (0..<5).map { index in ❸
    return names[index % nameCount] ❹
}

print(generatedNames) // ["John", "Mary", "Elizabeth", "John", "Mary"]
```

❶ Определяем набор имен для генерации.

❷ Выполняем измерение один раз (в отличие от каждой итерации с map).

❸ Используем метод map для диапазона, в данном случае от 0 до 4.

❹ Используя оператор модуля, мы генерируем значение 0, 1 или 2. После этого выбираем имя на основе этого значения.

С помощью данной последовательности можно генерировать числа, но, используя метод map, можно генерировать значения, которые могут не быть числами.

Помимо диапазонов, также можно использовать метод map с методами `zip` или `stride`, потому что они соответствуют протоколу `Sequence`. С помощью map можно генерировать множество полезных значений и экземпляров. В случае с последовательностями и коллекциями всегда возвращается новый массив, что следует учитывать при использовании данного метода с этими типами.

10.2.1. Упражнение

4

Создайте массив из букв «a», «b» и «c» 10 раз. Конечным результатом должно быть [«a», «b», «c», «a», «b», «c», «a» ...], пока массив не будет состоять из 30 элементов. Попробуйте использовать метод map, если это возможно.

10.3. Использование метода map для опционалов

Поначалу применение метода map может показаться необычным способом обновления значений внутри коллекций. Но концепция map не предназначена только для коллекций. Его также можно использовать для работы с опционалами. Наверное, звучит странно? Не совсем – это похоже на преобразование значения внутри контейнера.

В случае с массивом мы преобразуем его внутренние значения. Напротив, при работе с опционалом мы преобразуем его единственное значение (при условии

что оно одно). Использование метода map при работе с коллекциями и опционалами не сильно отличается. Вы будете чувствовать себя комфортно с ним и использовать его мощь.

10.3.1. Когда использовать метод map с опционалами

Представим, что нам нужно создать типографию, где можно распечатывать книги с фотографиями из социальных сетей и комментариями к ним. К сожалению, типография не поддерживает специальные символы, такие как смайлики, поэтому нужно убрать их из текстов.

Давайте посмотрим, как использовать метод map, чтобы очистить свой код, удалив смайлики. Но перед этим мы начнем с того, что извлечем смайлики из строк, которые будем использовать позже.

Извлечение смайликов

Прежде всего нам нужна функция, которая убирает смайлики из типа `String`; она делает это путем перебора представления `unicodeScalars` и удаляет любой скаляр, являющийся смайликом. Это можно сделать, передав метод `isEmoji` в метод `removeAll`, как показано здесь:

Листинг 10.9. Функция removeEmojis

```
func removeEmojis(_ string: String) -> String {
   var scalars = string.unicodeScalars ❶
   scalars.removeAll(where: isEmoji) ❷
   return String(scalars) ❸
}

func isEmoji(_ scalar: Unicode.Scalar) -> Bool { ❹
   // Мы наполним тело. Сперва давайте сосредоточимся на сигнатурах функций
   return true
}
```

❶ Создаем изменяемую копию скаляров строки. Она должна быть изменяемой, чтобы мы могли вызвать `removeAll`.

❷ Удаляем каждый скаляр, который является смайликом.

❸ Преобразуем скаляры обратно в тип `String` снова, прежде чем возвращать его.

❹ На этом этапе функция `isEmoji` является заполнителем, поэтому сначала можно подготовить функцию `removeEmojis`.

При создании новой функциональности полезно создавать фиктивные функции, такие как `isEmoji`, чтобы иметь возможность завершить работу над высокоуровневыми функциями, такими как, например, `removeEmojis`, не отвлекаясь от низкоуровневых функций.

На этом этапе функция `removeEmojis` готова и код компилируется. Мы готовы завершить работу над телом функции `isEmoji`.

Чтобы создать эту функцию, мы проверяем, является ли символ Юникода частью диапазона смайликов путем сопоставления с образцом.

Листинг 10.10. Определяем, находится ли символ Юникода в диапазоне смайликов

```
// Определяем, находится ли символ Юникода в диапазоне смайликов,
// на основе таблиц символов.
// https://apps.timwhitlock.info/emoji/tables/unicode
func isEmoji(_ scalar: Unicode.Scalar) -> Bool {
   switch Int(scalar.value) {
      case 0x1F601...0x1F64F: return true // Смайлики.
      case 0x1F600...0x1F636: return true // Дополнительные смайлики.
      case 0x2702...0x27B0: return true // Графические смайлики.
      case 0x1F680...0x1F6C0: return true // Транспортные и картографические
         // символы.
      case 0x1F681...0x1F6C5: return true // Дополнительные транспортные
         // и картографические символы.
      case 0x24C2...0x1F251: return true // Обрамленные символы.
      case 0x1F30D...0x1F567: return true // Другие дополнительные символы.
      default: return false
   }
}
```

Обе функции готовы. Давайте продолжим и посмотрим, как встроить это в операцию с использованием метода `map`, чтобы очистить код.

10.3.2. Создание обложки

Вернемся к нашей типографии. Чтобы создать печатную фотокнигу, нужно сделать обложку. На этой обложке всегда будет изображение, и у него будет заголовок (title), который будет отображаться в верхней части изображения.

Цель состоит в том, чтобы применить функцию `removeEmojis` к этому заголовку, дабы, когда на обложке будет заголовок, смайлики удалялись. Таким образом, смайлики не будут превращаться в квадратики (см. рис. 10.2).

```
let cover = Cover(image: image, title: "❤️ OMG Cute ⭐⭐babypics⭐⭐! 😍❤️🍼👶")
print(cover.title) // Optional("OMG Cute babypics!")
```

Рис. 10.2. Удаление смайликов из заголовка обложки

В приведенном ниже листинге мы вводим класс `Cover`, который содержит опциональное свойство `title`. Поскольку `removeEmojis` не принимает опционал, сначала нужно извлечь заголовок, чтобы применить функцию `removeEmojis`.

Листинг 10.11. Класс Cover

```
class Cover {
   let image: UIImage
```

```
    let title: String?

    init(image: UIImage, title: String?) {
       self.image = image
       var cleanedTitle: String? = nil ❶
       if let title = title { ❷
          cleanedTitle = removeEmojis(title) ❸
       }
       self.title = cleanedTitle ❹
    }
}
```

❶ Создаем временную переменную.

❷ Извлекаем заголовок.

❸ Применяем функцию `removeEmojis` к извлеченному заголовку.

❹ Устанавливаем заголовок в классе.

Можно объединить (и улучшить) эти четыре шага в один, используя метод `map`.

При использовании метода `map` можно применить функцию `removeEmojis()` к извлеченному значению внутри `map` (если оно есть). Если опционал равен nil, операция игнорируется (см. рис. 10.3).

Рис. 10.3. Использование метода map при работе с двумя опционалами

В приведенном ниже листинге видно, как будет выглядеть использование метода map для опционала внутри класса `Cover`.

Листинг 10.12. Использование метода map для опционала

```
class Cover {
    let image: UIImage
    let title: String?

    init(image: UIImage, title: String?) {
       self.image = image
```

```
        self.title = title.map { (string: String) -> String in ❶
            return removeEmojis(string) ❷
        }
    }
}
```

❶ Передаем замыкание, чтобы использовать метод map для опционала.

❷ Внутри замыкания у нас есть обычная строка, которую мы преобразуем с помощью функции removeEmojis.

Обе операции дают одинаковый вывод, то есть заголовок без смайликов, за исключением того, что мы сократили операцию, использующую метод map.

Внутри map

Обратите внимание, что значение внутри замыкания map не является опционалом. В этом состоит прелесть map: нам не нужно беспокоиться о том, является значение опционалом или нет.

10.3.3. Более короткий вариант нотации

Нам удалось убрать несколько строк; здорово, правда? Посмотрим, можно ли еще сократить код.

Можно начать с использования сокращенного обозначения замыкания:

```
self.title = title.map {removeEmojis ($ 0)}
```

Но все, что нужно map, – это функция, которая принимает аргумент и возвращает значение. Таким образом, вместо того чтобы создавать замыкание, можно передать существующую функцию removeEmojis напрямую map:

```
self.title = title.map (removeEmojis)
```

Это работает, потому что данная функция принимает один параметр и возвращает один параметр, а это именно то, чего ожидает map.

Примечание

Фигурные скобки {} заменены круглыми (), потому что мы не создаем замыкание; мы передаем ссылку на функцию в качестве обычного аргумента.

Конечный результат, как показано в этом листинге, теперь намного короче.

Листинг 10.13. Чистый вариант

```
class Cover {
    let image: UIImage
    let title: String?

    init(image: UIImage, title: String?) {
        self.image = image
```

```
        self.title = title.map(removeEmojis)
    }
}
```

Используя map, мы превратили свою многострочную логику в чистый неизменяемый однострочник. Свойство `title` остается опционалом, но его значения преобразуются, когда у переданного аргумента `title` есть значение.

Преимущества использования метода map с опционалами

У опционала есть контекст, а именно: равно значение nil или нет. Можно использовать метод map для этого контекста. Учитывая, что опционалы в Swift повсюду, использование метода map при работе с ними – мощное средство.

Можно выполнять действия над опционалом так, как если бы он был извлечен. Таким образом, его извлечение можно отложить. Кроме того, функции, которую вы передаете map, не нужно знать об опионалах или иметь с ними дело, что является еще одним преимуществом.

Применение метода map помогает избавиться от шаблонного кода. Благодаря ему больше не нужно возиться с временными переменными или ручным извлечением. Неизменяемое программирование – хорошая практика, потому что оно избавляет вас от переменных, которые меняются прямо у вас под носом.

Еще одним преимуществом метода map является то, что можно продолжать цепочку, как показано в приведенном ниже листинге. Например, помимо удаления смайликов, также можно удалить пробел из заголовка, дважды применив метод map.

Листинг 10.14. Цепочка

```
class Cover {
    let image: UIImage
    let title: String?

    init(image: UIImage, title: String?) {
        self.image = image
        self.title = title.map(removeEmojis).map { $0.trimmingCharacters(in:
            .whitespaces) }
    }
}
```

В листинге 10.14 метод map используется дважды. Сначала мы передаем функцию `removeEmojis`, а затем замыкание, чтобы обрезать пробел. Вторая операция с использованием map выполняется через замыкание, потому что в этом случае нельзя передать ссылку на функцию. Кроме того, обратите внимание, что мы не извлекали опционал, но выполнили с ним несколько действий.

В итоге получается небольшой конвейер, в котором мы неизменно применяем метод map для опционала. Где-нибудь еще в приложении можно извлечь опцио-

нал, чтобы прочитать значение. Но до тех пор можно делать вид, что значение опционала не равно nil, и работать с его внутренним значением.

10.3.4. Упражнение

5

Имеется словарь контактных данных. Приведенный ниже код получает улицу и город из данных и очищает строки. Посмотрите, можно ли избавиться от шаблонного кода (обязательно используйте метод map):

```
let contact =
   ["address":
      [
         "zipcode": "12345",
         "street": "broadway",
         "city": "wichita"
      ]
]

func capitalizedAndTrimmed(_ string: String) -> String {
   return string.trimmingCharacters(in: .whitespaces).capitalized
}

// Очищаем этот код:
var capitalizedStreet: String? = nil
var capitalizedCity: String? = nil
if let address = contact["address"] {
   if let street = address["street"] {
      capitalizedStreet = capitalizedAndTrimmed(street.capitalized)
   }
   if let city = address["city"] {
      capitalizedCity = capitalizedAndTrimmed(city.capitalized)
   }
}

print(capitalizedStreet) // Бродвей
print(capitalizedCity) // Уичито
```

10.4. map – это абстракция

С помощью метода map можно преобразовывать данные, минуя контейнеры или контексты, такие как массивы, словари или опционалы.

Вернемся к нашему методу, который удаляет смайлики из строки, из предыдущего раздела. Используя map, можно использовать функцию removeEmojis для всех типов контейнеров, таких как строки, словари или наборы (см. рис. 10.4).

```
let omgBabies: String? = "❤️ OMG Cute ⭐⭐babypics⭐⭐! 😍❤️🍼👶"
print(omgBabies.map(removeEmojis)) // Optional(" OMG Cute babypics! ")

let food = ["Favorite Meal": "🍕 Pizza", "Favorite Drink": "☕ Coffee"]
print(food.mapValues(removeEmojis)) // ["Favorite Meal": " Pizza", "Favorite Drink": " Coffee"]

let set: Set<String> = ["Great job 👍", "Excellent 🙌"]
print(set.map(removeEmojis)) // ["Great job ", "Excellent "]
```

Рис. 10.4. Удаление смайликов для нескольких типов

Независимо от того, имеете вы дело со словарями, массивами, опционалами или наборами, `removeEmoji` работает с любым типом благодаря методу `map`. Не нужно писать функцию `removeEmojis` отдельно для каждого типа.

Абстракция `map` носит название *функтор*. Этот термин происходит из раздела математики «теория категорий». Вам необязательно знать математику, чтобы работать с методом `map`. Но интересно посмотреть, как функтор определяет то, с чем можно использовать `map`, и что Swift заимствует эту функцию из мира функционального программирования.

10.5. Овладеваем методом flatMap

Изучение метода `flatMap` сродни обряду в Swift. Сначала `flatMap` похож на монстра у вас под кроватью: поначалу это пугает, но как только вы столкнетесь с ним, то увидите, что он не так уж и плох.

Цель этого раздела – развить понимание `flatMap` – создать ощущение типа «Что это было?», словно это неудавшийся магический трюк.

10.5.1. В чем преимущества flatMap?

Давайте уберем волшебство: `flatMap` выполняет операцию «разглаживания».

Операция «разглаживания» полезна, когда мы применяем map, но в итоге получаем вложенный тип. Например, при использовании метода map мы в конечном итоге получаем `Optional(Optional (4))`, а нам бы хотелось, чтобы было

Optional(4). Таким же образом мы получаем [[1, 2, 3], [4, 5, 6]], а нам нужно [1, 2, 3, 4, 5, 6].

Говоря проще: используя метод flatMap, мы комбинируем вложенные структуры.

Это немного сложнее – например, возможность чередовать операции во время переноса контекстов или когда вам нужно писать код без ошибок, – но вы скоро об этом узнаете.

На данный момент это все теория. Не так уж и плохо, правда? Давайте разберем интересные детали.

10.5.2. Когда метод map не подходит

Давайте посмотрим, как flatMap влияет на опционалы.

Рассмотрим приведенный ниже пример, в котором нам нужно преобразовать опциональный тип String в опциональный объект URL. Мы по наивности пытаемся использовать map и быстро убеждаемся, что он не подходит для этой ситуации.

Листинг 10.15. Преобразование типа String в URL

```
// Мы получили эти данные
let receivedData = ["url": "https://www.clubpenguinisland.com"]

let path: String? = receivedData["url"]

let url = path.map { (string: String) -> URL? in
   let url = URL(string: string) // Опционал (https://www.clubpenguinisland.com)
   return url // Возвращаем опциональный тип String
}

print(url) // Опционал (Опционал (http://www.clubpenguinisland.com))
```

В этом сценарии нам предоставляется опциональный тип String. Мы хотим преобразовать его в опциональный объект URL.

Проблема, однако, заключается в том, что в результате создания URL может вернуться значение nil. URL возвращает nil, когда мы передаем ему неверный URL.

Когда мы применяем метод map, то возвращаем объект URL? в функции. К сожалению, результатом этого становятся два опционала, вложенных друг в друга: Optional(Optional (http://www.clubpenguinisland.com)).

Когда вам нужно удалить один слой вложенности, можно принудительно извлечь опционал. Легко, правда? Посмотрим на результаты в приведенном ниже коде.

Листинг 10.16. Устранение двойной вложенности с помощью принудительного извлечения

```
let receivedData = ["url": "https://www.clubpenguinisland.com"]
```

```
let path: String? = receivedData["url"]

let url = path.map { (string: String) -> URL in ❶
   return URL(string: string)! ❷
}

print(url) // Опционал (http://www.clubpenguinisland.com). ❸
```

❶ Теперь мы возвращаем обычный URL.

❷ Выполняем принудительное извлечение – опасно!

❸ Опционал больше не является вложенным.

Попридержите своих лошадок.

Даже несмотря на то, что это решает проблему двойной вложенности опционала, использование принудительного извлечения теперь может привести к сбою. В нашем примере код работает нормально, потому что URL действителен. Но в реальных приложениях это может быть не так. Как только URL вернет nil, на этом все, и происходит сбой.

Вместо показанного здесь варианта можно использовать `flatMap`, чтобы удалить один слой вложенности.

Листинг 10.17. Используем метод flatMap, чтобы избавиться от двойной вложенности

```
let receivedData = ["url": "https://www.clubpenguinisland.com"]

let path: String? = receivedData["url"]

let url = path.flatMap { (string: String) -> URL? in ❶
   return URL(string: string) ❷
}

print(url) // Опционал (http://www.clubpenguinisland.com). ❸
```

❶ Снова возвращаем URL?.

❷ Возвращаем опционал URL!

❸ Двойная вложенность отсутствует.

Обратите внимание, что мы снова возвращаем URL? в функции `flatMap`. Как и в случае с map, оба замыкания одинаковы. Но поскольку мы используем `flatMap`, «разглаживание» происходит после преобразования. Эта операция удаляет один слой вложенности.

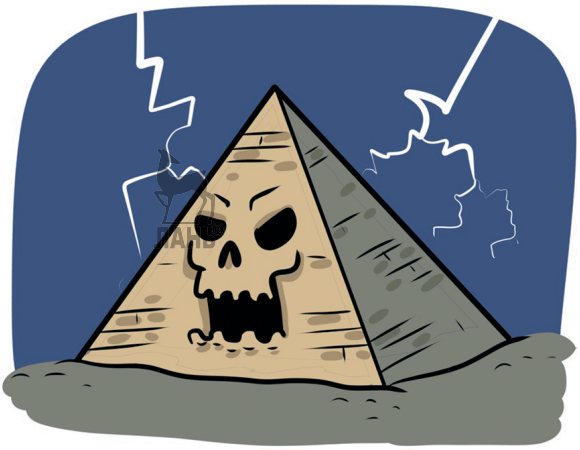
`FlatMap` сначала выполняет действие, аналогичное `map`, а затем «разглаживает» опционал. Можно рассматривать этот метод как `mapFlat` из-за последовательности операций.

Используя `flatMap`, можно продолжить трансформацию опционалов и воздержаться от использования принудительного извлечения.

10.5.3. Борьба с пирамидой гибели

Пирамида гибели – это код, который смещается вправо через отступ. Данная форма пирамиды является результатом большого количества вложений, например при извлечении нескольких опционалов.

Вот еще один пример, когда `map` не сможет помочь, а `flatMap` пригодится для борьбы с этой пирамидой.



Чтобы продемонстрировать, как это делается, поговорим о делении целых чисел.

Когда мы делим целое число, десятичные дроби обрезаются:

```
print (5/2) // 2
```

При делении типа `Int` деление 5 на 2 возвращает 2 вместо 2,5, потому что у `Int` нет такой точности, как у типа `Float`.

Функция в листинге 10.18 делит пополам значение `Int` только тогда, когда оно четное. Если значение нечетное, функция возвращает nil.

Функция безопасного деления на 2 принимает неопционал и возвращает опционал. Например, при делении 4 на 2 получается `Optional(2)`. Но при делении 5 на 2 возвращается nil, потому что десятичные дроби будут обрезаться.

Листинг 10.18. Функция безопасного деления на 2

```
func half(_ int: Int) -> Int? { // Делит только четные числа
   guard int % 2 == 0 else { return nil }
   return int / 2
}

print(half(4)) // Опционал (2)
print(half(5)) // nil
```

Далее мы будем постоянно применять эту функцию.

Мы создаем начальное значение и делим его пополам, в результате чего возвращается новый опционал. Если мы делим его на 2, сначала нужно извлечь вновь разделенное значение, прежде чем передать его функции `half`.

Такая цепочка событий создает неприятное дерево из выражений `if let` с отступом, также известное как пирамида гибели, как показано ниже.

Листинг 10.19. Пирамида гибели

```
var value: Int? = nil
let startValue = 80
if let halvedValue = half(startValue) {
   print(halvedValue) // 40
   value = halvedValue
   if let halvedValue = half(halvedValue) {
      print(halvedValue) // 20
      value = halvedValue
      if let halvedValue = half(halvedValue) {
         print(halvedValue) // 10
         if let halvedValue = half(halvedValue) {
            value = halvedValue
         } else {
            value = nil
         }
      } else {
         value = nil
      }
   } else {
      value = nil
   }
}

print(value) // Опционал (5)
```

Как видно из этого примера, когда вам нужно непрерывно применять функцию к значению, вы должны продолжать извлекать возвращаемое значение. Такое вложенное извлечение происходит потому, что `half` каждый раз возвращает опционал.

Кроме того, операторы `if let` в Swift можно группировать, что является идиоматическим подходом.

Листинг 10.20. Сочетание операторов if let

```
let startValue = 80
var endValue: Int? = nil

if
```

```
    let firstHalf = half(startValue),
    let secondHalf = half(firstHalf),
    let thirdHalf = half(secondHalf),
    let fourthHalf = half(thirdHalf) {
    endValue = fourthHalf
}

print(endValue) // Опционал(5)
```

Недостатком этого подхода является то, что приходится привязывать значения к константам для каждого шага, а именование каждого шага может быть громоздким. Кроме того, не все функции беспрекословно принимают одно значение и возвращают другое, а это означает, что вы будете соединять несколько замыканий, и в этом случае подход с *if let* не корректен.

10.5.4. Использование метода flatMap с опционалом

Теперь мы посмотрим, насколько выгодно использование `flatMap`. Помните, что этот метод похож на `map`, за исключением того, что он удаляет слой вложенности после действий, аналогичных методу `map`. Например, у нас есть `Optional(4)` и метод `flatMap`. Мы применяем функцию `half`, возвращающую новый опционал, который `flatMap` «разглаживает» (см. рис. 10.5 и листинг 10.21).

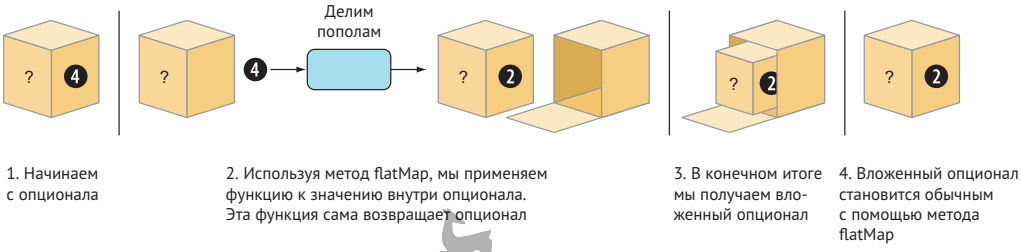


Рис. 10.5. Успешная операция с использованием метода flatMap

Видно, что если мы используем метод `flatMap` в функции `half` с `Optional(4)`, то получаем `Optional(2)`. При использовании метода `map` у нас было бы это: `Optional(Optional(2))`.

Листинг 10.21. Деление на 2 с использованием flatMap

```
let startValue = 8
let four = half(startValue) // Опционал (4)
let two = four.flatMap { (int: Int) -> Int? in
    print(int) // 4
    let nestedTwo = half(int)
    print(nestedTwo) // Опционал (2)
    return nestedTwo
}

print(two) // Опционал (2)
```

Прелесть использования flatMap состоит в том, что мы сохраняем обычный опционал, а это означает, что мы можем продолжать с ним операции по сцеплению.

Листинг 10.22. Многократные операции деления пополам

```
let startValue = 40

let twenty = half(startValue) // Опционал (20)

let five =
   twenty
      .flatMap { (int: Int) -> Int? in
         print(int) // 20
         let ten = half(int)
         print(ten) // Опционал (10)
         return ten
      }.flatMap { (int: Int) -> Int? in
         print(int) // 10
         let five = half(int)
         print(five) // Опционал (5)
         return five
      }

print(five) // Опционал (5)
```

Поскольку мы никогда не вкладываем опционал более одного раза, можно продолжать эту цепочку до бесконечности или всего два раза, как в этом примере. Мы избавляемся от уродливой пирамиды гибели. Кроме того, нам больше не нужно отслеживать временное значение вручную во время манипуляций с операторами *if let*.

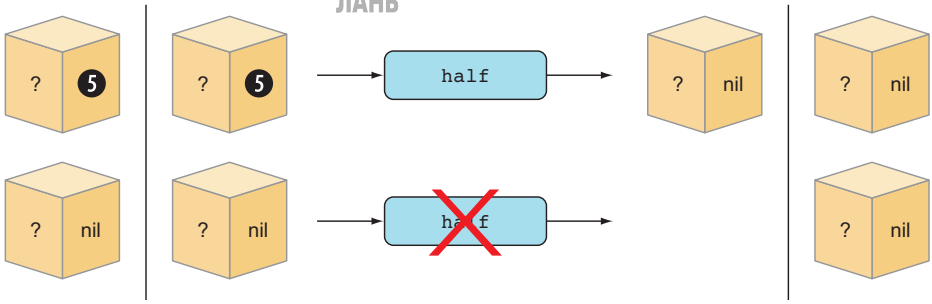
Разрыв цепочки с помощью flatMap

Поскольку flatMap позволяет сохранять цепочки операций, допускающих пустые значения, мы получаем еще одно преимущество: возможность разорвать цепочки, если это необходимо.

Когда мы возвращаем nil из операции с flatMap, то взамен получаем обычное значение nil. Это означает, что последующие операции такого рода игнорируются.

В приведенном ниже примере мы делим 5 пополам и применяем метод flatMap. В итоге мы получаем nil (см. рис. 10.6).

Если у нас нет нулевого опционала, последующие операции с flatMap игнорируются. Поскольку мы можем вернуть ноль из замыкания flatMap, то можем разорвать цепочку.



1. У нас есть два опционала: один со значением и один без

2. Используя метод flatMap, мы применяем функцию к значению внутри опционала. Функция возвращает опционал. В случае с 5 она возвращает nil. Таким образом, в конечном итоге мы получаем значение nil

Но метод flatMap ничего не делает с пустым опционалом. Nil остается nil

3. В любом случае мы получаем значение nil

Рис.10.6. Метод flatMap и значения nil

Давайте посмотрим в приведенном ниже примере, что произойдет, если продолжить цепочку, даже если в ходе операции с `flatMap` вернется nil.

Листинг 10.23. Разрыв цепочки

```
let startValue = 40
let twenty = half(startValue) // Опционал (20)
let someNil =
   twenty
      .flatMap { (int: Int) -> Int? in
         print(int) // 20
         let ten = half(int)
         print(ten) // Опционал (10)
         return ten
      }.flatMap { (int: Int) -> Int? in
         print(int) // 10
         let five = half(int)
         print(five) // Опционал (5)
         return five
      }.flatMap { (int: Int) -> Int? in ❶
         print(int) // 5
         let someNilValue = half(int)
         print(someNilValue) // nil
         return someNilValue ❷
      }.flatMap { (int: Int) -> Int? in ❸
         return half(int) ❹
      }
```

```
print(someNil) // nil
```

❶ Замыкание возвращает nil.

❷ Мы возвращаем nil.

❸ Замыкание в последней операции с `flatMap` вызываться не будет, поскольку значение опционала – nil.

❹ Этот код так и не вызывается, потому что мы вызываем `flatMap` при значении nil.

`flatMap` игнорирует переданные замыкания, как только найден nil. То же самое наблюдается при использовании метода `map` с опционалом, когда при обнаружении nil также ничего не происходит.

Обратите внимание, что в ходе третьей операции с `flatMap` возвращается nil, а четвертая операция игнорируется. Результат прежний, а это означает, что `flatMap` дает возможность прервать цепочку.

Мы позволяем `flatMap` обрабатывать любые неудачные преобразования, а вместо этого могли бы сосредоточиться на сценарии «счастливый путь».

Более того, чтобы завершить и очистить свой код, мы можем использовать более короткий вариант нотации, как делали это раньше, когда передавали именованную функцию методу `map`. То же самое можно сделать и с flatMap, как показано здесь.

Листинг 10.24. Более короткий вариант нотации

```
let endResult =
   half(80)
      .flatMap(half)
      .flatMap(half)
      .flatMap(half)

print(endResult) // Опционал (5)
```

Как правило, объединение операторов *if let* – приемлемый путь, который не требует глубоких знаний `flatMap`. Если вы не хотите создавать промежуточные константы или работаете с несколькими замыканиями, можно использовать подход с `flatMap` для написания краткого кода.

Представим себе сценарий, в котором мы находим пользователя по идентификатору, находим его любимый продукт и смотрим, есть ли какой-нибудь сопутствующий продукт. Последний шаг форматирует данные. Любой из этих шагов может быть функцией или замыканием. С помощью методов `flatMap` и `map` можно аккуратно сцеплять преобразования, не прибегая к большому количеству констант *if let* и промежуточных значений.

Листинг 10.25. Поиск сопутствующих продуктов

```
let alternativeProduct =
   findUser(3)
```

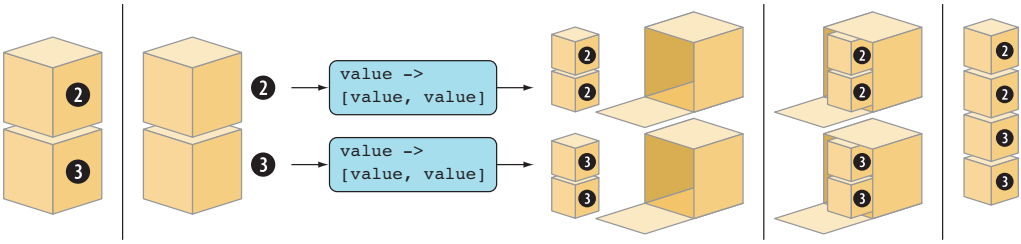
```
    .flatMap(findFavoriteProduct)
    .flatMap(findRelatedProduct)
    .map { product in
        product.description.trimmingCharacters(in: .whitespaces)
    }
```

10.6. flatMap и коллекции

Вы, наверное, уже догадались, что `flatMap` зарезервирован не только для опционалов, но и для коллекций.

Точно так же, как этот метод «разглаживает» опционалы, он делает то же самое с вложенной коллекцией после операции с `map`.

Например, у нас есть функция, которая генерирует новый массив из каждого значения внутри массива – например, превращая `[2, 3]` в `[[2, 2], [3, 3]]`. С помощью `flatMap` можно снова «разгладить» эти подмассивы, превращая их в один массив, например `[2, 2, 3, 3]` (см. рис. 10.7).



1. У нас есть обычный массив [2, 3]

2. Используя метод flatMap, мы применяем функцию к каждому значению в массиве. Эта функция превращает значение в [value, value], еще один массив

3. В конечном итоге мы получаем подмассивы [[2, 2], [3, 3]]

Рис. 10.7. «Разглаживание» коллекции с помощью метода flatMap

Вот как это будет выглядеть в коде:

Листинг 10.26. Повторяющиеся значения

```
let repeated = [2, 3].flatMap { (value: Int) -> [Int] in
    return [value, value]
}

print(repeated) // [2, 2, 3, 3]
```

Кроме того, посмотрите на приведенный ниже код, когда вы начинаете с вложенного массива значений. С помощью `flatMap` мы «разглаживаем» подмассивы, превращая их в один массив.

Листинг 10.27. Разглаживание вложенного массива

```
let stringsArr = [["I", "just"], ["want", "to"], ["learn", "about"],
    ["protocols"]]
```

```
let flattenedArray = stringsArr.flatMap { $0 } ❶
print(flattenedArray) // ["I", "just", "want", "to", "learn", "about",
   "protocols"]
```

❶ Возвращается только переданное значение; никакие преобразования не применяются.

Обратите внимание, что в этом случае мы не делаем ничего конкретного в замыкании `flatMap`.

10.6.1. flatMap и строки

Тип `String` – тоже коллекция. Как мы уже видели ранее, мы можем итерировать представление для `String`, такое как `unicodeScalars`. В зависимости от выбранного представления типа `String` мы также можем итерировать его кодовые единицы utf8 и utf16.

При итерации самого типа мы бы перебирали его символы. Поскольку тип `String` соответствует протоколу `Collection`, для него также можно использовать метод `flatMap`.

Например, можно создать лаконичный метод `interspersed`, который принимает элемент `Character` и разбрасывает или переплетает его между каждым символом строки.

Листинг 10.28. Метод interspersed

```
«Swift».interspersed(«-») // S-w-i-f-t

extension String { ❶
   func interspersed(_ element: Character) -> String {
      let characters = self.flatMap { (char: Character) -> [Character] in ❷
         return [char, element] ❸
      }.dropLast() ❹
      return String(characters) ❺
   }
}
```

❶ Расширяем `String`, добавляя функциональности.

❷ Вызываем `flatMap`.

❸ Замыкание возвращает массив, содержащий символ `String` и переданный элемент. Затем `flatMap` разглаживает массив.

❹ Нам не нужен элемент, добавленный к последнему символу. Можно удалить его с помощью метода `dropLast()`.

❺ Массив символов преобразуется в тип `String`.

Можно написать этот метод в сокращенном варианте, не указывая явные типы.

Листинг 10.29. Сокращенный вариант метода interspersed

```
extension String {
   func interspersed(_ element: Character) -> String {
      let characters = self.flatMap { return [$0, element] }.dropLast()
      return String(characters)
   }
}
```

10.6.2. Сочетание flatMap и map

Начав вложение `flatMap`, используя другие операции с `flatMap` или `map`, можно создать мощную функциональность с помощью нескольких строк кода. Представим, что нам нужно создать полную колоду карт. Можно сделать это с помощью нескольких строк, как только мы объединим методы `map` и `flatMap`.

Сначала определяем масти и достоинства. Затем мы итерируем масти и для каждой масти перебираем достоинства. Чтобы создать колоду, мы вкладываем `flatMap` и `map`, чтобы одновременно иметь доступ к масти и достоинству. Таким образом, можно легко создавать пары кортежей для каждой карты.

Как показано здесь, в конце мы используем метод `shuffle()`, который перетасовывает колоду (`deckOfCards`).

Листинг 10.30. Создание колоды

```
let suits = ["Hearts", "Clubs", "Diamonds", "Spades"] ❶
let faces = ["2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A"] ❶
var deckOfCards = suits.flatMap { suit in ❷
   faces.map { face in ❸
      (suit, face) ❹
   }
}
deckOfCards.shuffle() ❺

print(deckOfCards) // [("Diamonds", "5"), ("Hearts", "8"), ("Hearts", "K"),
➥ ("Clubs", "3"), ("Diamonds", "10"), ("Spades", "A"), ...
```

❶ Определяем возможные масти и достоинства.

❷ Итерируем масти с помощью метода `flatMap`, потому что «разглаживаем» массив.

❸ Применяем метод `map`.

❹ Поскольку мы выполняем вложение, и достоинство, и масть становятся доступными для создания нового типа (в этом случае кортежа).

❺ Перетасовываем карты – в противном случае игра становится скучной. Обратите внимание, что `deckOfCards` – это переменная, позволяющая тасовать колоду на месте.

Подсказка

Мы перемешиваем колоду (`deckOfCards`) на месте, но также можем получить свежую копию с помощью метода `shuffled()`.

Причина, по которой мы используем метод `flatMap` для массива suits, заключается в том, что при применении метода `map` (`faces.map`) возвращается массив, в результате чего мы получаем вложенные массивы. С помощью `flatMap` мы удаляем один слой вложенности так, чтобы в конечном итоге получить массив кортежей.

10.6.3. compactMap

Для работы с коллекциями, такими как `Array`, `String` или `Dictionary`, у `flatMap` есть младший брат – `compactMap`. Используя этот метод, вместо разглаживания вложенной коллекции можно «разглаживать» опционалы внутри коллекции.

Говоря проще, можно отфильтровывать значения nil из коллекции. Как вы уже видели, не каждый тип `String` можно превратить в `URL`, что приводит к появлению опционального типа `URL`, как в этом примере.

Листинг 10.31. Создание опциональных типов URL

```
let wrongUrl = URL(string: "OMG SHOES")
print(wrongUrl) // nil
let properUrl = URL(string: "https://www.swift.org")
print(properUrl) // Опционал (https://www.swift.org)
```

Если бы мы заполнили массив строками и попытались преобразовать их в типы `URL` с помощью метода `map`, мы бы получили массив опционалов `Int`, как показано здесь:

Листинг 10.32. Использование метода map с массивом

```
let strings = [
   "https://www.duckduckgo.com",
   "https://www.twitter.com",
   "OMG SHOES",
   "https://www.swift.org"
]
let optionalUrls = strings.map { URL(string: $0) }

print(optionalUrls) // [Опционал (https://www.duckduckgo.com),
➥ Опционал (https://www.twitter.com), nil, Опционал (https://www.swift.org)]
```

Обратите внимание, что `"OMG  SHOES"` нельзя превратить в `URL`, что приводит к значению nil внутри массива.

Опять же, с помощью `CompactMap` можно «разгладить» эту операцию, за исключением того, что на этот раз разглаживание массива будет означать удаление ну-

левых опционалов. Таким образом, как показано в этом листинге, мы получаем список URL-адресов, в котором нет опционалов.

Листинг 10.33. Разглаживание опционального массива

```
let urls = strings.compactMap(URL.init)
print(urls) // [https://www.duckduckgo.com, https://www.twitter.com,
➥ https://www.swift.org]
```

Поскольку не все строки могут быть преобразованы в типы URL, `URL.init` возвращает опционал URL. Затем `compactMap` отфильтровывает значения nil. Например, нельзя превратить `"OMG SHOES"` в URL, поэтому `compactMap` отфильтровывает это значение. В результате мы получаем правильные URL-ардеса, которые не являются опционалами.

Если вам больше по душе цикл *for*, можно использовать его для фильтрации опционального массива, а также с помощью выражения `let case`, как показано в листинге 10.34. Это выражение позволяет использовать сопоставление с образцом для кейса перечисления. К примеру, это могут быть опционалы, потому что опционалы – это перечисления. Например, можно получить массив с опциональными типами URL и итерировать их, отфильтровывая все значения nil, и в результате получить извлекаемое значение внутри цикла.

Листинг 10.34. Использование цикла for для фильтрации опционального массива

```
let optionalUrls: [URL?] = [
   URL(string: "https://www.duckduckgo.com"),
   URL(string: "Bananaphone"),
   URL(string: "https://www.twitter.com"),
   URL(string: "https://www.swift.org")
]

for case let url? in optionalUrls {
   print("The url is \(url)") // Здесь извлекается url.
}

// Вывод:
// url - https://www.duckduckgo.com
// url - https://www.twitter.com
// url - https://www.swift.org
```

Как упоминалось ранее, с помощью циклов *for* мы получаем преимущество использования операторов `break`, `continue` или `return`, чтобы разорвать цикл на полпути.

10.6.4. Вложенность или цепочки

Хитрость при использовании `flatMap` и `compactMap` заключается в том, что не имеет значения, будете вы вкладывать или объединять операции с `flatMap` или

compactMap. Например, можно использовать метод flatMap с Optional два раза подряд. Кроме того, можно вложить две операции с flatMap. В любом случае, результат будет тот же:

```
let value = Optional(40)
let lhs = value.flatMap(half).flatMap(half)
let rhs = value.flatMap { int in half(int).flatMap(half) }
lhs == rhs // Истинно.
print(rhs) // Опционал (10)
```

Еще одно преимущество использования вложенных flatMap или compactMap заключается в том, что можно обращаться к инкапсулированным значениям внутри вложенного замыкания.

Например, можно использовать flatMap и compactMap для одного и того же типа String. Вначале каждый элемент внутри строки привязан к a внутри операции с flatMap; затем мы снова применяем метод compactMap для той же строки, но привязываем каждый элемент к b. В итоге мы получаем способ объединить a и b для создания нового списка.

Используя эту технику, как показано ниже, вы создаете массив уникальных кортежей из символов в одной строке.

```
let string = "abc"
let results = string.flatMap { a -> [(Character, Character)] in ❶
   string.compactMap { b -> (Character, Character)? in ❷
      if a == b { ❸
         return nil ❹
      } else {
         return (a, b)
      }
   }
}
print(results) // [("a", "b"), ("a", "c"), ("b", "a"), ("b", "c"),
   ("c", "a"), ("c", "b")] ❺
```


❶ Используем метод flatMap.

❷ Вкладываем операцию с compactMap, снова для той же строки.

❸ Благодаря вложенности можно обращаться к константам a и b из внутренней операции с compactMap.

❹ Если a и b одинаковы, мы возвращаем nil, поэтому в итоге получаем только уникальные комбинации.

❺ Результатом является уникальная комбинация символов, отфильтрованных и «разглаженных» благодаря вложенным операциям.

Это маленькая хитрость, но знание того, что flatMap и compactMap можно объединить в цепочку или вложить, может помочь вам по-разному изменять код для получения похожих результатов.

10.6.5. Упражнения

6

Создайте функцию, которая превращает массив целых чисел в массив со значением, вычтенным и прибавленным для каждого целого числа, например [20, 30, 40] превращается в [19, 20, 21, 29, 30, 31, 39, 40, 41]. Попробуйте решить эту задачу с помощью `map` или `flatMap`.

7

Создайте значения от 0 до 100, используя только четные числа, пропуская все десятки без единиц, например 10, 20 и т. д. В конечном итоге у вас должно получиться [2, 4, 6, 8, 12, 14, 16, 18, 22...]. Можно ли решить эту задачу с помощью `map` или `flatMap`?

8

Создайте функцию, которая удаляет все гласные из строки. Опять же, попробуйте решить эту проблему с помощью `map` или `flatMap`.

9

Имеется массив кортежей. Создайте массив с кортежами всех возможных пар кортежей этих значений – например, [1, 2] превращается в [(1, 1), (1, 2), (2, 1), (2, 2)]. Опять же, попробуйте сделать это с помощью `map` и/или `flatMap` и убедитесь, что нет дубликатов.

10

Напишите функцию, которая дублирует каждое значение внутри массива – например, [1, 2, 3] превращается в [1, 1, 2, 2, 3, 3] и [["a", "b"], ["C", "d"]] превращается в [["a", "b"], ["a", "b"], ["c", "d"], ["c", "d" "]]. Попробуйте использовать для этого `map` или `flatMap`.

10.7. В заключение

В зависимости от вашего опыта функциональный стиль программирования может показаться немного чуждым. К счастью, для создания впечатляющих приложений не обязательно владеть функциональным программированием. Но, добавив методы `map` и `flatMap` в свой инструментарий, можно использовать их возможности и лаконично добавлять мощные неизменяемые абстракции в свой код.

Предпочитаете ли вы императивный или функциональный стиль программирования, я надеюсь, что вы будете чувствовать себя уверенно, выбирая подход, создающий тонкий баланс между удобочитаемостью, надежностью и скоростью.

Резюме

- Методы `map` и `flatMap` – это концепции, заимствованные из мира функционального программирования.
- Метод `map` – это абстракция, которая носит название *функтор*.

- Функтор обозначает контейнер – или контекст, – из которого можно преобразовать его значение с помощью метода `map`.
- Метод `map` используется со многими типами, включая `Array`, `Dictionary`, протоколы `Sequence` и `Collections`, а также `Optional`.
- Метод `map` является важным элементом при преобразовании данных внутри конвейера.
- Программирование в императивном стиле – прекрасная альтернатива функциональному стилю программирования.
- Программирование в императивном стиле может быть более производительным. Напротив, программирование в функциональном стиле может включать в себя неизменяемые преобразования и иногда может быть более удобочитаемым и демонстрировать более четкие намерения.
- Метод `flatMap` – это операция «разглаживания», которая следует после операции с map.
- С помощью `flatMap` можно «разгладить» вложенный опционал, превратив его в один.
- Используя `flatMap`, можно изменять последовательность операций для опционала неизменным образом.
- Если опционал равен nil, `map` и `flatMap` игнорируют любые цепочки.
- Если мы возвращаем nil в ходе операции с `flatMap`, то можно разорвать цепочки.
- Используя `flatMap`, можно преобразовывать массивы и последовательности мощным образом при наличии очень небольшого количества кода.
- С помощью `CompactMap` можно отфильтровывать значения nil из массивов и последовательностей опционалов.
- Также можно отфильтровывать значения nil, используя императивный стиль, с помощью цикла *for*.
- Можно вкладывать операции с `flatMap` и `compactMap`, когда речь идет об одинаковых результатах.
- В коллекциях и последовательностях можно комбинировать `flatMap` и `map`, чтобы объединить все их значения.

Ответы

1

Создайте функцию, которая превращает массив во вложенный массив. Убедитесь, что вы используете метод map.

```
func makeSubArrays<T>(_ arr: [T]) -> [[T]] {
    return arr.map { [$0] }
}

makeSubArrays(["a", "b", "c"]) // [[a], [b], [c]]
makeSubArrays([100, 50, 1]) // [[100], [50], [1]]
```

2

Создайте функцию, которая преобразует значения в словаре фильмов. Каждую оценку от 1 до 5 необходимо преобразовать в удобочитаемый формат.

```
// Например
// Рейтинг 1.2 - "Очень плохо", 3 - "Средне", 4.5 - "Превосходно".
func transformRating<T>(_ dict: [T: Float]) -> [T: String] {
   return dict.mapValues { (rating) -> String in
      switch rating {
         case ..<1: return "Very weak"
         case ..<2: return "Weak"
         case ..<3: return "Average"
         case ..<4: return "Good"
         case ..<5: return "Excellent"
         default: fatalError("Unknown rating")
      }
   }
}

let moviesAndRatings: [String : Float] = ["Home Alone 4" : 1.2,
➥ "Who framed Roger Rabbit?" : 4.6, "Star Wars: The Phantom Menace"
➥ : 2.2, "The Shawshank Redemption" : 4.9]
let moviesHumanRadable = transformRating(moviesAndRatings)
```

3

Все еще просматривая фильмы и рейтинги, преобразуйте словарь в описание для каждого фильма с добавлением рейтинга к заголовку. Например:

```
let movies: [String : Float] = ["Home Alone 4" : 1.2, "Who framed Roger
➥ Rabbit?" : 4.6, "Star Wars: The Phantom Menace" : 2.2, "The Shawshank
➥ Redemption" : 4.9]

func convertRating(_ rating: Float) -> String {
   switch rating {
      case ..<1: return "Very weak"
      case ..<2: return "Weak"
      case ..<3: return "Average"
      case ..<4: return "Good"
      case ..<5: return "Excellent"
      default: fatalError("Unknown rating")
   }
}

let movieDescriptions = movies.map { (tuple) in
```

```
return "\(tuple.key) (\(convertRating(tuple.value)))"
}
print(movieDescriptions) // ["Home Alone 4 (Weak)", "Star Wars: The
➥ Phantom Menace (Average)", "Who framed Roger Rabbit? (Excellent)",
➥ "The Shawshank Redemption (Excellent)"]
```

4

Создайте массив из букв «a», «b» и «c» 10 раз. Конечным результатом должно быть ["a", "b", "c", "a", "b", "c", "a" ...], пока массив не будет состоять из 30 элементов. Попробуйте использовать метод map, если это возможно.

```
let values = (0..<30).map { (int: Int) -> String in
   switch int % 3 {
      case 0: return "a"
      case 1: return "b"
      case 2: return "c"
      default: fatalError("Not allowed to come here")
   }
}
print(values)
```

5

Имеется словарь контактных данных. Приведенный ниже код получает улицу и город из данных и очищает строки. Посмотрите, можно ли избавиться от шаблонного кода (и обязательно используйте метод map).

```
let someStreet = contact["address"]?["street"].map(capitalizedAndTrimmed)
let someCity = contact["address"]?["city"].map(capitalizedAndTrimmed)
```

6

Создайте функцию, которая превращает массив целых чисел в массив со значением, вычтенным и прибавленным для каждого целого числа, например [20, 30, 40] превращается в [19, 20, 21, 29, 30, 31, 39, 40, 41]. Попробуйте решить эту задачу с помощью map или flatMap.

```
func buildList(_ values: [Int]) -> [Int] {
   return values.flatMap {
      [$0 - 1, $0, $0 + 1]
   }
}
```

7

Создайте значения от 0 до 100, используя только четные числа, пропуская все десятки без единиц, например 10, 20 и т. д. В конечном итоге у вас должно получиться [2, 4, 6, 8, 12, 14, 16, 18, 22...]. Можно ли решить эту задачу с помощью `map` или `flatMap`?

```
let strideSequence = stride(from: 0, through: 30, by: 2).flatMap { int in
    return int % 10 == 0 ? nil : int
}
```

8

Создайте функцию, которая удаляет все гласные из строки. Опять же, попробуйте решить эту проблему с помощью `map` или `flatMap`.

```
func removeVowels(_ string: String) -> String {
    let characters = string.flatMap { char -> Character? in
        switch char {
            case "e", "u", "i", "o", "a": return nil
            default: return char
        }
    }
    return String(characters)
}
removeVowels("Hi there!") // H thr!
```

9

Имеется массив кортежей. Создайте массив с кортежами всех возможных пар кортежей этих значений, например [1, 2] превращается в [(1, 1), (1, 2), (2, 1), (2, 2)]. Опять же, попробуйте сделать это с помощью `map` и/или `flatMap` и убедитесь, что нет дубликатов.

```
func pairValues(_ values: [Int]) -> [(Int, Int)] {
    return values.flatMap { lhs in
        values.map { rhs -> (Int, Int) in
            return (lhs, rhs)
        }
    }
}
```

10

Напишите функцию, которая дублирует каждое значение внутри массива, например [1, 2, 3] превращается в [1, 1, 2, 2, 3, 3] и [["a", "b"], ["C", "d"]], превращается в [["a", "b"], ["a", "b"], ["c", "d"], ["c", "d"]]. Попробуйте использовать для этого `map` или `flatMap`.

```
func double<T>(_ values: [T]) -> [T] {
return values.flatMap { [$0, $0] }
}

print(double([1,2,3]))
print(double([["a", "b"], ["c", "d"]]))
```

Глава 11. Асинхронная обработка ошибок с помощью типа Result

В этой главе:

- изучение проблем с обработкой ошибок в стиле Cocoa;
- знакомство с типом `Result`;
- как `Result` обеспечивает безопасность на этапе компиляции;
- предотвращение ошибок, связанных с забытыми обратными вызовами;
- устойчивое преобразование данных с помощью `map`, `mapError` и `flatMap`;
- использование подхода «счастливый путь» при обработке ошибок;
- смешивание функций, генерирующих ошибку, и типа `Result`;
- как `AnyError` делает `Result` менее ограничительным;
- как выразить намерение с помощью типа `Never`.

Вы рассмотрели множество механизмов обработки ошибок в Swift и, возможно, заметили в главе 6, что мы генерировали ошибки синхронно. В этой главе основное внимание уделяется обработке ошибок из асинхронных процессов, что, к сожалению, представляет собой совершенно иную идиому в Swift.

Асинхронными действиями может быть код, работающий в фоновом режиме, пока выполняется текущий метод. Например, можно выполнить асинхронный API-вызов для получения данных в формате JSON с сервера. Когда вызов завершается, он запускает функцию обратного вызова, давая вам данные или ошибку.

Swift пока не предлагает официального решения для асинхронной обработки ошибок. По слухам, Swift не будет делать этого до тех пор, пока не будет внедрен шаблон async/await где-то в версии 7 или 8. К счастью, сообщество, похоже, предпочитает обработку асинхронных ошибок с помощью типа `Result` (что подкрепляется включением Apple неофициального типа `Result` в диспетчер пакетов Swift). Возможно, вы уже работали с ним и даже реализовывали его в проектах. В этой главе мы будем использовать один из типов, предложенных компанией Apple, который, возможно, немного более расширенный, по сравнению с большинством примеров, которые можно найти в интернете. Чтобы получить максимальную отдачу от работы с `Result`, мы проберемся поглубже и посмотрим, что такое распространение, т. н. монадическую обработку ошибок и связанный с ней тип `AnyError`. Тип `Result` – это перечисление, подобное `Optional`, но с некоторыми отличиями. Поэтому если вы комфортно себя чувствуете при работе с `Optional`, `Result` не должен быть слишком трудным.

Мы начнем с изучения преимуществ, которые предоставляет тип `Result`, и того, как можно добавить его в свои проекты. Мы создадим сетевой API, а за-

тем будем дальше улучшать его в последующих разделах. После этого мы заново напишем API, но будем использовать тип Result, чтобы воспользоваться его преимуществами.

Далее мы увидим, как распространять асинхронные ошибки и поддерживать чистоту своего кода, ориентируясь на подход «счастливый путь», с помощью `map`, `mapError` и `flatMap`.

Рано или поздно мы снова воспользуемся обычными функциями, генерирующими ошибку, для преобразования своих асинхронных данных. Вы увидите, как смешать две идиомы обработки ошибок, работая с такими функциями в сочетании с `Result`.

После создания надежного API мы познакомимся с уникальным типом `AnyError`, который компания Apple предлагает в сочетании с `Result`. Этот тип дает возможность хранить несколько типов ошибок внутри `Result`. Преимущество заключается в том, что можно ослабить строгость обработки ошибок без оглядки на Objective-C с помощью `NSError`. Мы опробуем множество удобных функций, чтобы сделать код лаконичным.

После этого мы познакомимся с типом `Never`, чтобы указать, что ваш код не сможет завершиться неудачно или успешно. Звучит немного теоретически, но прекрасно подходит для завершения главы. Считайте, что это бонус.

К концу главы вы будете чувствовать себя комфортно, применяя мощные преобразования к своему асинхронному коду и справляясь с любыми ошибками, которые могут возникнуть. Вы также сможете избежать страшной пирамиды гибели и сосредоточиться на подходе «счастливый путь». Но существенным преимуществом является то, что ваш код будет безопасным и лаконичным с изящной обработкой ошибок. Так давайте приступим!

11.1. Зачем использовать тип Result?

Присоединяйтесь!

Будет более познавательно и веселее, если вы будете знакомиться с кодом по мере прохождения этой главы. Исходный код можно скачать по адресу `http://mng.bz/5YP1`.

Механизм обработки ошибок в Swift не очень хорошо подходит для асинхронной обработки. На момент написания этих строк асинхронная обработка ошибок Swift до сих пор не реализована. Как правило, разработчики используют стиль обработки ошибок, предлагаемый Cocoa, – как в старые добрые времена Objective-C, когда сетевой вызов возвращает несколько значений. Например, можно получить данные в формате JSON из API, а функция обратного вызова даст вам и значение, и ошибку, при этом вам придется выполнить проверку на предмет наличия nil и там, и там.

К сожалению, у этого метода есть проблемы – о которых вы скоро узнаете, – и тип `Result` решает их. Тип `Result`, основанный на типе `Result` из языка Rust

и типе Either из языков Haskell и Scala, представляет собой идиому функционального программирования, принятую сообществом Swift, что делает его неофициальным стандартом обработки ошибок.

На момент написания этих строк разработчики неоднократно переосмысливали тип Result, поскольку официального стандарта пока еще не существует. Хотя Swift официально не использует тип Result, менеджер пакетов Swift предлагает его неофициально. Таким образом, компания Apple (косвенно) использует тип Result, который оправдывает свою реализацию в ваших кодовых базах. Мы будем работать с Result, используя полезные пользовательские функции.

11.1.1. Как раздобыть Result

Данный тип можно найти в playground-проекте этой главы, особом файле с расширением playground. Но его также можно получить напрямую из диспетчера пакетов Swift, еще известного как SwiftPM, на GitHub по адресу http://mng.bz/6GPD или с помощью зависимостей SwiftPM. Эта глава не содержит полного руководства о том, как создавать инструмент командной строки Swift через SwiftPM, но приведенные ниже команды должны помочь вам.

Для начала, чтобы настроить папку и исполняемый проект Swift, откройте командную строку и введите следующее:

```
mkdir ResultFun
cd ResultFun
swift package init --type executable
```

Затем откройте файл Package.swift и измените его, написав это:

```
// swift-tools-version:4.2
// Эта цифра указывает на минимальную версию Swift, необходимую
// для сборки данного пакета.

import PackageDescription

let package = Package(
   name: "ResultFun",
   dependencies: [
      .package(url: "https://github.com/apple/swift-package-manager",
      from: "0.2.1") ❶
   ],
   targets: [
      .target(
         name: "ResultFun",
         dependencies: ["Utility"]), ❷
   ]
)
```

❶ Мы ссылаемся на проект SwiftPM из самого SwiftPM.

❷ Нужны зависимости от пакета Utility, чтобы получить необходимые исходные файлы.

В папке своего проекта откройте файл `Sources/ResultFun/main.swift` и измените его следующим образом:

```
import Basic ❶

let result = Result<String, AnyError>("It's working, hooray!") ❷ ❸
print(result)
```

❶ SwiftPM предлагает пакет Basic.

❷ Мы будем рассматривать тип `AnyError` позже в этой главе.

❸ Создаем тип `Result`, чтобы убедиться, что импорт сработал правильно.

Наберите `swift run`, и увидите это: `Result(It's working, hooray!)`. Готовы? Давайте продолжим.

11.1.2. Result похож на Optional, но с изюминкой

`Result` *во многом* похож на `Optional`, и это здорово, потому что если вам комфортно с опционалами (см. главу 4), вы будете чувствовать себя как дома, имея дело с типом `Result`.

Тип `Result` представляет собой перечисление с двумя кейсами: `success` и `failure`. Но пусть это не вводит вас в заблуждение. `Optional` – это тоже «просто» перечисление с двумя кейсами, такой же мощный, как и `Result`.

В самом простом варианте тип `Result` выглядит так:

Листинг 11.1. Тип Result

```
public enum Result<Value, ErrorType: Swift.Error> { ❶
   /// Указывает на success со значением в ассоциированном объекте.
   case success(Value) ❷
   /// Указывает на failure с ошибкой внутри ассоциированного объекта.
   case failure(ErrorType) ❸
   // ... Остальное оставлено на потом
}
```

❶ Тип `Result` требует двух обобщенных значений.

❷ В случае успеха привязывается значение (`Value`).

❸ Обобщение `ErrorType` привязывается в случае сбоя.

Отличие от `Optional` заключается в том, что вместо присутствующего значения (кейс `some`) или `nil` (кейс `none`) `Result` сообщает, что у него либо есть значение (кейс `success`), либо есть ошибка (кейс `failure`). По сути, тип `Result` указывает на возможный сбой вместо `nil`. Другими словами, с помощью этого типа можно ука-

зать контекст, по которому операция завершилась неудачей, вместо того чтобы пропустить значение.

`Result` содержит значение для каждого кейса, в то время как в случае с `Optional` значение есть только у кейса some. Кроме того, обобщение `ErrorType` ограничено протоколом `Error`, а это значит, что для кейса `failure` подходят лишь типы `Error`. Это ограничение пригодится для некоторых вспомогательных функций, о которых вы узнаете в следующем разделе. Обратите внимание, что кейс `success` подходит для любого типа, потому что для него нет ограничений.

Мы еще не видели полный вариант типа `Result`, в котором содержится множество методов, но этого кода достаточно, чтобы начать работу. Достаточно скоро мы познакомимся с другими методами, такими как преобразование из и в генерирующие функции и изменение значений и ошибок в `Result` неизменяемым образом.

Давайте быстро перейдем к сути типа `Result`: обработке ошибок.

11.1.3. Преимущества Result

Чтобы лучше понять преимущества типа `Result` в асинхронных вызовах, сначала посмотрим на недостатки асинхронных API в стиле Cocoa Touch, прежде чем увидим, как улучшить ситуацию с помощью типа `Result`. На протяжении всей главы мы будем вносить улучшения в этот API.

Давайте посмотрим на класс `URLSession` внутри фреймворка Foundation. Мы будем использовать его для выполнения сетевого вызова, как показано в листинге 11.2, и нас интересуют данные и ошибка ответа. У iTunes App store нет «популярного» настольного приложения, поэтому мы создадим API для осуществления поиска в iTunes Store без настольного приложения.

Для начала мы используем жестко закодированную строку для поиска «железного человека» (iron man) – будем использовать ручное процентное кодирование и функцию `callURL` для выполнения сетевого вызова.

Листинг 11.2. Выполнение сетевого вызова

```
func callURL(with url: URL, completionHandler: @escaping (Data?, Error?) ❶
➥ -> Void) { ❷
   let task = URLSession.shared.dataTask(with: url, completionHandler:
   ➥ { (data, response, error) -> Void in
      completionHandler(data, error) ❸
   })
   task.resume()
}

let url = URL(string: «https://itunes.apple.com/search?term=iron%20man»)!

callURL(with: url) { (data, error) in ❹
   if let error = error { ❺
```

```
        print(error)
    } else if let data = data { ❻
        let value = String(data: data, encoding: .utf8) ❼
        print(value)
    } else {
        // Что тут? ❽
    }
}
```

❶ В этой ситуации требуется ключевое слово `@escaping`; оно указывает на то, что замыкание `completeHandler` может быть потенциально сохранено и сохранить память.

❷ У функции `callURL` есть обработчик `completionHandler`, который вызывается, когда `URLSession.dataTask` завершает работу.

❸ Мы получаем данные из URL-адреса и передаем `data` и `error` вызывающей программе.

❹ Мы вызываем функцию `callURL`, чтобы получить `data` и `error`, которые возвращаются в определенный момент времени (асинхронно).

❺ Как только вызывается функция обратного вызова, извлекается любая ошибка.

❻ При наличии `data` можно работать с `response`.

❼ Мы превращаем данные в тип `String`, чтобы прочитать необработанное значение.

❽ Вот в чем проблема: если и `error`, и `data` равны nil, что тогда делать?

Но проблема состоит в том, что нам нужно проверить, равны ли `error` и/или `data` nil. Кроме того, что произойдет, если оба значения равны nil? В документации по классу `URLSession` (http://mng.bz/oVxr) сказано, что у `data` или `error` есть значение; однако в коде это не отражено, и нам все равно нужно проверять все значения.

При возврате нескольких значений из асинхронного вызова из `URLSession` значения success и failure не являются взаимоисключающими. Теоретически можно было бы получить и данные ответа, и ошибку с ошибкой, либо ни то, ни другое. Или у вас может быть одно из двух, но вы можете ошибочно предположить, что если нет ошибки, вызов должен быть успешным. В любом случае, у нас нет гарантии на этапе компиляции, чтобы обеспечить безопасную обработку возвращаемых данных. Но мы изменим это и увидим, что тип `Result` даст нам эти гарантии.

11.1.4. Создание API с использованием типа Result

Вернемся к API-вызову. С помощью типа `Result` можно принудительно установить на этапе компиляции, что ответ является либо успешным (есть значение), либо неудачным (есть ошибка). В качестве примера давайте обновим асинхронный вызов, чтобы он передавал `Result`.

Мы воспользуемся перечислением NetworkError и заставим функцию callURL использовать тип Result.

Листинг 11.3. Ответ с Result

```
enum NetworkError: Error {
case fetchFailed(Error) ❶
}

func callURL(with url: URL, completionHandler: @escaping (Result<Data,
➥ NetworkError>) -> Void) { ❷
   let task = URLSession.shared.dataTask(with: url, completionHandler: {
   ➥ (data, response, error) -> Void in
      // ... детали будут заполнены в ближайшее время
   })
   task.resume()
}

let url = URL(string: «https://itunes.apple.com/search?term=iron%20man»)!

callURL(with: url) { (result: Result<Data, NetworkError>) in ❸
   switch result {
      case .success(let data): ❹
         let value = String(data: data, encoding: .utf8)
         print(value)
      case .failure(let error): ❺
         print(error)
   }
}
```

❶ Определяем пользовательскую ошибку для передачи внутри Result. Можно сохранить низкоуровневую ошибку из URLSession внутри кейса fetchFailed, чтобы справиться с устранением неполадок.

❷ На этот раз функция callURL передает тип Result, содержащий либо Data, либо NetworkError.

❸ Вызываем функцию callURL, чтобы получить Result через замыкание.

❹ Выполняем сопоставление с образцом, чтобы получить значение из Result.

❺ Выполняем сопоставление с образцом, чтобы перехватить любую ошибку.

Как вы убедились, мы получаем тип Result<Data, NetworkError> при вызове callURL(). Но на этот раз вместо сопоставления для error и data значения теперь взаимоисключающие. Если нам нужно значение из Result, мы *должны* обработать оба кейса, взамен обеспечивая себе безопасность на этапе компиляции и устраняя любые неловкие ситуации, когда data и error могут быть равны nil или заполнены одновременно. Кроме того, большим преимуществом является то, что вы заранее знаете, что ошибка в кейсе failure имеет тип NetworkError, в от-

личие от функций, генерирующих ошибку, когда нам известен только тип ошибки во время выполнения.

Мы также можем использовать систему обработки ошибок, в которой тип данных содержит замыкание onSuccess или onFailure. Но я хочу подчеркнуть, что, используя Result, если вам нужно значение, вы *должны* что-то сделать с ошибкой.

Как избежать обработки ошибок

Конечно, нельзя полностью принудительно обработать ошибку внутри Result, если мы выполним сопоставление для единственного кейса перечисления с помощью оператора if case let. Кроме того, можно игнорировать ошибку с помощью печально известного комментария //TODO handle error, но тогда получится, что мы сознательно пытаемся избежать обработки ошибки. Как правило, если нам нужно получить значение из Result, компилятор также попросит нас обработать ошибку.

В качестве другого варианта, если причина сбоя нас не интересует, но мы все же хотим получить значение из Result, можно сделать это с помощью метода dematerialize. Эта функция либо возвращает значение, либо выдает ошибку в Result. При использовании ключевого слова try?, как показано в приведенном ниже листинге, можно мгновенно преобразовать Result в Optional.

Листинг 11.4. Использование метода dematerialize

```
let value: Data? = try? result.dematerialize()
```

11.1.5. Из Cocoa Touch в Result

Идем дальше. Ответ от dataTask URLSession возвращает три значения: data, response и error.

Листинг 11.5. Ответ URLSession

```
URLSession.shared.dataTask(with: url, completionHandler: { (data, response,
   error) -> Void in ... }
```

Но если мы хотим работать с Result, нам придется преобразовать значения из обработчика URLSession в Result самостоятельно. Давайте воспользуемся этой возможностью, чтобы конкретизировать функцию callURL и преобразовать обработку ошибок в стиле Cocoa Touch с обработкой ошибок в стиле Result.

Одним из способов преобразования значения и ошибки в Result является добавление пользовательского инициализатора, который выполняет преобразование за нас, как показано в приведенном ниже листинге. Можно передать этому инициализатору данные и ошибку, а затем использовать их для создания нового типа Result. В функции callURL можно потом вернуть Result через замыкание.

Листинг 11.6. Преобразование ответа и ошибки в результат

```
public enum Result<Value, ErrorType> {
   // ... Пропускаем часть кода
```

```
    init(value: Value?, error: ErrorType?) { ❶
       if let error = error {
          self = .failure(error)
       } else if let value = value {
          self = .success(value)
       } else {
          fatalError("Could not create Result") ❷
       }
    }
}

func callURL(with url: URL, completionHandler: @escaping (Result<Data,
➥ NetworkError>) -> Void) {
    let task = URLSession.shared.dataTask(with: url, completionHandler:
    ➥ { (data, response, error) -> Void in

    let dataTaskError = error.map { NetworkError.fetchFailed($0)} ❸

    let result = Result<Data, NetworkError>(value: data, error:
    ➥ dataTaskError) ❹

    completionHandler(result) ❺
})

task.resume()
}
```

❶ Создаем инициализатор, который принимает опциональные значения `value` и `error`.

❷ Если `value` и `error` равны нулю, дело плохо, и это приводит к сбою, потому что мы можем быть уверены, что `URLSession` возвращает либо `value`, либо `error`.

❸ Превращаем текущую ошибку в ошибку `NetworkError` более высокого уровня и передаем ошибку более низкого уровня из `URLSession` в кейс `fetchFailed`, чтобы помочь с устранением неполадок.

❹ Создаем `Result` из значений `data` и `error`.

❺ Передайте `Result` обратно в замыкание `completionHandler`.

Если API не возвращает значение

Не все API возвращают значение, но мы по-прежнему можем использовать `Result` с так называемым *модульным типом*, обозначенным как `Void` или `()`. Можно использовать `Void` или `()` в качестве значения для `Result`, например `Result<(), MyError>`.

11.2. Распространение Result

Давайте сделаем свой API уровнем чуть выше, чтобы вместо создания URL-адресов вручную можно было искать товары в iTunes Store, передавая строки. Помимо этого, вместо того чтобы иметь дело с ошибками нижнего уровня, давайте поработаем с высокоуровневой ошибкой `SearchResultError`, которая лучше соответствует новой абстракции поиска, которую мы создаем. Данный раздел – хорошая возможность увидеть, как распространять и преобразовывать любые типы `Result`.

API, который мы создадим, позволит вводить поисковый запрос, а назад мы будем получать результаты в формате JSON в виде `[String: Any]`.

Листинг 11.7. Вызов API поиска

```
enum SearchResultError: Error {
   case invalidTerm(String) ❶
   case underlyingError(NetworkError) ❷
   case invalidData ❸
}

search(term: «Iron man») { result: Result<[String: Any], SearchResultError> in ❹
   print(result)
}
```

❶ Кейс `invalidTerm` используется, когда нельзя создать URL.

❷ В кейсе `underlyingError` находится низкоуровневая ошибка `NetworkError` для устранения неполадок или для восстановления после ошибки.

❸ Кейс `invalidData` предназначен для того случая, когда необработанные данные нельзя преобразовать в формат JSON.

❹ Выполняем поиск по термину и получаем `Result` через замыкание.

11.2.1. Создание псевдонимов типов

Перед созданием реализации `search` для удобства мы создадим несколько псевдонимов типов, которые пригодятся нам при многократной работе с одним и тем же типом `Result`.

Например, если вы работаете со множеством функций, которые возвращают `Result<Value, SearchResultError>`, можно определить псевдоним типа для `Result`, содержащего `SearchResultError`. Этот псевдоним должен убедиться, что Result требует только одно обобщение вместо двух.

Листинг 11.8. Создание псевдонима типов

```
typealias SearchResult<Value> = Result<Value, SearchResultError> ❶

let searchResult = SearchResult("Tony Stark") ❷
print(searchResult) // success("Tony Stark")
```

❶ Определяем обобщенный псевдоним типа, который прикрепляет обобщение Error к SearchResultError.

❶ Result предлагает удобный инициализатор, чтобы создать Result из значения, если он может вывести ошибку.

Частичные псевдонимы типа

У псевдонима типа все еще есть обобщение Value для Result, а это означает, что SearchResult прикреплен в SearchResultError, но его значение может быть любым, например [String: Any], Int и т. д.

Можно создать SearchResult, только передав ему значение. Но его истинный тип – Result<Value, SearchResultError>.

Еще один псевдоним типов, который мы можем ввести, относится к типу JSON, а именно к словарю типа [String: Any]. Этот псевдоним помогает сделать наш код более читабельным, т. е. мы будем работать с SearchResult<JSON> вместо громоздкого типа SearchResult<[String: Any]>.

Листинг 11.9. JSON

```
typealias JSON = [String: Any]
```

Теперь, когда у нас есть два этих типа, мы будем работать с типом SearchResult <JSON>.

11.2.2. Функция search

Новая функция search использует функцию callURL и выполняет две дополнительные задачи: она преобразует данные в формат JSON и низкоуровневую NetworkError в SearchResultError, что делает функцию более высокоуровневой для использования, как показано в приведенном ниже листинге.

Листинг 11.10. Реализация функции search

```
func search(term: String, completionHandler: @escaping (SearchResult<JSON>)
    -> Void) { ❶
  let encodedString = term.addingPercentEncoding(withAllowedCharacters:
    .urlHostAllowed) ❷
  let path = encodedString.map { "https://itunes.apple.com/search?term="
    + $0 } ❸

  guard let url = path.flatMap(URL.init) else { ❹
    completionHandler(SearchResult(.invalidTerm(term))) ❺
    return
  }

  callURL(with: url) { result in ❻
    switch result {
```

```
        case .success(let data): ❼
            if
                let json = try? JSONSerialization.jsonObject(with: data,
                    options: []),
                let jsonDictionary = json as? JSON {
                let result = SearchResult<JSON>(jsonDictionary)
                completionHandler(result) ❽
            } else {
                let result = SearchResult<JSON>(.invalidData)
                completionHandler(result) ❾
            }
        case .failure(let error):
            let result = SearchResult<JSON>(.underlyingError(error)) ❿
            completionHandler(result)
        }
    }
}
```

❶ Функция использует JSON и псевдонимы типов `SearchResult`.

❷ Функция преобразует термин (term) в формат URL-кодировки. Обратите внимание, что `encodedString` – это опционал.

❸ Добавляем закодированную строку в путь API iTunes. Мы используем метод map, чтобы отложить извлечение.

❹ Преобразуем полный путь в URL с помощью метода `flatMap`. Оператор `guard` выполняет действие по извлечению.

❺ Убеждаемся, что URL создан; в случае неудачи мы вызываем замыкание раньше.

❻ Вызываем исходную функцию `callURL` для получения необработанных данных.

❼ В случае успеха функция пытается преобразовать данные в формат JSON `[String: Any]`.

❽ Если данные успешно преобразуются в формат JSON, можно передать их обработчику завершения (`completionHandler`).

❾ Если не удается преобразовать данные в формат JSON, мы передаем `SearchResultError`, обернутую в `Result`. Можно опустить SearchResultError, потому что Swift самостоятельно может определить тип ошибки.

❿ В случае сбоя `callURL` мы преобразуем низкоуровневую `NetworkError` в высокоуровневую `SearchResultError`, передавая исходную `NetworkError` в `SearchResultError` для устранения неполадок.

Благодаря функции `search` мы получаем функцию высшего уровня для поиска в iTunes API. Однако она все еще выглядит несколько неуклюже, потому что

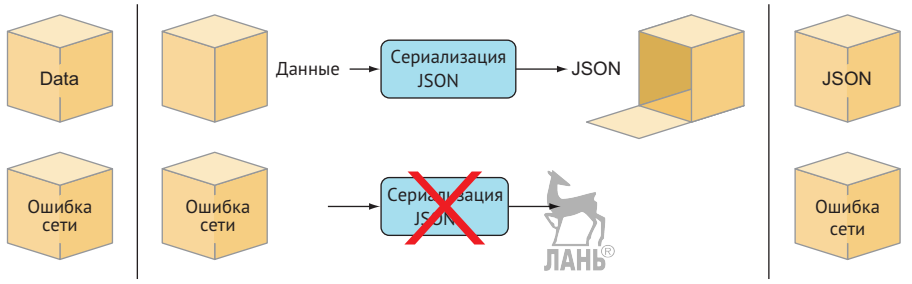
мы вручную создаем несколько типов Result и вызываем обработчик ошибок в нескольких местах. Это шаблонный код, и можно забыть вызвать обработчик в более крупных функциях. Давайте избавимся от этого с помощью методов map, mapError и flatMap, чтобы иметь возможность преобразовывать и распространять один тип Result, а обработчика завершения нужно будет вызвать только один раз.

11.3. Преобразование значений внутри Result

Подобно тому, как можно создавать опционалы через приложение и применять к ним метод map (чтобы отложить извлечение), также можно создавать Result с помощью своих функций и методов, ориентируясь на подход «счастливый путь». По сути, после получения Result можно передавать его, преобразовывать и использовать оператор switch, только когда вам нужно извлечь его значение или обработать ошибку.

Один из способов преобразовать его – использовать метод map, аналогично тому, как это делается в случае с типом Optional. Помните, как можно преобразовать его внутреннее значение с помощью этого метода (в случае если оно есть)? То же и с типом Result: мы преобразуем его значение success, если оно есть. Используя метод map, в этом случае мы превратим Result<Data, NetworkError> в Result<JSON, NetworkError>.

Подобно тому, как этот метод игнорирует значения nil в опционалах, он так же игнорирует ошибки в Result (см. рис. 11.1).



1. У нас есть типы Result: один со значением и один с ошибкой

2. Используя метод map, мы применяем функцию к значению в Result

map ничего не делает в случае неудачи

3. map снова оборачивает преобразованное значение в Result

В случае неудачи тип Result остается тем же

Рис. 11.1. Методы map и Result

В качестве специального дополнения также можно использовать метод map для ошибки вместо значения внутри Result. Наличие метода mapError удобно, потому что мы преобразуем NetworkError внутри Result в SearchResultError.

Таким образом, с помощью mapError мы превращаем Result<JSON, NetworkError> в Result<JSON, SearchResultError>, который соответствует типу, который мы передаем обработчику завершения (см. рис. 11.2).

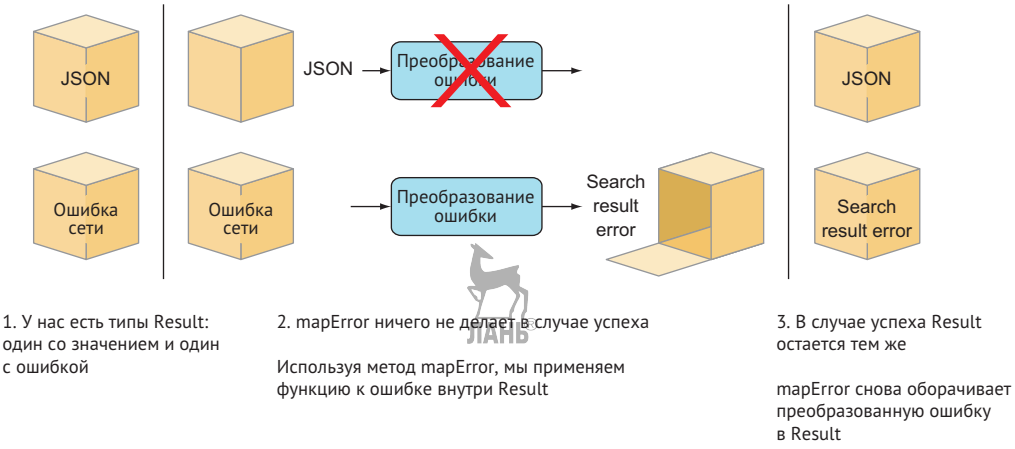


Рис. 11.2. Использование метода mapError() внутри типа Result

Объединив возможности map и mapError, можно превратить Result<Data, NetworkError> в Result<JSON, SearchResultError>, он же SearchResult<JSON>, без необходимости использовать оператор switch только один раз (см. рис. 11.3). В листинге 11.11 приведен пример применения метода map для ошибки и значения.

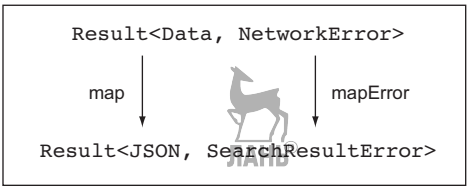


Рис. 11.3. Использование методов map и mapError для значения и ошибки

Используя методы mapError и map, можно удалить шаблонный код из функции search, о которой шла речь ранее.

Листинг 11.11. Использование метода map

```
func search(term: String, completionHandler: @escaping (SearchResult<JSON>)
➥ -> Void) {
   // ... Пропускаем часть кода.

   callURL(with: url) { result in
      let convertedResult: SearchResult<JSON> =
         result ❶
            // Преобразование данных в JSON
            .map { (data: Data) -> JSON in ❷
               guard
                  let json = try? JSONSerialization.jsonObject(with:
                  ➥ data, options: []),
                  let jsonDictionary = json as? JSON else {
```

```
                    return [:] ❸
                }
            return jsonDictionary
        }
        // Преобразование NetworkError в SearchResultError
        .mapError { (networkError: NetworkError) ->
        ➥ SearchResultError in ❹
        return SearchResultError.underlyingError(networkError)
        ➥ // Обработка ошибки из нижнего уровня
    }
    completionHandler(convertedResult) ❺
  }
}
```

❶ Это тип `Result<Data, NetworkError>`.

❷ В случае успеха мы используем метод map для преобразования данных в формат JSON.

❸ В случае неудачи мы получаем пустой JSON вместо ошибки, которую можно решить с помощью `flatMap`.

❹ Используем метод map, чтобы тип `Error` совпадал с `SearchResultError`.

❺ Мы передаем тип `SearchResult<JSON>` обработчику после того, как все будет сделано.

Теперь, вместо того чтобы вручную извлекать типы `Result` и передавать их обработчику в несколько потоков, мы преобразуем `Result` в `SearchResult` и передаем его обработчику только один раз. Как и в случае с опционалами, мы откладываем обработку ошибок до тех пор, пока не захотим получить значение.

К сожалению, `mapError` не является частью типа `Result`, предлагаемого компанией Apple. Нужно определять метод самостоятельно (см. предстоящее упражнение), но также можно заглянуть в соответствующий файл с расширением playground.

В качестве следующего шага давайте исправим ошибку, потому что в настоящее время мы возвращаем пустой словарь, вместо того чтобы выдавать ошибку. Используем для этого метод `flatMap`.

11.3.1. Упражнение

1

Посмотрите на функцию map в Result и скажите, можно ли создать mapError.

11.3.2. Использование метода flatMap для типа Result

Одним из недостатков функции `search` является то, что когда данные нельзя преобразовать в формат JSON, необходимо получить ошибку. Ее можно сгенериро-

вать, но это будет выглядеть неуклюже, потому что в этом случае нам бы пришлось смешивать идиому выдачи ошибки в Swift с идиомой `Result`. Мы увидим это в следующем разделе.

Чтобы придерживаться концепции `Result`, давайте вернем еще один тип `Result` из метода `map`. Но вы, возможно, догадались, что в этом случае у нас будет вложенный `Result`, такой как этот, например `SearchResult<SearchResult<JSON >>`. Можно использовать метод `flatMap`, определенный в `Result`, чтобы избавиться от одного дополнительного уровня вложенности.

Точно так же, как можно использовать `flatMap` для превращения `Optional <Optional<JSON>>` в `Optional<JSON>`, можно превратить `SearchResult <Search Result<JSON >>` в `SearchResult<JSON>` (см. рис. 11.4).

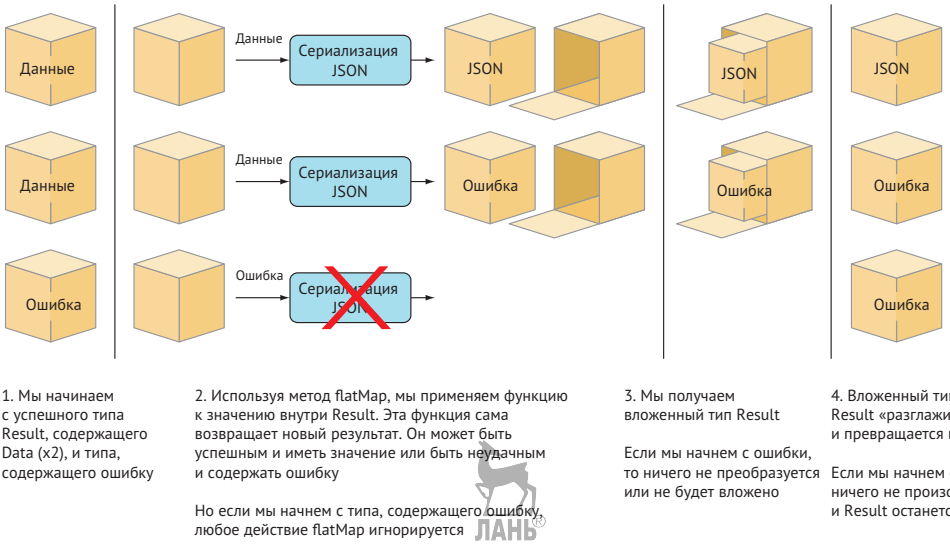


Рис. 11.4. Как flatMap работает с типом Result

Заменив `map` на `flatMap` при преобразовании `Data` в JSON, можно вернуть `Result` из операции с методом `flatMap` в случае неудачного преобразования, как показано в листинге 11.12.

Листинг 11.12. Использование метода flatMap для типа Result

```
func search(term: String, completionHandler: @escaping (SearchResult<JSON>)
➥ -> Void) {
   // ... Пропускаем часть кода

   callURL(with: url) { result in
      let convertedResult: SearchResult<JSON> =
         result
            // Преобразование в SearchResultError
            .mapError { (networkError: NetworkError) ->
            ➥ SearchResultError in ❶
```

```
            return SearchResultError.underlyingError(networkError)
        }
        // Выполняем преобразование Data в JSON или возвращаем
        // SearchResultError
        .flatMap { (data: Data) -> SearchResult<JSON> in ❷
            guard
                let json = try? JSONSerialization.jsonObject(with:
                ➥ data, options: []),
                let jsonDictionary = json as? JSON else {
                    return SearchResult(.invalidData) ❸
                }
                return SearchResult(jsonDictionary)
    }
    completionHandler(convertedResult)
  }
}
```

❶ mapError перемещается вверх по цепочке, поэтому тип Error – это SearchResultError, прежде чем мы применим метод flatMap для значения. Это помогает flatMap, поэтому он также может вернуть SearchResultError вместо NetworkError.

❷ Операция с методом map заменяется на flatMap.

❸ Теперь можно вернуть Result из операции с методом flatMap.

flatMap не меняет тип Error

При использовании метода flatMap тип ошибки не меняется с одного на другой. Например, нельзя превратить Result<Value, SearchResultError> в Result<Value, NetworkError>. Об этом следует помнить, а также почему mapError перемещается вверх по цепочке.

11.3.3. Упражнения

2

Используя техники, которые вы изучили, попробуйте подключиться к реальному API. Посмотрите, можно ли реализовать API FourSquare (http://mng.bz/nxVg) и получить список мест в формате JSON. Вы можете зарегистрироваться, чтобы получить учетные данные разработчика.

Обязательно используйте Result для возврата любых мест, которые можно получить из API.

Чтобы разрешить асинхронные вызовы внутри playground-проектов, добавьте следующее:

```
import PlaygroundSupport
PlaygroundPage.current.needsIndefiniteExecution = true
```

3

Можно ли использовать методы `map`, `mapError` и даже `flatMap` для преобразования результата, чтобы вызывать обработчик завершения только один раз?

4

Сервер может вернуть ошибку, даже если вызов успешен. Например, если вы будете передавать широту и долготу 0, то получите значения `errorType` и `errorDetail` в ключе `meta` в JSON:

```
{"meta":{"code":400,"errorType":"param_error","errorDetail":"Must
➥ provide parameters (ll and radius) or (sw and ne) or (near and
➥ radius)","requestId":"5a9c09ba9fb6b70cfe3f2e12"},"response":{}}
```

Убедитесь, что эта ошибка отражена в типе `Result`.

11.4. Смешивание Result с функциями, генерирующими ошибку

Раньше мы избегали генерации ошибки внутри операций с методами `map` или `flatMap` в типе `Result`, чтобы иметь возможность сосредоточиться на одной идиоме за раз.

Давайте поднимем ставки. Как только вы начнете работать с возвращаемыми данными, то, скорее всего, будете использовать синхронные «обычные» функции для обработки данных, такие как преобразование или хранение данных либо проверка значений. Другими словами, вы будете применять функции, генерирующие ошибку, к значению внутри `Result`. По сути, это означает смешивание двух идиом обработки ошибок.

11.4.1. От генерации ошибки к типу Result

Ранее мы проводили преобразование данных (`Data`) в формат JSON из операции с методом `flatMap`. Чтобы имитировать реальный сценарий, давайте используем этот метод, дабы на этот раз конвертировать данные в JSON, используя `parseData`, функцию, которая генерирует ошибку. Чтобы быть более реалистичной, `parseData` идет вместе с ошибкой `ParsingError`, которая отличается от используемой нами ошибки `SearchResultError`.

Листинг 11.13. Функция parseData

```
enum ParsingError: Error { ❶
   case couldNotParseJSON }

func parseData(_ data: Data) throws -> JSON { ❷
   guard
      let json = try? JSONSerialization.jsonObject(with: data, options: []),
      let jsonDictionary = json as? JSON else {
```

```
            throw ParsingError.couldNotParseJSON
        }
        return jsonDictionary
}
```

❶ Используется конкретная ошибка для преобразования данных.

❷ Функция parseData превращает Data в JSON и может сгенерировать ошибку ParsingError.

Можно превратить эту функцию в Result с помощью инициализатора. Инициализатор принимает закрытие, которое может выбросить ошибку; затем он перехватывает все ошибки, генерируемые замыканием, и создает Result. Этот тип Result может быть успешным или неудачным (если была выброшена ошибка).

Вот как это работает: мы передаем функцию, генерирующую ошибку, в Result и в этом случае преобразуем ее в Result<JSON, SearchResultError>.

Листинг 11.14. Преобразование функции в Result

```
let searchResult: Result<JSON, SearchResultError> =
    Result(try parseData(data))
```

Мы почти у цели, но чего-то не хватает. Мы пытаемся преобразовать функцию parseData в Result с SearchResultError через инициализатор. Тем не менее функция не генерирует ошибку SearchResultError. Можно просмотреть тело функции, чтобы удостовериться. Но только во время выполнения Swift знает, какую ошибку сгенерирует функция parseData.

Если во время преобразования выскальзывает какая-либо ошибка, которая не является SearchResultError, инициализатор генерирует ошибку из parseData, а это значит, что ее нам также нужно перехватить. Более того, именно поэтому инициализатор генерирует ошибку, потому что он поступает так с любыми ошибками, которые не в состоянии преобразовать. Такая ситуация причиняет некоторое неудобство, когда вы превращаете известную во время выполнения ошибку в ошибку, известную на этапе компиляции.

Чтобы завершить преобразование, нужно добавить оператор *do catch*. В случае успеха или когда Result получает SearchResultError, мы остаемся в блоке do. Но как только функция parseData выдает ошибку ParsingError, как показано в приведенном ниже примере, мы попадаем в блок catch, что дает возможность вернуться к ошибке по умолчанию.

Листинг 11.15. Передача генерирующей функции в Result

```
do {
    let searchResult: Result<JSON, SearchResultError> = Result(try
        parseData(data)) ❶
} catch {
    print(error) // ParsingError.couldNotParseData
    let searchResult: Result<JSON, SearchResultError> =
```

```
        Result(.invalidData(data)) ❷
    }
```

❶ Вызываем функцию parseData(); если все удачно, мы получаем searchResult.

❷ Если преобразование завершится неудачно, мы попадем в оператор catch, где возвращаемся к SearchResult с ошибкой по умолчанию.

11.4.2. Преобразование генерирующей функции внутри flatMap

Теперь, когда мы знаем, как преобразовать функцию, генерирующую ошибку, в Result, можно приступить к их смешиванию с вашим конвейером с помощью метода flatMap.

Внутри метода flatMap, созданного ранее, создадим Result из функции parseData.

Листинг 11.16. Создание Result из parseData

```
func search(term: String, completionHandler: @escaping (SearchResult<JSON>)
➥ -> Void) {
    // ... Пропускаем часть кода
    callURL(with: url) { result in
        let convertedResult: SearchResult<JSON> =
            result
                .mapError { SearchResultError.underlyingError($0) }
                .flatMap { (data: Data) -> SearchResult<JSON> in ❶
                    do {
                        // Выполняем перехват, если метод parseData генерирует
                        ➥ ошибку ParsingError.
                        let searchResult: SearchResultError<JSON> =
                        ➥ Result(try parseData(data)) ❷
                        return searchResult
                    } catch {
                        // Игнорируем любые ошибки, которые генерирует функция
                        /➥ parseData, и возвращаемся к SearchResultError.
                        return SearchResult(.invalidData(data)) ❸
                    }
                }
                completionHandler(convertedResult)
    }
}
```

❶ Запускаем метод flatMap.

❷ Функция parseData передается инициализатору.

❷ Если преобразование `parseData` не удается, в конечном итоге мы получаем оператор `catch` и по умолчанию возвращаемся к `SearchResultError.invalidData`.

11.4.3. Пропускаем ошибки через конвейер

Создавая тип `Result` с помощью методов `map` и `flatMap`, мы выполняем так называемую *монадическую* обработку ошибок. Пусть этот термин не пугает вас – метод `flatMap` основан на законах монады из функционального программирования. Прелесть этого состоит в том, что мы можем сосредоточиться на подходе «счастливый путь» при преобразовании своих данных.

Как и в случае с опционалами, `flatMap` не вызывается, если `Result` не содержит значения. Можно работать с реальным значением (независимо от того, является `Result` ошибочным или нет), сохраняя контекст ошибки и распространяя `Result` выше – вплоть до того места, где какой-нибудь код сможет выполнить сопоставление с образцом, например в случае с кодом, вызывающим функцию.

Допустим, если мы продолжим преобразование данных, то можем получить несколько цепочек. В этом конвейере благодаря методу `map` вы всегда будете следовать подходу «счастливый путь», а с помощью `flatMap` вы сможете двигаться либо в правильном направлении, либо в ошибочном.

Например, вам нужно добавить дополнительные шаги, такие как проверка данных, их фильтрация и хранение в базе данных (возможно, в кеше). У вас будет несколько шагов, и здесь метод `flatMap` может повести нас по пути, который приведет к ошибке. В отличие от этого, благодаря методу `map` вы всегда будете следовать подходу «счастливый путь» (см. рис. 11.5).

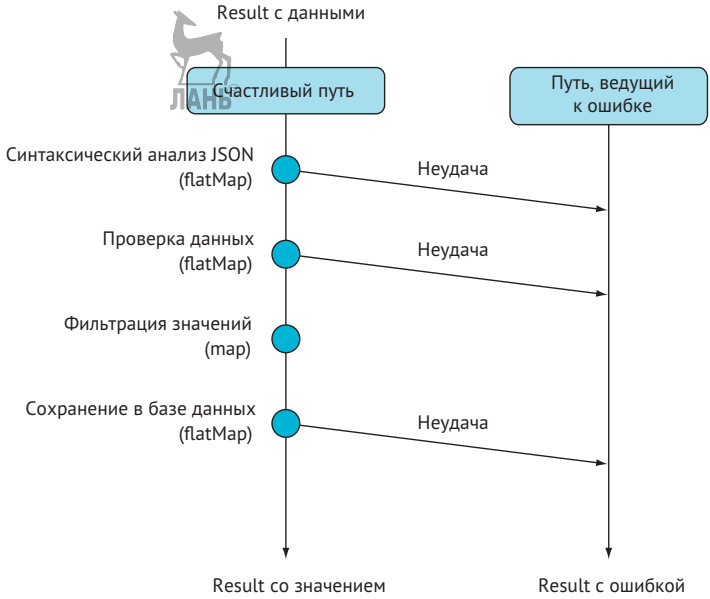


Рис. 11.5. Программирование с использованием подхода «счастливый путь»

Чтобы сэкономить время, мы не будем реализовывать все эти методы, но суть состоит в том, что можно создать сложный конвейер, как показано в приведенном листинге, пропустить через него ошибку и вызвать обработчик завершения только один раз.

Листинг 11.17. Более длинный конвейер

```
func search(term: String, completionHandler: @escaping (SearchResult<JSON>)
   -> Void) {
   // ... Пропускаем часть кода

   callURL(with: url) { result in
      let convertedResult: SearchResult<JSON> =
         result
            // Преобразование типа error в SearchResultError
            .mapError { (networkError: NetworkError) ->
               SearchResultError in
               // Код опущен
            }
            // Преобразуем Data в JSON или возвращаем SearchResultError
            .flatMap { (data: Data) -> SearchResult<JSON> in
               // Код опущен
            }
            // Оценка данных
            .flatMap { (json: JSON) -> SearchResult<JSON> in
               // Код опущен
            }
            // Фильтрация значений
            .map { (json: JSON) -> [JSON] in
               // Код опущен
            }
            // Сохранение в базу данных
            .flatMap { (mediaItems: [JSON]) -> SearchResult<JSON> in
               // Код опущен
               database.store(mediaItems)
            }
      completionHandler(convertedResult)
   }
}
```


Разрыв цепочки

Обратите внимание, что `map` и `flatMap` игнорируются, если `Result` содержит ошибку. Если какая-либо операция с `flatMap` возвращает `Result` с ошибкой, любые последующие операции с `flatMap` и `map` также игнорируются.

С помощью `flatMap` можно разрывать цепочки, так же как в случае применения `flatMap` для типа `Optional`.

11.4.4. Подходя к концу

Возможно, на вид ничего особенного, но наш API достаточно хорош. Он обрабатывает сетевые ошибки и ошибки преобразования, его легко читать и расширять. Нам удалось избежать ужасной пирамиды гибели, и наш код придерживается концепции «счастливый путь».

Получение простого перечисления `Result` выглядит не особо впечатляюще после всей этой работы. Но чистые API время от времени кажутся простыми.

11.4.5. Упражнение

5

Посмотрите на приведенные ниже генерирующие функции. Можно ли использовать их для преобразования Result в FourSquare API:

```
func parseData(_ data: Data) throws -> JSON {
   guard
      let json = try? JSONSerialization.jsonObject(with: data,
      options: []),
      let jsonDictionary = json as? JSON else {
         throw FourSquareError.couldNotParseData
      }
   return jsonDictionary
}

func validateResponse(json: JSON) throws -> JSON {
   if
      let meta = json["meta"] as? JSON,
      let errorType = meta["errorType"] as? String,
      let errorDetail = meta["errorDetail"] as? String {
         throw FourSquareError.serverError(errorType: errorType,
            errorDetail: errorDetail)
      }
   return json
}

func extractVenues(json: JSON) throws -> [JSON] {
   guard
      let response = json["response"] as? JSON,
      let venues = response["venues"] as? [JSON]
```

```
        else {
            throw FourSquareError.couldNotParseData
        }
    return venues
}
```

11.5. Несколько ошибок внутри Result

Работа с Result может показаться обременительной, когда целый ряд действий может окончиться неудачей. Ранее мы преобразовывали каждую ошибку в Result, содержащий один тип ошибки – SearchResultError в примерах. Преобразование ошибок в один тип – неплохой подход. Но это может стать довольно проблематичным, если вы имеете дело со множеством различных ошибок, особенно когда вы начинаете новый проект и вам необходимо соединить воедино все способы генерации ошибок. Преобразование каждой ошибки в правильный тип может замедлить работу.

Но не стоит беспокоиться; если вы хотите продвигаться быстро и делать так, чтобы ошибки были известными во время выполнения, можно использовать обобщенный тип AnyError, который также предлагает диспетчер пакетов Swift.

11.5.1. Знакомство с AnyError

Тип AnyError обозначает любую ошибку, которая может быть внутри Result, позволяя смешивать и сопоставлять все типы ошибок в одном и том же типе Result. Используя AnyError, вы избавляете себя от необходимости выяснять каждую ошибку на этапе компиляции.

AnyError оборачивает Error и сохраняет ошибку внутри; затем Result может использовать AnyError в качестве типа ошибки, например Result<String, AnyError>. Можно создать AnyError вручную, а также можно создать Result типа Result<String, AnyError> несколькими способами.

Обратите внимание, что в Result есть два инициализатора: один преобразует обычную ошибку в AnyError, а другой принимает генерирующую функцию, в которой ошибка преобразуется в AnyError.

Листинг 11.18. Тип AnyError

```
enum PaymentError: Error {
    case amountTooLow
    case insufficientFunds
}

let error: AnyError = AnyError(PaymentError.amountTooLow) ❶

let result: Result<String, AnyError> = Result(PaymentError.amountTooLow) ❷

let otherResult: Result<String, AnyError> = Result(anyError: { () throws ->
```

```
    String in ❸
    throw PaymentError.insufficientFunds
})
```

❶ Можно передать ошибку типу AnyError.

❷ Также можно напрямую передать ошибку в Result, в результате чего ошибка автоматически преобразуется в AnyError, поскольку Result имеет тип Result<String, AnyError>.

❸ Можно даже передать функции, выдающие ошибку, в Result. Поскольку AnyError обозначает все возможные ошибки, преобразование всегда завершается успешно.

Функции, возвращающие Result с AnyError, аналогичны генерирующей функции, в которой вам известен только тип ошибки во время выполнения.

Наличие AnyError имеет смысл, когда вы разрабатываете API и не хотите слишком фокусироваться на соответствующих ошибках. Представьте, что вы создаете функцию для перевода денег под названием processPayment. Можно возвращать разные типы ошибок на каждом шаге, что освобождает вас от необходимости преобразовывать разные ошибки в один конкретный тип. Обратите внимание, что мы также получаем специальный метод mapAny.

Листинг 11.19. Возвращение разных ошибок

```
func processPayment(fromAccount: Account, toAccount: Account, amountInCents:
➥ Int, completion: @escaping (Result<String, AnyError>) -> Void) {

guard amountInCents > 0 else {
    completion(Result(PaymentError.amountTooLow)) ❶
    return
}

guard isValid(toAccount) && isValid(fromAccount) else {
    completion(Result(AccountError.invalidAccount)) ❷
    return
}

// Обработка платежа.
moneyAPI.transfer(amountInCents, from: fromAccount, to: toAccount) {
➥ (result: Result<Data, AnyError>) in
        let response = result.mapAny(parseResponse) ❸
    completion(response)
    }
}
```

❶ Возвращаем PaymentError.

❷ Но можно вернуть и другую ошибку типа AccountError.

❸ Используем специальный метод mapAny.

Интересно отметить, что если Result имеет тип AnyError, то мы получаем специальный mapAny бесплатно. Метод mapAny работает аналогично map, за исключением того, что он может принимать любую функцию, которая генерирует ошибку. Если функция внутри mapAny генерирует ошибку, метод автоматически оборачивает эту ошибку внутри AnyError. Данная техника позволяет передавать генерирующие функции методу map без необходимости перехватывать какие-либо ошибки.

Кроме того, есть большое отличие этого метода от flatMap, которое заключается в том, что нельзя изменить ErrorType изнутри операций. В случае с flatMap вам придется создавать и возвращать новый тип Result вручную. Используя метод mapAny, можно передать обычную функцию, выдающую ошибку, и позволить mapAny обрабатывать перехват и обертку в AnyError. Применение метода mapAny позволяет работать со значением и даже изменить ошибку в Result.

map или mapAny: что выбрать

Разница между map и mapAny заключается в том, что первый метод работает для всех типов Result, но не перехватывает ошибки из функций, которые их генерируют, в то время как mapAny работает с различными функциями, но доступен только для типов Result, содержащих AnyError. Попробуйте использовать map, когда это возможно; он сообщает, что функция не может выбросить ошибку. Кроме того, если вы реорганизуете AnyError обратно в обычный тип Error в Result, map все еще будет доступен.

Сопоставление для AnyError

Чтобы вывести ошибку при работе с AnyError, можно использовать свойство underlyingError для выполнения сопоставления для фактической ошибки внутри него.

Листинг 11.20. Сопоставление для AnyError

```
processPayment(fromAccount: from, toAccount: to, amountInCents: 100) {
➥ (result: Result<String, AnyError>) in
   switch result {
   case .success(let value): print(value)
   case .failure(let error) where error.underlyingError is AccountError:
      print("Account error")
   case .failure(let error):
      print(error)
   }
}
```

AnyError – полезный заполнитель, позволяющий выполнять «правильную» обработку ошибок в более позднее время. Если время позволяет и ваш код становит-

ся устойчивым, можно начать заменять общие ошибки более строгими преобразованиями для получения дополнительных преимуществ на этапе компиляции.

Работа с AnyError дает гораздо больше гибкости, но вы страдаете от эрозии кода, потому что теряете большое преимущество, которое дает Result, когда можно увидеть, какие ошибки ожидать, еще до запуска кода. Также можно использовать тип NSError вместо AnyError, потому что он тоже довольно гибок. Но потом вы посмотрите на Objective-C и увидите, что лишились таких преимуществ, как строгое сопоставление с образцом для ошибок «перечисление–тип». Прежде чем идти по маршруту NSError, можно подумать еще раз и определить, сможете ли вы и дальше работать с ошибками Swift в сочетании с AnyError.

11.6. Невообразимый провал и Result

Иногда вам может понадобиться соответствовать протоколу, который хочет, чтобы вы использовали тип Result. Но тип, который реализует протокол, возможно, так и не потерпит неудачу. Давайте посмотрим, как улучшить свой код в этом случае с помощью уникального приема. Этот раздел выглядит несколько эзотерическим, но он оказывается полезным, когда вы сталкиваетесь с подобной ситуацией.

11.6.1. Когда протокол определяет Result

Представим, что у вас есть протокол Service, представляющий тип, который загружает для нас данные. Этот протокол определяет, что данные должны загружаться асинхронно, и использует Result.

У нас есть несколько типов ошибок и данные, которые можно загрузить, поэтому протокол определяет их как ассоциированные типы.

Листинг 11.21. Протокол Service

```
protocol Service {
   associatedtype Value ❶
   associatedtype Err: Error ❷
   func load(complete: @escaping (Result<Value, Err>) -> Void) ❸
}
```

❶ Значение, которое загружает протокол.

❷ Это ошибка, которую может предоставить протокол. Обратите внимание, что наш ассоциированный тип носит название Err и ограничен протоколом Error.

❸ Метод load возвращает Result, содержащий Value и Err, переданные замыканием.

Теперь нам нужно реализовать этот протокол с помощью типа Subscription Loader, который загружает подписки клиентов на журналы. Это показано в листинге 11.22. Обратите внимание, что загрузка подписок *всегда* завершает-

ся успешно. Это можно гарантировать благодаря тому, что они загружаются из памяти. Но тип Service объявляет, что мы используем Result, а это требует ошибки, поэтому нужно объявить, какую ошибку выдает SubscriptionLoader. У SubscriptionLoader нет ошибок. Чтобы устранить эту проблему, давайте создадим пустое перечисление, соответствующее Error, которое будет называться BogusError, чтобы SubscriptionLoader мог соответствовать протоколу Service. Обратите внимание, что у этого перечисления нет кейсов, а это означает, что на самом деле оно пустое.

Листинг 11.22. Реализация протокола Service

```
struct Subscription { ❶
   // ... детали опущены
}

enum BogusError: Error {} ❷

final class SubscriptionsLoader: Service {
func load(complete: @escaping (Result<[Subscription], BogusError>) ->
   Void) { ❸
      // ... загрузка данных. Всегда успешно
      let subscriptions = [Subscription(), Subscription()]
      complete(Result(subscriptions))
   }
}
```

❶ Subscription – это тип данных, извлекаемых из SubscriptionLoader.

❷ Мы создаем фиктивный тип Error, поэтому можно определить его в типе Result, чтобы угодить протоколу.

❸ Метод load теперь возвращает Result, возвращающий массив подписок. Обратите внимание, что мы определили пустой тип BogusError, чтобы угодить протоколу.

Мы создали пустое перечисление, которое соответствует типу Error, просто чтобы угодить компилятору. Но поскольку в BogusError нет кейсов, нельзя создать его экземпляр, и Swift это знает. После вызова метода load для SubscriptionLoader и получения Result можно выполнить сопоставление только для кейса success, а Swift достаточно умен, чтобы понять, что у вас не будет кейса failure. BogusError нельзя создать, поэтому нам не нужно выполнять сопоставление, как показано в приведенном ниже примере.

Листинг 11.23. Сопоставление только для кейса success

```
let subscriptionsLoader = SubscriptionsLoader ()
subscriptionsLoader.load { (result: Result<[Subscription], BogusError>) in
   switch result {
      case .success(let subscriptions): print(subscriptions)
         // Нам не нужен .failure ❶
```

```
    }
}
```

❶ Нам сошло это с рук. В обычном случае мы бы получили ошибку компилятора!

Данный метод позволяет исключить на этапе компиляции кейсы, для которых нужно выполнить сопоставление. С помощью него можно очистить свои API, демонстрируя более явное намерение. Но официальное решение – тип Never – позволяет избавиться от BogusError.

Тип Never

Чтобы угодить компилятору, мы создали тип BogusError, который нельзя инстанцировать. На самом деле такой тип уже существует в Swift, и называется он Never.

Тип Never – это так называемый *нижний тип*; он сообщает компилятору, что определенный путь к коду не может быть достигнут. Этот механизм также можно встретить и в других языках программирования, например тип Nothing в Scala или когда функция в Rust возвращает восклицательный знак (!).

Never – это скрытый тип, используемый Swift для обозначения невозможных путей. Например, когда функция вызывает fatalError, она может возвращать тип Never, указывая, что возврат чего-либо является невозможным путем.

Листинг 11.24. Из исходного кода Swift

```
func crashAndBurn() -> Never { ❶
    fatalError("Something very, very bad happened")
}
```

❶ Тип Never возвращается, но код гарантирует, что он никогда не вернется.

Если заглянуть внутрь исходного кода Swift, то можно увидеть, что Never – это не что иное, как пустое перечисление.

Листинг 11.25. Тип Never

```
public enum Never {}
```

В нашей ситуации можно заменить BogusError на Never и получить тот же результат. Однако все же нужно убедиться, что Never реализует Error.

Листинг 11.26. Реализация Never

```
extension Never: Error {} ❶
final class SubscriptionsLoader: Service {
    func load(complete: @escaping (Result<[Subscription], Never>) -> Void) {❷
        // ... загрузка данных. Всегда успешно
        let subscriptions = [Subscription(), Subscription()]
        complete(Result(subscriptions))
    }
}
```

❶ Мы расширяем тип Never, чтобы он соответствовал протоколу Error.

❷ Теперь мы используем тип Never, чтобы указать, что SubscriptionLoader никогда не дает сбоя.

Примечание

Начиная с пятой версии Swift Never соответствует некоторым протоколам, таким как Error.

Обратите внимание, что Never также может указывать на то, что Service все время терпит неудачу. Например, можно поставить Never в качестве кейса success в Result.

11.7. В заключение

Я надеюсь, что вы убедились в преимуществах обработки ошибок с помощью Result. Result может дать вам представление о том, какую ошибку ожидать на этапе компиляции. По ходу дела мы воспользовались своими знаниями методов map и flatMap и написали код, который казался безошибочным, но все же содержал в себе ошибки. Теперь вы знаете, как применять монадическую обработку ошибок.

Вот спорная мысль: можно использовать тип Result для обработки всех ошибок в своем проекте. Вы получаете больше преимуществ на этапе компиляции, но за счет более сложного программирования. Обработка ошибок с помощью Result более жесткая, но в качестве награды вы получите более безопасный и строгий код. А если вы хотите немного ускориться, всегда можно создать тип Result, содержащий AnyError.

Резюме

- Использование операций с данными в URLSession по умолчанию – небезопасный способ обработки ошибок.
- Тип Result предоставляется диспетчером пакетов Swift и является хорошим способом асинхронной обработки ошибок.
- У Result есть два обобщения, и он во многом напоминает тип Optional, но объясняет, почему что-то не получилось.
- Result – безопасный способ обработки ошибок на этапе компиляции, и можно увидеть, какую ошибку ожидать до запуска программы.
- Используя методы map, flatMap и mapError, вы можете чисто соединять преобразования своих данных, сохраняя контекст ошибки.
- Функции, генерирующие ошибки, можно преобразовать в Result с помощью специального инициализатора. Этот инициализатор позволяет смешивать и сочетать две идиомы генерирования ошибок.
- Можно отложить строгую обработку ошибок, используя тип AnyError.

- С помощью AnyError многократные ошибки могут находиться в Result.
- Если вы работаете с большим количеством типов ошибок, работа с AnyError может быть быстрее за счет того, что вы не знаете, каких ошибок ожидать на этапе компиляции.
- AnyError может быть хорошей альтернативой NSError, поэтому можно воспользоваться преимуществами типов ошибок Swift.
- Можно использовать тип Never, чтобы указать, что у Result не может быть кейса failure или success.

Ответы

1

Посмотрите на функцию map в Result и скажите, можно ли создать mapError.

```
extension Result {
   public func mapError<E: Error>(_ transform: (ErrorType) throws
   ➥ -> E) rethrows -> Result<Value, E> {
      switch self {
         case .success(let value):
            return Result<Value, E>(value)
         case .failure(let error):
            return Result<Value, E>(try transform(error))
      }
   }
}
```

Следующая часть – ответы на упражнения 2 и 3.

2

Используя техники, которые вы изучили, попробуйте подключиться к реальному API. Посмотрите, можно ли реализовать API FourSquare (http://mng.bz/nxVg) и получить список мест в формате JSON. Вы можете зарегистрироваться, чтобы получить учетные данные разработчика.

3

Можно ли использовать методы map, mapError и даже flatMap для преобразования результата, чтобы вызывать обработчик завершения только один раз?

4

Сервер может вернуть ошибку, даже если вызов успешен. Например, если вы будете передавать широту и долготу 0, то получите значения errorType и errorDetail в ключе meta в JSON. Убедитесь, что эта ошибка отражена в типе Result.

```
// Нам нужна ошибка.
enum FourSquareError: Error {
```

```
    case couldNotCreateURL
    case networkError(Error)
    case serverError(errorType: String, errorDetail: String)
    case couldNotParseData
}

let clientId = ENTER_YOUR_ID
let clientSecret = ENTER_YOUR_SECRET
let apiVersion = "20180403"

// Вспомогательная функция для создания URL
func createURL(endpoint: String, parameters: [String: String]) -> URL? {
    let baseURL = https://api.foursquare.com/v2/
    // Преобразуем словарь parameters в массив URLQueryItems.
    var queryItems = parameters.map { pair -> URLQueryItem in
        return URLQueryItem(name: pair.key, value: pair.value)
    }

    // Добавляем в запрос параметры по умолчанию
    queryItems.append(URLQueryItem(name: "v", value: apiVersion))
    queryItems.append(URLQueryItem(name: "client_id", value: clientId))
    queryItems.append(URLQueryItem(name: "client_secret", value:
        clientSecret))
    var components = URLComponents(string: baseURL + endpoint)
    components?.queryItems = queryItems
    return components?.url
}

// Вызов функции getvenues
func getVenues(latitude: Double, longitude: Double, completion:
➥ @escaping (Result<[JSON], FourSquareError>) -> Void) {
    let parameters = [
        "ll": "\(latitude),\(longitude)",
        "intent": "browse",
        "radius": "250"
    ]

guard let url = createURL(endpoint: "venues/search", parameters:
    parameters)
    else {
        completion(Result(.couldNotCreateURL))
        return
    }
    let task = URLSession.shared.dataTask(with: url) { data, response,
```

```
        error in
        let translatedError = error.map { FourSquareError.networkError($0) }
        // Преобразуем опционал data и опционал в Result
        let result = Result<Data, FourSquareError>(value: data, error:
           translatedError)

           // Преобразование Data в JSON
           .flatMap { data in
              guard
                 let rawJson = try?
                    JSONSerialization.jsonObject(with: data, options: []),
                 let json = rawJson as? JSON
                 else {
                    return Result(.couldNotParseData)
              }
              return Result(json)
           }

           // Проверка на предмет наличия ошибок сервера
           .flatMap { (json: JSON) -> Result<JSON, FourSquareError> in
              if
                 let meta = json["meta"] as? JSON,
                 let errorType = meta["errorType"] as? String,
                 let errorDetail = meta["errorDetail"] as? String {
                    return Result(.serverError(errorType: errorType,
                    errorDetail: errorDetail))
              }
              return Result(json)
           }

           // Извлекаем местоположения (venues).
           .flatMap { (json: JSON) -
              > Result<[JSON], FourSquareError> in
              guard
                 let response = json["response"] as? JSON,
                 let venues = response["venues"] as? [JSON]
                 else {
                    return Result(.couldNotParseData)
              }
              return Result(venues)
        }
        completion(result)
    }
```

```
    task.resume()
}

// Times Square
let latitude = 40.758896
let longitude = -73.985130

// Вызываем метод getVenues
getVenues(latitude: latitude, longitude: longitude) { (result:
    Result<[JSON], FourSquareError>) in
        switch result {
            case .success(let categories): print(categories)
            case .failure(let error): print(error)
        }
}
```

5

Посмотрите на приведенные ниже генерирующие функции. Можно ли использовать их для преобразования `Result` в FourSquare API?

```
enum FourSquareError: Error {
    // ... Пропускаем часть кода
    case unexpectedError(Error) // Добавляем новую ошибку, если
    // преобразование в Result прошло неудачно.
}

func getVenues(latitude: Double, longitude: Double, completion:
➥ @escaping (Result<[JSON], FourSquareError>) -> Void) {
    // ... Пропускаем часть кода
    let result = Result<Data, FourSquareError>(value: data, error:
        translatedError)

    // Преобразование в формат JSON
    .flatMap { data in
        do {
            return Result(try parseData(data))
        } catch {
            return Result(.unexpectedError(error))
        }
    }

    // Проверка на предмет наличия ошибок сервера.
    .flatMap { (json: JSON) -> Result<JSON, FourSquareError> in
        do {
            return Result(try validateResponse(json: json))
```

```
        } catch {
            return Result(.unexpectedError(error))
        }
    }

    // Извлекаем местоположения (venues).
    .flatMap { (json: JSON) -> Result<[JSON], FourSquareError> in
        do {
            return Result(try extractVenues(json: json))
        } catch {
            return Result(.unexpectedError(error))
        }
}
```

Глава 12. Расширения протоколов

В этой главе:

- гибкое моделирование данных с протоколами вместо подклассов;
- добавление поведения по умолчанию с помощью расширений протоколов;
- расширение обычных типов с протоколами;
- работа с наследованием протоколов и реализациями по умолчанию;
- применение композиции протоколов для создания очень гибкого кода;
- как Swift назначает приоритеты для вызовов методов;
- расширение типов, содержащих ассоциированные типы;
- расширение жизненно важных протоколов, таких как `Sequence` и `Collection`.

В предыдущих главах было показано, как работать с протоколами, ассоциированными типами и обобщениями. Чтобы еще лучше ориентироваться в мире протоколов, эта глава проливает свет на расширения протоколов. Для некоторых возможность расширения протокола является наиболее важной особенностью Swift, как вы увидите позже в этой главе.

Помимо объявления сигнатуры метода, с помощью протоколов можно предоставлять полную реализацию. Расширение протокола означает, что вы можете использовать реализации протокола по умолчанию, чтобы типам не нужно было реализовывать определенные методы. Это дает огромные преимущества. Можно изящно обходить жесткие структуры подклассов и в конечном итоге получить в своих приложениях гибкий и многократно используемый код.

В чистом виде термин «расширение протокола» звучит просто. Используйте реализацию по умолчанию и будьте счастливы. Кроме того, расширения протокола можно легко понять, если не лезть вглубь. Но по мере прохождения этой главы вы обнаружите множество разных вариантов использования, подводных камней, передовых практик и хитростей, связанных с правильно расширяемыми протоколами.

Вы увидите, что для компиляции требуется гораздо больше, чем просто получение кода. Кроме того, существует проблема, когда связанность кода может быть сильно уменьшена, и понимание того, какие методы вы вызываете, может быть непростым делом, тем более когда вы смешиваете протоколы с наследованием протоколов, переопределяя методы. Вы убедитесь, что протоколы не всегда легко понять. Но при правильном применении они позволяют создавать очень гибкий код. Сначала вы увидите, как протоколы позволяют моделировать данные по *горизонтали*, а не по *вертикали*, например с помощью подклассов, а также узнаете, как работают расширения протокола и как их переопределять.

Затем мы смоделируем почтовый API двумя способами и рассмотрим связанные с этим компромиссы. Сперва мы используем наследование протоколов для

предоставления реализации по умолчанию, которая более специализирована. Затем мы смоделируем тот же API с помощью функции сигнатуры под названием *композиция протокола*. Вы увидите преимущества и недостатки обоих подходов.

Далее наступает время для теории, чтобы лучше понять то, какие методы потом вызываются. Мы рассмотрим методы переопределения, наследование от протоколов и приоритеты вызовов Swift. Это немного теоретически, если вы увлечены этим.

На следующем этапе вы увидите, как расширять типы в нескольких направлениях. Вы найдете компромисс между расширением типа для соответствия протоколу и расширением протокола, ограниченного типом. Это тонкое, но важное различие.

Продвигаясь дальше, вы увидите, как расширять типы с помощью ассоциированных типов. Затем вы узнаете, как расширить протокол `Collection` и как Swift назначает приоритеты методам, основанным на ограниченном ассоциированном типе.

Напоследок мы перейдем на более низкий уровень и увидим, как Swift расширяет протокол `Sequence`. Мы будем применять эти знания для создания расширений многократного использования. Мы создадим метод `take(while:)`, который представляет собой противоположность методу `drop(while:)`, а также создадим метод `inspect`, который поможет отлаживать итераторы. Далее последует краткий обзор функций высшего порядка наряду с массивом `ContiguousArray` для написания расширений более низкого уровня.

После того как вы закончите читать эту главу, вы можете поймать себя на том, что пишете код с более уменьшенной связанностью, поэтому давайте начнем.

12.1. Наследование классов в сравнении с наследованием протоколов

В мире объектно-ориентированного программирования типичным способом достижения наследования является использование подклассов. Подклассы – это легитимный способ достижения полиморфизма. Они предлагают разумные значения по умолчанию для подклассов. Но, как вы видели в этой книге, наследование может быть жесткой формой моделирования данных. В качестве альтернативы наследованию на основе классов Swift предлагает наследование протоколов, известное как протокольно-ориентированное программирование, которое поразило многих разработчиков во время презентаций на всемирной конференции для разработчиков на платформах (*Apple Apple World Wide Developers Conference* – WWDC). С помощью расширений протоколов можно использовать метод с полной реализацией (существующих) типов без необходимости использовать иерархию подклассов, предлагая высокую возможность повторного использования.

В этом разделе вы увидите, что моделирование может работать горизонтально, а не вертикально, когда вы используете протоколы.

12.1.1. Моделирование данных по горизонтали, а не по вертикали

Присоединяйтесь!

Будет более познавательно и веселее, если вы будете знакомиться с кодом по мере прохождения этой главы. Исходный код можно скачать по адресу `http://mng.bz/vOvJ`.

Можно рассматривать подклассы как вертикальный способ моделирования данных. У вас есть суперкласс, и вы можете создать его подкласс и переопределить методы и поведение, чтобы добавить новую функциональность, а затем можно пойти еще ниже и снова создать подкласс. Представьте, что мы создаем класс `RequestBuilder`, который создает типы `URLRequest` для сетевого вызова. Подкласс может расширить функциональность, добавив заголовки по умолчанию, а другой подкласс может зашифровать данные внутри запроса. Через подклассы мы получаем тип, который может создавать зашифрованные сетевые запросы (см. рис. 12.1).

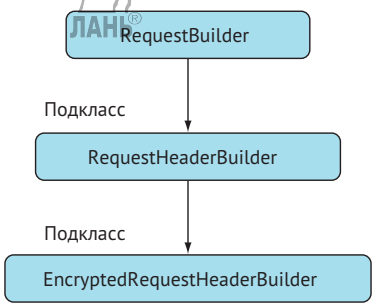


Рис. 12.1. Наследование на базе классов

С другой стороны, протоколы можно представить как горизонтальный способ моделирования данных. Мы берем тип и добавляем к нему дополнительную функциональность, заставляя его соответствовать протоколам, например добавляя строительные блоки в свою структуру. Вместо того чтобы создавать суперкласс, мы создаем отдельные протоколы с реализацией по умолчанию для создания запросов и заголовков и для шифрования.

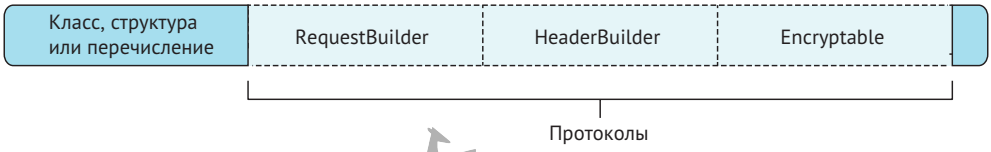


Рис. 12.2. Реализация протоколов

Концепция отделения функциональности от типа дает много гибкости и позволяет использовать код повторно. Вы больше не ограничены одним суперклассом. Пока тип соответствует `RequestBuilder`, он получает функциональность по умолчанию. Это может быть любой тип: перечисление, структура, класс, подкласс – не важно.

12.1.2. Создание расширения протокола

Создание расширения протокола – безболезненный процесс. Продолжим рассматривать наш пример с протоколом RequestBuilder. Сначала мы определяем протокол, а затем расширяем его, таким образом получая возможность добавить реализацию по умолчанию для каждого типа, соответствующего этому протоколу.

Листинг 12.1. Расширение протокола

```
protocol RequestBuilder {
   var baseURL: URL { get } ❶
      func makeRequest(path: String) -> URLRequest ❷
   }

extension RequestBuilder { ❸
   func makeRequest(path: String) -> URLRequest { ❹
      let url = baseURL.appendingPathComponent(path) ❺
      var request = URLRequest(url: url)
      request.httpShouldHandleCookies = false
      request.timeoutInterval = 30
      return request
   }
}
```

❶ Определяем протокол RequestBuilder со свойством baseURL.

❷ Также определяем метод makeRequest. У метода в определении протокола не может быть тела.

❸ Расширяем протокол RequestBuilder, потому что методы внутри расширений могут иметь тела.

❹ Используем реализацию метода makeRequest по умолчанию.

❺ Расширение использует свойство baseURL.

Примечание

В протоколе реализация по умолчанию всегда добавляется через расширение.

Чтобы получить реализацию makeRequest бесплатно, нужно всего лишь соответствовать протоколу RequestBuilder. Мы соответствуем RequestBuilder, сохраняя свойство baseURL. Например, представим себе приложение для стартапа, которое перечисляет захватывающие поездки на велосипеде. Приложение должно делать запросы для получения данных. Для этого тип BikeRequestBuilder соответствует протоколу RequestBuilder, и мы обязательно должны реализовать свойство baseURL. В результате мы получаем метод makeRequest.

Листинг 12.2. Реализация протокола с реализацией по умолчанию

```
struct BikeRequestBuilder: RequestBuilder { ❶
```

```
    let baseURL: URL = URL(string: "https://www.biketriptracker.com")! ❶
}

let bikeRequestBuilder = BikeRequestBuilder()
let request = bikeRequestBuilder.makeRequest(path: "/trips/all") ❷
print(request) // https://www.biketriptracker.com/trips/all ❸
```

❶ Тип BikeRequestBuilder соответствует протоколу RequestBuilder и реализует требование свойства baseURL.

❷ Тип BikeRequestBuilder получает метод makeRequest бесплатно благодаря реализации по умолчанию.

❸ Подтверждаем, что был сделан успешный запрос.

12.1.3. Несколько расширений

Тип может соответствовать нескольким протоколам. Представим, что у нас есть BikeAPI, который создает запросы и обрабатывает их. Он может соответствовать двум протоколам: RequestBuilder из предыдущего примера и новому протоколу, ResponseHandler, из приведенного ниже примера. BikeAPI теперь может соответствовать обоим протоколам и получать методы бесплатно.

Листинг 12.3. ResponseHandler

```
enum ResponseError: Error {
    case invalidResponse
}

protocol ResponseHandler { ❶
    func validate(response: URLResponse) throws
}

extension ResponseHandler {
    func validate(response: URLResponse) throws {
        guard let httpresponse = response as? HTTPURLResponse else {
            throw ResponseError.invalidResponse
        }
    }
}

class BikeAPI: RequestBuilder, ResponseHandler { ❷
    let baseURL: URL = URL(string: "https://www.biketriptracker.com")!
}
```


❶ Вводим еще один протокол.

❷ Класс BikeAPI может соответствовать двум протоколам.

12.2. Наследование в сравнении с композицией

Мы уже видели, как использовать реализации по умолчанию через расширение протокола. Можно моделировать данные с расширениями иными способами, а именно с помощью *наследования протокола* и *композиции*.

Мы создадим гипотетический фреймворк, который может отправлять электронные письма через SMTP. Мы сосредоточимся на API и опустим реализацию. Давайте начнем с подхода, использующего наследование протоколов, а затем создадим более гибкий подход с помощью композиции протоколов. Таким образом можно будет увидеть весь процесс в обоих подходах и связанные с этим компромиссы.

12.2.1. Протокол Mailer

Сперва, как показано в приведенном ниже листинге, мы определяем тип `Email`, который использует структуру `MailAddress` для определения своих свойств. `MailAddress` выражает больше намерения, чем просто использование `String`. Помимо этого, мы определяем протокол `Mailer` с реализацией по умолчанию через расширение протокола (реализация опущена).

Листинг 12.4. Типы Email и Mailer

```
struct MailAddress { ❶
   let value: String
}

struct Email { ❷
   let subject: String
   let body: String
   let to: [MailAddress]
   let from: MailAddress
}

protocol Mailer {
   func send(email: Email) ❸
}

extension Mailer {
   func send(email: Email) { ❹
      // Опущено: Соединение с сервером
      // Опущено: Отправка сообщения
      print(«Email is sent!»)
   }
}
```

❶ `MailAddress` обозначает адрес электронной почты.

❷ `Email` содержит значения для отправки электронной почты.

❸ Протокол `Mailer` может отправлять электронные письма.

❹ По умолчанию протокол `Mailer` сможет отправлять электронную почту (реализация опущена).

Неплохо для начала. Теперь представим, что нам нужно добавить реализацию по умолчанию для протокола `Mailer`, который *также* проверяет `Email` перед отправкой. Но не все почтовые программы проверяют электронную почту. Возможно, почтальон основан на команде `sendmail` из UNIX или другом сервисе, который не проверяет электронную почту как таковую. Не все почтовые программы выполняют проверку, поэтому нельзя предполагать, что `Mailer` проверяет электронную почту по умолчанию.

12.2.2. Наследование протокола

Если мы все же хотим использовать реализацию по умолчанию, которая позволяет отправлять проверенные электронные письма, можно воспользоваться как минимум двумя подходами. Мы начнем с наследования протоколов, а затем перейдем на композиционный подход, чтобы увидеть плюсы и минусы обоих подходов.

При использовании наследования можно расширить протокол, чтобы добавить дополнительные требования. Для этого нужно создать подпротокол, который наследует от суперпротокола, аналогично тому, как протокол `Hashable` наследует от `Equatable`.

В качестве следующего шага мы начнем с создания протокола `ValidatingMailer`, который наследует от протокола `Mailer`. `ValidatingMailer` переопределяет метод `send(email:)`, делая его генерирующим, а также вводит новый метод под названием `validate(email:)` (см. рис. 12.3).

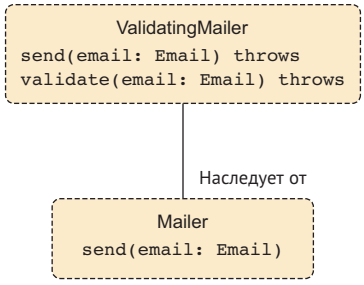


Рис. 12.3. ValidatingMailer наследует от Mailer

Чтобы облегчить жизнь разработчикам, которые будут заниматься реализацией `ValidatingMailer`, мы расширяем его и используем метод `send(email:)` по умолчанию, который использует метод `validate(email:)` перед отправкой. Опять же, чтобы сосредоточиться на API, мы опустим реализацию, как показано в этом листинге:

Листинг 12.5. Протокол ValidatingMailer

```
protocol ValidatingMailer: Mailer {
```

```
    func send(email: Email) throws // Функция Send теперь может генерировать
    // ошибку ❶
    func validate(email: Email) throws
}

extension ValidatingMailer {
    func send(email: Email) throws {
        try validate(email: email) ❷
        // Соединение с сервером
        // Отправка почты
        print(«Email validated and sent.»)
    }
    func validate(email: Email) throws {
        // Проверяем адрес электронной почты и отсутствует ли тема письма.
    }
}
```

❶ Снова объявляем метод `send(email :)`, чтобы он мог генерировать ошибку.

❷ Метод `send(email :)` использует метод `validate(email :)` для проверки электронного письма перед отправкой.

Теперь структура `SMTPClient` реализует `ValidatingMailer` и автоматически получает проверенный метод `send(email:)`.

Листинг 12.6. SmtpClient

```
struct SMTPClient: ValidatingMailer {
    // Реализация опущена.
}

let client = SMTPClient()
try? client.send(email: Email(subject: "Learn Swift",
    body: "Lorem ipsum",
    to: [MailAdress(value: "john@appleseed.com")],
    from: MailAdress(value: "stranger@somewhere.com"))
```

Недостатком наследования протокола является то, что мы не разделяем функциональность и семантику. Например, из-за наследования протокола все, что проверяет электронную почту автоматически, должно быть `Mailer`. Можно ослабить это ограничение, применив композицию протокола, – давайте сделаем это.

12.2.3. Композиционный подход

В случае с композиционным подходом мы сохраняем протокол `Mailer`, но вместо `ValidatingMailer`, который наследует от `Mailer`, мы будем использовать автономный протокол `MailValidator`, который ни от кого не наследует. Протокол

MailValidator также использует реализацию по умолчанию через расширение. Для краткости мы это опустим:

Листинг 12.7. Протокол MailValidator

```
protocol MailValidator {
   func validate(email: Email) throws
}

extension MailValidator {
   func validate(email: Email) throws {
      // Опущено: проверяем адрес электронной почты и отсутствует ли тема
      // письма.
   }
}
```

Теперь можно приступить к композиции. Сделаем так, чтобы SMTPClient соответствовала обоим протоколам. Mailer не знает о существовании MailValidator, и наоборот (см. рис. 12.4).

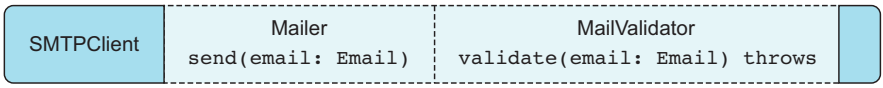


Рис. 12.4. SMTPClient реализует протоколы Mailer и MailValidator

При наличии двух протоколов можно создать расширение, которое работает только на пересечении протоколов. Расширение на пересечении означает, что типы, соответствующие протоколам Mailer и MailValidator, получают определенную реализацию или даже бонусный метод. Внутри пересечения send(email:) сочетает в себе функциональность Mailer и MailValidator (см. рис. 12.5).

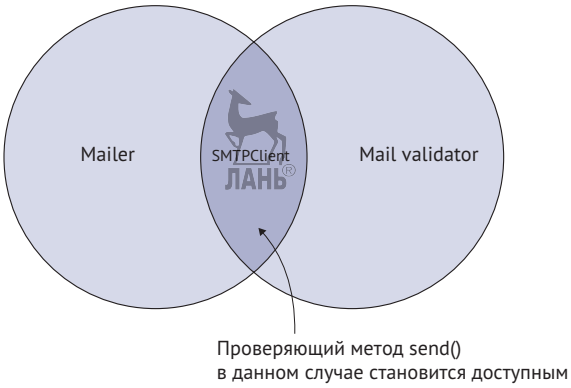


Рис. 12.5. Пересечение расширения

Чтобы создать расширение с пересечением, мы расширяем один протокол, который соответствует другому, с помощью ключевого слова Self.

Листинг 12.8. Пересечение Mailer и MailValidator

```
extension MailValidator where Self: Mailer { ❶
```

```
    func send(email: Email) throws { ❷
        try validate(email: email)
        // Соединение с сервером
        // Отправка почты
        print("Email validated and sent.")
    }
}
```

❶ Можно определить пересечение с помощью оператора `Self`.

❷ Можно использовать реализацию по умолчанию, когда реализованы методы `send(email:)` и `validate(email:)`. Обратите внимание, что в данном случае метод `send(email:)` может генерировать ошибку.

Примечание

Расширяете ли вы `MailValidator` или `Mailer`, не имеет значения – подойдет любое направление.

Еще одно преимущество этого подхода состоит в том, что мы можем использовать новые методы, такие как `send(email:, at:)`, который позволяет проверять почту и ставить ее в очередь. Мы проверяем электронную почту, чтобы в очереди была уверенность, что письмо можно отправить. Можно определить новые методы в пересечении протокола.

Листинг 12.9. Добавление бонусных методов

```
extension MailValidator where Self: Mailer {
// ... Пропускаем часть кода
    func send(email: Email, at: Date) throws { ❶
        try validate(email: email)
        // Соединение с сервером
        // Добавляем письмо в отложенную очередь
        print("Email validated and stored.")
    }
}
```

❶ В пересечении можно ввести новые методы.

12.2.4. Высвобождаем энергию пересечения

Теперь мы сделаем так, чтобы структура `SMTPClient` соответствовала протоколам `Mailer` и `MailValidator`, что позволит разблокировать код внутри пересечения протоколов. Другими словами, `SMTPClient` получает методы `send(email:)` и `send(email:, at:)` бесплатно.

Листинг 12.10. Реализация двух протоколов, чтобы получить бесплатный метод

```
struct SMTPClient: Mailer, MailValidator {} ❶
```

```
let client = SMTPClient()

let email = Email(subject: "Learn Swift",
   body: "Lorem ipsum",
   to: [MailAdress(value: "john@appleseed.com")],
   from: MailAdress(value: "stranger@somewhere.com"))

try? client.send(email: email) // Сообщение проверено и отправлено ❷
try? client.send(email: email, at: Date(timeIntervalSinceNow: 3600))
➥ //Сообщение проверено и поставлено в очередь. ❸
```

❶ SMTPClient соответствует обоим протоколам, что позволяет разблокировать методы пересечения.

❷ Используется метод пересечения `send (email:)`.

❸ Бонусный метод также разблокирован для использования.

Еще один способ увидеть преимущества – воспользоваться обобщенной функцией, как показано в листинге 12.11. Когда вы ограничены обоими протоколами, реализация пересечения становится доступной. Обратите внимание, что мы определяем обобщение T и ограничиваем его обоими протоколами. При этом становится доступным метод `send(email:, at:)`.

Листинг 12.11. Обобщение с пересечением

```
func submitEmail<T>(sender: T, email: Email) where T: Mailer,
➥ T: MailValidator {
   try? sender.send(email: email, at: Date(timeIntervalSinceNow: 3600))
}
```

Использование композиционного подхода значительно уменьшает связанность вашего кода. Фактически, может быть, даже слишком. Другие разработчики могут не знать точно, какая реализация метода используется «под капотом», и также могут быть не в состоянии определить, когда бонусные методы разблокированы. Еще одним недостатком является то, что такой тип, как `SMTPClient`, должен реализовывать несколько протоколов. Но преимущества огромны. Если действовать аккуратно, то с помощью композиции можно создать элегантный, многократно используемый код, в котором связанность значительно уменьшена.

Второй способ рассматривать пересекающиеся протоколы – использовать бесплатные преимущества под видом «Поскольку вы соответствуете и A, и B, вы также могли бы бесплатно использовать C».

При наследовании протокола `SMTPClient` нужно реализовать только один протокол, и он более жесткий. Определить, что получает разработчик, также немного проще. Когда вы работаете с протоколами, попытка найти лучшую абстракцию может быть непростой задачей.

12.2.5. Упражнение

1

Создайте расширение, которое активирует функцию `explode()`, но только для типов, соответствующих протоколам `Mentos` и `Coke`:

```
protocol Mentos {}
protocol Coke {}

func mix<T>(concoction: T) where T: Mentos, T: Coke {
   // concoction.explode() // Приводим это в действие, но только при
   ➥ условии, что T соответствует обоим протоколам, а не только одному.
}
```

12.3. Переопределение приоритетов

При реализации протоколов необходимо соблюдать несколько правил. Мы отойдем от протокола Mailer, о котором шла речь в предыдущем разделе, и станем более концептуальными.

12.3.1. Переопределение реализации по умолчанию

Чтобы увидеть, как работает наследование протоколов, представьте себе протокол Tree с методом `grow()`. Этот протокол предлагает реализацию по умолчанию посредством расширения. Между тем структура `Oak` реализует `Tree`, а также реализует метод `grow()` (см. рис. 12.6).

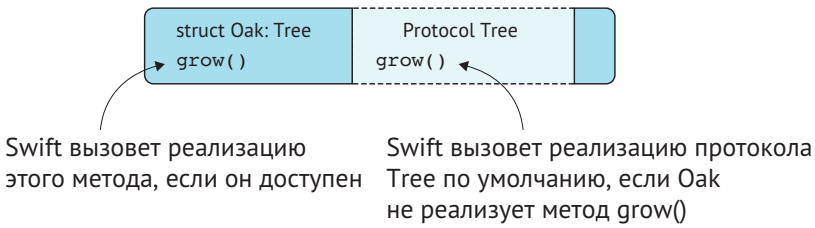


Рис. 12.6. Переопределение протокола

Swift выбирает самый конкретный метод, который может найти. Если тип реализует тот же метод, что и метод в расширении протокола, Swift игнорирует метод расширения. Другими словами, Swift вызывает функцию `grow()` для `Oak`, а не для `Tree`. Это позволяет переопределять методы, которые определены в расширении протокола.

Имейте в виду, что расширение протокола не может переопределять методы из реальных типов, например пытаясь предоставить существующему типу новую реализацию через протокол. Кроме того, на момент написания этих строк не существует специального синтаксиса, который позволял бы определить, переопределяет ли тип метод протокола. Когда тип, такой как класс, структура или пере-

числение, реализует протокол и реализует тот же метод, что и расширение протокола, не ясно, как определить, какой метод Swift вызывает внутри.

12.3.2. Переопределение и наследование протоколов

Чтобы усложнить задачу, мы введем наследование протокола. На этот раз мы познакомимся с еще одним протоколом под названием `Plant`, от которого наследует протокол `Tree`. `Oak` по-прежнему реализует `Tree`, а `Plant` также использует стандартную реализацию функции `grow()`, которую `Tree` переопределяет.

Swift снова вызывает наиболее специализированную реализацию `grow()` (см. рис. 12.7). Он вызывает этот метод для `Oak`, если тот доступен. В противном случае вызывает его для `Tree`. Если два предыдущих варианта терпят неудачу, Swift вызывает метод `grow()` для `Plant`. Если ничего не помогает, компилятор выдает ошибку.

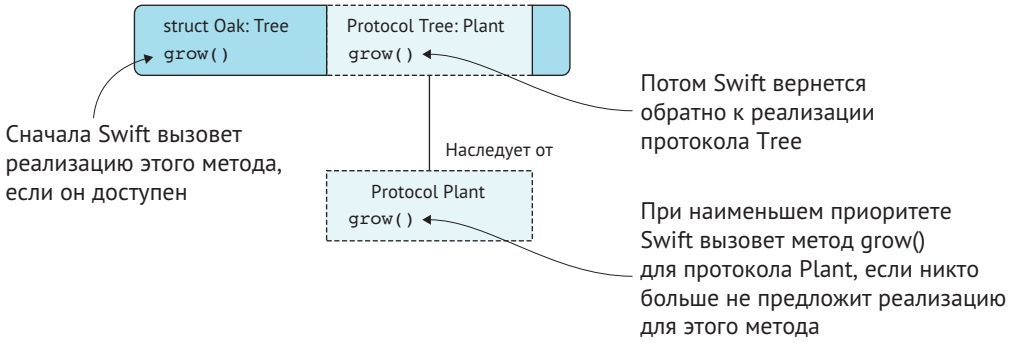


Рис. 12.7. Переопределение с наследованием протокола

Вы видите переопределения, которые можно наблюдать в примере кода. В приведенном ниже листинге мы определяем функцию `growPlant`; обратите внимание, что она принимает `Plant`, а не `Tree` или `Oak`. Swift выбирает наиболее специализированную реализацию в любом случае.

Листинг 12.12. Переопределение в действии

```
func growPlant<P: Plant>(_ plant: P) { ❶
   plant.grow()
}

protocol Plant {
   func grow()
}

extension Plant { ❷
   func grow() {
      print("Growing a plant")
   }
}
```

```
protocol Tree: Plant {} ❸

extension Tree {
   func grow() { ❸
      print("Growing a tree")
   }
}

struct Oak: Tree {
   func grow() { ❹
      print("The mighty oak is growing")
   }
}

struct CherryTree: Tree {} ❺
struct KiwiPlant: Plant {} ❻

growPlant(Oak()) // The mighty oak is growing
growPlant(CherryTree()) // Growing a tree ❼
growPlant(KiwiPlant()) // Growing a plant
```

❶ Определяем функцию `growPlant`. Обратите внимание, что она принимает `Plant`.

❷ Протокол `Plant` использует метод по умолчанию для `grow`.

❸ Протокол `Tree` наследует от `Plant` и использует свою реализацию для `grow()`.

❹ Структура `Oak` переопределяет метод `grow()` путем реализации собственной версии.

❺ Структура `CherryTree` не переопределяет `grow()`.

❻ `KiwiPlant` также не переопределяет `grow()`.

❼ Несмотря на то что `growPlant` принимает типы `Plant`, Swift вызывает наиболее специализированную версию методов.

При наследовании протокола интересная особенность состоит в том, что вы получаете поведение переопределения, подобное классам и подклассам. В протоколах это поведение доступно не только для классов, но и для структур и перечислений.

12.3.3. Упражнение

2

Каким будет вывод приведенного ниже кода?

```
protocol Transaction {
   var description: String { get }
}
```

```
extension Transaction {
   var   description: String { return "Transaction" }
}

protocol RefundableTransaction: Transaction {}

extension RefundableTransaction {
   var description: String { return "RefundableTransaction" }
}

struct CreditcardTransaction: RefundableTransaction {}
   func printDescription(transaction: Transaction) {
      print(transaction.description)
   }

printDescription(transaction: CreditcardTransaction())
```

12.4. Расширение в двух направлениях

В целом потребность в подклассе становится менее необходимой благодаря расширениям протоколов, за некоторыми исключениями. Иногда подклассы могут быть правильным подходом, например когда речь идет о преобразовании и Objective-C и делении `NSObject` на подклассы или при работе с конкретными фреймворками, такими как `UIKit`, которые используют представления и подпредставления, кроме всего прочего. Несмотря на то что эта книга вообще-то не затрагивает фреймворки, давайте сделаем небольшое исключение для реального случая использования, связанного с расширениями, пользовательским интерфейсом и подклассами.

Типичное использование протоколов в сочетании с подклассами включает в себя применение класса `UIViewController` из `UIKit`, который представляет собой экран (его часть) на iPhone, iPad или AppleTV. `UIViewController` предназначен для работы с подклассами и прекрасно подойдет в качестве примера.

12.4.1. Выбор расширений

Представим, что у нас есть протокол `AnalyticsProtocol`, который помогает отслеживать аналитические события для пользовательских метрик. Можно реализовать этот протокол для `UIViewController`, который использует реализацию по умолчанию, что добавляет функциональность `AnalyticsProtocol` всем типам `UIViewController` и его подклассам.

Но если предположить, что *все* они должны соответствовать этому протоколу, то, вероятно, это будет небезопасно. Если вы занимаетесь разработкой фреймворка с этим расширением, разработчик, реализующий этот фреймворк, получает это расширение автоматически, нравится ему это или нет. Хуже того, расширение из фреймворка может конфликтовать с существующим расширением в приложении, если у них одно и то же имя!

Один из способов избежать подобных проблем – переключить расширение. Переключение расширения означает, что, вместо того чтобы расширить UIViewController с помощью протокола, можно расширить протокол, ограниченный UIViewController.

Листинг 12.13. Переключение направлений расширения

```
protocol AnalyticsProtocol {
   func track(event: String, parameters: [String: Any])
}

// Неправильно:
extension UIViewController: AnalyticsProtocol {
   func track(event: String, parameters: [String: Any]) {
      // ... Пропускаем остальной код }
}

// Правильно:
extension AnalyticsProtocol where Self: UIViewController {
func track(event: String, parameters: [String: Any]) {
   // ... Пропускаем остальной код }
}
```

Теперь, если UIViewController явно соответствует этому протоколу, он выбирает преимущества протокола – например, NewsViewController может явно соответствовать протоколу AnalyticsProtocol и использовать его методы. Таким образом, мы делаем так, чтобы контроллеры (viewcontrollers) не соответствовали протоколу по умолчанию.

Листинг 12.14. Выбираем преимущества

```
extension NewsViewController: UIViewController, AnalyticsProtocol {
   // ... Пропускаем часть кода
   override func viewDidAppear(_ animated: Bool) {

      super.viewDidAppear(animated)
      track("News.appear", params: [:])
   }
}
```

Данная техника становится еще более важной при использовании фреймворка. Расширения не являются пространством имен, поэтому будьте осторожны с добавлением общедоступных расширений внутри фреймворка, поскольку разработчики могут не захотеть, чтобы их классы соответствовали протоколу по умолчанию.

12.4.2. Упражнение

3

В чем разница между этими двумя расширениями:

```
extension UIViewController: MyProtocol {}
extension MyProtocol where Self: UIViewController {}
```

12.5. Расширение с использованием ассоциированных типов

Давайте посмотрим, как Swift отдает приоритет вызовам методов для протоколов, особенно для протоколов с ассоциированными типами.

Начнем с массива (`Array`). Он реализует протокол `Collection`, у которого есть ассоциированный тип `Element`, представляющий элемент внутри коллекции. Если мы расширяем массив с помощью специальной функции, такой как `unique()`, которая удаляет все дубликаты, можно сделать это, обращаясь к `Element` как к его внутреннему значению.

Листинг 12.15. Использование функции unique () в массиве

```
[3, 2, 1, 1, 2, 3].unique() // [3, 2, 1]
```

Давайте расширим массив. Чтобы иметь возможность проверить каждый элемент на предмет равенства, нам нужно убедиться, что `Element` соответствует протоколу `Equatable`. Это можно выразить с помощью ограничения. Ограничение `Element` до `Equatable` означает, что функция `unique()` доступна только для массивов с элементами `Equatable`.

Листинг 12.16. Расширение массива

```
extension Array where Element: Equatable { ❶
   func unique() -> [Element] { ❷
      var uniqueValues = [Element]()
      for element in self {
         if !uniqueValues.contains(element) { ❸
            uniqueValues.append(element)
         }
      }
      return uniqueValues
   }
}
```

❶ Нужно ограничить `Element`, чтобы можно было сравнивать элементы.

❷ Метод `unique()` возвращает массив без повторяющихся элементов.

❸ Можно передать элементы Equatable в метод contains.

Расширение массива – хорошее начало. Но, вероятно, имеет смысл предоставить это расширение многим типам коллекций, а не только массиву, а возможно, также значениям словаря или даже строкам. Можно перейти на более низкий уровень и расширить протокол Collection, как показано здесь, чтобы несколько типов могло извлечь выгоду из этого метода. Вскоре вы обнаружите недостаток данного подхода.

Листинг 12.17. Расширение протокола Collection

```
// На этот раз мы расширяем протокол Collection, а не массив
extension Collection where Element: Equatable {
   func unique() -> [Element] {
      var uniqueValues = [Element]()
      for element in self {
         if !uniqueValues.contains(element) {
            uniqueValues.append(element)
         }
      }
   return uniqueValues
   }
}
```

Теперь каждый тип, соответствующий протоколу Collection, наследует метод unique(). Давайте проверим его.

Листинг 12.18. Тестирование метода unique ()

```
// У массива по-прежнему есть метод unique()
[3, 2, 1, 1, 2, 3].unique() // [3, 2, 1]

// То же касается и строк.
"aaaaaaabcdef".unique() // ["a", "b", "c", "d", "e", "f"]

// Или значений словаря
let uniqueValues = [1: "Waffle",
   2: "Banana",
   3: "Pancake",
   4: "Pancake",
   5: "Pancake"
].values.unique()

print(uniqueValues) // ["Banana", "Pancake", "Waffle"]
```

В случае расширения протокола Collection вместо Array речь идет о преимуществе расширения протокола по сравнению с конкретным типом.

12.5.1. Специализированное расширение

Остался один момент. Метод `unique()` не очень эффективен. Для каждого значения в коллекции нужно проверять, есть ли это значение в новом массиве, а это означает, что для каждого элемента вам нужно перебрать (возможно) все элементы массива `uniqueValues`. У нас было бы больше контроля, если бы `Element` соответствовал протоколу `Hashable`. Тогда можно было бы выполнить проверку на предмет уникальности с помощью значения хеша, используя набор (`Set`), что намного быстрее, чем поиск в массиве, потому что набор не хранит свои элементы в определенном порядке.

Чтобы поддерживать поиск с помощью `Set`, создадим еще одно расширение `unique()` для протокола `Collection`, где его элементы соответствуют `Hashable`. `Hashable` – это подпротокол протокола `Equatable`, а это означает, что Swift выбирает расширение с `Hashable` вместо расширения с `Equatable`, если это возможно (см. рис. 12.8). Например, если в массиве есть типы, соответствующие протоколу `Hashable`, Swift использует быстрый метод `unique()`, но если элементы соответствуют протоколу `Equatable`, Swift будет использовать более медленную версию.

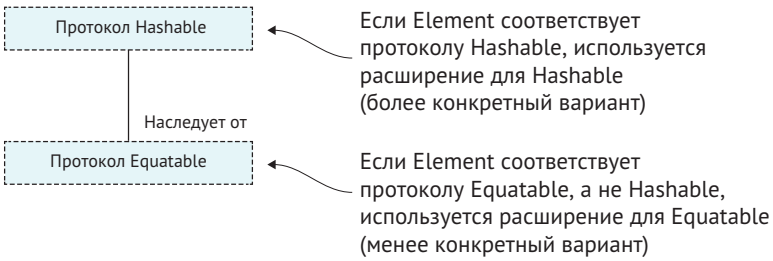


Рис. 12.8. Специализация ассоциированного типа

Во втором расширении, которое также называется `unique()`, можно поместить каждый элемент в набор для улучшения скорости.

Листинг 12.19. Расширение протокола Collection с ограничением Hashable для Element

```
// Данное расширение является дополнением, оно НЕ заменяет другое расширение.
extension Collection where Element: Hashable { ❶
   func unique() -> [Element] {
      var set = Set<Element>() ❷
      var uniqueValues = [Element]()
      for element in self {
         if !set.contains(element) { ❸
            uniqueValues.append(element)
            set.insert(element)
         }
      }
      return uniqueValues
   }
}
```

❶ Мы расширяем протокол Collection только для элементов, которые соответствуют протоколу Hashable.

❷ Создаем набор для очень быстрого (неупорядоченного) поиска элементов.

❸ Проверяем, существует ли элемент внутри набора; если нет, то можно добавить его в массив uniqueValues.

Теперь у вас есть два расширения. Одно ограничивает Element протоколом Equatable, а другое – протоколом Hashable. Swift выбирает самый специализированный вариант.

12.5.2. Недостаток расширения

Выбор абстракции иногда может быть сложным. На самом деле в этом API есть один недостаток. На данный момент Set по своей природе уже уникален, и все же он получает метод unique, потому что соответствует протоколу Collection. Можно переопределить метод для быстрого преобразования в массив (Array).

Листинг 12.20. Метод unique

```
extension Set {
   func unique() -> [Element] {
      return Array(self)
   }
}
```

Метод unique не добавляет реального значения, но, по крайней мере, теперь у нас есть быстрый способ конвертировать Set в Array. Найти баланс между расширением наименьшего общего знаменателя без ослабления API конкретных типов – это искусство.

Интересно то, что Swift снова выбирает наиболее конкретную реализацию. Swift выбирает Equatable в качестве наименьшего знаменателя, Hashable, если элементы соответствуют протоколу Hashable, а в случае с Set Swift использует его конкретную реализацию, игнорируя любые расширения метода с тем же именем в Collection.

12.6. Расширение с конкретными ограничениями

Мы также можем ограничить ассоциированные типы конкретным типом вместо ограничения протоколом. Например, предположим, у нас есть структура Article со свойством viewCount, которое отслеживает количество просмотров статьи.

Листинг 12.21. Структура Article

```
struct Article: Hashable {
   let viewCount: Int
}
```

Можно расширить Collection, чтобы получить общее количество просмотров внутри коллекции. На этот раз мы ограничиваем Element структурой Article, как показано ниже. Поскольку речь идет об ограничении конкретным типом, можно использовать оператор ==.

Листинг 12.22. Расширение Collection

```
// Неправильно
extension Collection where Element: Article { ... }

// Правильно
extension Collection where Element == Article {
   var totalViewCount: Int {
      var count = 0
      for article in self {
         count += article.viewCount
      }
   return count
   }
}
```

После этого можно получать общее количество просмотров всякий раз, когда у вас будет коллекция со статьями, будь то Array, Set или что-то еще.

Листинг 12.23. Расширение в действии

```
let articleOne = Article(viewCount: 30)
let articleTwo = Article(viewCount: 200)

// Получение общего количества для массива
let articlesArray = [articleOne, articleTwo]
articlesArray.totalViewCount // 230

// Получение общего количества для набора
let articlesSet: Set<Article> = [articleOne, articleTwo]
articlesSet.totalViewCount // 230
```

Всякий раз, когда вы создаете расширение, может быть сложно решить, насколько низко уровнем нужно идти. Конкретное расширение для типа Array достаточно для 80 % случаев, и в этом случае вам не нужно переходить в Collection. Если вы заметили, что вам нужна такая же реализация для других типов, то можете перейти на более низкий уровень, где будете расширять Collection. При этом вы будете работать с более абстрактными типами. Если вам нужно перейти еще ниже, можно обратиться к протоколу Sequence, чтобы использовать расширения для еще большего количества типов. Чтобы увидеть, как перейти на сверхнизкий уровень, давайте попробуем расширить протокол Sequence, чтобы использовать полезные расширения для множества типов.

12.7. Расширение протокола Sequence

Sequence – очень интересный протокол в плане расширения. Расширяя его, можно активировать несколько типов одновременно, таких как Set, Array, Dictionary, на ваш выбор.

Когда вы освоитесь, Sequence может помочь снизить барьер для создания расширений методов, которые вы хотели бы видеть. Можно подождать обновлений для Swift, но если вы нетерпеливы или у вас особые требования, то можете создать свои собственные обновления. Расширение протокола Sequence, в отличие от конкретного типа, такого как Array, означает, что можно активировать несколько типов одновременно.

Swift любит заимствовать концепции из Rust; эти два языка во многом очень похожи. Как насчет того, чтобы поступить так же и добавить несколько полезных методов в словарь Sequence? Расширение Sequence не просто программирование, потому что эти методы являются полезными утилитами, которые можно использовать в своих проектах.

12.7.1. Заглянем внутрь метода filter

Прежде чем приступить к расширению протокола Sequence, давайте подробнее рассмотрим несколько интересных моментов, касающихся того, как Swift делает это. Вначале метод filter принимает функцию. Эта функция является замыканием, которое мы передаем методу.

Листинг 12.24. Метод filter

```
let moreThanOne = [1,2,3].filter { (int: Int) in
   int > 1
}
print(moreThanOne) // [2, 3]
```

Глядя на сигнатуру метода filter, видно, что он принимает функцию, которая делает его функцией высшего порядка. Функция высшего порядка – это десятидолларовая концепция для названия за один доллар, указывающая на то, что функция может принимать или возвращать другую функцию. Данная функция является замыканием, которое мы передаем filter.

Также обратите внимание на то, что у метода filter есть ключевое слово rethrows, как показано в листинге 12.2. Это означает, что любые ошибки, сгенерированные из замыкания, передаются обратно вызывающей программе. Наличие метода с ключевым словом rethrows аналогично обычному распространению ошибок, за исключением того, что слово rethrows зарезервировано для функций высшего порядка, таких как filter. Преимущество тут состоит в том, что filter принимает как функции, которые могут сгенерировать ошибку, так и функции, которые этого не делают; любые ошибки должна обрабатывать вызывающая программа.

Листинг 12.25. Просмотр сигнатуры метода filter

```
public func filter(
   _ isIncluded: (Element) throws -> Bool ❶
   ) rethrows -> [Element] { ❷
      // ... Пропускаем остальную часть кода.
}
```

❶ Метод `filter` принимает другую функцию `isIncluded`. Это замыкание, которое мы передаем.

❷ Определяем ключевое слово `rethrows`.

Давайте теперь посмотрим на тело функции. Видно, что `filter` создает массив `results`, который имеет скрытый тип `ContiguousArray` (подробнее об этом через минуту), и перебирает каждый элемент с помощью низкоуровневого метода `makeIterator`. Для каждого элемента `filter` вызывает метод `isIncluded`. Это замыкание, которое мы передаем `filter`. Если `isIncluded` – также известный как *передаваемое замыкание* – возвращает значение `true`, `filter` добавляет элемент в массив `results`.

В конце `filter` преобразует `ContiguousArray` обратно в обычный массив.

Листинг 12.26. Метод filter

```
public func filter(
   _ isIncluded: (Element) throws -> Bool ❶
   ) rethrows -> [Element] {
   var result = ContiguousArray<Element>() ❷

   var iterator = self.makeIterator() ❸
   while let element = iterator.next() { ❸
      if try isIncluded(element) { ❹
         result.append(element) ❺
      }
   }

   return Array(result) ❻
}
```

❶ Метод `filter` принимает другую функцию для проверки каждого элемента.

❷ Внутри тела функции отфильтрованные результаты сохраняются в результате типа `ContiguousArray`.

❸ В методе используется низкоуровневый итерационный механизм без дополнительных затрат.

❹ Каждый элемент сопоставляется с функцией `isIncluded`.

❺ Если `isIncluded` возвращает для элемента значение `true`, элемент добавляется в результат.

❻ Результат с отфильтрованными элементами преобразуется в обычный массив и возвращается.

Примечание

В исходном коде Swift метод `filter` перенаправляет вызов другому методу `_filter`, который выполняет эту работу. Поскольку это пример, мы по-прежнему называли этот метод `filter`.

ContiguousArray

Когда вы видите здесь тип `ContiguousArray` вместо `Array`, это может показаться неуместным. Метод `filter` использует `ContiguousArray` для дополнительной производительности, что имеет смысл для такого низкоуровневого, многократно используемого метода.

`ContiguousArray` может потенциально повысить производительность, если содержит классы или протокол Objective-C; в противном случае производительность будет такой же, как у обычного типа `Array`. Но `ContiguousArray` не преобразуется в код Objective-C.

Когда `filter` делает свою работу, он возвращает обычный массив. Обычные массивы могут преобразовываться в `NSArray`, если это необходимо для Objective-C, в то время как `ContiguousArray` этого делать не может. Таким образом, использование `ContiguousArray` может помочь выжать последние капли из производительности, что имеет значение для таких низкоуровневых методов, как `filter`.

Теперь, когда вы увидели, как создать расширение для протокола `Sequence`, давайте приступим.

12.7.2. Создание метода take (while :)

В дополнение к методу `drop(while:)`, который удаляет первое число элементов, можно использовать противоположный метод под названием `take(while:)`.

Чтобы быстро освежить в памяти сведения о `drop(while:)`, напомним, что этот метод используется для удаления первого объема непригодных данных, таких как пустые строки. Мы передаем замыкание методу `drop`, который продолжает игнорировать, или отбрасывать, строки, пока мы не найдем текст. В этот момент `drop` перестает отбрасывать строки и возвращает остальную часть последовательности, как показано здесь:

Листинг 12.27. drop(while:)

```
let silenceLines =
    """

    The silence is finally over.
    """.components(separatedBy: "\n")

    let lastParts = silenceLines.drop(while: { (line) -> Bool in
```

```
        line.isEmpty
    })

print(lastParts) // ["The silence is finally over."]
```

В случае с методом `take(while:)` у нас есть другой вариант использования. Нам нужны строки до тех пор, пока мы не столкнемся с пустой строкой. Это противоположно методу `drop(while:)` и дополняет его. Прежде чем приступить к созданию метода `take(while:)`, давайте посмотрим, как он работает. Обратите внимание, что итерация продолжается до тех пор, пока не обрывается на пустой строке.

Листинг 12.28. Получаем первые строки


```
let lines =
    """
    We start with text...
    ... and then some more

    This is ignored because it came after empty space
    and more text
    """.components(separatedBy: "\n")

let firstParts = lines.take(while: { (line) -> Bool in
    !line.isEmpty
})

print(firstParts) // ["We start with text...", "... and then some more"]
```

В этой реализации мы имитируем внутреннюю часть `filter`, где используем `ContiguousArray` и `makeIterator`.

Листинг 12.29. Расширение протокола Sequence с помощью take (while :)


```
extension Sequence {
    public func take(
        while predicate: (Element) throws -> Bool ❶
        ) rethrows -> [Element] {

            var iterator = makeIterator() ❷
            var result = ContiguousArray<Element>() ❸

            while let element = iterator.next() { ❹
                if try predicate(element) { ❺
                    result.append(element)
                } else {
                    break ❻
                }
            }

            return Array(result)
```

```
        }
}
```

❶ Метод `take(while:)` также принимает закрытие для проверки каждого элемента.

❷ Мы также можем использовать метод `makeIterator`.

❸ И также можем использовать `ContiguousArray`.

❹ Итерируем каждый элемент.

❺ Если элемент соответствует `predicate`, продолжаем.

❻ Как только замыкание возвращает значение `false`, мы прекращаем итерацию и возвращаем свой результат.

12.7.3. Создание метода Inspect

Еще один полезный метод, который можно добавить в свою библиотеку, – это метод `inspect`. Он очень похож на `forEach`, с одним отличием: он возвращает последовательность. Этот метод особенно полезен для отладки конвейера, где вы объединяете операции.

Можно втиснуть его посередине операции конвейера, сделать что-то со значениями, например записать их в журнал, и продолжить работу с конвейером, как будто ничего и не было, тогда как `forEach` завершит итерацию, как показано здесь.

Листинг 12.30. Метод inspect в действии


```
["C", "B", "A", "D"]
   .sorted()
   .inspect { (string) in
      print("Inspecting: \(string)")
   }.filter { (string) -> Bool in
      string < "C"
   }.forEach {
      print("Result: \($0)")
}

// Вывод:
// Inspecting: A
// Inspecting: B
// Inspecting: C
// Inspecting: D
// Result: A
// Result: B
```

Чтобы добавить метод `inspect` в свою кодовую базу, можно воспользоваться приведенным ниже кодом. Обратите внимание, что вам не нужно использовать метод `makeIterator`, если вы этого не хотите.

Листинг 12.31. Расширение Sequence с использованием метода inspect

```
extension Sequence {
   public func inspect(
      _ body: (Element) throws -> Void ❶
   ) rethrows -> Self {
      for element in self {
         try body(element) ❷
      }
      return self ❸
   }
}
```

❶ Метод inspect также принимает функцию.

❷ Вызываем функцию body с каждым элементом.

❸ Снова возвращаем тот же тип, чтобы продолжить цепочку.

Расширение протокола Sequence означает, что мы находимся на довольно низком уровне. Не только тип Array получает дополнительные методы. Это делают и иные типы: Set, Range, zip, String и другие. Мы поверхностно коснулись данной темы. Компания Apple использует множество приемов оптимизации и определенные последовательности для дальнейшей оптимизации. Однако подход, о котором идет речь в этом разделе, должен охватывать множество случаев.

Расширение протокола Sequence с помощью методов, которые, по вашему мнению, отсутствуют, является хорошим и полезным упражнением. Какие еще расширения вы можете создать?

12.7.4. Упражнение

4

Готовы к испытанию? Создайте расширение с методом scan. Метод scan аналогичен reduce, но, кроме возврата конечного значения, он также возвращает промежуточные результаты в виде одного массива, что очень полезно для отладки метода reduce! Обязательно используйте makeIterator и ContiguousArray для повышения скорости:

```
let results = (0..<5).scan(initialResult: "") { (result: String, int:
➥ Int) -> String in
   return "\(result)\(int)"
}
print(results) // ["0", "01", "012", "0123", "01234"]

let lowercased = ["S", "W", "I", "F", "T"].scan(initialResult: "") {
➥ (result: String, string: String) -> String in
   return "\(result)\(string.lowercased())"
```

```
}
print(lowercased) // ["s", "sw", "swi", "swif", "swift"]
```

12.8. В заключение

Соблазн использования расширений и протоколов велик. Имейте в виду, что иногда конкретный тип – верный путь, прежде чем погрузиться в хитроумные абстракции. Расширение протоколов – одно из самых мощных, если не самое мощное свойство Swift, позволяющее писать код с уменьшенной связанностью, который можно использовать многократно. Найти подходящую абстракцию для решения проблемы – непростой процесс. Теперь, когда вы прочитали эту главу, я надеюсь, вы знаете, как выполнять расширение по горизонтали и вертикали, используя хитрые способы, которые помогут вам создавать краткий и чистый код.

Резюме

- Протоколы могут предоставлять реализацию по умолчанию через расширения.
- С помощью расширений можно рассматривать моделирование данных по горизонтали, тогда как с помощью подклассов вы моделируете данные более жестко по вертикали.
- Можно переопределить реализацию по умолчанию, предоставив реализацию для конкретного типа.
- Расширения протокола не могут переопределять конкретный тип.
- Через наследование протокола можно переопределить реализацию протокола по умолчанию.
- Swift всегда выбирает наиболее конкретную реализацию.
- Можно создать расширение протокола, которое будет доступно, только когда тип реализует два протокола. Это называется пересечение протоколов.
- Пересечение протоколов является более гибким, чем наследование, но оно также более абстрактно для понимания.
- При смешивании подклассов с расширениями протоколов расширение протокола и ограничение его классом является хорошей эвристикой (в отличие от расширения класса, чтобы соответствовать протоколу). Таким образом, разработчик может выбрать реализацию протокола.
- В случае с ассоциированными типами, такими как `Element` в протоколе `Collection`, Swift выбирает наиболее специализированную абстракцию, такую как `Hashable`, вместо элементов протокола `Equatable`.
- Расширение низкоуровневого протокола, такого как `Sequence`, означает, что вы используете новые методы сразу для нескольких типов.
- Swift использует специальный тип `ContiguousArray` при расширении протокола Sequence для повышения производительности.

Ответы

1

Создайте расширение, которое активирует функцию `explode()`, но только для типов, соответствующих протоколам `Mentos` и `Coke`:

```
extension Mentos where Self: Coke {
   func explode() {
      print("BOOM!")
   }
}
```

2

Каким будет вывод приведенного ниже кода?

```
"RefundableTransaction"
```

3

В чем разница между этими двумя расширениями?

```
extension UIViewController: MyProtocol {}
extension MyProtocol where Self: UIViewController {}
```

В первой строке все классы, содержащие в названии `ViewController`, и их подклассы соответствуют протоколу `MyProtocol`. Вторая строка заставляет класс соответствовать протоколу только по мере необходимости.

4

Создайте расширение с методом `scan`. Обязательно используйте `makeIterator` и `ContiguousArray` для повышения скорости:

```
extension Sequence {
   func scan<Result>(
      initialResult: Result,
      _ nextPartialResult: (Result, Element) throws -> Result
   ) rethrows -> [Result] {
      var iterator = makeIterator()

      var results = ContiguousArray<Result>()
      var accumulated: Result = initialResult
      while let element = iterator.next() {
         accumulated = try nextPartialResult(accumulated, element)
         results.append(accumulated)
      }
      return Array(results)
   }
}
```

Глава 13. Шаблоны Swift

В этой главе:

- мокирование типов с использованием протоколов и ассоциированных типов;
- условное соответствие и его преимущества;
- использование условного соответствия с пользовательскими протоколами и типами;
- обнаружение недостатков, связанных с протоколами;
- стирание типов;
- использование альтернатив протоколам.

В этой главе вы познакомитесь с полезными шаблонами, которые можно использовать в своей работе. Я позаботился о том, чтобы эти шаблоны были более ориентированы на Swift, а не на более традиционное ООП. Мы сосредоточимся на шаблонах Swift, поэтому протоколы и обобщения займут центральное место.

Вначале мы будем мокировать существующие типы с помощью протоколов. Вы увидите, как заменить сетевой API на поддельный автономный, а также как заменить этот API интерфейсом, ориентированным на тестирование. Я расскажу о маленьком трюке, связанном с ассоциированными типами, который поможет вам мокировать типы, в том числе из других фреймворков.

Условное соответствие – это мощное свойство, которое позволяет расширять типы с помощью протокола, но только при определенных условиях. Вы увидите, как расширить существующие типы с помощью условного соответствия, даже если у них есть ассоциированные типы. Затем мы пойдем дальше и создадим обобщенный тип, который мы также активируем, используя условное соответствие.

После этого мы рассмотрим недостатки, связанные с протоколами с ассоциированными типами и требованиями `Self`, которые можно неофициально рассматривать как протоколы на этапе компиляции. Вы увидите, как лучше всего справиться с этими недостатками. Сначала мы рассмотрим подход с использованием перечислений, а затем последует более сложный, но одновременно с этим и более гибкий подход, использующий *стирание типов*.

В качестве альтернативы протоколам вы увидите, как создать обобщенную структуру, которая может заменить протокол с ассоциированными типами. Данный тип будет очень гибким и отличным инструментом.

Цель этой главы – познакомить вас с новыми подходами к существующим проблемам и помочь понять протоколы и обобщения на более глубоком уровне. Давайте начнем с мокирования типов – это метод, который можно применять в различных проектах.

13.1. Внедрение зависимости

В главе 8 мы видели, как использовать протоколы в качестве интерфейса или типа, когда можно передавать различные типы методу, если эти типы соответствуют одному и тому же протоколу.

Можно пойти дальше и выполнить *инверсию управления* или *внедрение зависимости*. Это причудливые слова, которые говорят о том, что мы передаем реализацию типу. Рассматривайте это как поставку двигателя для мотоцикла.

Целью этого упражнения является создание взаимозаменяемой реализации, в которой можно переключаться между тремя сетевыми уровнями: реальным уровнем, автономным уровнем и уровнем тестирования.

Реальный уровень – это когда мы отправляем приложение для эксплуатации. Автономный уровень загружает подготовленные файлы, что полезно для работы с фиксированными ответами, когда вы управляете всеми данными, например когда бэкэнд-сервер еще не закончен или когда вам нужно продемонстрировать работу приложения без сетевого подключения. Уровень тестирования облегчает написание модульных тестов.

Присоединяйтесь!

Будет более познавательно и веселее, если вы будете знакомиться с кодом по мере прохождения этой главы. Исходный код можно скачать по адресу `http://mng.bz/4vPa`.

13.1.1. Замена реализации

Мы создадим класс с именем `WeatherAPI`, который будет получать информацию о последнем состоянии погоды. Для создания сетевых вызовов `WeatherAPI` будет использовать класс `URLSession` из фреймворка Foundation. Но мы сделаем по-другому. `WeatherAPI` принимает тип, соответствующий протоколу `Session`, потому что протокол позволяет передавать пользовательскую реализацию. Таким образом, `WeatherAPI` вызывает сетевые методы для протокола, не зная, какую конкретную реализацию он получил. Мы мокируем `URLSession`, потому что если мы знаем, как мокировать тип, который нам не принадлежит, то в случае с типами, которые принадлежат нам, это будет еще проще.

Вначале давайте определим протокол под названием `Session`, представляющий `URLSession`, а также другие сессии, такие как автономная сессия. Таким образом можно запустить приложение в демонстрационном режиме без подключения к сети, а также сеанс тестирования, который позволяет нам проверить, что наш API вызывает методы, когда мы этого ожидаем (см. рис. 13.1).

`WeatherAPI` вызывает методы в протоколе `Session`. Давайте отразим общедоступный метод для `URLSession`: создадим метод `dataTask`, который можно запустить для извлечения данных.

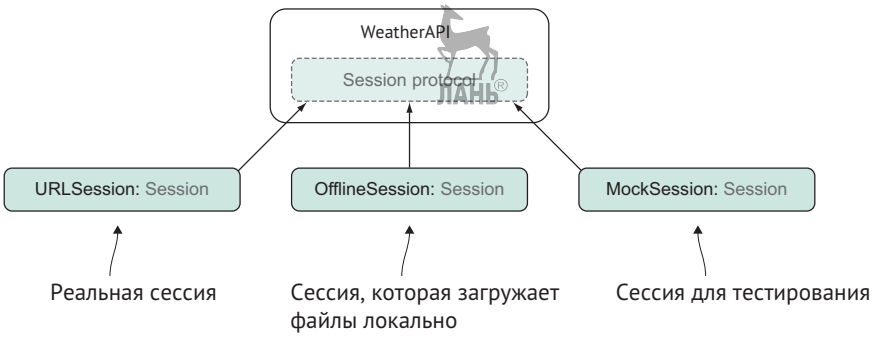


Рис. 13.1. Мокирование

В методе dataTask мы обычно получаем тип URLSessionDataTask, но мы применим маленький трюк.

Вместо возврата URLSessionDataTask или другого протокола мы вернем ассоциированный тип. Причина состоит в том, что ассоциированный тип преобразуется в конкретный тип на этапе компиляции. Это может быть URLSessionDataTask или другой тип, который может выбрать разработчик. Давайте назовем этот ассоциированный тип Task.

Листинг 13.1. Протокол Session

```
protocol Session {
   associatedtype Task ❶
   func dataTask(with url: URL, completionHandler: @escaping (Data?,
   ➥ URLResponse?, Error?) -> Void) -> Task ❷
}
```

❶ Определяем тип Task, который возвращает метод dataTask.

❷ Метод dataTask зеркально отображает метод в URLSession.

Поскольку протокол Session зеркально отображает URLSession, нам всего лишь нужно, чтобы URLSession соответствовал Session, не прибегая к написанию кода реализации. Давайте теперь расширим URLSession и сделаем так, чтобы он соответствовал Session.

Листинг 13.2. URLSession соответствует протоколу Session

```
extension URLSession: Session {}
```

13.1.2. Передача пользовательской версии Session

Теперь мы можем создать класс WeatherAPI и определить обобщение с именем Session, чтобы обеспечить возможность реализации, поддерживающей замену. Мы почти у цели, но пока еще не можем вызвать метод resume для Session.Task.

Листинг 13.2. Класс WeatherAPI

```
final class WeatherAPI<S: Session> { ❶
```

```
    let session: S ❷

    init(session: S) { ❸
        self.session = session
    }

    func run() {
        guard let url = URL(string: "https://www.someweatherstartup.com")
        ➥ else {
            fatalError(«Could not create url»)
        }

        let task = session.dataTask(with: url) { (data, response, error) in ❹
            // Работаем с полученными данными.
        }

        task.resume() // Не работает, task имеет тип S.Task ❺
    }
}

let weatherAPI = WeatherAPI(session: URLSession.shared) ❻
weatherAPI.run()
```

❶ Класс WeatherAPI принимает обобщение Session.

❷ Сохраняем Session в свойстве.

❸ В инициализаторе мы передаем и сохраняем Session.

❹ Можно вызвать метод dataTask.

❺ К сожалению, пока еще нельзя вызвать метод resume() для Task.

❻ Передаем URLSession в WeatherAPI через инициализатор.

Мы пытаемся запустить task, но не можем. Ассоциированный тип Task, который возвращает dataTask(with: completionHandler:), имеет тип Session.Task, у которого нет никаких методов. Давайте исправим это.

13.1.3. Ограничение ассоциированного типа

У URLSessionDataTask есть метод resume(), а у Session.Task – нет. Чтобы это исправить, мы введем новый протокол DataTask, который содержит метод resume(). Затем мы ограничим Session.Task протоколом DataTask.

Чтобы завершить реализацию, нужно сделать так, чтобы URLSessionDataTask также соответствовал протоколу DataTask (см. рис. 13.2).

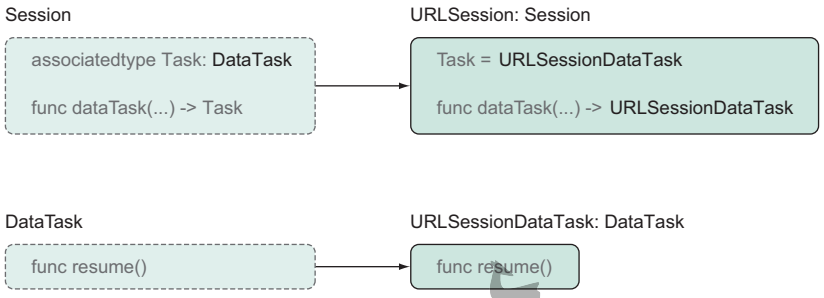


Рис. 13.2. Зеркальное отображение URLSession

Давайте создадим протокол DataTask и ограничим им Session.Task.

Листинг 13.4. Создание протокола DataTask

```
protocol DataTask { ❶
   func resume() ❷
}

protocol Session {
   associatedtype Task: DataTask ❸
   func dataTask(with url: URL, completionHandler: @escaping (Data?,
   ➥ URLResponse?, Error?) -> Void) -> Task
}
```

❶ Вводим новый протокол DataTask.

❷ DataTask теперь может использовать метод resume(), как и URLSession.

❸ Ассоциированный тип теперь соответствует DataTask, поэтому dataTask(with: completionHandler:) возвращает задачу (task), которую можно возобновить.

Для завершения настройки делаем так, чтобы URLSessionDataTask соответствовал протоколу DataTask, чтобы все снова компилировалось.

Листинг 13.5. Реализация DataTask

```
extension URLSessionDataTask: DataTask {} ❶
```

❶ URLSessionDataTask теперь соответствует протоколу DataTask.

Все снова хорошо. Теперь URLSession возвращает тип, соответствующий протоколу DataTask. На этом этапе весь код компилируется. Мы можем вызвать свой API с реальным URLSession для выполнения запроса.

13.1.4. Замена реализации

Можно передать URLSession своему WeatherAPI, но реальная сила заключается в возможности поменять реализацию. У нас есть фальшивый URLSession, который мы назовем OfflineURLSession. Он загружает локальные файлы, вместо того чтобы создавать реальное сетевое подключение. Таким образом, мы не зависим

от бэкэнда, если нам нужно поэкспериментировать с классом WeatherAPI. Каждый раз при вызове метода dataTask(with url:) класс OfflineURLSession создает OfflineTask, который загружает локальный файл, как показано в приведенном ниже листинге.

Листинг 13.6. OfflineTask

```
final class OfflineURLSession: Session { ❶
   var tasks = [URL: OfflineTask]() ❷

   func dataTask(with url: URL, completionHandler: @escaping (Data?,
   ➥ URLResponse?, Error?) -> Void) -> OfflineTask {
      let task = OfflineTask(completionHandler: completionHandler) ❸
      tasks[url] = task
      return task
   }
}

enum ApiError: Error { ❹
   case couldNotLoadData
}

struct OfflineTask: DataTask {
   typealias Completion = (Data?, URLResponse?, Error?) -> Void ❺
   let completionHandler: Completion

   init(completionHandler: @escaping Completion) { ❶
      self.completionHandler = completionHandler
   }

   func resume() { ❼
      let url = URL(fileURLWithPath: "prepared_response.json") ❼
      let data = try! Data(contentsOf: url) ❼
      completionHandler(data, nil, nil) ❼
   }
}
```

❶ OfflineURLSession соответствует протоколу Session и имитирует URL Session.

❷ Сохраняем tasks, чтобы убедиться, что они не были освобождены напрямую.

❸ OfflineURLSession создает и возвращает типы OfflineTask.

❹ Нам нужна ошибка для возврата. Также можно полностью имитировать ошибку URLSession.

❺ Для удобства мы определяем псевдоним типа для имитации замыкания Completion.

❻ Замыкание Completion из OfflineURLSession передается и сохраняется OfflineTask.

❼ После запуска OfflineTask выполняется загрузка локальных данных в формате JSON и вызывается обработчик completionHandler, который изначально был передан OfflineURLSession.

Примечание

При более зрелой реализации tasks также были бы освобождены и была бы дана возможность дополнительной конфигурации загрузки файлов.

Теперь, когда у вас есть несколько реализаций, соответствующих протоколу Session, можно начать менять их, не затрагивая остальную часть кода. Мы можем создать эксплуатационную версию WeatherAPI или автономную.

Листинг 13.7. Обмен реализаций

```
let productionAPI = WeatherAPI(session: URLSession.shared)
let offlineApi = Weath+rAPI(session: OfflineURLSession())
```

С помощью протоколов можно менять реализацию, например передавать различные сессии в WeatherAPI. При использовании ассоциированного типа можно сделать так, чтобы он представлял другой тип внутри протокола, что может упростить имитацию конкретного типа. Не всегда нужно использовать ассоциированные типы, но это помогает, когда нет возможности создавать экземпляры определенных типов, например когда вам нужно мокировать код из сторонних фреймворков.

Это все, что нужно. Теперь WeatherAPI работает с эксплуатационным или автономным слоем, что дает большую гибкость. При такой настройке тестирование также не должно вызывать вопросов. Перейдем к созданию слоя тестирования, чтобы у нас была возможность правильно протестировать WeatherAPI.

13.1.5. Модульное тестирование и мокирование с использованием ассоциированных типов

При наличии реализации, поддерживающей замену, можно перейти к реализации, которая поможет при тестировании. Можно создать определенный тип, соответствующий протоколу Session. В качестве примера мы создадим класс MockSession и структуру MockTask, которые смотрят, вызываются ли определенные URL-адреса.

Листинг 13.8. Создание MockTask и MockSession

```
class MockSession: Session { ❶
   let expectedURLs: [URL]
   let expectation: XCTestExpectation

   init(expectation: XCTestExpectation, expectedURLs: [URL]) {
      self.expectation = expectation
```

```
        self.expectedURLs = expectedURLs
    }

    func dataTask(with url: URL, completionHandler: @escaping (Data?,
    ➥ URLResponse?, Error?) -> Void) -> MockTask {
        return MockTask(expectedURLs: expectedURLs, url: url, expectation:
        ➥ expectation) ❷
    }
}

struct MockTask: DataTask {
    let expectedURLs: [URL]
    let url: URL
    let expectation: XCTestExpectation ❸

    func resume() {
        guard expectedURLs.contains(url) else {
            return
        }
        self.expectation.fulfill()
    }
}
```

❶ MockSession также соответствует протоколу Session.

❷ MockSession возвращает MockTask.

❸ MockTask содержит ожидания, которым должен соответствовать наш тест.

Теперь, если нам нужно протестировать свой API, можно проверить, что вызываются ожидаемые URL-адреса.

Листинг 13.9. Тестирование API

```
class APITestCase: XCTestCase {

    var api: API<MockSession>! ❶

    func testAPI() {
        let expectation = XCTestExpectation(description: «Expected
        ➥ someweatherstartup.com») ❷
        let session = MockSession(expectation: expectation, expectedURLs:
        ➥ [URL(string: «www.someweatherstartup.com»)!]) ❸
        api = API(session: session)
        api.run() ❹
        wait(for: [expectation], timeout: 1) ❺
    }
}
```

```
let testcase = APITestCase()
testcase.testAPI()
```

❶ Создаем API и определяем его как API, у которого есть MockSession.

❷ Создаем ожидание, которое должно быть оправдано.

❸ Создаем сессию с ожиданием.

❹ Как только API будет запущен, мы надеемся, что ожидание будет оправдано.

❺ Проверяем, оправдано ли ожидание. Время ожидания составляет одну секунду.

13.1.6. Использование типа Result

Поскольку мы используем протокол, то можно воспользоваться реализацией по умолчанию для всех сеансов с помощью расширения протокола. Возможно, вы заметили, что мы использовали обычный метод обработки ошибок Session. Но у нас есть мощный тип Result, доступный из диспетчера пакетов Swift, как описано в главе 11. Можно расширить протокол Session и выбрать вариант, где используется тип Result, как показано в приведенном ниже листинге. Таким образом, все типы, соответствующие Session, смогут вернуть Result.

Листинг 13.10. Расширение протокола Session с типом Result

```
protocol Session {
   associatedtype Task: DataTask

   func dataTask(with url: URL, completionHandler: @escaping (Data?,
   ➥ URLResponse?, Error?) -> Void) -> Task

   func dataTask(with url: URL, completionHandler: @escaping (Result<Data,
   ➥ AnyError>) -> Void) -> Task ❶
}

extension Session {
   func dataTask(with url: URL, completionHandler: @escaping (Result<Data,
   ➥ AnyError>) -> Void) -> Task { ❷
      return dataTask(with: url, completionHandler: { (data, response,
         error) in ❸
         if let error = error {
            let anyError = AnyError(error)
            completionHandler(Result.failure(anyError))
         } else if let data = data {
            completionHandler(Result.success(data))
         } else {
            fatalError()
         }
```

```
        })
    }
}
```

❶ Добавляем новый метод в протокол Session.

❷ Можно предоставить реализацию по умолчанию, которая использует Result.

❸ Реализация по-прежнему вызывает для нас обычный метод dataTask и преобразует ответ в тип Result.

Теперь разработчики, реализующие протокол Session, включая класс URLSession, могут возвращать тип Result.

Листинг 13.11. Сессии, получающие тип Result

```
URLSession.shared.dataTask(with: url) { (result: Result<Data, AnyError>) in
    // ...
}

OfflineURLSession().dataTask(with: url) { (result: Result<Data, AnyError>) in
     //...
}
```

Благодаря силе протоколов у нас есть возможность переходить от одной реализации к другой, переключаясь между эксплуатацией, отладкой и тестированием. С помощью расширений также можно использовать Result.

13.1.7. Упражнение

1

Посмотрите на эти типы. Можно ли сделать структуру WaffleHouse тестируемой, чтобы проверить, что Waffle была подана:

```
struct Waffle {}

class Chef {
    func serve() -> Waffle {
        return Waffle()
    }
}

struct WaffleHouse {
    let chef = Chef()
    func serve() -> Waffle {
        return chef.serve()
    }
}
```

```
let waffleHouse = WaffleHouse()
let waffle = waffleHouse.serve()
```

13.2. Условное соответствие

Условное соответствие – неоспоримое свойство, представленное в Swift версии 4.1. С помощью условного соответствия можно заставить тип соответствовать протоколу, но только при определенных условиях. В главе 7 кратко описывается условное соответствие, но давайте воспользуемся этой возможностью, чтобы получше познакомиться с ним и увидеть более продвинутые варианты использования. После этого раздела вы будете точно знать, когда и как применять условное соответствие. Приступим!

13.2.1. Бесплатная функциональность

Самый простой способ увидеть условное соответствие в действии – это создать структуру, реализовать протоколы `Equatable` или `Hashable`, и, не прибегая к реализации какого-либо из методов протокола, мы бесплатно получим соответствие этим двум протоколам.

Ниже приводится структура `Movie`. Обратите внимание, что мы уже можем сравнивать фильмы, всего лишь делая так, чтобы `Movie` соответствовала протоколу `Equatable`, без реализации необходимой функции ==; это тот же метод, который мы применяли при работе с `Pair` в главе 7.

Листинг 13.12. Автопротокол Equatable

```
struct Movie: Equatable {
   let title: String
   let rating: Float
}

let movie = Movie(title: "The princess bride", rating: 9.7)
movie == movie // Истинно. Мы уже можем выполнять сравнение,
               // не прибегая к реализации протокола Equatable
```


Переопределение вручную

Мы по-прежнему можем реализовать функцию == из протокола `Equatable`, если хотим предоставить собственную логику.

Возможно автоматическое получение типа, соответствующего протоколу `Equatable`, поскольку все свойства соответствуют этому протоколу. Swift синтезирует это бесплатно для некоторых протоколов, таких как `Equatable` и `Hashable`, но, например, в случае с `Comparable` или протоколами, которые вы представляете самостоятельно, это будет невозможно.

Предупреждение

К сожалению, Swift не синтезирует методы для классов.

13.2.2. Условное соответствие для ассоциированных типов

Еще один доступный способ увидеть условное соответствие в действии – посмотреть на `Array` и его тип `Element`, представляющий элемент внутри массива. `Element` – это ассоциированный тип из протокола Sequence, который использует `Array`, как было описано в главе 9.

Представим, что у нас есть протокол `Track`, обозначающий дорожку, используемую в звуковом программном обеспечении, такую как звуковой файл в формате wav или эффект искажения. Для реализации этого протокола можно использовать `AudioTraca`; он может воспроизводить аудиофайлы по определенному URL-адресу.

Листинг 13.13. Протокол Track

```
protocol Track {
   func play()
}

struct AudioTrack: Track {
   let file: URL
   func play() {
      print("playing audio at \(file)")
   }
}
```


Если у нас есть массив дорожек и нам нужно одновременно воспроизвести эти дорожки для музыкальной композиции, можно расширить `Array` только там, где `Element` соответствует протоколу `Track`, а затем ввести метод `play()`. Таким образом, можно инициировать этот метод для всех элементов `Track` внутри массива. Однако у данного подхода есть один недостаток, с которым мы разберемся чуть позже.

Листинг 13.14. Расширяем Array (имеется недостаток)

```
extension Array where Element: Track {
   func play() {
      for element in self {
         element.play()
      }
   }
}

let tracks = [
```

```
    AudioTrack(file: URL(fileURLWithPath: "1.mp3")),
    AudioTrack(file: URL(fileURLWithPath: "2.mp3"))
]
tracks.play() // Используем метод play()
```

У этого подхода есть недостаток. Array не соответствует протоколу Track; он всего лишь реализует метод с таким же именем, как и у метода внутри протокола, а именно метод play().

Поскольку Array не соответствует протоколу, мы больше не можем вызывать метод play() для вложенного массива. Кроме того, если у вас есть функция, принимающая Track, вы также не можете передать Array с типами, которые соответствую протоколу Track.

Листинг 13.15. Нельзя использовать Array в качестве типа Track

```
let tracks = [
    AudioTrack(file: URL(fileURLWithPath: "1.mp3")),
    AudioTrack(file: URL(fileURLWithPath: "2.mp3"))
]

// Если массив является вложенным, мы больше не можем вызывать метод play().
[tracks, tracks].play() // ошибка: тип выражения неоднозначен без
// дополнительного контекста.

// Либо нельзя передать массив, если что-либо находится в ожидании
// протокола Track.
func playDelayed<T: Track>(_ track: T, delay: Double) {
    // ... Пропускаем часть кода
}

playDelayed(tracks, delay: 2.0) // Тип аргумента '[AudioTrack]' не
    соответствует ожидаемому типу 'Track'
```

13.2.3. Делаем так, чтобы Array условно соответствовал пользовательскому протоколу

Начиная с Swift версии 4.1 эту проблему можно решить, и Array будет соответствовать пользовательскому протоколу. Это возможно, но только при условии, что его элементы соответствуют протоколу Track. Единственное отличие от предыдущего примера состоит в том, что мы добавляем :Track после Array.

Листинг 13.16. Делаем так, чтобы Array условно соответствовал пользовательскому протоколу

```
// Прежде. Условное соответствие отсутствует.
extension Array where Element: Track {
    // ... Пропускам часть кода
```

```
}

// После. Есть условное соответствие.
extension Array: Track where Element: Track {
   func play() {
      for element in self {
         element.play()
      }
   }
}
```

Предупреждение

Если вы делаете так, чтобы тип условно соответствовал протоколу, используя ограничение, например `where Element: Track`, вам необходимо предоставить реализацию самостоятельно. Swift не будет делать это за вас.

Теперь `Array` – это тип, соответствующий протоколу `Track`. Можно передать его функциям, ожидающим `Track`, или вложить массивы с другими данными, и вы по-прежнему сможете вызывать метод `play()`, как показано здесь:

Листинг 13.17. Условное соответствие в действии

```
let nestedTracks = [
   [
      AudioTrack(file: URL(fileURLWithPath: "1.mp3")),
      AudioTrack(file: URL(fileURLWithPath: "2.mp3"))
   ],
   [
      AudioTrack(file: URL(fileURLWithPath: "3.mp3")),
      AudioTrack(file: URL(fileURLWithPath: "4.mp3"))
   ]
]

// Вложенность работает.
nestedTracks.play()

// И вы можете передать этот массив функции, ожидающей Track!
playDelayed(tracks, delay: 2.0)
```

13.2.4. Условное соответствие и обобщения

Вы видели, как расширить `Array` с помощью ограничения для ассоциированного типа. Прелесть состоит в том, что мы также можем использовать ограничения при работе с параметрами обобщенного типа.

В качестве примера давайте возьмем тип `Optional`, поскольку у него есть только обобщенный тип `Wrapped` и нет ассоциированных типов. Мы можем заставить

Optional реализовать Track с помощью условного соответствия для его обобщения следующим образом:

Листинг 13.18. Расширение Optional

```
extension Optional: Track where Wrapped: Track {
   func play() {
      switch self {
         case .some(let track): ❶
            track.play() ❶
         case nil:
            break // Ничего не делаем
      }
   }
}
```

❶ Если в Optional есть значение Track, можно вызвать для него метод play.

Теперь Optional соответствует Track, но только если его внутреннее значение соответствует Track. Без условного соответствия мы бы уже могли вызывать метод play() для опционалов, соответствующих протоколу.

Листинг 13.19. Вызов метода play () для опционала

```
let audio: AudioTrack? = AudioTrack(file: URL(fileURLWithPath: "1.mp3"))
audio?.play() ❶
```

❶ Вызов метода для опционала.

Но теперь Optional официально также является Track, что позволяет нам передавать его типам и функциям, ожидающим Track. Другими словами, используя условное соответствие, можно передать опционал AudioTrack? методу, ожидающему Track, который не является опционалом.

Листинг 13.20. Передача опционала playDelayed

```
let audio: AudioTrack? = AudioTrack(file: URL(fileURLWithPath: «1.mp3»))
playDelayed(audio, delay: 2.0) ❶
```

❶ playDelayed ожидает трек. Теперь он принимает опционал.

13.2.5. Условное соответствие для типов

Условное соответствие проявляется, когда вы работаете с обобщенными типами, такими как Array, Optional или своим собственным. Условное соответствие набирает силу, когда у вас есть обобщенный тип, хранящий внутренний тип, и вам нужно, чтобы обобщенный тип имитировал поведение внутреннего типа (см. рис. 13.3). Ранее мы видели, как Array становится Track, если его элементы соответствуют протоколу.

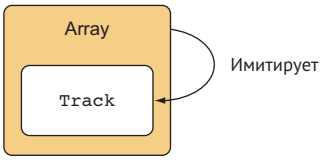


Рис. 13.3. Тип, имитирующий внутренний тип

Давайте посмотрим, как это работает, если мы самостоятельно создадим обобщенный тип. Одним из таких типов может быть CachedValue. CachedValue – это класс, который хранит значение. По истечении определенного периода времени значение обновляется замыканием, которое содержится в CachedValue.

Преимущество CachedValue состоит в том, что он может на некоторое время кешировать затратные вычисления перед обновлением. Это может помочь ограничить многократную загрузку больших файлов или повторное выполнение затратных вычислений.

Например, можно хранить значение со значением времени жизни до двух секунд. Если мы запросим значение через три секунды или более, замыкание будет вызвано снова, чтобы обновить значение, как показано здесь.

Листинг 13.21. CachedValue в действии

```
let simplecache = CachedValue(timeToLive: 2, load: { () -> String in ❶
   print("I am being refreshed!") ❷
   return "I am the value inside CachedValue"
})

// Будет выведено: "I am being refreshed!"
simplecache.value // "I am the value inside CachedValue" ❸
simplecache.value // "I am the value inside CachedValue"
sleep(3) // wait 3 seconds ❹

// Будет выведено: "I am being refreshed!" ❹
simplecache.value // "I am the value inside CachedValue"
```

❶ Создаем CachedValue и передаем его пользовательскому замыканию в инициализаторе. Обратите внимание, что мы устанавливаем значение timeToLive на две секунды.

❷ Если значение получено, мы выводим строку для отладки.

❸ Значение кешируется. Можно запросить его и получить строку.

❹ Через две секунды замыкание вызывается снова и дает новую строку для CachedValue, чтобы ее можно было сохранить.

Давайте посмотрим на внутреннее устройство CachedValue, чтобы увидеть, как он работает, а затем перейдем к условному соответствию.

Листинг 13.22. Внутри CachedValue

```
final class CachedValue<T> { ❶
```

```
    private let load: () -> T ❷
    private var lastLoaded: Date

    private var timeToLive: Double
    private var currentValue: T

    public var value: T {
       let needsRefresh = abs(lastLoaded.timeIntervalSinceNow) > timeToLive ❸
       if needsRefresh {
          currentValue = load() ❹
          lastLoaded = Date()
       }
       return currentValue ❺
    }

    init(timeToLive: Double, load: @escaping (() -> T)) {
       self.timeToLive = timeToLive
       self.load = load
       self.currentValue = load()
       self.lastLoaded = Date()
    }
}
```

❶ CachedValue может хранить все, что угодно, представленное как тип T.

❷ Сохраняем замыкание load, которое используется для обновления значения внутри.

❸ При каждом доступе к значению мы проверяем, требуется ли обновление.

❹ Если нужно выполнить обновление, мы обновляем значение и устанавливаем дату lastLoaded.

❺ Возвращаем значение.

Как сделать тип условно соответствующим

Здесь начинается самое интересное. Теперь, когда у нас есть обобщенный тип, мы можем занять свои начальные позиции и приступить к использованию условного соответствия. Таким образом, CachedValue отражает возможности его значения внутри. Например, можно сделать так, чтобы CachedValue соответствовал протоколу Equatable, если его значение внутри соответствует Equatable. Можно сделать так, чтобы CachedValue соответствовал протоколу Hashable, если его значение внутри соответствует Hashable, и то же самое можно сделать в случае с протоколом Comparable (см. рис. 13.4).

Можно добавить расширения для всех протоколов, которые можно себе представить (если это имеет смысл). Готовы? Поехали!

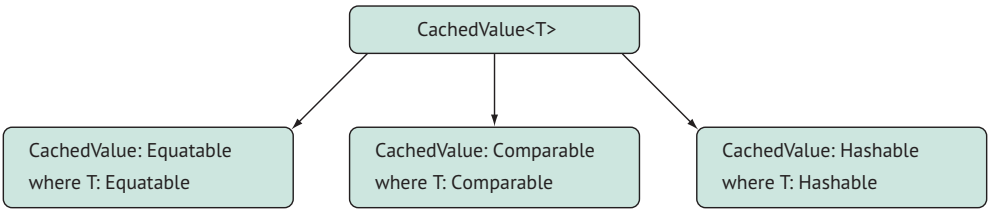


Рис. 13.4. Приведение CachedValue в условное соответствие протоколам Equatable, Hashable и Comparable

Листинг 13.23. Условное соответствие для CachedValue

```
// Соответствие протоколу Equatable
extension CachedValue: Equatable where T: Equatable {
   static func == (lhs: CachedValue, rhs: CachedValue) -> Bool {
      return lhs.value == rhs.value
   }
}

// Соответствие протоколу Hashable
extension CachedValue: Hashable where T: Hashable {
   func hash(into hasher: inout Hasher) {
      hasher.combine(value)
   }
}

// Соответствие протоколу Comparable
extension CachedValue: Comparable where T: Comparable {
   static func <(lhs: CachedValue, rhs: CachedValue) -> Bool {
      return lhs.value < rhs.value
   }
   static func ==(lhs: CachedValue, rhs: CachedValue) -> Bool {
      return lhs.value == rhs.value
   }
}
```

Это только начало. Вы можете использовать свои собственные протоколы и множество других протоколов, которые есть в Swift.

После этого CachedValue принимает свойства своего внутреннего типа. Давайте посмотрим, соответствует ли CachedValue протоколам Equatable, Hashable и Comparable надлежащим образом.

Листинг 13.24. Теперь CachedValue соответствует всем трем протоколам

```
let cachedValueOne = CachedValue(timeToLive: 60) {
   // Выполняем затратную операцию.
   // Например, вычисляем цель жизни.
   return 42
```

```
}

let cachedValueTwo = CachedValue(timeToLive: 120) {
   // Выполняем еще одну затратную операцию.
   return 1000
}

// Equatable: Можно провести проверку на предмет равенства.
cachedValueOne == cachedValueTwo
// Comparable: Можно сравнить два кешированных значения.
cachedValueOne > cachedValueTwo

let set = Set(arrayLiteral: cachedValueOne, cachedValueTwo) // Hashable:
➥ You can store CachedValue in a set
```

Можно продолжить. Например, мы можем заставить `CachedValue` реализовать `Track` или любые другие пользовательские реализации.

Условное соответствие лучше всего работает при хранении наименьшего общего знаменателя внутри обобщения, а это означает, что в этом случае не следует добавлять слишком много ограничений для `T`.

Если обобщенный тип по умолчанию не слишком ограничен, будет проще расширить тип с помощью условного соответствия. В случае с `CachedValue` `T` не ограничен, поэтому помещаются все типы, а затем мы добавляем функциональность с помощью условного соответствия. Таким образом, и простые, и расширенные типы помещаются в `CachedValue`. Если вы ограничите `T` 10 протоколами, внутри `CachedValue` поместится очень мало типов, и тогда от условного соответствия будет мало пользы.

13.2.6. Упражнение

2

Какая польза от обобщения, имеющего несколько ограничений при применении условного соответствия?

3

Сделайте так, чтобы `CachedValue` соответствовал пользовательскому протоколу `Track`, о котором идет речь в этой главе.

13.3. Что делать с недостатками

Протоколы – это рецепт отношений «любовь–ненависть». Это фантастический инструмент, но иногда то, что «должно всего-навсего работать», просто невозможно.

Например, распространенной проблемой для разработчиков Swift является желание хранить типы, соответствующие протоколу `Hashable`. Вы очень скоро обнаружите, что это не так просто, как кажется.

Представим себе, что мы моделируем игровой сервер для игры в покер. У нас есть протокол PokerGame, а типы StudPoker и TexasHoldem соответствуют этому протоколу. В приведенном ниже листинге обратите внимание на то, что PokerGame соответствует протоколу Hashable, поэтому мы можем хранить игры внутри наборов и использовать их в качестве словарных ключей.

Листинг 13.25. PokerGame

```
protocol PokerGame: Hashable {
   func start()
}

struct StudPoker: PokerGame {
   func start() {
      print("Starting StudPoker")
   }
}
struct TexasHoldem: PokerGame {
   func start() {
      print("Starting Texas Holdem")
   }
}
```

Можно хранить типы StudPoker и TexasHoldem в массивах, наборах и словарях. Но если вам нужно смешивать и сопоставлять различные типы PokerGame в качестве ключей в словаре или в массиве, вы столкнетесь с проблемой.

Например, допустим, нам нужно сохранить количество активных игроков для каждой игры в словаре, где PokerGame – это ключ.

Листинг 13.26. PokerGame в качестве ключа выдает ошибку

```
// Этот вариант не сработает!
var numberOfPlayers = [PokerGame: Int]()

// Ошибка, которую генерирует компилятор Swift:
error: using 'PokerGame' as a concrete type conforming to protocol 'Hashable'
is not supported
var numberOfPlayers = [PokerGame: Int]()
```

Звучит правдоподобно для хранения протокола Hashable в качестве ключа словаря. Но компилятор выдает ошибку, потому что нельзя использовать этот протокол в качестве конкретного типа. Hashable представляет собой подпротокол Equatable, и, следовательно, у него есть требования Self, что не позволяет хранить Hashable во время выполнения. Swift хочет, чтобы на этапе компиляции Hashable был преобразован в конкретный тип. Однако протокол не является конкретным типом.

Можно хранить один из типов PokerGame в качестве словарного ключа, например [TexasHoldem: Int], но тогда у вас не будет возможности смешивать их.

Также можно попробовать использовать обобщения, которые преобразуются в конкретный тип. Но это тоже не сработает.

Листинг 13.27. Попытка смешать игры

```
func storeGames<T: PokerGame>(games: [T]) -> [T: Int] {
   /// ... Пропускаем остальную часть кода
}
```

К сожалению, это обобщение может преобразоваться в один тип для каждой функции, например:

Листинг 13.28. Преобразование обобщения

```
func storeGames(games: [TexasHoldem]) -> [TexasHoldem: Int] {
   /// ... Пропускаем остальную часть кода
}
func storeGames(games: [StudPoker]) -> [StudPoker: Int] {
   /// ... Пропускаем остальную часть кода
}
```

Опять же, нельзя просто так смешивать и сопоставлять типы `PokerGame` в одном контейнере, таком как словарь.

Давайте используем два разных подхода для решения этой проблемы. Первый включает в себя перечисление, а второй – стирание типов.

13.3.1. Как избежать использования перечисления

Вместо того чтобы использовать протокол `PokerGame`, мы рассмотрим возможность применения конкретного типа. Создание суперкласса `PokerGames` заманчиво, но мы уже встречались с недостатками наследования на основе классов. Вместо этого давайте воспользуемся перечислением.

Как показано в листинге 13.29, вначале `PokerGame` становится перечислением и сохраняет `StudPoker` и `TexasHoldem` в каждом кейсе в качестве ассоциированных значений. Потом мы делаем так, чтобы `StudPoker` и `TexasHoldem` соответствовали протоколу `Hashable`, потому что `PokerGame` больше не является протоколом. Мы также делаем так, чтобы и `PokerGame` соответствовал протоколу `Hashable` для хранения его в словаре. Таким образом, у нас есть конкретные типы, и мы можем хранить игры внутри словаря.

Листинг 13.29. PokerGame

```
enum PokerGame: Hashable {
   case studPoker(StudPoker)
   case texasHoldem(TexasHoldem)
}
struct StudPoker: Hashable {
```

```
    // ... Реализация опущена
}
struct TexasHoldem: Hashable {
    // ... Реализация опущена
}

// Теперь все работает
var numberOfPlayers = [PokerGame: Int]()
```

Примечание

Обратите внимание, что Swift генерирует реализацию `Hashable`, избавляя нас от написания шаблонного кода.

13.3.2. Стирание типов

Перечисления – быстрый и безболезненный способ решения проблемы, когда у вас нет возможности использовать протоколы в качестве типа. Но перечисления плохо масштабируются; поддержание большого количества кейсов может стать проблемой. Более того, на момент написания этих строк нельзя позволить другим расширять перечисления с помощью новых кейсов, если вам нужно создать общедоступный API. Кроме того, протоколы – это способ добиться внедрения зависимостей, который фантастически подходит для тестирования.

Если вы хотите придерживаться протокола, то также можете рассмотреть использование техники, которая носит название *стирание типов*. Иногда ее еще называют *упаковкой*. Используя стирание типов, можно переместить протокол на этапе компиляции во время выполнения, обернув тип в тип контейнера. Таким образом, у вас могут быть протоколы с требованиями `Self`, например `Hashable` или `Equatable`, или протоколы с ассоциированными типами в качестве ключей словаря.

Прежде чем начать, я должен предупредить вас. Стирание типов в Swift – такое же веселое занятие, как и езда на автомобиле в час пик в Лос-Анджелесе. Стирание типов показывает, что протоколы Swift еще не полностью сформировались, поэтому не расстраивайтесь, если это покажется вам сложным. Стирание типов – это обходной путь, который можно использовать, пока инженеры Swift не предложат нативное решение.

Примечание

Внутри исходного кода Swift также используется стирание типов. Вы не единственные, кто сталкивается с этой проблемой!

Мы введем новую структуру, которая называется `AnyPokerGame`. Этот конкретный тип оборачивает тип, соответствующий протоколу `PokerGame`, и скрывает внутренний тип. `AnyPokerGame` соответствует `PokerGame` и перенаправляет методы в сохраняемый тип (см. рис. 13.5).

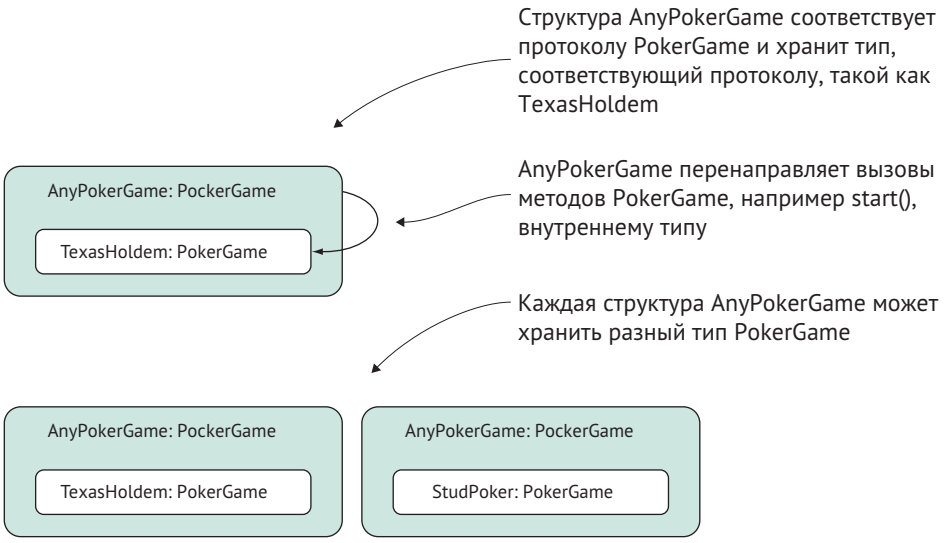


Рис. 13.5. Стирание типов

Поскольку AnyPokerGame – это конкретный тип, а именно структура, можно использовать ее для хранения различных игр внутри одного массива, набора или словарей и других мест (см. рис. 13.6).

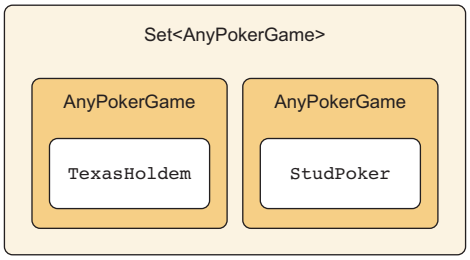


Рис. 13.6. Хранение AnyPokerGame внутри набора

В коде видно, что AnyPokerGame оборачивает PokerGame. Обратите внимание на то, что мы оборачиваем StudPoker и TexasHoldem в один массив и набор, а также как ключи словаря. Задача решена!

Листинг 13.30. AnyPokerGame в действии

```
let studPoker = StudPoker()
let holdEm = TexasHoldem()

// Можно смешивать несколько игр в массиве.
let games: [AnyPokerGame] = [
   AnyPokerGame(studPoker),
   AnyPokerGame(holdEm)
]

games.forEach { (pokerGame: AnyPokerGame) in
```

```
    pokerGame.start()
}

// Вы также можете хранить их внутри набора
let setOfGames: Set<AnyPokerGame> = [
    AnyPokerGame(studPoker),
    AnyPokerGame(holdEm)
]

// Вы даже можете использовать игры в качестве ключей!
var numberOfPlayers = [
    AnyPokerGame(studPoker): 300,
    AnyPokerGame(holdEm): 400
]
```

Примечание

Помните AnyError из главы 11? Он также выполняет стирание типов. То же делает и итератор AnyIterator, который мы видели в главе 9.

Создание AnyPokerGame

Теперь, когда мы увидели AnyPokerGame в действии, пришло время создать ее. Мы начнем с введения AnyPokerGame, которая соответствует протоколу PokerGame. В инициализаторе мы передаем тип PokerGame, ограниченный обобщением, а затем сохраняем метод start из переданного типа PokerGame в закрытом свойстве _start. После вызова start() мы перенаправляем его в сохраненный метод _start().

Листинг 13.31. Структура AnyPokerGame

```
struct AnyPokerGame: PokerGame { ❶
    init<Game: PokerGame>(_ pokerGame: Game) { ❷
        _start = pokerGame.start ❸
    }

    private let _start: () -> Void ❹

    func start() {
        _start() ❺
    }
}
```

❶ Вводим структуру AnyPokerGame, соответствующую протоколу PokerGame.

❷ Принимаем обобщение PokerGame.

❸ Сохраняем метод start в свойстве _start.

❹ Тут мы определяем свойство _start.

❶ Всякий раз при вызове метода start() мы вызываем внутреннее свойство _start().

Чтобы стереть тип, нужно зеркально отображать каждый метод из протокола и сохранить их в своих функциях в качестве свойств. Но нам повезло, потому что у PokerGame только один метод.

Мы почти закончили. Поскольку PokerGame также соответствует протоколу Hashable, мы должны сделать так, чтобы и AnyPokerGame соответствовала ему. В этом случае Swift не может синтезировать реализацию Hashable, потому что мы храним замыкание. Как и класс, замыкание является ссылочным типом, который Swift не будет синтезировать, поэтому нам нужно реализовать Hashable самостоятельно. К счастью, в Swift есть тип AnyHashable, который представляет собой стираемый тип, соответствующий протоколу Hashable. Можно сохранить игру внутри AnyHashable и перенаправить в него методы Hashable.

Давайте посмотрим на полную реализацию AnyPokerGame.

Листинг 13.32. Реализация Hashable

```
struct AnyPokerGame: PokerGame {
   private let _start: () -> Void
   private let _hashable: AnyHashable ❶

   init<Game: PokerGame>(_ pokerGame: Game) {
      _start = pokerGame.start
      _hashable = AnyHashable(pokerGame) ❷
   }

   func start() {
      _start()
   }
}

extension AnyPokerGame: Hashable { ❸
   func hash(into hasher: inout Hasher) {
      _hashable.hash(into: &hasher) ❹
   }

   static func ==(lhs: AnyPokerGame, rhs: AnyPokerGame) -> Bool {
      return lhs._hashable == rhs._hashable
   }
}
```

❶ Свойство hashable хранит тип AnyHashable.

❷ Мы сохраняем игру внутри _hashable.

❸ AnyPokerGame соответствует протоколу Hashable.

❹ Перенаправляем необходимые методы в методы AnyHashable.

Примечание

AnyPokerGame расширен как выбор стиля, чтобы отделить код для соответствия Hashable.

Поздравляю, мы стерли тип! AnyPokerGame оборачивает любой тип PokerGame, и теперь мы можем свободно использовать AnyPokerGame внутри коллекций. С помощью этой техники можно использовать протоколы с требованиями Self или ассоциированными типами *и* работать с ними во время выполнения!

К сожалению, рассмотренное здесь решение – всего лишь верхушка айсберга. Чем сложнее ваш протокол, тем сложнее стирание типов. Но это может стоить компромисса; те, кто будет работать с вашим кодом, выиграют, если вы скроете от них эти внутренние сложности.

13.3.3. Упражнение

4

Готовы к сложному испытанию? Создадим небольшой фреймворк Publisher/Subscriber (или Pub/Sub), где издатель (Publisher) может уведомить всех своих подписчиков (Subscribers) о событии:

```
// Вначале вводим протокол PublisherProtocol.
protocol PublisherProtocol {
    // По умолчанию тип Message - это String, но это может быть и что-то еще.
    // Избавляет вас от объявления псевдонимов.
    associatedtype Message = String

    // У протокола PublisherProtocol есть тип Subscriber, ограниченный
    ➥ протоколом SubscriberProtocol.
    // Они используют один и тот же тип Message.
    associatedtype Subscriber: SubscriberProtocol
    where Subscriber.Message == Message

    func subscribe(subscriber: Subscriber)
}

// Далее мы вводим протокол SubscriberProtocol, напоминающий подписчика,
➥ который реагирует на события от издателя.
protocol SubscriberProtocol {
    // По умолчанию тип Message - это String, но это может быть и что-то еще.
    ➥ С помощью этого мы сохраняем объявление псевдонима типа.
    associatedtype Message = String
    func update(message: Message)
}

// Создаем класс Publisher, в котором хранится единственный тип Subscriber.
```

```
➥ Но здесь нельзя смешивать и сопоставлять подписчиков.
final class Publisher<S: SubscriberProtocol>: PublisherProtocol where
➥ S.Message == String {
   var subscribers = [S]()
   func subscribe(subscriber: S) {
      subscribers.append(subscriber)
   }

   func sendEventToSubscribers() {
      subscribers.forEach { subscriber in
         subscriber.update(message: "Here's an event!")
      }
   }
}
```

На данный момент `Publisher` может поддерживать массив подписчиков одного типа. Можно ли стереть `SubscriberProtocol`, чтобы `Publisher` мог хранить разные типы подписчиков?

Подсказка: поскольку `SubscriberProtocol` имеет ассоциированный тип, можно создать обобщение `AnySubscriber<Msg>`, где `Msg` – это ассоциированный тип `Message`.

13.4. Альтернатива протоколам

Мы видели множество вариантов использования при работе с протоколами. При этом у вас может появиться желание, чтобы большая часть кода была на основе протоколов. Протоколы являются убедительными, если у вас есть сложный API; тогда вашим потребителям нужно только придерживаться протокола, чтобы воспользоваться преимуществами, в то время как вы, будучи автором, можете скрыть основные сложности системы.

Однако обычно ловушка начинается с протокола, прежде чем вы узнаете, что он вам нужен. Как только в руке у вас появляется молоток – и при этом он блестит, – вещи могут начать выглядеть как гвозди. В своих видеороликах, которые транслируются на Всемирной конференции для разработчиков, компания Apple выступает за использование протоколов. Но протоколы также могут быть и подводным камнем. Если вы неукоснительно соблюдаете парадигму, основанную на протоколе, но пока не уверены, что он вам нужен, то можете столкнуться с излишней сложностью. В предыдущем разделе вы также видели, что у протоколов есть недостатки, которые могут затруднить поиск подходящего решения.

Давайте воспользуемся этим разделом, чтобы рассмотреть другую альтернативу.

13.4.1. Чем мощнее, тем хуже код

Во-первых, подумайте, действительно ли вам нужен протокол. Иногда небольшое дублирование не так уж плохо. Используя протоколы, вы платите за абстрак-

цию. Пройти грань между чрезмерными абстракциями и жестким кодом – это искусство.

Простой подход – придерживаться конкретных типов, когда вы не уверены, нужен ли вам протокол. Например, иметь дело с типом `String` проще, чем со сложным обобщенным ограничением с тремя операторами *where*.

Давайте рассмотрим еще один вариант, когда вы считаете, что вам нужен протокол, но, возможно, его использования можно избежать.

Один из протоколов, который является обычным явлением в том или ином виде, – это протокол `Validator`, представляющий некий фрагмент данных, который можно проверить, например, когда речь идет о формах. Прежде чем показывать альтернативу, мы смоделируем его следующим образом:

Листинг 13.33. Validator

```
protocol Validator {
   associatedtype Value ❶
   func validate(_ value: Value) -> Bool ❷
}
```


❶ Тип значения, которое можно проверить.

❷ Логический тип, указывающий на то, была проверка успешной или нет.

Теперь можно использовать этот валидатор, например чтобы проверить, имеет ли String минимальное количество символов.

Листинг 13.34. Реализация протокола валидатора

```
struct MinimalCountValidator: Validator {
   let minimalChars: Int

   func validate(_ value: String) -> Bool {
      guard minimalChars > 0 else { return true }
      guard !value.isEmpty else { return false } // isEmpty быстрее.
      return value.count >= minimalChars
   }
}

let validator = MinimalCountValidator(minimalChars: 5)
validator.validate("1234567890") // true
```

Теперь для каждой отдельной реализации нужно ввести новый тип, соответствующий типу `Validator`, что является хорошим подходом, но требует дополнительного количества шаблонного кода. Давайте рассмотрим альтернативный вариант, чтобы доказать, что протоколы не всегда нужны.

13.4.2. Создание обобщенной структуры

С помощью обобщений мы заставляем тип – такой как структура – работать со многими другими типами, при этом реализация остается той же. Если мы добавим в эту смесь функцию высшего порядка, то также можем поменять реализации местами. Вместо того чтобы создавать протокол `Validator` и множество типов, соответствующих ему, можно использовать обобщенную структуру `Validator`. Теперь вместо протокола и нескольких реализаций у нас может быть одна обобщенная структура.

Давайте начнем с создания структуры `Validator`. Обратите внимание, что в ней хранится замыкание, которое позволяет менять реализацию.

Листинг 13.35. Структура Validator

```
struct Validator<T> {
   let validate: (T) -> Bool

   init(validate: @escaping (T) -> Bool) {
      self.validate = validate
   }
}

let notEmpty = Validator<String>(validate: { string -> Bool in
   return !string.isEmpty
})

notEmpty.validate("") // Ложно
notEmpty.validate("Still reading this book huh? That's cool!") // Истинно
```

В итоге мы получаем тип, у которого могут быть различные реализации и который работает со многими типами. Для того чтобы активировать `Validator`, не нужно много усилий. Можно объединить несколько маленьких валидаторов в один с помощью метода `combine`.

Листинг 13.36. Объединение валидаторов

```
extension Validator {
   func combine(_ other: Validator<T>) -> Validator<T> { ❶
      let combinedValidator = Validator<T>(validate: { (value: T) ->
         Bool in ❷
         let ownResult = self.validate(value) ❸
         let otherResult = other.validate(value)
         return ownResult && otherResult
      })

      return combinedValidator ❹
   }
}
```

```
let notEmpty = Validator<String>(validate: { string -> Bool in
   return !string.isEmpty
})

let maxTenChars = Validator<String>(validate: { string -> Bool in ❺
   return string.count <= 10
})

let combinedValidator: Validator<String> = notEmpty.combine(maxTenChars) ❻
combinedValidator.validate("") // Ложно
combinedValidator.validate("Hi") // Истинно
combinedValidator.validate("This one is way too long") // Ложно
```

❶ Мы объявляем метод, который принимает два валидатора и возвращает новый. Обратите внимание, что мы не определяем обобщение; `T` из `Validator <T>` в определении типа используется повторно.

❷ Передаем замыкание новому `combinedValidator`.

❸ Внутри замыкания мы запускаем обе валидации и возвращаем результат.

❹ Возвращаем `combinedValidator`.

❺ Чтобы продемонстрировать новую функциональность, создаем новый валидатор, которому нужно максимум десять символов для строки.

❻ Теперь можно легко объединить два валидатора в один.

Мы объединили два валидатора. Поскольку метод `combine` возвращает новый валидатор, можно продолжить цепочку, например комбинируя валидатор регулярного выражения с валидатором непустой строки и т. д. Кроме того, поскольку `Validator` является обобщением, он работает с любыми типами, такими как `Int` и другие. Это один из примеров того, как добиться гибкости, не прибегая к использованию протоколов.

13.4.3. Эмпирические правила полиморфизма

Вот несколько правил, которые следует учитывать при рассмотрении вопросов, касающихся полиморфизма в Swift.

Требования	Предлагаемый подход
Легкий полиморфизм	Используйте перечисления
Тип, который должен работать с несколькими типами	Создайте обобщенный тип
Тип, которому необходима единственная настраиваемая реализация	Сохраняйте замыкание
Тип, который работает с несколькими типами и имеет единственную настраиваемую реализацию	Используйте обобщенную структуру или класс, где хранится замыкание
Когда вам нужен расширенный полиморфизм, расширения по умолчанию и другие расширенные варианты использования	Используйте протоколы

13.5. В заключение

Мы вооружены и готовы написать тестируемый код, применить условное соответствие Swift, обобщенные типы со стиранием типов и знаем, когда использовать перечисления, а когда – обобщенные структуры и протоколы!

В этой главе изложены сложные темы, но мы были настойчивы и дошли до конца. Пришло время похлопать себя по спине! Самое сложное позади, и мы заработали свой значок от Swift за освещение сложных теоретических тем в этой книге.

Резюме

- Можно использовать протоколы в качестве интерфейса для замены реализаций, тестирования или других случаев использования.
- Ассоциированный тип может быть преобразован в тип, который вам не принадлежит.
- При условном соответствии тип может соответствовать протоколу, если его обобщенный или ассоциированный тип соответствует ему.
- Условное соответствие хорошо работает, когда у вас есть обобщенный тип с очень небольшим количеством ограничений.
- Нельзя использовать протокол с ассоциированными типами или требования `Self` в качестве конкретного типа.
- Иногда можно заменить протокол на перечисление и использовать его в качестве конкретного типа.
- Можно использовать протокол с ассоциированными типами или требования `Self` во время выполнения посредством стирания типов.
- Часто обобщенная структура является отличной альтернативой протоколу.
- Объединение функции высшего порядка с обобщенной структурой позволяет создавать очень гибкие типы.

Ответы

1

Можно ли сделать структуру `WaffleHouse` тестируемой, чтобы проверить, что `Waffle` была подана?

Один из способов – сделать `Chef` протоколом, и тогда можно поменять реализацию `Chef` внутри `WaffleHouse`. Потом можно передать тестовый `chef` в WaffleHouse:

```
struct Waffle {}

protocol Chef {
   func serve() -> Waffle
}

class TestingChef: Chef {
```

```
    var servedCounter: Int = 0
    func serve() -> Waffle {
        servedCounter += 1
        return Waffle()
    }
}

struct WaffleHouse<C: Chef> {
    let chef: C
    init(chef: C) {
        self.chef = chef
    }

    func serve() -> Waffle {
        return chef.serve()
    }
}

let testingChef = TestingChef()
let waffleHouse = WaffleHouse(chef: testingChef)

waffleHouse.serve()
testingChef.servedCounter == 1 // Истинно
```

2

Какая польза от обобщения, имеющего несколько ограничений при применении условного соответствия?

У вас будет больше гибкости в том отношении, что у вас будет база, которая работает со множеством типов, и по-прежнему будет возможность получить преимущества, когда тип соответствует протоколу.

3

Сделайте так, чтобы `CachedValue` соответствовал пользовательскому протоколу `Track`, о котором идет речь в этой главе.

```
extension CachedValue: Track where T: Track {
    func play() {
        currentValue.play()
    }
}
```

4

Создайте небольшой фреймворк Publisher/Subscriber (или Pub/Sub), где издатель (Publisher) может уведомить всех своих подписчиков (Subscribers) о событии.

Эту задачу можно решить, создав структуру `AnySubscriber`. Обратите внимание, что вам нужно сделать ее обобщенной из-за ассоциированного типа

Message из Subscriber. В этом случае Publisher хранит AnySubscriber типа AnySubscriber<String>:

```
struct AnySubscriber<Msg>: SubscriberProtocol {
    private let _update: (_ message: Msg) -> Void

    typealias Message = Msg

    init<S: SubscriberProtocol>(_ subscriber: S) where S.Message == Msg {
        _update = subscriber.update
    }

    func update(message: Msg) {
        _update(message)
    }
}
```

Publisher больше не является обобщением. Теперь он может смешивать и сопоставлять подписчиков:

```
final class Publisher: PublisherProtocol {
    // Publisher использует типы AnySubscriber <String>.
    ➥ По сути, он привязывает ассоциированный тип Message к типу String.
    var subscribers = [AnySubscriber<String>]()
    func subscribe(subscriber: AnySubscriber<String>) {
        subscribers.append(subscriber)
    }

    func sendEventToSubscribers() {
        subscribers.forEach { subscriber in
            subscriber.update(message: «Here's an event!»)
        }
    }
}
```

Глава 14. Написание качественного кода на языке Swift

В этой главе:

- документирование кода с помощью Quick Help;
- как написать хорошие комментарии, которые не отвлекают;
- почему стиль не так уж и важен;
- обеспечение согласованности и уменьшение количества ошибок с помощью SwiftLint;
- разделение больших классов по способу Swift;
- делаем типы обобщенными.

Написание кода на Swift – веселое занятие, но в более крупных проектах соотношение между написанием и сопровождением кода смещается в сторону обслуживания. В зрелых проектах важную роль играют правильное именование, повторное использование кода и документирование. В этой главе мы рассмотрим эти моменты, чтобы сделать жизнь программистов более комфортной, если смотреть на код с точки зрения обслуживания.

Это наименее ориентированная на код глава, но одна из наиболее важных глав, когда речь идет о работе над проектами в командах или о попытке передать присваивание кода для нового задания. В ней также рассказывается о подходах с использованием рефакторинга, чтобы сделать свой код более универсальным и многократно используемым.

Эта глава начинается с документирования – кто не любит писать строки документации? Я знаю, что не люблю. Но вы действительно можете помочь другим программистам набрать скорость, предоставив им хорошие примеры, описания, обоснования и требования. Вы увидите, как добавить документирование в код с помощью Quick Help.

Затем мы разберемся, когда и как добавлять комментарии в свой код, и узнаем, как добавлять, а не отвлекать, размещая ценные комментарии с хирургической точностью. Комментарии не ориентированы на Swift, но они по-прежнему являются важной частью ежедневной работы программиста.

Swift предлагает множество различных способов написания кода. Но вы увидите, что стиль не так уж и важен и что для команд важнее согласованность. Вы увидите, как установить и настроить SwiftLint для обеспечения согласованности в кодовой базе, чтобы каждый человек, как новичок, так и эксперт, мог писать код, как если бы его писал один разработчик.

Далее вы узнаете, что можно по-разному относиться к классам очень большого размера. Как правило, это классы «Manager». Исходники от компании Apple и примеры кодов включают в себя множество этих классов, и следование примеру может быть заманчивым. Но чаще всего у этих классов много обязанностей, которые могут повредить удобству сопровождения. Вы узнаете, как можно разделить большие классы и проложить путь к созданию обобщенных типов.

Найти подходящее имя сложно. Вы узнаете, что типы могут получать излишне специфицированные имена и что подходящие имена более ориентированы на будущее и могут подготовить вас к созданию обобщенных типов.

14.1. Документация по API

Написание документации может быть утомительным делом. Вы знаете это, я знаю это, мы все это знаем. Но даже добавление небольшого количества документации к вашим свойствам и типам может помочь коллеге быстро набрать скорость.

Документация удобна для внутренних типов в проекте, но становится особенно важной, когда речь идет об элементах, помеченных как `public` (*открытый*), таких как открытые функции, переменные и классы. Код, помеченный этим словом, доступен, если он предоставляется через фреймворк, который можно использовать повторно в разных проектах, что является еще одной причиной для предоставления полной документации.

В этом разделе вы увидите, как добавить полезные примечания для сопровождения своего кода, чтобы направлять читателей с помощью комментариев к документу, которые представлены в XCode посредством Quick Help.

Затем, в конце, вы увидите, как создать веб-сайт с документацией на основе Quick Help с помощью инструмента под названием Jazzy.

14.1.1. Как работает Quick Help

Документация Quick Help (*Быстрая справка*) – это краткая нотация Markdown, которая сопровождает типы, свойства, функции, кейсы перечислений и т. д. Xcode может показывать всплывающие подсказки после добавления документации Quick Help.

Представим, что мы создаем пошаговую онлайн-игру. Мы используем небольшое перечисление для обозначения каждого шага, который может сделать игрок, например пропуск хода, атака местоположения или лечение.

Xcode может отображать подсказки Quick Help двумя способами. Всплывающее окно, показанное на рис. 14.1, активируется при наведении курсора на слово и нажатии клавиши **Option** (Mac).

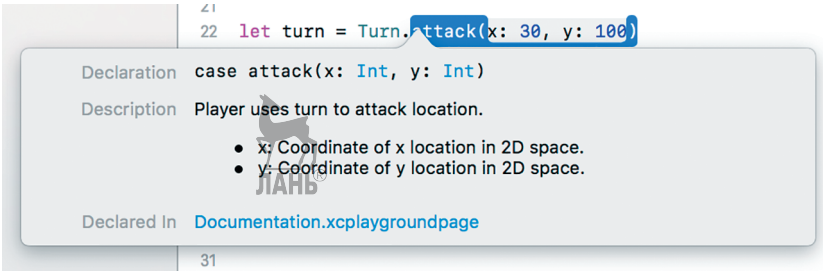


Рис. 14.1. Всплывающая подсказка Quick Help

Quick Help также доступна на боковой панели Xcode (см. рис. 14.2).

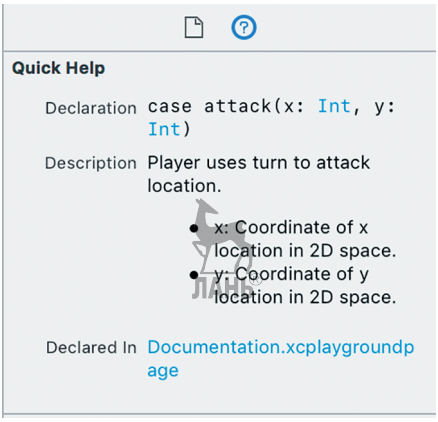


Рис. 14.2. Quick Help на боковой панели

Перечисление в нотации Quick Help форматируется следующим образом:

Листинг 14.1. Нотации Quick Help

```
/// Ход игрока в пошаговой онлайн-игре
enum Turn {
   /// Игрок пропускает ход и получает золото.
   case skip
   /// Игрок использует ход для атаки местоположения.
   /// - x: Coordinate of x location in 2D space.
   /// - y: Coordinate of y location in 2D space.
   case attack(x: Int, y: Int)
   /// Игрок использует раунд для исцеления и не получает золото.
   case heal
}
```

Используя нотацию ///, можно добавлять нотации Quick Help к своим типам. Добавив документацию Quick Help, можно добавить больше информации к написанному вами коду.

14.1.2. Добавление выносок в Quick Help

Quick Help предлагает множество вариантов разметки, называемых *выносками*, которые можно добавить в Quick Help. Выноски дают небольшую поддержку подсказкам Quick Help, например когда речь идет о типах ошибок, которые может генерировать функция, или примере использования вашего кода.

Давайте рассмотрим, как добавить пример кода и кода, где генерируется ошибка, введя новую функцию. Эта функция будет принимать несколько перечислений Turn и возвращать строку действий, происходящих внутри нее.

На рис. 14.3 показан пример кода.

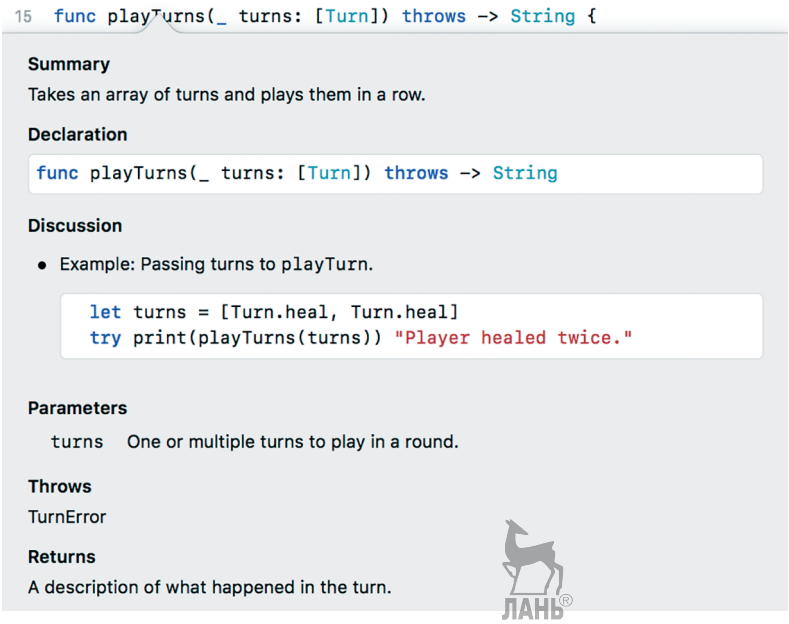


Рис. 14.3. Quick Help с выносками

Чтобы создать пример внутри Quick Help, нужно поместить текст в нотацию Quick Help, как показано здесь.

Листинг 14.2. Добавление выносок

```
/// Takes an array of turns and plays them in a row.
///
/// - Parameter turns: One or multiple turns to play in a round.
/// - Returns: A description of what happened in the turn.
/// - Throws: TurnError
/// - Example: Passing turns to 'playTurn'.
///
///     let turns = [Turn.heal, Turn.heal]
///     try print(playTurns(turns)) "Player healed twice."
func playTurns(_turns: [Turn]) throws -> String {
```

Swift может сообщить вам на этапе компиляции, что функция может генерировать ошибку, но сгенерированные ошибки известны только во время выполнения. Добавление выноски Throws может, по крайней мере, дать читателю больше информации о том, каких ошибок ждать.

Можно даже включить дополнительные выноски, такие как *Attention*, *Author*, *Authors*, *Bug*, *Complexity*, *Copyright*, *Date*, *Note*, *Precondition*, *Requires*, *See AlsoVersion*, *Warning* и некоторые другие.

Полный список всех параметров выноски см. в рекомендациях по разметке XCode (http://mng.bz/Qgow).

14.1.3. Документация в качестве HTML с использованием Jazzy

Jazzy (https://github.com/realm/jazzy) – отличный инструмент для преобразования нотаций Quick Help в веб-сайт документации наподобие Apple. Например, представим, что мы используем фреймворк Analytics. Можно попросить Jazzy сгенерировать для нас документацию на основе общедоступных типов и доступной информации (см. рис. 14.4).

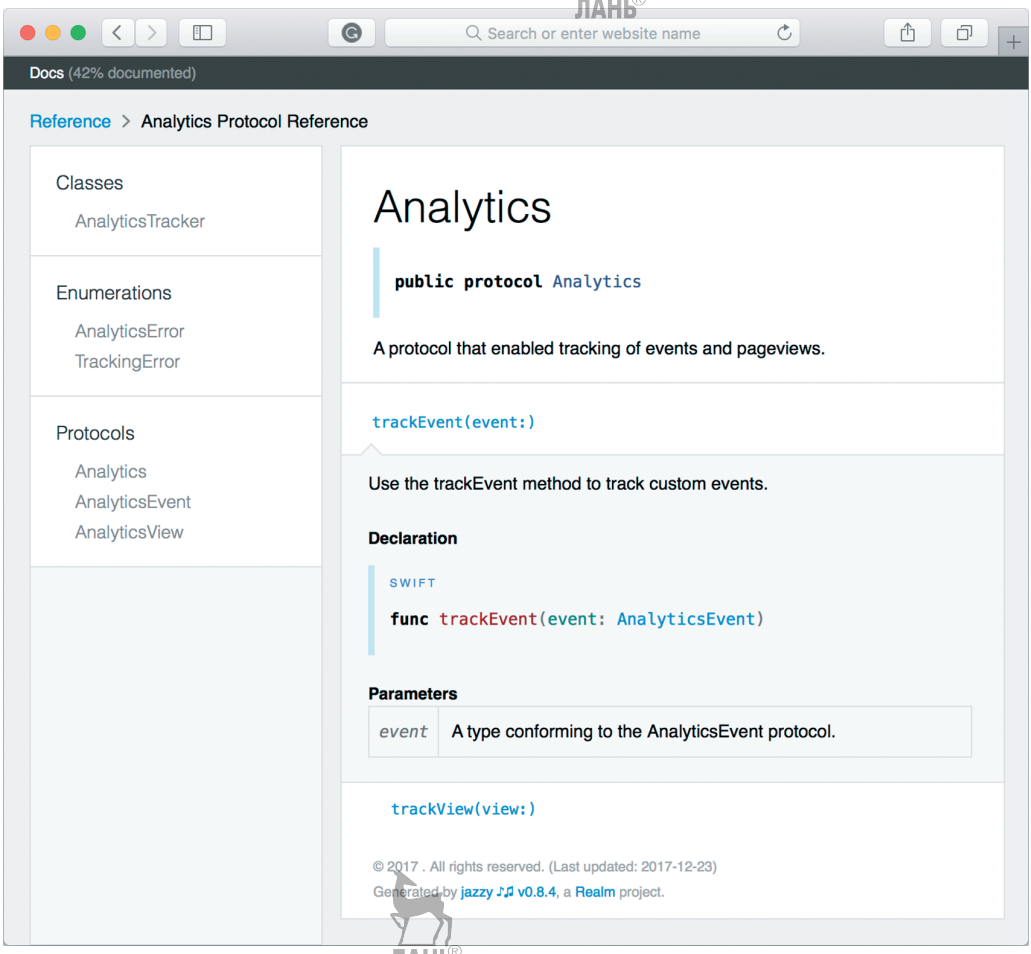


Рис. 14.4. Документирование с использованием Jazzy

Jazzy – это инструмент командной строки, который устанавливается как gem в Ruby. Чтобы установить и запустить Jazzy, используем эти команды из терминала:

```
gem install jazzy
jazzy
```

Вот что мы увидим на выводе:

```
Running xcodebuild
building site
building search index
downloading coverage badge
jam out to your fresh new docs in 'docs'
```

Теперь мы найдем папку docs, в которой содержится прекрасно документированный исходный код. Можно использовать Jazzy при работе с любым проектом, в котором вы хотите создать веб-сайт с документацией.

14.2. Комментарии

Несмотря на то что комментарии относятся не только к Swift, написание кода, который приятно читать и можно быстро понять, по-прежнему полезен.

Комментарии, или явное знание того, когда добавить значимости, используя комментарии, являются жизненно важным компонентом повседневного программирования. К сожалению, комментарии могут испортить ваш код, если вы не будете проявлять осторожность. В этом разделе вы узнаете, как написать полезные комментарии.

14.2.1. Объясняем причину

Простое правило, которому нужно следовать, состоит в том, что комментарий сообщает читателю, «почему» этот элемент присутствует. Напротив, код сообщает читателю, «что» он делает.

Представим себе структуру Message, которая позволяет пользователю отправлять сообщения другому пользователю в частном порядке. В этом примере видно, что когда комментарии сфокусированы на том, чтобы дать ответ на вопрос «что», они не особо помогают.

Листинг 14.3. Структура с комментариями, отвечающая на вопрос «что»

```
struct Message {
   // Идентификатор
   let id: String

   // Дата
   let date: Date

   // Содержимое
   let contents: String
}
```

Такие комментарии излишни и запутывают код. Можно определить значение этих переменных, основываясь на именах свойств, и, кроме того, они ничего не добавляют в Quick Help.

Вместо этого попытаемся ответить на вопрос «почему» с помощью комментария, как показано здесь.

Листинг 14.4. Структура с комментарием типа «почему»

```
struct Message {
   let id: String
   let date: Date
   let contents: String

   // Сообщения могут быть обрезаны сервером при достижении 280 символов
   let maxLength = 280
}
```

Здесь мы объясняем, что там делает `maxLength` и почему 280 – это не просто произвольное число, которое нельзя определить из имени свойства.

14.2.2. Объясняем только непонятные элементы

Не все «почему» нужно объяснять.

Читателей вашего кода, как правило, интересуют только неясные отклонения от стандартного кода, например почему вы храните данные в двух местах вместо одного (возможно, для повышения производительности) или почему вы меняете имена в списке перед поиском (возможно, так вы можете проводить поиск по фамилии).

Не нужно добавлять комментарии к коду, который ваши коллеги уже знают. Тем не менее не стесняйтесь использовать их при значительных отклонениях. Как правило, будьте скупы на комментарии.

14.2.3. Код несет истину

Независимо от того, сколько комментариев вы пишете, истина содержится в коде. Поэтому, когда вы реорганизуете код, вам придется реорганизовывать комментарии, которые сопровождают его. Устаревшие комментарии – это мусор и шум в чистой среде.

14.2.4. Комментарии – не повязка для неудачных имен

Комментарии могут ошибочно использоваться в качестве повязки для компенсации неудачных имен функций или переменных. Вместо этого объясните все в коде, и тогда комментарии вам не понадобятся.

В следующем примере комментарии добавляются в виде повязки, которую можно опустить, предоставив дополнительный контекст с помощью кода:

```
// Убеждаемся, что пользователь может получить доступ к контенту
if user.age > 18 && (user.isLoggedIn || user.registering) { ... }
```

Вместо этого предоставьте контекст с помощью кода, чтобы комментарий не понадобился:

```
let canUserAccessContent = user.age > 18 && (user.isLoggedIn ||
    user.registering)

if canUserAccessContent {
    ...
}
```

Теперь код дает контекст, и комментарии больше не нужны.

14.2.5. Зомби-код

Зомби-код – это закомментированный код, который ничего не делает. Он мертв, но по-прежнему обитает на вашем рабочем месте.

Его легко узнать по закомментированным фрагментам:

```
// func someThingUsefulButNotAnymore() {
    // user.save()
// }
```

Иногда такой код может задерживаться внутри кодовых баз, возможно, из-за того, что о нем забыли, или потому, что кто-то думает, что он может понадобиться снова в какой-то момент.

Помимо проблем с вложениями, в какой-то момент вам придется отказаться от старого кода. Он все еще будет жить в ваших воспоминаниях, за исключением того, что эти воспоминания называются системой управления версиями, например git. Если вам снова нужна старая функция, можно использовать нужную вам систему управления версиями, чтобы вернуть ее, сохраняя при этом кодовую базу чистой.

14.3. Правила стиля

В этом разделе вы увидите, как соблюдать правила стиля с помощью инструмента SwiftLint.

Редко можно найти разработчика, у которого нет сильного предпочтения стиля. Джон любит использовать `forEach` везде, где это возможно, в то время как Карл хочет решить все с помощью циклов `for`. Мэри предпочитает фигурные скобки на одной строке, а Джеффу нравится размещать каждый параметр функции на новой строке, чтобы упростить чтение кода, где используется команда git diff.

Эти предпочтения стиля могут быть глубокими и вызвать множество споров внутри команд.

Однако в конечном итоге стиль не так уж важен. Если код выглядит блестяще чистым, но полон ошибок, или никто его не использует, то какой смысл в хорошем стиле? Цель программного обеспечения – не выглядеть читабельным, а удовлетворить потребность или решить проблему.

Конечно, иногда стиль имеет значение для того, чтобы сделать кодовую базу более приятной для чтения и более легкой для понимания. В конце концов, стиль – это средство для достижения цели, а не сама цель.

14.3.1. Согласованность – это ключ

При работе в команде более ценной, чем сам стиль, является его согласованность. Вы экономите время, когда код предсказуем и, казалось бы, написан одним разработчиком.

Полезно иметь общее представление о том, как выглядит файл Swift, прежде чем вы откроете его. Вы не хотите тратить много времени на расшифровку кода, который просматриваете, что снижает когнитивную нагрузку на вас и членов вашей команды. Новички, работающие над проектом, быстрее набирают скорость и принимают стилистические решения, когда узнают четкий стиль, которому нужно соответствовать.

Удалив споры о стиле из уравнения, можно сосредоточиться на важных аспектах, а именно на создании программного обеспечения, которое решает проблемы или удовлетворяет потребности.

14.3.2. Обеспечение соблюдения правил с помощью линтера

Можно повысить согласованность кодовой базы и свести к минимуму обсуждение стилей, добавив в проект линтер. Линтер пытается обнаружить проблемы, подозрительный код и отклонения от стиля до того, как вы скомпилируете или запустите свою программу. Использование линтера для анализа кода гарантирует, что вы сохраняете согласованную кодовую базу для команд и проектов.

С помощью линтера как новички, так и ветераны проекта могут работать над новым и старым кодом и поддерживать согласованный стиль, соблюдаемый линтером. Линтер даже помогает выполнять проверку на предмет наличия неудачных практик, которые могут иметь место время от времени.

Компания Realm предлагает отличный инструмент SwiftLint (`https://github.com/realm/SwiftLint`) для этой цели. Можно использовать SwiftLint, чтобы соблюдать правила стиля, которые вы настраиваете. По умолчанию он использует правила, определенные сообществом Swift (`https://github.com/github/swift-style-guide`).

Например, код на рис. 14.5 нарушает некоторые правила, такие как принудительное извлечение и пустой пробел. SwiftLint сообщает об этом с помощью предупреждений и сообщений об ошибках. Можно рассматривать SwiftLint как расширение компилятора.

Дело не только в стиле. Сообщение **Empty Count Violation**, показанное на рис. 14.5, предупреждает вас об использовании `isEmpty` вместо `.count == 0`, потому что `isEmpty` лучше работает на больших массивах.

```
9  import UIKit
10
11 class ViewController: UIViewController {
12     var lastLogin: Date?
13     var list: [Int] = []
14
15     override func viewDidLoad() {
16         super.viewDidLoad()
17
18         if list.count == 0 {          Empty Count Violation: Prefer checking `isEmpty` over comparing `count` to zero. (empty_count)
19             print(lastLogin!)         Force Unwrapping Violation: Force unwrapping should be avoided. (force_unwrapping)
20         }
21                                       Trailing Whitespace Violation: Lines should not have trailing whitespace. (trailing_whitespace)
22     }
23
24                                   Vertical Whitespace Violation: Limit vertical whitespace to a single empty line. Currently 2. (vertical_whitespace)
25 }
26                                       Trailing Newline Violation: Files should have a single trailing newline. (trailing_newline)
27
```

Рис. 14.5. SwiftLint в действии

Не беспокойтесь, если некоторые правила покажутся вам слишком строгими. Можно настроить каждое правило по своему вкусу. После того как команда определилась с правилами, каждый может двигаться вперед, и можно убрать обсуждение стиля из уравнения при изучении кода.

Читайте дальше, чтобы узнать, как установить SwiftLint и настроить его правила.

14.3.3. Установка SwiftLint

SwiftLint можно установить с помощью HomeBrew (`https://brew.sh`). В командной строке введите следующее:

```
brew install swiftlint
```

Кроме того, можно напрямую установить пакет SwiftLint из репозитория Github (`https://github.com/realm/Swiftlint/releases`).

SwiftLint – это утилита командной строки, которую можно запустить с помощью команды swiftlint. Но вы, скорее всего, будете использовать XCode. Давайте настроим его, чтобы SwiftLint там тоже работал.

В Xcode найдите вкладку **Build Phases** (см. рис. 14.6). Нажмите на знак «плюс» и выберите **New Run Script Phase** (Новый этап запуска сценария). Затем добавьте новый сценарий и код, показанный на рис. 14.7.

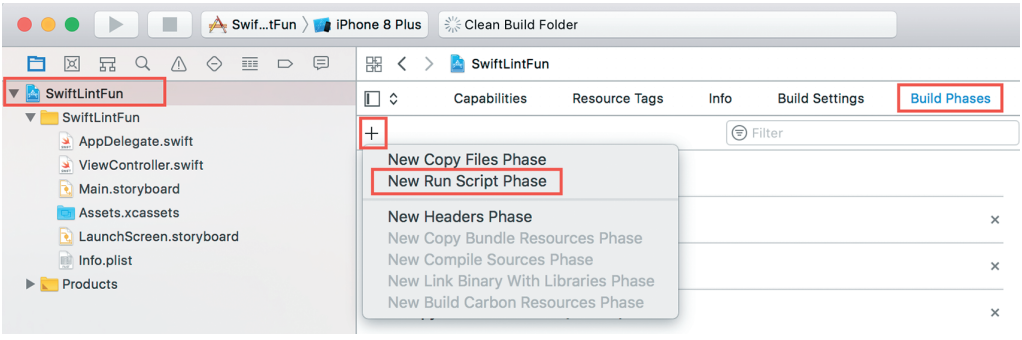


Рис. 14.6. Этапы сборки XCode

Рис. 14.7. Скрипт Xcode

То, что нужно! В следующий раз, когда вы будете создавать свой проект, SwiftLint выдаст предупреждения и ошибки, где это применимо, на основе конфигурации по умолчанию.

14.3.4. Настройка SwiftLint

Далее вы, вероятно, захотите настроить SwiftLint по своему вкусу. Файл конфигурации – это файл с расширением yml – или yaml – .swiftlint.yml. Он содержит правила, которые можно включать и отключать.

```
disabled_rules: # rule identifiers to exclude from running
   - variable_name
   - nesting
   - function_parameter_count
opt_in_rules: # some rules are only opt-in
   - control_statement
   - empty_count
   - trailing_newline
   - colon
   - comma
included: # paths to include during linting. `--path` is ignored if present.
   - Project
   - ProjectTests
   - ProjectUITests
excluded: # paths to ignore during linting. Takes precedence over 'included'.
   - Pods
   - Project/R.generated.swift

# configurable rules can be customized from this configuration file
# binary rules can set their severity level
force_cast: warning # implicitly. Give warning only for force casting
```

```
force_try:
severity: warning # explicitly. Give warning only for force try

type_body_length:
   - 300 # warning
   - 400 # error

# or they can set both explicitly
file_length:
   warning: 500
   error: 800

large_tuple: # warn user when using 3 values in tuple, give error if there
➥ are 4
   - 3
   - 4

# naming rules can set warnings/errors for min_length and max_length
# additionally they can set excluded names
type_name:
   min_length: 4 # only warning
      error: 35
   excluded: iPhone # excluded via string
reporter: «xcode»
```

Кроме того, можно проверить все правила, которые предлагает SwiftLint, с помощью команды `rules swiftlint`.

Переместите файл .swiftlint.yml в корневой каталог вашего исходного кода, например туда, где можно найти файлы main.swift или AppDelegate.swift. Теперь вся команда использует одни и те же правила, и вы можете включать и отключать их по своему усмотрению.

14.3.5. Временное отключение правил SwiftLint

Некоторые правила созданы для того, чтобы их нарушали. Например, принудительное извлечение может быть установлено как нарушение, но, возможно, иногда вам нужно, чтобы оно было включено. Можно использовать модификаторы SwiftLint, чтобы отключить определенные правила для каких-либо строк в коде.

Применяя конкретные комментарии в своем коде, можно отключить правила SwiftLint в соответствующих строках. Например, можно отключать и включать правило таким образом:

```
// swiftlint:disable <rule1> [<rule> <rule>...]
// .. Нарушение кода.
// swiftlint:enable <rule1> [<rule> <rule>...]
```

Можно изменять эти правила с помощью ключевых слов `:previous`, `:this` или `:next`.

Например, можно отключить правила нарушения, которые мы видели в начале этого раздела:

```
if list.count == 0 { // swiftlint:disable:this empty_count
    // swiftlint:disable:next force_unwrapping
    print(lastLogin!)
}
```

14.3.6. Автозамена правил SwiftLint

Как только вы добавите SwiftLint в существующий проект, скорее всего, посыплется град предупреждений. К счастью, SwiftLint может автоматически исправлять предупреждения с помощью приведенной ниже команды, которую можно выполнить из терминала:

```
swiftlint autocorrect
```

Команда `autocorrect` корректирует файлы и исправляет предупреждения, когда она чувствует себя достаточно уверенно, чтобы сделать это.

Конечно, это не помешает проверить файл diff в системе управления версиями, чтобы убедиться, был ли ваш код исправлен соответствующим образом.

14.3.7. Синхронизация SwiftLint

Если SwiftLint использует несколько членов команды, существует вероятность, что у одного из них может быть установлена другая версия SwiftLint, особенно когда SwiftLint обновляется с течением времени. Можно добавить небольшую проверку, чтобы XCode выдавал предупреждение, когда вы используете не ту версию.

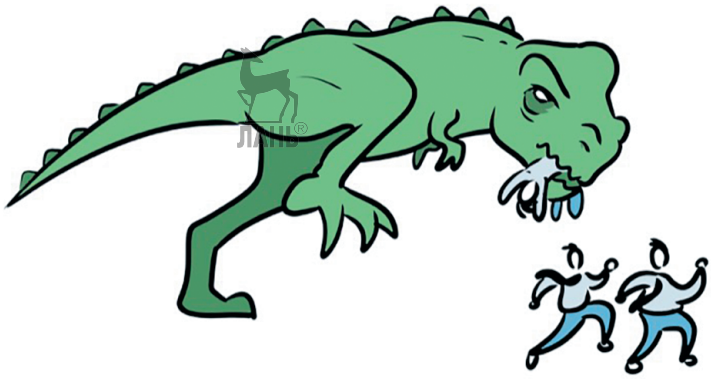
Это еще один этап сборки, который вы можете добавить в XCode. Например, приведенная ниже команда выдает предупреждение, если у вас не установлен SwiftLint версии 0.23.1:

```
EXPECTED_SWIFTLINT_VERSION=»0.23.1»
if swiftlint version | grep -q ${EXPECTED_SWIFTLINT_VERSION}; then
echo "Correct version"
else
echo «warning: SwiftLint is not the right version
➥ ${EXPECTED_SWIFTLINT_VERSION}. Download from
➥ https://github.com/realm/SwiftLint»
```

Теперь всякий раз, когда кто-то из членов команды вовремя не выполнил обновление, XCode выдаст предупреждение.

Этот раздел предоставил более чем достаточно информации для начала работы с SwiftLint. SwiftLint предлагает гораздо больше настроек и регулярно обновляется новыми правилами. Обязательно следите за проектом!

14.4. Избавляемся от менеджеров



Классы Manager, которые можно узнать по наличию суффикса `-Manager`, довольно часто появляются в iOS и сообществе Swift, скорее всего, потому, что код Apple в качестве примера использует типы менеджеров. Но у классов менеджеров, как правило, много (или слишком много) обязанностей. Испытайте подход типа «вы всегда так делали» и посмотрите, удастся ли улучшить читабельность кода, используя большие классы менеджера.

Мы увидим, как пересмотреть большой класс со множеством обязанностей, разделив его на более мелкие повторно используемые типы. Работа в более модульном режиме открывает дорогу к более модульному и гибкому подходу к архитектуре, включая обобщенные типы.

Поскольку классы менеджеров имеют тенденцию быть венцом крупногабаритных классов, они послужат ярким примером в этом разделе.

14.4.1. Ценность менеджеров

Менеджеры, как правило, – это классы со множеством обязанностей, например:

- `BluetoothManager` – проверка соединений, хранение списка устройств, помощь в переподключении и служба обнаружения;
- `ApiRequestManager` – выполнение сетевых вызовов, хранение ответов в кеше, наличие механизма очереди и поддержка WebSocket.

Избегать подобных классов в реальном мире легче сказать, чем сделать; это нельзя предотвратить на 100 %. Кроме того, типы менеджеров имеют смысл, когда вы составляете их из типов поменьше, по сравнению с классами, содержащими множество обязанностей.

Однако если у вас есть большой класс менеджера, можно удалить суффикс `Manager`, и имя типа сообщит читателю то же, что и раньше.

Например, переименуйте `BluetoothManager` в `Bluetooth` и оставьте те же обязанности. Также можно переименовать `PhotosManager` в `Photos` или `Stack`. Опять же, без суффикса `-Manager` тип предоставляет тот же объем информации о своих задачах.

14.4.2. Атака на менеджеров

Обычно когда тип получает суффикс `-Manager`, это показатель того, что у этого класса много важных обязанностей. Давайте посмотрим, как это исправить.

Во-первых, четко именуйте эти обязанности, чтобы менеджер не скрывал их. Например, `ApiRequestManager` можно называть `ApiRequestNetworkCacheQueue Websockets`. Это лишь малость, но истина вышла наружу! Явное обозначение обязанностей означает, что вы раскрыли все, что делает тип. Теперь ясно, насколько ответствен `ApiRequestManager`.

В качестве следующего шага рассмотрим возможность разделения обязанностей на более мелкие такие как типы `ResponseCache`, `RequestQueue`, `Network` и `Websockets` (см. рис. 14.8). Кроме того, теперь можно переименовать `ApiRequestManager` в нечто более точное, например в `Network`.

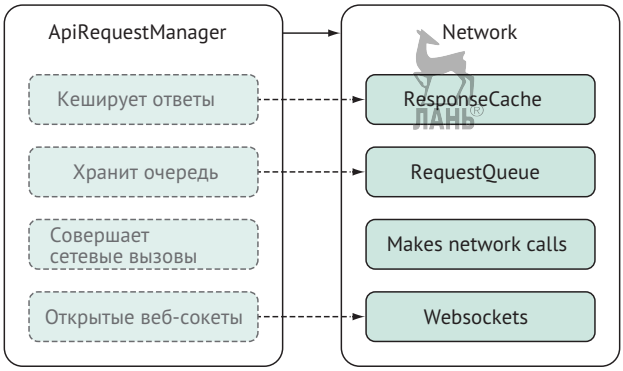


Рис.14.8. Рефакторинг ApiRequestManager

`ApiRequestManager` разделен на точные части и состоит из отдельных типов, которые вы только что извлекли.

Теперь можно рассматривать каждый тип в отдельности. Кроме того, если существует ошибка, например в механизме кеширования, не нужно заглядывать в гигантский класс `ApiRequestManager`. Скорее всего, вы найдете ошибку внутри типа `ResponseCache`.

Меньшее число менеджеров не только отлично подходит для того, чтобы работа шла быстрее, – я сказал это вслух? – вы также получаете ясность при написании программного обеспечения в небольших компонентах.

14.4.3. Прокладываем дорогу для обобщений

Теперь, когда большой класс `ApiRequestManager` был разбит на более мелкие типы, проще выбрать, какие из этих типов можно использовать повторно, чтобы они не ограничивались сетевым доменом.

Например, типы `ResponseCache` и `RequestQueue` явно ориентированы на сетевые функциональные возможности. Но не так уж и сложно представить, что кеш и очередь работают и для других типов. Если механизмы очередей и кеширования требуются в нескольких частях приложения, можно потратить немного уси-

лий, чтобы сделать ResponseCache и ResponseQueuegeneric обобщениями, на что будет указывать символ <T> (см. рис. 14.9).

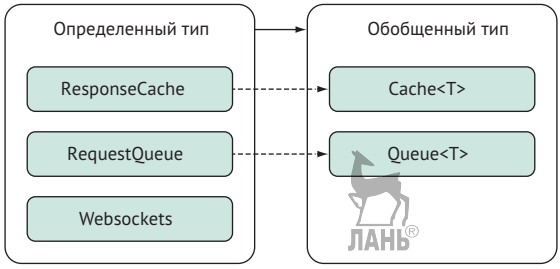


Рис. 14.9. Создание обобщенных типов

Мы убираем имена Response и Request, чтобы получить типы Queue и Cache для элементов за пределами сетевого домена, а также реорганизуем их, чтобы сделать их обобщенными по функциональности (см. рис. 14.10).

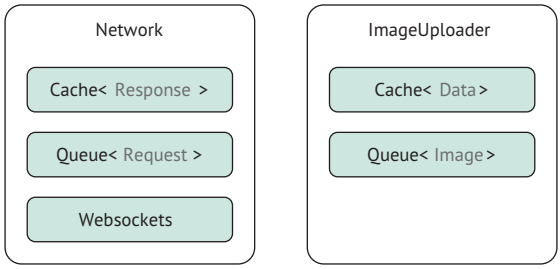


Рис. 14.10. Повторное использование обобщенных типов

На этот раз можно использовать обобщенные типы Queue и Cache для постановки в очередь и кеширования сетевых запросов и ответов. Но в какой-то другой части вашего приложения, благодаря обобщениям, можно использовать очередь и кеш для хранения и загрузки, например данных изображения.

Наличие небольших типов с четкой ответственностью облегчает создание больших типов. Добавьте к этому обобщения, и вы получите подход в стиле Swift для работы с базовой функциональностью внутри приложения.

В качестве контрапункта можно добавить, что создание обобщенного типа с самого начала не всегда плодотворно. Можно посмотреть на стул и диван и подумать: «Мне нужен только обобщенный тип Seat». Вполне нормально время от времени сопротивляться принципу *DRY* (*Don't repeat yourself* – Не повторяйся). Иногда дублирование является хорошим выбором, когда вы не уверены, в каком направлении движется ваш проект.

14.5. Именование абстракций

Именование классов, переменных и файлов в программировании может быть сложнее, чем дать имя первенцу или питомцу, особенно когда вы даете имена сложным вещам, которые охватывают несколько различных типов поведения.

В этом разделе показано, как можно называть типы ближе к абстрактному, а также рассматривать некоторые из них как обобщенные кандидаты.

14.5.1. Обобщенное или конкретное

Если вы даете программисту задание создать кнопку для отправки регистрационной формы изнутри программы, думаю, было бы справедливо предположить, что вы не хотите, чтобы он создавал абстрактный класс `Something` для обозначения этой кнопки. Равно как и причудливое сверхконкретное имя, такое как `CommonRegistrationButtonUsedInFourthScreen`, также не будет приветствоваться.

Учитывая этот пример, я надеюсь, вы согласитесь, что подходящее имя, скорее всего, находится где-то между абстрактным и конкретным примерами.

14.5.2. Хорошие имена не меняются

При разработке программного обеспечения может произойти несоответствие между тем, что может делать тип, и тем, как он используется.

Представьте, что вы создаете приложение, которое отображает пять лучших кофеен. Приложение считывает прошлые местоположения пользователей и выясняет, где пользователь бывал чаще всего.

Первая мысль, которая может прийти на ум, – создать тип `FavoritePlaces`. Вы даете ему большое количество мест, а он возвращает пять самых посещаемых кофеен. После этого можно выполнить фильтрацию по типам, как показано в этом примере.

```
let locations = ... // извлеченные местоположения.
let favoritePlaces = FavoritePlaces(locations: locations)
let topFiveFavoritePlaces = favoritePlaces.calculateMostCommonPlaces()

let coffeePlaces = topFiveFavoritePlaces.filter { place in place.type ==
   "Coffee" }
```

Но теперь клиент звонит и хочет добавить новую функциональность в приложение, а также он хочет, чтобы приложение показывало, какие кофейни пользователь посетил реже всех, чтобы у него была возможность побудить пользователя снова посетить эти места.

К сожалению, нельзя опять использовать тип `FavoritePlaces`. У этого типа есть все внутренние механизмы для группировки местоположений и выполнения нового требования, но в названии особо упоминается, что он использует только избранные места, а не те, что посещают реже всего.

Получается, что имя типа слишком точное. Тип назван в честь того, как он используется, то есть поиск любимых мест. Но было бы лучше, если бы вы дали ему имя, основываясь на том, что он делает, а именно находит и группирует вхождения мест (см. рис. 14.11).

Дело в том, что хорошие имена не меняются. Если бы вы начали с типа `LocationGrouper`, то могли бы использовать его напрямую в обоих случаях, доба-

вив свойство `lessVisitedPlaces`; это бы предотвратило рефакторинг именования или новый тип.

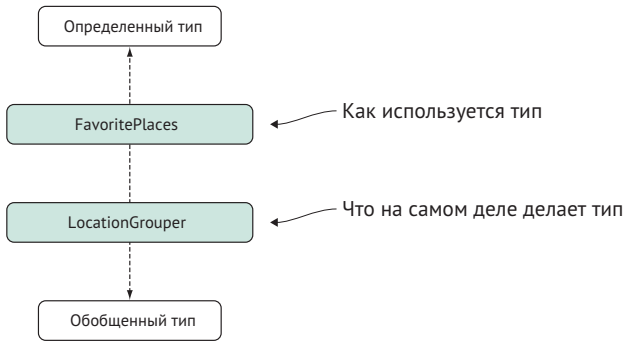


Рис. 14.11. Именование типа в соответствии с тем, что он делает

Получить в итоге неудачное имя – это незначительная деталь, но это может произойти легко и незаметно. Прежде чем вы об этом узнаете, в вашей базе кода может появиться слишком много неподходящих имен. Например, не стоит использовать что-то вроде `redColor` в качестве свойства кнопки для состояния предупреждения; лучше подойдет слово warning (предупреждение), потому что дизайн предупреждения может измениться, но цель не изменится. Также при создании `UserheaderView`, который представляет собой не что иное, как изображение и метку, которую вы можете повторно использовать в качестве чего-то еще, имя `ImageDescriptionView`, возможно, будет более подходящим и пригодным для повторного использования.

14.5.3. Именование обобщений

Чем абстрактнее вы делаете свои типы, тем легче сделать их обобщенными. Например, `LocationGrouper` не слишком далек от типа `Grouper<T>`, где T может представлять `CLLocation`, например `Grouper<CLLocation>`. Но `Grouper` в качестве обобщенного типа также можно использовать для группировки чего-то другого, например оценки мест, и в этом случае вы будете повторно использовать тип как `Grouper<Rating>`.

Создание обобщенного типа не должно быть целью. Чем конкретнее вы можете сделать тип, тем легче его понять. Чем более обобщен тип, тем чаще его можно использовать повторно (но труднее понять). Однако до тех пор, пока имя типа соответствует его цели, – это лучшее решение.

14.6. Контрольный список

В следующий раз при создании проекта проверьте, можете ли вы следовать короткому контрольному списку, чтобы повысить качество своих результатов:

- документация Quick Help;
- редкие, но содержательные комментарии, объясняющие причину;

- добавление SwiftLint в свой проект;
- сокращение классов со слишком большим количеством обязанностей;
- именование типов в соответствии с тем, что они делают, а не тем, как используются.

14.7. В заключение

Хотя эта глава и не была сосредоточена на коде, ее идеи по-прежнему ценны. Когда вы сохраняете последовательную, чистую кодовую базу, то можете сделать жизнь более комфортной для себя и окружающих вас разработчиков.

Если вам понравилась эта глава, я рекомендую прочитать рекомендации по разработке API в Swift (`http://mng.bz/XAMG`). Тут полно полезных соглашений об именах.

Мы почти закончили; в следующей главе дам несколько советов касательно того, что можно делать дальше.

Резюме

- Документация Quick Help – полезный способ добавления небольших фрагментов документации в базу кода.
- Документация Quick Help особенно полезна для общедоступных и внутренних элементов, используемых внутри проекта и фреймворка.
- Quick Help поддерживает множество полезных выносок, которые обогащают документацию.
- Для создания документации Quick Help можно использовать Jazzy.
- Комментарии объясняют причину, а не дают определение тому, что и так ясно.
- Будьте скупы на комментарии.
- Комментарии не являются повязкой для неудачных имен.
- Нет необходимости позволять закомментированному коду, так называемому коду-зомби, находиться где-либо.
- Согласованность кода важнее стиля.
- Согласованность можно обеспечить путем установки SwiftLint.
- SwiftLint поддерживает конфигурации, которые ваша команда может выбирать, что помогает устранить дискуссии и разногласия в отношении вопросов, связанных со стилем.
- Классы менеджеров могут отбрасывать суффикс `-Manager` и по-прежнему передавать то же значение.
- Большой тип может состоять из более мелких типов с целенаправленными обязанностями.
- Мелкие компоненты являются хорошими кандидатами для того, чтобы стать обобщениями.

- Называйте свои типы в соответствии с тем, что они делают, а не тем, как они используются.
- Чем более абстрактен тип, тем легче сделать его обобщенным и использовать повторно.

Глава 15. Что дальше?

Мы дошли до конца; пришло время хорошенько похлопать себя по спине! Мы затронули множество вопросов и рекомендаций по конкретным языкам и расширили свой технический арсенал, чтобы быстро решать различные проблемы в стиле Swift. Более того, мы увидели, что Swift – это коллекция современных и не очень современных концепций в современном обличье. Эти концепции останутся с вами на протяжении вашей карьеры программиста. В следующий раз, когда вы увидите новый захватывающий фреймворк или даже новый язык, который хотите выучить, сможете распознать обобщения, методы `map` и `flatMap`, типы `Sum` и опционалы и сразу же применить эти концепции. Их понимание весомее, чем искусный трюк Swift или превосходный фреймворк.

Я надеюсь, что вы приняли эти концепции и ваша повседневная работа получила большой качественный удар и что это поможет вам получить интересную работу или увлекательное продвижение по службе, или же вы сможете быстрее вливать запросы на принятие изменений в основной репозиторий проекта, обучая других мощным способам написания кода на Swift.

А что же теперь? Читайте дальше, чтобы познакомиться с предложенными идеями.

15.1. Создавайте фреймворки, предназначенные для Linux

На момент написания этих строк Swift уделял большое внимание фреймворкам OS от компании Apple, таким как iOS, tvOS и MacOS. У этих платформ уже тонны фреймворков. Но сообщество Swift могло бы помочь, чтобы убедиться, что существуют фреймворки, предназначенные для Linux. Компиляция кода Swift для проектов Linux поможет части сообщества, ориентированной на вопросы, связанные с сервером. Речь идет о таких вещах, как инструменты командной строки, веб-фреймворки и т. д. Если вы – iOS-разработчик, создание инструментов для Linux откроет вам новый мир программирования, и вы прикоснетесь к различным концепциям. Заставить свой код работать с менеджером пакетов Swift – неплохое начало.

15.2. Изучите диспетчер пакетов Swift

Диспетчер пакетов Swift поможет вам быстро приступить к работе и сборке. Для начала ознакомьтесь с руководством по началу работы (`http://mng.bz/XAMG`).

Я также рекомендую изучить исходный код диспетчера (`https://github.com/apple/swift-package-manager`). В папках Utility и Basic есть несколько полезных расширений и типов, которые можно использовать для ускорения разработки. Один из них – `Result`, о котором уже шла речь. Другие интересные типы – `OrderedSet`, `Version` или помощники для написания инструментов командной строки, такие как `ArgumentParser` или `ProgressBar`.

Вы также можете найти полезные расширения (http://mng.bz/MWP7), такие как `flatMapValue` для типа `Dictionary`, который преобразует значение, если оно не равно nil.

К сожалению, у диспетчера пакетов Swift есть свои проблемы. На момент написания этих строк он не поддерживает iOS для управления зависимостями и не очень хорошо работает с XCode. Со временем создание системных приложений станет проще и привлекательнее с развитием диспетчера пакетов и ростом числа фреймворков, которые предлагает сообщество.

15.3. Изучайте фреймворки

Еще одно учебное упражнение – поэкспериментировать с конкретными фреймворками или заглянуть внутрь них:

- Kitura (https://github.com/IBM-Swift/Kitura);
- Vapor (https://github.com/vapor/vapor);
- SwiftNIO – для низкоуровневых сетей (https://github.com/apple/swift-nio);
- TensorFlow for Swift – для машинного обучения (https://github.com/tensorflow/swift).

Если вас больше интересует метапрограммирование, предлагаю взглянуть на Sourcery (https://github.com/krzysztofzablocki/Sourcery). Это мощная утилита, с помощью которой можно избавиться от шаблонного кода в своей кодовой базе, помимо прочего.

15.4. Бросьте себе вызов

Если вы всегда подходили к новому приложению определенным образом, возможно, сейчас самое время попробовать себя в совершенно ином подходе.

Например, попробуйте настоящее протокольно-ориентированное программирование, где вы моделируете свои приложения, начиная с протоколов. Посмотрите, как далеко вы можете зайти без создания конкретных типов. Вы войдете в мир абстрактных объектов и хитроумных обобщенных ограничений. В итоге вам понадобится конкретный тип, например класс или структура, но, возможно, вы сможете использовать большинство реализаций по умолчанию в протоколе.

Проверьте, можете ли вы бросить себе вызов, используя протоколы, но только на этапе компиляции, а это означает, что вы будете применять обобщения и ассоциированные типы. С ними будет сложнее работать, но видеть, как ваш код компилируется, и наблюдать за тем, как все работает, очень полезно. Используя протокольно-ориентированный подход, у вас может возникнуть ощущение, что вы программируете, надев наручники, но это напрягает ваше мышление и будет плодотворным упражнением.

Еще одна идея состоит в том, чтобы вообще избегать протоколов. Посмотрите, как далеко вы можете продвинуться, используя только перечисления, структуры и функции высшего порядка.

15.4.1. Присоединяйтесь к эволюции Swift

Приятно следить за прогрессом Swift, когда каждые несколько месяцев появляются новые обновления. Но не нужно наблюдать со стороны. Вы можете принять участие в обсуждении эволюции Swift и даже представить свои собственные предложения, касающиеся изменений. Все начинается со страницы на Github (https://github.com/apple/swift-evolution).

15.4.2. Заключительные слова

Большое спасибо за то, что приобрели и прочитали эту книгу. Я надеюсь, что ваши навыки в программировании на языке Swift значительно улучшились. Не стесняйтесь связаться со мной в Twitter (@tjeerdintveen). Увидимся там!

Предметный указатель

Н

О

П

С

Т

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Оптовые закупки: тел. +7 (499) 782-38-89

Электронный адрес: **books@alians-kniga.ru**.

Чейрд ин’т Вейн

Swift. Подробно

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Беликов Д. А.*
Корректор *Синяева Г. И.*
Верстка *Орлов И. Ю.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 34,29. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**