ВАСИЛИЙ УСОВ

SWIFT

Основы разработки приложений под iOS, iPadOS и macOS

5-е издание, дополненное и переработанное

ВАСИЛИЙ УСОВ

SWIFT

Основы разработки приложений под iOS, iPadOS и macOS

5-е издание,
дополненное и переработанное

ПИТЕР®

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

2020

ББК 32.973.2-018.1
УДК 004.438
У76

Усов Василий

У76 Swift. Основы разработки приложений под iOS, iPadOS и macOS. 5-е изд., дополненное и переработанное. — СПб.: Питер, 2020. — 496 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1402-3

Язык Swift молод, он растет, развивается и изменяется, хотя основные подходы к программированию и разработке уже сформировались. В новом, пятом издании книги была полностью переработана первая часть книги, что делает знакомство с языком Swift более комфортным, а весь текст актуализирован в соответствии с возможностями Swift 5.

В ходе долгого и плодотворного общения с читателями появилось множество идей, благодаря которым новое издание стало еще более полезным и насыщенным учебными материалами. Теперь вы не только изучите Swift, но и получите начальные сведения о принципах разработки полноценных приложений.

Мир Swift постоянно меняется, людей со значительным багажом знаний и опыта за плечами еще просто не существует в силу возраста языка, поэтому вы можете стать одним из первых специалистов.

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

ББК 32.973.2-018.1
УДК 004.438

ISBN 978-5-4461-1402-3

Оглавление

Олег Фролов

На момент написания этого отзыва, у меня есть три приложения в AppStore. За плечами небольшой опыт программирования на PHP. До этой книги пару раз пробовал изучать Swift, но не хватало времени и терпения. В ноябре 2018 года приобрел 4-е издание книги. Через 4 месяца изучения и параллельного написания приложения, выложил его в AppStore. Лично для меня было продуктивно учиться по книге, выполнять домашние задания в конце каждой главы. Затраты на книгу полностью окупились.

Станислав Слипченко

Как-то я решил, что было бы классно создать свое приложение для телефона. Так как у меня был айфон, то выбор пал на Swift. Случайно наткнулся на 3-е издание книги Василия Усова в интернете и сразу же ее заказал.

После прочтения книги я самостоятельно сделал приложение из последнего задания — оно работало. Я очень гордился им! Хотя я уже знаю Swift, изучаю фреймворки и работу с сетью, все равно с нетерпением жду пятое издание и обязательно его приобрету и изучу.

С каждым новым прочтением нахожу что-то новое в уже изученной книге.

Дмитрий Ухаткин

Хочу сказать автору спасибо, что подтолкнул меня изучать язык Swift. Я работал на станках с программным управлением на заводе, но со временем понял, что мне нравится программировать и надо в этом развиваться. Я начал с нуля изучать Swift и сейчас работаю почти 4 месяца в компании, где мы разрабатываем приложение.

Сергей Литвинов, разработчик на Swift.

Когда я начинал знакомство со Swift, опыта в программировании у меня не было совсем. Было только желание. Я смотрел на «Ютубе» разные курсы, но как только дело доходило до сложных тем, терялось понимание, что сильно демотивировало. Я пересматривал ролики, пытаясь вникнуть в суть, но это мало помогало. Дважды я бросал обучение.

Потом случайно ко мне попала книга «Swift. Основы разработки приложений под iOS и macOS». Почти сразу я понял, что информации в книге гораздо больше, чем в видеокурсах. Когда я добрался до сложных тем, читать стало еще интересней — все усваивалось и все было понятно. Я считаю, что это лучшая книга по Swift — у автора талант писать простым языком о сложных вещах. Эта книга стала моим третьим и успешным подходом к обучению. Я научился писать приложения и осуществил свою маленькую мечту — написал аудиопроигрыватель (AMP player) для macOS, который сейчас пользуется спросом у покупателей.

Всем у кого нет опыта программирования, настоятельно рекомендую «Swift. Основы разработки приложений под iOS и macOS». Она лучшая.

Посвящается моему отцу и моей матери.
Благодаря вашим советам и стараниям я выбрал
путь, определивший мою дальнейшую судьбу.
Нет на свете слов, способных выразить
мою благодарность и любовь к вам!

Отдельное спасибо Sergey Litvinov, Олег MRRO8OT,
Nikolai Sinyov, Constantin Horoshun, Alexey mdauzh
и остальным участникам чата в Telegram
за активность, поддержку и взаимовыручку!

Читателю

Эта книга — венец моих стараний, начатых несколько лет назад с написания первого издания учебника по Swift. Работая над книгой, я ставил для себя две основные задачи: популяризацию apple-dev (от *англ.* develop — разработка) и языка программирования Swift, а также поиск наиболее эффективных способов и подходов к обучению. Благодаря нашей пока еще небольшой «армии» последователей я вижу, что мой многолетний труд не напрасен и уже приносит свои плоды. Многие из моих добрых товарищей, которые не так давно купили одно из изданий книги и ввязались в эту вечную гонку за знаниями, уже разрабатывают свои приложения или нашли отличную работу, где могут в полной мере реализовываться.

Я рад каждому новому участнику, регистрирующемуся на сайте или добавляющемуся в Telegram-канал, и неважно, сколько этому участнику лет: 15 или 40. Желание изучить Swift уже само по себе бесценно, так как кто-то из вас, читателей, рано или поздно создаст «то самое приложение», которое совершит очередную революцию.

Если ты юноша или девушка, то это прекрасно, что поздно вечером ты не заставляешь своих родителей переживать, а мирно сидишь перед монитором с этой книгой в руках и изучаешь невероятный язык Swift. Я также уверен, что ты не против купить новую модель iPhone, на который можно заработать на своем собственном приложении. И вот что я скажу: у тебя все получится!

Если ты уже учишься в институте, то Swift — это то, что будет держать твой ум в тонусе. Помни, что многие из известных разработчиков придумали и реализовали свои удивительные идеи именно в твоем возрасте. Сейчас ты находишься на пике активности, воспользуйся этим с умом!

Если ты старше, то наверняка в твоем мозге созрела гениальная идея, для реализации которой ты и пришел в apple-dev. Swift — это именно

тот инструмент, который тебе необходим. Главное, погрузившись в разработку, не забывай о своих близких, хотя, уверен, что они с полным пониманием отнесутся к твоему начинанию.

Дари этому миру добро и прекрасные приложения! Добро пожаловать во вселенную Swift.

Наш портал — https://swiftme.ru.

Telegram-канал — https://t.me/usovswift.

Василий Усов

Введение

На ежегодной всемирной конференции разработчиков на платформе Apple (Worldwide Developers Conference, WWDC) 2 июня 2014 года «яблочная» компания приятно удивила iOS-общественность, представив новый язык программирования, получивший название Swift. Это стало большой неожиданностью: максимум, на что рассчитывали разработчики, привыкшие к теперь уже уходящему в прошлое языку Objective-C, — это обзор новых возможностей iOS 8 и новые прикладные программные интерфейсы для работы с ними. Оправившись от шока, разработчики подступились к Swift, изучая и, конечно же, критикуя его. Все это время Swift активно развивался. И вот теперь мы ожидаем выхода уже пятой версии языка программирования, привносящей в него много нового и интересного.

Если вы когда-либо писали приложения на языке Objective-C, то после изучения Swift с его многообразием возможностей вы, вероятно, захотите переписать свои приложения на новом языке программирования[1]. После выхода Swift многие разработчики пошли именно по этому пути, понимая, что в будущем наибольшее внимание Apple будет уделять развитию нового языка. Более того, Swift стал первой разработкой Apple с открытым исходным кодом, что говорит о скором внедрении его поддержки и другими операционными системами (ждем поддержки Windows)[2].

---

[1] Swift в значительной мере отличается от Objective-C по части удобства программирования. Однако в редких случаях при разработке программ вам, возможно, придется использовать вставки, написанные на Objective-C.

[2] В настоящее время приложения на Swift можно разрабатывать не только для операционных систем iOS и OS X, но и для watchOS (операционная система «умных» часов Apple Watch) и tvOS (операционная система телевизионной приставки Apple TV четвертого поколения). Однако изучение приемов разработки приложений для различных операционных систем выходит за рамки темы данной книги.

Если вы когда-либо программировали на других языках, то могу предположить, что после знакомства со Swift и со всем многообразием его возможностей вы не захотите возвращаться в «старый лагерь».

Помните, что Swift затягивает и не отпускает!

О Swift

Swift — это быстрый, современный, безопасный и удобный язык программирования. С его помощью процесс создания программ становится очень гибким и продуктивным, так как Swift вобрал в себя лучшее из таких языков, как C, Objective-C и Java. Swift на редкость удобен для изучения, восприятия и чтения кода. У него очень перспективное будущее.

Изучая этот замечательный язык, вы удивитесь, насколько он превосходит другие языки программирования, на которых вы раньше писали. Его простота, лаконичность и невероятные возможности просто поразительны!

Язык Swift создан полностью с нуля и обладает рядом особенностей.

Современность

Swift является результатом комбинации последних изысканий в области программирования и опыта, полученного в процессе работы по созданию продуктов экосистемы Apple.

Объектоориентированность

Swift — объектно-ориентированный язык программирования, придерживающийся парадигмы «всё — это объект». Если в настоящий момент данное утверждение показалось вам непонятным, не переживайте: чуть позже мы еще вернемся к нему.

Читабельность, экономичность и лаконичность кода

Swift просто создан для того, чтобы быть удобным в работе и максимально понятным. Он имеет простой и прозрачный синтаксис, позволяющий сокращать многострочный код, который вы, возможно, писали в прошлом, до однострочных (а в некоторых случаях — односимвольных!) выражений.

Безопасность

В рамках Swift разработчики попытались создать современный язык, свободный от уязвимостей и не требующий излишнего напряжения программиста при создании приложений. Swift имеет строгую типизацию: в любой момент времени вы точно знаете, с объектом какого типа работаете. Более того, при создании приложений вам практически не требуется думать о расходуемой оперативной памяти, Swift все делает за вас в автоматическом режиме.

Производительность

Swift все еще очень молод, но по производительности разрабатываемых программ он приближается (а в некоторых случаях уже и обгоняет) ко всем известному «старичку» — языку программирования C++[1].

Актуальность

Swift — современный язык программирования, и поэтому он должен поддерживать свою репутацию на протяжении всего жизненного цикла. Это стало возможным благодаря активно растущему сообществу swift-разработчиков, штаб-квартирой которых стал портал swift.org, на котором размещены все необходимые данные о том, как стать членом этой быстро растущей семьи. Для русскоязычного сообщества уже созданы несколько профильных сайтов, одним из которых является swiftme.ru.

Эти особенности делают Swift по-настоящему удивительным языком программирования. А сейчас для вас самое время погрузиться в мир Swift: он еще очень и очень молод, людей со значительным багажом знаний и опыта за плечами пока просто не существует в силу возраста языка, поэтому в перспективе вы можете стать одним из них.

О книге

Использование смартфонов для решения возникающих задач стало нормой. В связи с этим многие компании обращают все более при-

[1] Соответствующие тесты периодически проводит и размещает на своем портале компания Primate Tabs — разработчик популярного тестера производительности Geekbench.

стальное внимание на обеспечение функционального доступа к предлагаемым ими услугам посредством мобильных приложений (будь то оптимизированный интернет-сайт, открываемый в браузере, или специальная программа). iOS является одной из популярнейших мобильных операционных систем в мире, и в такой ситуации спрос на мобильное программирование растет небывалыми темпами.

Книга содержит исчерпывающую информацию для всех желающих научиться программировать на замечательном языке Swift с целью создания собственных iOS-приложений (в том числе и для macOS, tvOS и watchOS) или программ для операционной системы Linux. В ходе чтения книги вы встретите не только теоретические сведения, но и большое количество практических примеров и заданий, выполняя которые вы углубите свои знания изучаемого материала. Вам предстоит пройти большой путь, и это будет нужный и очень важный опыт. Хотя книга предназначена в первую очередь для изучения языка Swift, вы получите некоторые начальные сведения о принципах разработки полноценных приложений. Можете быть уверены, эта информация будет очень полезной. Книга даст возможность освоить новый язык и в скором времени приступить к написанию собственных приложений для App Store или Mac App Store. Изучив язык, в дальнейшем вы сможете самостоятельно выбрать, для какой платформы создавать программы. Несмотря на то что здесь приводится информация о разработке и под операционную систему Linux, основной упор делается именно на разработку на платформе от Apple. Примеры кода соответствуют Swift версии не ниже 5, iOS версии не ниже 12 и Xcode версии не ниже 10. Если у вас более свежие версии, не беспокойтесь, весь описанный материал с большой долей вероятности будет без каких-либо ошибок работать и у вас. Но небольшая возможность того, что Apple незначительно изменит синтаксис Swift, все же существует. Если вы встретитесь с такой ситуацией, прошу отнестись с пониманием и сообщить мне об этом в Telegram-канал или оставить сообщение на портале swiftme.ru.

Swiftme.ru — это развивающееся сообщество программистов Swift. Здесь вы найдете ответы на различные вопросы, возникающие в ходе обучения и разработки, а также уроки и курсы, которые помогут вам глубоко изучить тему разработки приложений.

О домашних заданиях

В пятом издании мне пришлось убрать из книги практически все задания для самостоятельного решения. Причиной этому стало увеличение количества учебного материала и ограничения на размер книги. Но не переживайте, все задания из старых изданий, включая новые и интересные, размещены на сайте swiftme.ru. Все, что вам нужно сделать, — зайти на swiftme.ru, зарегистрироваться, после чего получить доступ к базе учебных материалов.

Советую ознакомиться с представленной на сайте информацией прямо сейчас, так как я не буду ограничиваться одними лишь заданиями. Со временем (возможно, уже сейчас) там появятся дополнительные учебные видео, которые, без сомнения, смогут значительно упростить изучение Swift.

Заглядывайте в учебный раздел перед каждой главой книги, так вы будете знать об имеющихся дополнительных материалах и заданиях для самостоятельного решения. Не пренебрегайте этим советом.

Очень часто при изучении языка или написании собственных приложений начинающие разработчики пользуются нехитрым приемом «копировать/вставить». Они копипастят все, начиная от решения домашних заданий и заканчивая найденными рецептами. Этот подход плох тем, что чаще всего человек не разбирается в том, что копирует. Решение задачи проходит мимо и не оседает в его голове.

Конечно, в некоторых случаях такой подход может ускорить достижение конечной цели: написание программы или последней страницы книги. Но в действительности ваша цель в получении глубоких знаний для повышения собственного уровня так и не будет полностью достигнута.

Я настоятельно советую разбирать каждый пример или рецепт и не гнаться за готовыми решениями. Каждое нажатие на клавиши, каждое написание символа должно быть осознанным.

Если у вас возникают проблемы с решением какого-либо задания, вы можете обратиться к нашему сообществу в Telegram.

Тем не менее советую изо всех сил стараться решать задания самостоятельно, используя при этом помощь сообщества, книгу, а также другие справочные материалы. Но не ориентируйтесь на то, чтобы

посмотреть (или, может, правильнее — подсмотреть?) готовое решение.

Экспериментируйте, пробуйте, тестируйте — среда разработки выдержит даже самый некрасивый и неправильный код!

Исправления в пятом издании

Вы держите в руках пятое издание книги «Swift: Основы разработки под iOS и macOS». Каждый автор хочет создать действительно ценный продукт, и не замечать мысли и предложения читателей было бы большой глупостью и проявлением эгоизма. В ходе долгого и плодотворного общения со многими из вас была выработана масса идей, благодаря которым новое издание стало по-настоящему полезным. Огромное спасибо всем участникам каналов в Telegram — с вашей помощью книга становится лучше и интереснее.

В сравнении с предыдущим изданием эта книга содержит следующие изменения и дополнения:

- Весь материал актуализирован в соответствии со Swift версии 5 и Xcode 10.
- Добавлено большое количество нового учебного материала, в частности, связанного с принципами функционирования типа данных String «под капотом».
- Учтены пожелания и замечания пользователей по оформлению и содержанию.
- Исправлены найденные опечатки.
- Домашние задания перенесены на портал swiftme.ru, что позволило увеличить количество учебного материала.

Для кого написана книга

Ответьте для себя на следующие вопросы:

- Имеете ли вы минимальные знания о программировании на любом языке высокого уровня?
- Хотите ли вы научиться создавать программы для операционной системы iOS (для вашего гаджета iPhone и iPad), macOS, watchOS или tvOS?

- Предпочитаете ли вы обучение в практической форме скучным и монотонным теоретическим лекциям?

Если вы ответили на них утвердительно, то эта книга для вас.

Изучаемый материал в книге подкреплен практическими домашними заданиями. Мы вместе пройдем путь от самых простых понятий до решения интереснейших задач.

Не стоит бояться, Swift вовсе не отпугнет вас (как это мог сделать Objective-C), а процесс создания приложений окажется очень увлекательным. А если у вас есть идея потрясающего приложения, то совсем скоро вы сможете разработать его для современной мобильной системы iOS, стационарной системы macOS или Linux, смарт-часов Apple Watch или телевизионной приставки AppleTV.

Очень важно, чтобы вы не давали своим рукам «простаивать». Тестируйте весь предлагаемый код и выполняйте все задания, так как учиться программировать, просто читая текст, — не лучший способ. Если в процессе изучения нового материала у вас появится желание поиграть с кодом из листингов — делайте это не откладывая. Постигайте Swift!

Не бойтесь ошибаться: пока вы учитесь, ошибки — ваши друзья. А исправлять их и избегать в будущем вам помогут среда разработки Xcode (о ней мы поговорим позже) и моя книга.

Помните: чтобы стать великим программистом, требуется время! Будьте терпеливы и внимательно изучайте материал. Желаю увлекательного путешествия!

Что нужно знать, прежде чем начать читать

Единственное и самое важное требование — вы должны иметь навыки работы с компьютером: уметь скачивать, устанавливать и открывать программы, пользоваться мышью и клавиатурой, а также иметь общие навыки работы с операционной системой. Как видите, я прошу не так уж много.

Если вы раньше программировали на каких-либо языках, это очень поможет, так как у вас уже достаточно базовых знаний для успешного освоения материала. Если же это не так, не переживайте — я попытаюсь дать максимально полный материал, который позволит проходить урок за уроком.

Структура книги

Книга состоит из семи частей:

- **Часть I. Подготовка к разработке Swift-приложений**. В первой части вы начнете путешествие в мир Swift, выполните самые важные и обязательные шаги, предшествующие началу разработки собственных приложений. Вы узнаете, как завести собственную учетную запись Apple ID, как подключиться к программе apple-разработчиков, где взять среду разработки Swift-приложений, как с ней работать.
- **Часть II. Базовые возможности Swift**. После знакомства со средой разработки Xcode, позволяющей приступить к изучению языка программирования, вы изучите базовые возможности Swift. Вы узнаете, какой синтаксис имеет Swift, что такое переменные и константы, какие типы данных существуют и как всем этим пользоваться при разработке программ.
- **Часть III. Контейнерные типы данных**. Что такое последовательности и коллекции и насколько они важны для создания ваших программ? В этой части книги вы познакомитесь с наиболее важными элементами языка программирования.
- **Часть IV. Основные возможности Swift.** Четвертая часть фокусируется на рассмотрении и изучении наиболее простых, но очень интересных средств Swift, позволяющих управлять ходом выполнения приложений.
- **Часть V. Введение в разработку приложений**. Эта часть посвящена изучению основ среды разработки Xcode, а также созданию двух первых консольных приложений.
- **Часть VI. Нетривиальные возможности Swift.** В шестой части подробно описываются приемы работы с наиболее мощными и функциональными средствами Swift. Материал этой части вы будете использовать с завидной регулярностью при создании собственных приложений в будущем. Также отличительной чертой данной части является большая практическая работа по созданию первого интерактивного приложения в Xcode Playground.
- **Часть VII. Введение в мобильную разработку.** В конце долгого и увлекательного пути изучения языка и создания некоторых простых приложений вам предстоит окунуться в мир разработки

полноценных программ. Из этой части вы узнаете основы создания интерфейсов и работы программ в Xcode «под капотом». Все это в будущем позволит вам с успехом осваивать новый материал и создавать прекрасные проекты.

Условные обозначения

ПРИМЕЧАНИЕ В данном блоке приводятся примечания и замечания.

Листинг

```
А это примеры кода (листинги)
```

СИНТАКСИС

В таких блоках приводятся синтаксические конструкции с объяснением вариантов их использования.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Часть I. Подготовка к разработке Swift-приложений

В первой части вы узнаете о том, что необходимо для начала разработки приложений на языке Swift. В настоящее время существует возможность разработки Swift-приложений под операционными системами macOS и Linux. Для полноценного обучения необходим именно macOS, две другие ОС будут сильно ограничивать ваши возможности в ходе изучения.

Так как на данном этапе нашей целью является лишь изучение данного языка программирования, то вы можете выбрать наиболее подходящую для вас платформу, но если в будущем вы ставите своей целью разработку собственных программ для продукции фирмы Apple, то работа в системе macOS является обязательным условием. Позже мы вернемся к этому вопросу.

В этой части книги приведены инструкции для начала разработки под каждой из доступных систем, но в дальнейшем упор все же будет сделан именно на работу под macOS.

ПРИМЕЧАНИЕ Есть способ приступить к полноценному изучению Swift и разработке собственных приложений, не имея Mac. Вы можете узнать об этом подробнее на нашем Telegram-канале.

- ✓ Глава 1. Подготовка к разработке в macOS
- ✓ Глава 2. Подготовка к разработке в Linux
- ✓ Глава 3. Подготовка к разработке в Windows

1 Подготовка к разработке в macOS

1.1. Компьютер Mac

Прежде чем приступить к разработке программ на языке Swift в macOS, вам потребуется несколько вещей. Для начала понадобится компьютер iMac, MacBook, Mac mini или Mac Pro с установленной операционной системой macOS. Лучше, если это будет macOS Mojave (10.14). В этом случае вы сможете использовать последнюю версию среды разработки Xcode и языка Swift.

Это первое и базовое требование связано с тем, что среда разработки приложений Xcode создана компанией Apple исключительно с ориентацией на собственную платформу.

Если вы ранее никогда не работали с Xcode, то будете поражены широтой возможностей данной среды и необычным подходом к разработке приложений. Естественно, далеко не все возможности рассмотрены в книге, поэтому я советую вам в дальнейшем самостоятельно продолжить ее изучение.

В том случае, если вы уже имеете опыт работы с Xcode, можете пропустить данную главу и перейти непосредственно к изучению языка Swift.

1.2. Зарегистрируйтесь как Apple-разработчик

Следующим шагом должно стать получение учетной записи Apple ID и регистрация в центре Apple-разработчиков. Для этого необходимо пройти по ссылке https://developer.apple.com/register/ в вашем браузере (рис. 1.1).

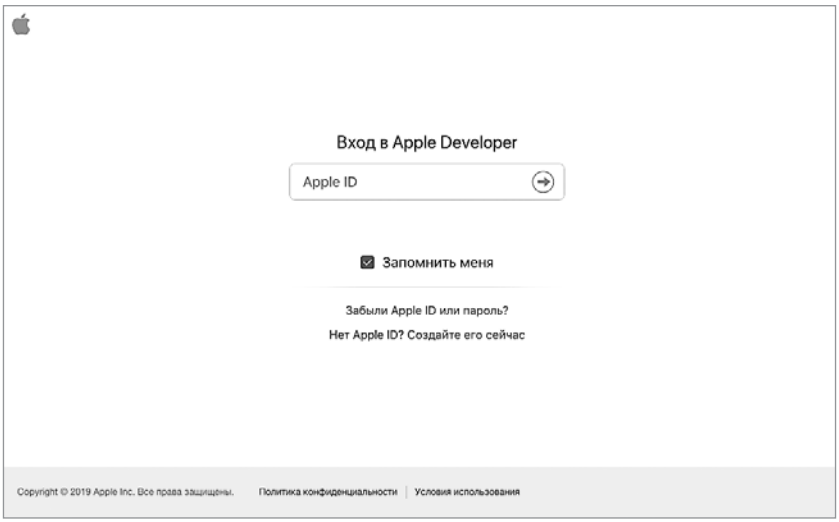

Рис. 1.1. Страница входа в Центр разработчиков

Примечание Apple ID — это учетная запись, которая позволяет получить доступ к сервисам, предоставляемым фирмой Apple. Возможно, вы уже имеете личную учетную запись Apple ID. Она используется, например, при покупке мобильных приложений в AppStore или при работе с порталом iCloud.com.

Если у вас уже есть учетная запись Apple ID, то используйте данные своей учетной записи для входа в Центр разработчиков. В ином случае нажмите кнопку Создать Apple ID сейчас и введите требуемые для регистрации данные.

На стартовой странице Центра разработчиков необходимо перейти по ссылке Developer, расположенной в верхней части страницы, после чего откроется страница, содержащая ссылки на основные ресурсы для разработчиков (рис. 1.2).

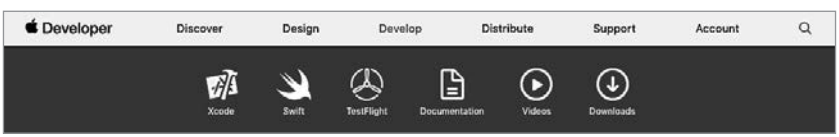

Рис. 1.2. Перечень ресурсов, доступных в Центре разработчиков

В Центре разработчиков вы можете получить доступ к огромному количеству различной документации, видео, примеров кода — ко всему, что поможет создавать отличные приложения.

Регистрация в качестве разработчика бесплатна. Таким образом, каждый может начать разрабатывать приложения, не заплатив за это ни копейки (если не учитывать стоимость компьютера). Тем не менее за 99 долларов в год вы можете участвовать в платной программе iOS-разработчиков (iOS Developer Program), которую вам предлагает Apple. Это не обязательно, но если вы хотите распространять свои приложения, то есть использовать мобильный магазин приложений App Store, то участие в программе становится обязательным.

Лично я советую вам пока не задумываться об этом, так как все навыки iOS-разработки можно получить с помощью бесплатной учетной записи и Xcode.

1.3. Установите Xcode

Теперь все, что вам необходимо, — скачать Xcode. Для этого перейдите в раздел Xcode и нажмите кнопку Download в правом верхнем углу страницы. Перед вами откроется страница, содержащая доступные для скачивания версии данной среды разработки (рис. 1.3).

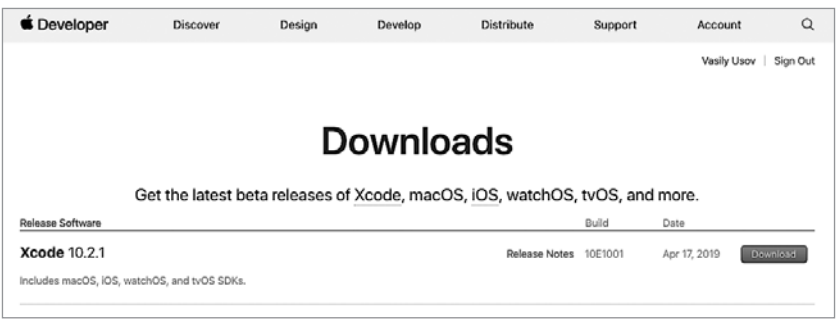

Рис. 1.3. Страница Xcode в Центре разработчиков

После щелчка по ссылке Download произойдет автоматический переход в магазин приложений Mac App Store на страницу Xcode. Здесь вы можете увидеть краткое изложение всех основных возможностей среды разработки, обзор последнего обновления и отзывы пользователей. Для скачивания Xcode просто щелкните на кнопке

Загрузить и при необходимости введите данные своей учетной записи Apple ID.

После завершения процесса установки вы сможете найти Xcode в Launchpad или в папке Программы в Доке.

1.4. Введение в Xcode

Изучение программирования на языке Swift мы начнем со знакомства со средой разработки Xcode.

ПРИМЕЧАНИЕ Интегрированная среда разработки (Integrated Development Environment, IDE) — набор программных средств, используемый программистами для разработки программного обеспечения (ПО).

Среда разработки обычно включает в себя:

- текстовый редактор;
- компилятор и/или интерпретатор;
- средства автоматизации сборки;
- отладчик.

Xcode — это IDE, то есть среда создания приложений для iOS и macOS. Xcode — это наиболее важный инструмент, который использует разработчик. Среда Xcode удивительна! Она предоставляет широкие возможности, и изучать их следует постепенно, исходя из поставленных и возникающих задач. Внешний вид рабочей среды приведен на рис. 1.4.

Именно с использованием этого интерфейса разрабатываются любые приложения для «яблочных» продуктов. При изучении Swift вы будете взаимодействовать с иной рабочей областью — рабочим интерфейсом playground-проектов. О нем мы поговорим чуть позже.

Xcode распространяется на бесплатной основе. Это полифункциональное приложение без каких-либо ограничений в своей работе. В Xcode интегрированы: пакет iOS SDK, редактор кода, редактор интерфейса, отладчик и многое другое. Также в него встроены симуляторы iPhone, iPad, Apple Watch и Apple TV. Это значит, что все создаваемые приложения вы сможете тестировать прямо в Xcode (без необходимости загрузки программ на реальные устройства). Более подробное изучение состава и возможностей данной IDE отложим до момента изучения процесса разработки приложений. Сейчас все это не столь важно.

Я надеюсь, что вы уже имеете на своем компьютере последнюю версию Xcode, а значит, мы можем перейти к первому знакомству с этой

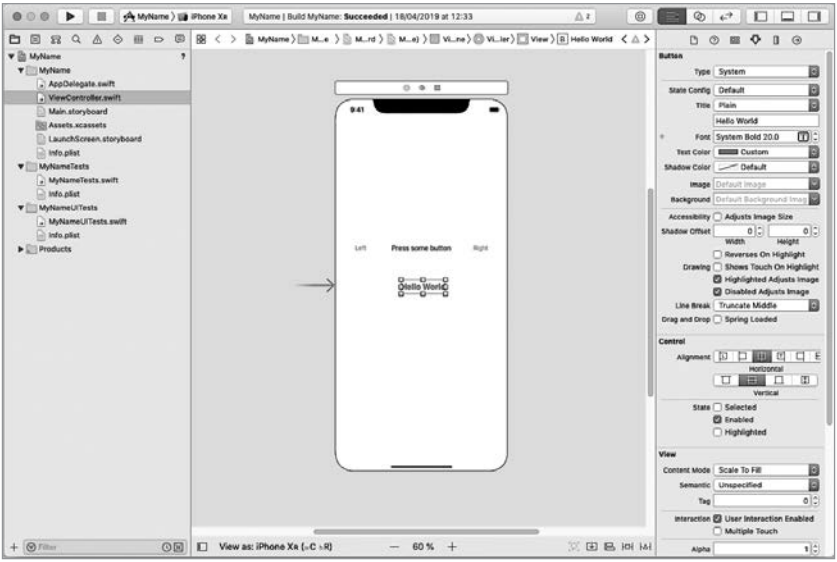

Рис. 1.4. Интерфейс Xcode

замечательной средой. Для начала необходимо запустить Xcode. При первом запуске, возможно, вам придется установить некоторые дополнительные пакеты (все пройдет в автоматическом режиме при щелчке на кнопке install в появившемся окне).

После скачивания и полной установки Xcode вы можете приступить к ее использованию. Чуть позже вы создадите свой первый проект, а сейчас просто взгляните на появившееся при запуске Xcode стартовое окно (рис. 1.5).

Стартовое окно служит для двух целей: создания новых проектов и организации доступа к созданным ранее. В данном окне можно выделить две области. Нижняя левая область представляет собой меню, состоящее из следующих пунктов:

- Get started with a playground — создание нового playground-проекта. О том, что это такое, мы поговорим чуть позже.
- Create a new Xcode project — создание нового приложения для iOS или OS X.
- Clone an existing project — подключение внешнего репозитория для поиска размещенных в нем проектов.

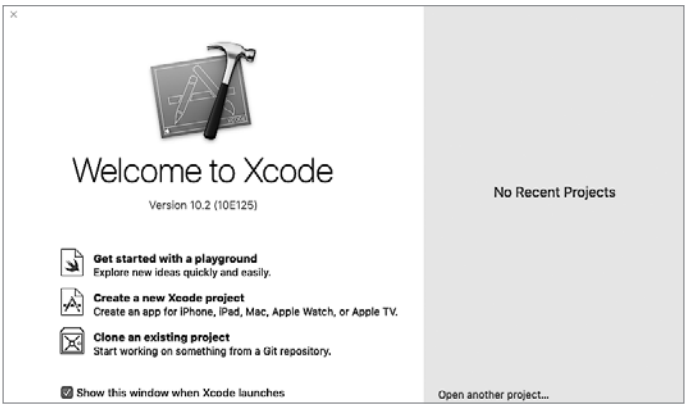

Рис. 1.5. Стартовое окно Xcode

Правая часть окна содержит список созданных ранее проектов. В вашем случае, если вы запускаете Xcode впервые, данный список будет пустым. Но не переживайте, в скором времени он наполнится множеством различных проектов.

ПРИМЕЧАНИЕ В названиях всех создаваемых в ходе чтения книги проектов я советую указывать номера глав и/или листингов. В будущем это позволит навести порядок в списке проектов и оградит вас от лишней головной боли.

Одной из потрясающих возможностей Xcode является наличие playground-проектов. Playground-проект — это интерактивная среда разработки, своеобразная «песочница» или «игровая площадка», где вы можете комфортно тестировать создаваемый вами код и видеть результат его исполнения в режиме реального времени. С момента своего появления playground активно развивается. С каждой новой версией Xcode в него добавляются все новые и новые возможности.

Представьте, что вам нужно быстро проверить небольшую программу. Для этой цели нет ничего лучше, чем playground-проект! Пример приведен на рис. 1.6.

Как вы можете видеть, внешний вид интерфейса playground-проекта значительно отличается от рабочей области Xcode, которую вы видели ранее в книге. Повторю, что playground-проект позволяет писать код и незамедлительно видеть результат его исполнения, хотя и не служит для создания полноценных самостоятельных программ. Каждый playground-проект хранится в системе в виде особого файла с расши-

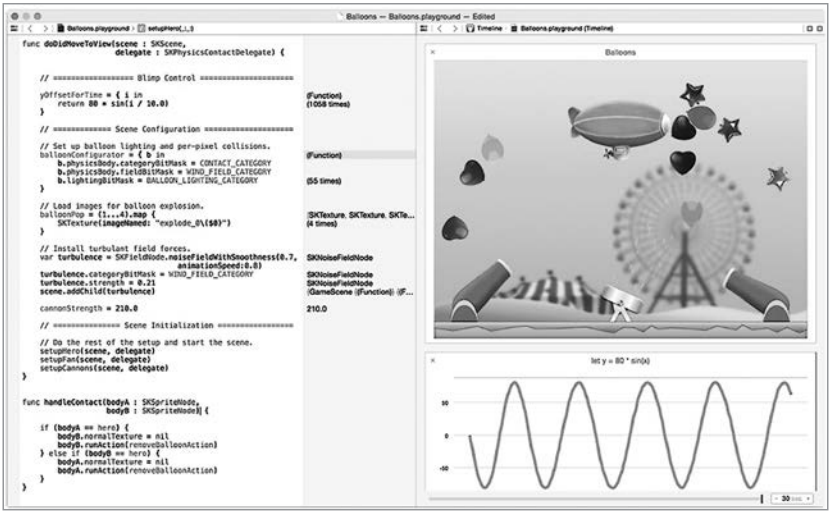

Рис. 1.6. Пример playground-проекта

рением playground. Хочу отметить, что при разработке полноценных приложений вы можете использовать playground-проекты в качестве их составных частей и использовать реализованные в них механизмы.

1.5. Интерфейс playground-проекта

Нет способа лучше для изучения языка программирования, чем написание кода. Playground-проект предназначен именно для этого. Выберите вариант Get started with a playground в стартовом окне для создания нового playground-проекта. Далее Xcode предложит вам выбрать один из типов создаваемого проекта (рис. 1.7), которые отличаются между собой лишь предустановленным в проекте кодом.

ПРИМЕЧАНИЕ Обратите внимание, что в верхней части окна есть возможность выбора платформы (iOS, tvOS, macOS). В книге будет описываться разработка под iOS и использоваться только первая вкладка, но если вы из числа любителей изучать что-то новое, то советую со временем заглянуть в каждую из вкладок и типов проектов. Это будет очень полезно.

Сейчас вам необходимо выбрать тип Blank, который содержит минимальное количество кода. После нажатия кнопки Next среда разработки попросит вас ввести имя создаваемого playground-проекта. Измените имя

на «Part 1 Basics», выберите платформу iOS и щелкните на кнопке Create. После этого откроется рабочий интерфейс playground-проекта (рис. 1.8).

С первого взгляда окно playground-проекта очень похоже на обыкновенный текстовый редактор. Но это только с первого взгляда. Xcode Playground имеет куда более широкие возможности.

Рис. 1.7. Окно выбора типа создаваемого проекта

Рис. 1.8. Рабочий интерфейс playground-проекта

Как показано на рис. 1.8, рабочее окно состоит из 5 основных частей:

- В левой части экрана расположен *редактор кода*, в котором вы можете писать и редактировать свой swift-код. В только что созданном нами файле имеется один комментарий и две строки кода.
- Как только код будет написан, Xcode моментально обработает его, отобразит ошибки и выведет результат в правой части экрана в области результатов (рис. 1.8а).

Рис. 1.8а. Отображение значения в области результатов

ПРИМЕЧАНИЕ Если Playground не отображает результаты, то для принудительного запуска обработки кода вы можете использовать либо кнопку слева от строчки кода, либо кнопку, расположенную в нижней части окна (рис. 1.8б). Бывают случаи, что playground зависает во время обработки кода или не запускает ее в автоматическом режиме.

Вы можете видеть, что результат параметра str отображается в области результатов. В дальнейшем мы вместе будем писать код и обсуждать результаты его выполнения. Помните, что основная цель — повышение вашего уровня владения Swift.

Если навести указатель мыши на строку `"Hello, playground"` в области результатов, то рядом появятся две кнопки, как показано на рис. 1.9.

Левая кнопка (изображение глаза) позволяет отобразить результат в отдельном всплывающем окне, правая — прямо в области кода. Попробуйте щелкнуть на каждой из них и посмотрите на результат.

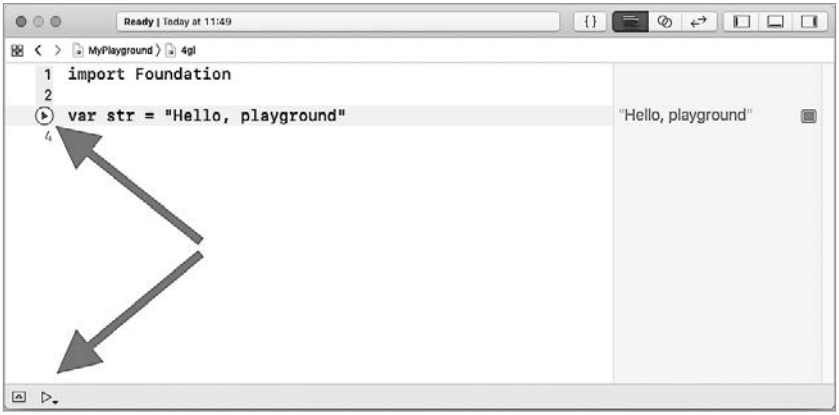

Рис. 1.8б. Кнопки запуска обработки кода

Рис. 1.9. Дополнительные кнопки в области результатов

1.6. Возможности playground-проекта

Playground — это потрясающая платформа для разработки кода и написания обучающих материалов. Она просто создана для того, чтобы тестировать появляющиеся мысли и находить решения возникающих в процессе разработки проблем. Playground-проекты обладают рядом возможностей, благодаря которым процесс разработки можно значительно улучшить.

Начиная с версии 6.3, в Xcode появилась поддержка markdown-синтаксиса для комментариев. На рис. 1.10 приведен пример изменения внешнего вида комментариев после выбора в меню пункта Editor ▸ Show Rendered Markup. В верхней части изображен исходный код комментария, а в нижней — отформатированный.

В скором времени вы увидите, что в качестве результатов могут выводиться не только текстовые, но и графические данные (рис. 1.11).

Строки формата *N* `times` в области результатов, где *N* — целое число, говорят о том, что данная строка кода выводится *N* раз. Пример такой строки вы можете видеть на рис. 1.11. Подобные выводы результатов

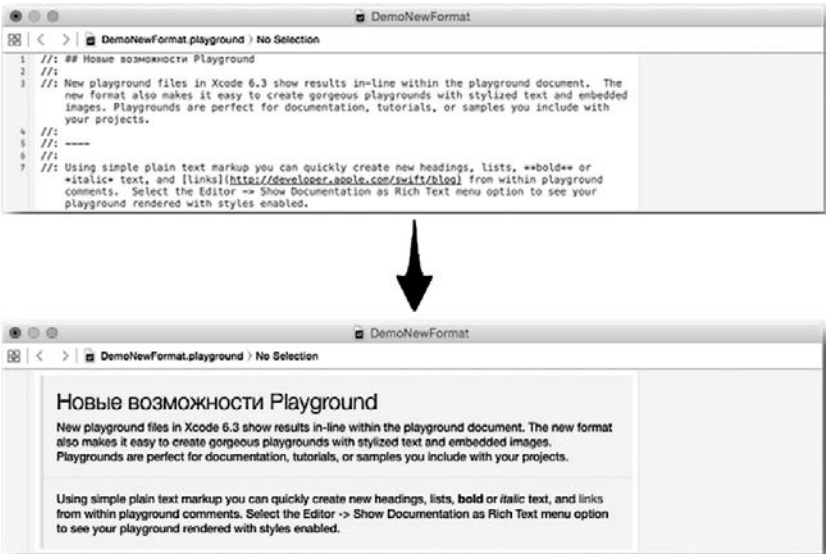

Рис. 1.10. Форматированный комментарий

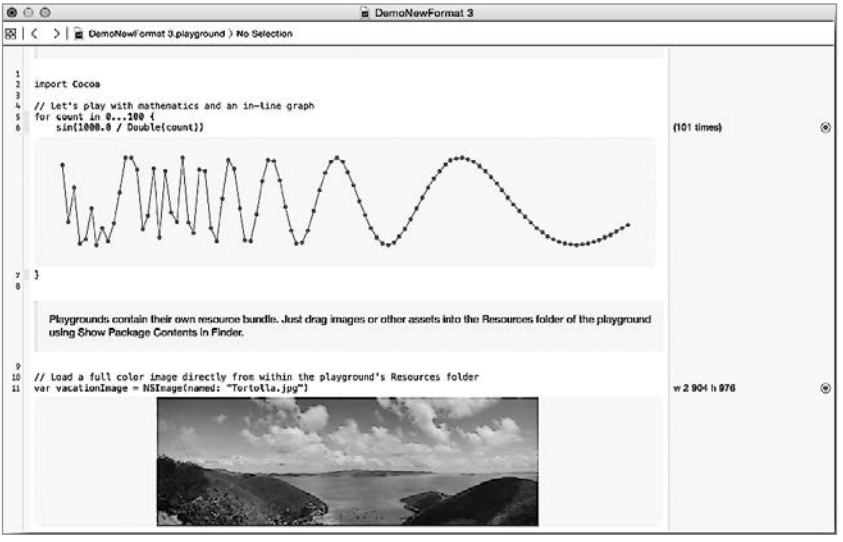

Рис. 1.11. Пример вывода результирующей информации

можно отобразить в виде графиков и таблиц. Со всеми возможными вариантами отображения результатов исполнения swift-кода вы познакомитесь в ходе работы с playground-проектами в Xcode.

Также Xcode имеет в своем арсенале такой полезный механизм, как автодополнение (в Xcode известное как автокомплит). Для примера в рабочей части только что созданного playground-проекта на новой строке напишите латинский символ «a» — вы увидите, что всплывет окно автодополнения (рис. 1.12).

Все, что вам нужно, — выбрать требуемый вариант и нажать клавишу ввода, и он появится в редакторе кода. Список в окне автодополнения меняется в зависимости от введенных вами символов. Также все создаваемые элементы (переменные, константы, типы, экземпляры и т. д.) автоматически добавляются в список автодополнения.

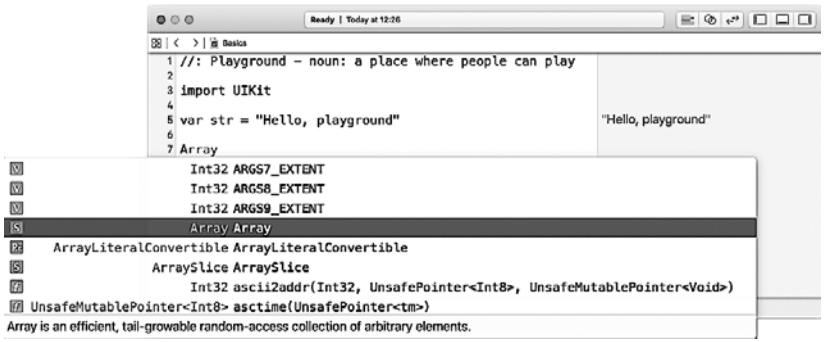

Рис. 1.12. Окно автодополнения в Xcode

Одной из возможностей Xcode, которая значительно упрощает работу, является указание на ошибки в программном коде. Для каждой ошибки выводится подробная вспомогательная информация, позволяющая внести ясность и исправить недочет. Ошибка показывается с помощью красного значка в форме кружка слева от строки, где она обнаружена. При щелчке на этом значке появляется описание ошибки (рис. 1.13).

Дополнительно информация об ошибке выводится на консоли в области отладки (Debug Area). Вывести ее на экран можно, выбрав в меню пункт View ▸ Debug Area ▸ Show Debug Area или щелкнув на кнопке с направленной вниз стрелкой в левом нижнем углу области кода. Вы будете регулярно взаимодействовать с консолью в процессе разработки программ.

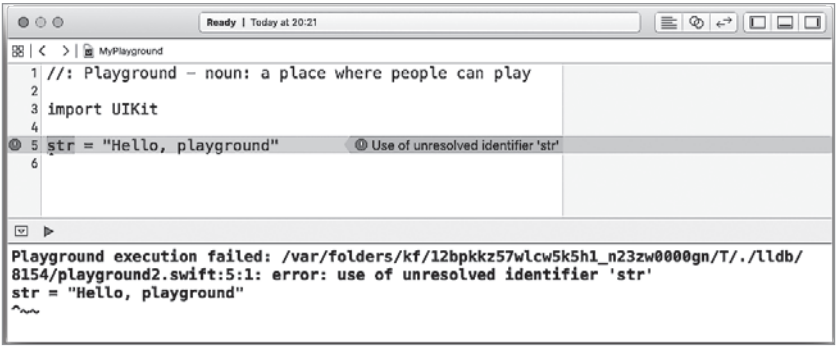

Рис. 1.13. Отображение ошибки в окне playground-проекта

Возможно, что при открытии области отладки консоль будет пуста. Данные в ней появятся после появления в вашем коде первой ошибки или первого вывода информации по требованию.

Swift позволяет также получать исчерпывающую информацию об используемых в коде объектах. Если нажать клавишу Alt и щелкнуть на любом объекте в области кода (например, на `str`), то появится вспомогательное окно, позволяющее узнать тип объекта, а также имя файла, в котором он расположен (рис. 1.14).

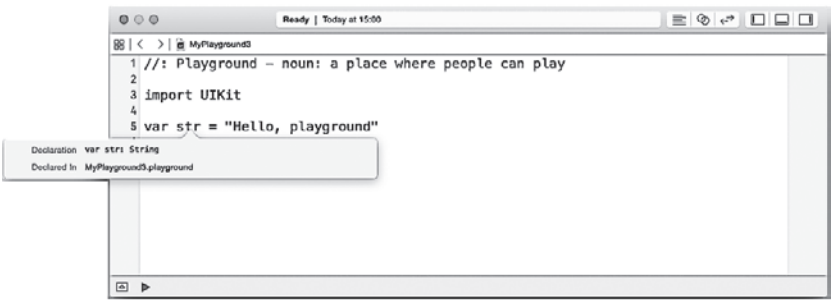

Рис. 1.14. Всплывающее окно с информацией об объекте

Среда Xcode вместе с playground-проектами дарит вам поистине фантастические возможности для реализации своих идей!

2 Подготовка к разработке в Linux

Если вы решили программировать на Swift в Linux, то сначала вам следует установить набор компонентов, которые дадут возможность создавать swift-приложения.

Программирование под Linux в значительной мере отличается от разработки в OS X, в особенности из-за отсутствия среды разработки Xcode. Со временем и в Linux обязательно появятся среды разработки и редакторы кода, позволяющие писать и компилировать Swift-код, но в данный момент придется обойтись имеющимися средствами.

ПРИМЕЧАНИЕ В этой главе описывается установка Swift 3 на Ubuntu 14.04. С момента ее написания в процессе установки принципиально ничего не изменилось. Если вам потребуется более свежая версия языка или вы используете новые редакции Ubuntu, то все равно сможете с легкостью применить описанные ниже инструкции.

В качестве операционной системы мной была выбрана Ubuntu 14.04. Это проверенная временем, очень стабильная сборка, имеющая максимально удобный интерфейс. В настоящее время на портале swift.org имеются версии и для других версий ОС.

В первую очередь необходимо скачать обязательные пакеты, которые обеспечивают работу компилятора. Для этого откройте Terminal и введите следующую команду:

```
sudo apt-get install clang libicu-dev
```

После скачивания при запросе установки введите Y и нажмите Enter (рис. 2.1).

Установка может продолжаться длительное время, но весь процесс будет отражен в консоли.

Теперь необходимо скачать последнюю доступную версию Swift. Для этого посетите страницу http://swift.org/download и выберите соответствующую вашей операционной системе сборку (рис. 2.2).

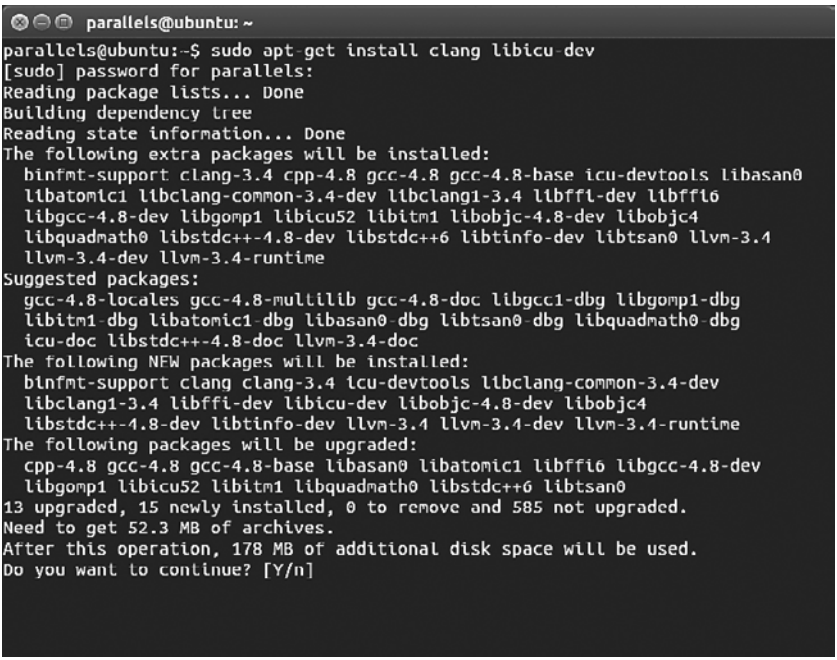

Рис. 2.1. Установка пакетов в Ubuntu

После скачивания дважды щелкните на скачанном архиве в формате tar.gz и распакуйте его в произвольную папку. На рис. 2.3 архив распакован в папку Home. Выбранная вами папка будет домашней директорией для Swift.

Теперь введите в терминале следующую команду:

```
gedit .profile
```

Перед вами откроется текстовый редактор. Прокрутите текстовое содержимое файла и в самом конце, пропустив одну пустую строку, добавьте следующий текст:

```
export PATH=/path/to/usr/bin:"${PATH}"
```

где `/path/to/usr/bin` — это путь к папке bin внутри распакованного архива со Swift. В случае, если вы распаковали архив в папку home, путь будет выглядеть примерно так:

```
home/parallels/swift-3.0-RELEASE-ubuntu14.04/usr/bin
```

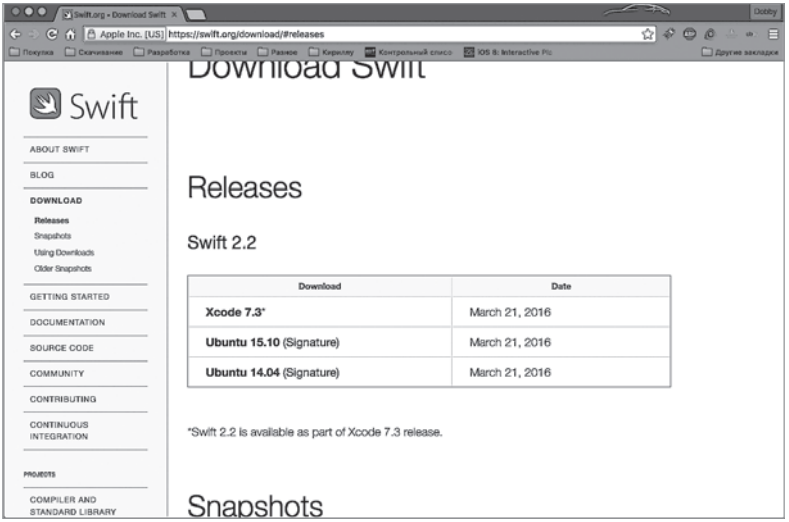

Рис. 2.2. Доступные релизы Swift

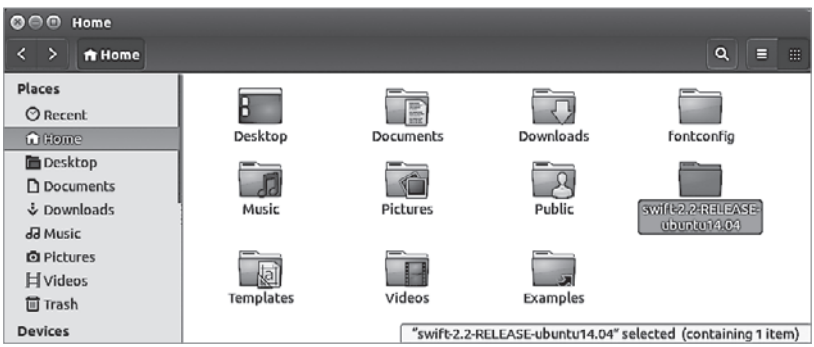

Рис. 2.3. Распакованный архив в папке Home

Скопируйте данную строку в буфер обмена, сохраните файл и закройте текстовый редактор. В окне терминала вставьте скопированную ранее команду и нажмите Enter (рис. 2.4).

На этом установка Swift завершена! Для проверки работоспособности вы можете ввести команду

```
swift –version
```

Она выведет версию установленной сборки Swift (рис. 2.4).

Чтобы выполнить swift-код, в произвольном месте диска создайте новый файл с расширением swift, содержащий некоторый swift-код. После этого в терминале введите команду

```
swift '/home/parallels/my.swift'
```

где

```
/home/parallels/my.swift
```

Рис. 2.4. Вывод информации о версии Swift

Это путь к созданному файлу.

После нажатия Enter в терминале отобразится результат выполнения кода (рис. 2.5).

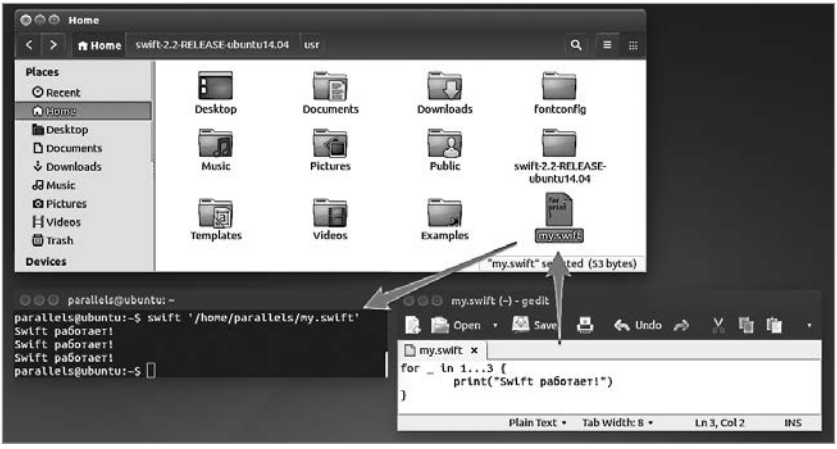

Рис. 2.5. Пример выполненного swift-кода

Вот и всё! Установка Swift в Linux такая же простая, как и установка Xcode в OS X. Помните, что весь приведенный далее материал ориентируется на OS X-версию.

3 Подготовка к разработке в Windows

С недавних пор у программистов появилась возможность попрактиковаться в разработке на Swift под ОС Windows. Как и в случае с Linux, разработка под Windows сейчас не может называться полноценной, так как пока не существует какой-либо среды разработки, позволяющей строить пользовательские интерфейсы.

Для программирования под Windows существует пакет Swift for Windows, который устанавливает необходимое окружение. Последнюю версию (сейчас это 1.6) вы можете найти на портале http://swiftforwindows.github.com. Скачав установочный файл, запустите его и инсталлируйте программу в произвольную папку (рис. 3.1).

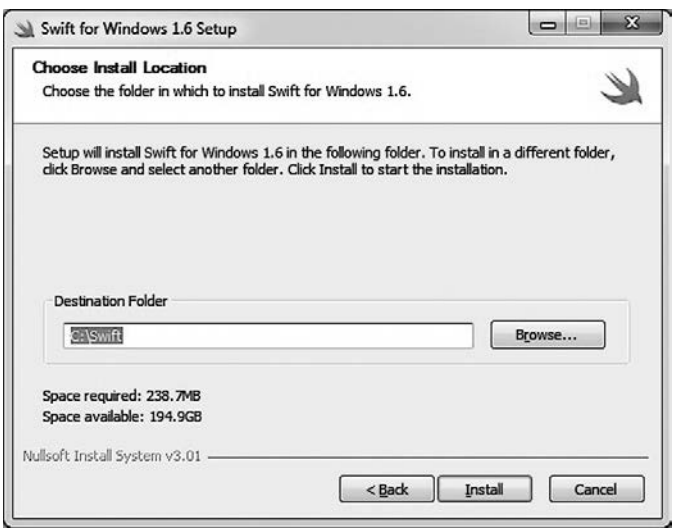

Рис. 3.1. Окно установки компилятора Swift

После установки будет запущено приложение, с помощью которого вы сможете скомпилировать написанный код (рис. 3.2).

Теперь вам достаточно просто написать произвольный код в файле с расширением swift и сохранить его на жестком диске. После этого с помощью кнопки Select File необходимо выбрать данный файл.

При нажатии на Compile отобразятся все ошибки, а при их отсутствии программа будет скомпилирована и готова к запуску, который можно осуществить нажатием на кнопку Run.

На этом всё! Установка Swift в Windows такая же простая, как и установка Xcode в OS X. Помните, что весь приведенный далее материал относится к OS X-версии.

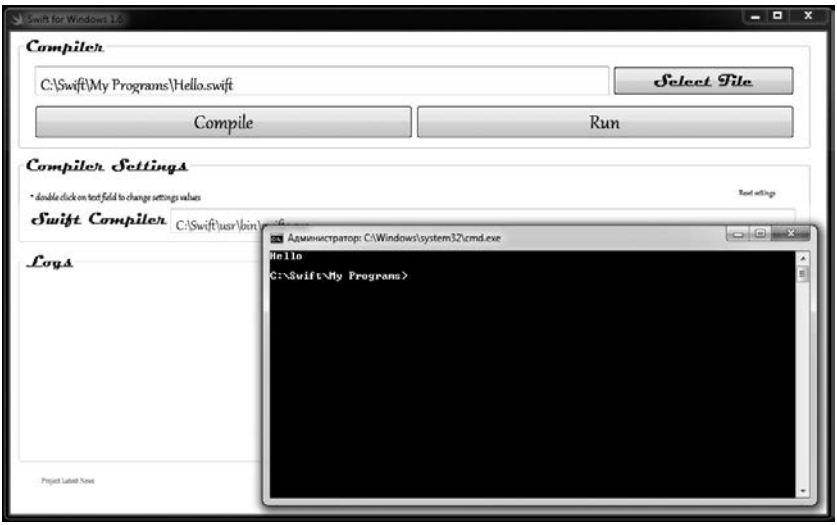

Рис. 3.2. Рабочее окно компилятора

Часть II. Базовые возможности Swift

Добро пожаловать во вселенную Swift 5!

Пришло время перейти к изучению этого прекрасного языка программирования. С каждым пройденным разделом, с каждой изученной страницей и каждым выполненным заданием вы будете все ближе к своей цели — к разработке потрясающих приложений и, возможно, к хорошему заработку.

Swift — крайне интересный язык программирования. Если ранее вы работали с другими языками программирования, то уже скоро заметите их сходство с детищем Apple. Данная часть книги расскажет вам о базовых понятиях, которые предшествуют успешному программированию, и обучит основам синтаксиса и работы с различными фундаментальными механизмами, которые легли в основу всей системы разработки программ на языке Swift.

Как и многие другие языки, Swift активно использует переменные и константы для хранения значений, а для доступа к ним служат идентификаторы (имена). Более того, Swift вывел функциональность переменных и констант на новый качественный уровень. И скоро вы в этом убедитесь. По ходу чтения книги вы узнаете о многих потрясающих нововведениях, которые не встречались вам в других языках, но на отсутствие которых вы, возможно, сетовали.

- ✓ Глава 4. Отправная точка
- ✓ Глава 5. Фундаментальные типы данных

4 Отправная точка

У каждого из вас свой уровень подготовки и разный опыт за плечами, каждый прошел свой собственный путь независимо от других. Все это накладывает определенные сложности при написании учебного материала: для кого-то он может быть слишком простым, а для кого-то чрезвычайно сложным! Найти золотую середину порой не так просто.

ПРИМЕЧАНИЕ В первую очередь эта книга ориентирована на начинающих разработчиков. Помните, что вы всегда можете пропустить то, что уже знаете. Не стоит выражать свое недовольство из-за наличия излишнего, с вашей точки зрения, материала.

4.1. Как компьютер работает с данными

С момента изобретения микропроцессора основные принципы работы персонального компьютера не претерпели существенных изменений. Он все так же орудует нулями и единицами, проводя с ними совсем нехитрые операции. Возможно, вы удивитесь, но современные электронные устройства не такие уж и умные, как нам рассказывают в рекламе: весь смарт-функционал определяется инженерами, проектирующими схемы, и программистами.

В отличие от обыкновенного калькулятора, компьютер предназначен для выполнения широкого (можно сказать, неограниченного) спектра задач. В связи с этим в его аппаратную архитектуру уже заложены большие функциональные возможности, а вся логика работы определяется программным обеспечением, создаваемым программистами.

В общем случае при максимальном упрощении можно выделить три основных функциональных уровня, обеспечивающих работу компьютера (рис. 4.1).

Рис. 4.1. Функциональные уровни компьютера

Аппаратный уровень представляет собой набор типовых блоков (физического оборудования), самыми важными из которых являются центральный процессор (*CPU*, Central Processing Unit) и оперативная *память*.

CPU — сердце любого широкофункционального электронного устройства (персональные компьютеры, планшеты, смартфоны и т. д.). Его изобретение стало настоящей революцией, ознаменовавшей собой появление современных вычислительных машин. Кстати, компьютеры называются вычислительными машинами неспроста. Их работа заключается в обработке чисел, в вычислении различных значений через выполнение простейших арифметических и логических операций, называемых инструкциями. Все данные в любом компьютере в итоге превращаются в числа. Причем неважно, говорим мы о фотографии, любимой песне или книге — для компьютера все это представляется в виде чисел.

CPU как раз и предназначен для работы с числами. Он получает на вход несколько значений и в зависимости от переданной команды выполняет с ними заданные операции.

В ходе этой работы в качестве временного хранилища данных выступает оперативная память. CPU постоянно производит с ней обмен данными.

Память — это основное средство хранения данных, используемое программами в ходе их функционирования. Память компьютера состоит из миллионов отдельных ячеек, каждая из которых может хранить информацию. Также у каждой ячейки присутствует свой уникальный адрес.

Рассмотрим принцип работы оперативной памяти. На рис. 4.2 приведен схематичный вид памяти с ячейками № 1669–1680. В настоящее время эти ячейки пусты, то есть не имеют записанной в них информации.

Рис. 4.2. Память с набором ячеек

ПРИМЕЧАНИЕ Приведенные схемы и описания максимально упрощены. На самом деле компьютер на аппаратном уровне оперирует исключительно двумя цифрами: 0 и 1. Он знает всю их мощь и по полной их использует.

Приведенные далее примеры хранения и обработки десятичных чисел показаны для лучшего восприятия вами учебного материала. Повторюсь, что в действительности эти числа переводятся в двоичную систему, после чего происходит их запись в память и обработка в процессоре в виде последовательностей 0 и 1.

Предположим, что вы запустили программу «Калькулятор». При ее использовании возникает необходимость временно хранить в памяти различные числовые значения. Каждое значение при этом, скорее всего, будет занимать не одну, а целую группу ячеек (логически объединенных). Каждая такая группа будет иметь уникальное имя, она может быть названа *хранилищем данных*. Данные об используемых в программе хранилищах записываются в специальный реестр. При этом указывается имя хранилища, а также адреса используемых ячеек.

Предположим, было создано два хранилища с именами «Группа 1» и «Группа 2». В первом содержится числовое значение 23, а во втором — 145 (рис. 4.3).

В будущем для того, чтобы получить значение 23 (хранящееся в «Группе 1»), нет необходимости сканировать все ячейки оперативной памяти на предмет записанных в них данных — достаточно лишь обратиться к «Группе 1» по имени и получить записанное в ее ячейках значение.

Рис. 4.3. Реестр с двумя хранилищами данных

Если вернуться к рис. 4.1, то вторым идет уровень **операционной системы**. В ходе разработки вы будете не так часто задумываться о нем.

В общем случае операционная система — это посредник между вашей программой и аппаратной частью. Благодаря ОС, приложения могут использовать все возможности компьютера: выполнение математических операций, передача данных по Wi-Fi и Bluetooth, отображение графики, воспроизведение песен группы Queen и многое-многое другое.

Операционная система предоставляет интерфейсы, которые могут использоваться в ходе работы программного обеспечения. Хотите сложить два числа? Не вопрос! Напишите соответствующую команду, и ОС сама передаст ваши байты куда следует, после чего вернет результат.

Для того чтобы отдавать различные команды на выполнение каких-либо операций, как раз и нужен третий уровень (см. рис. 4.1).

«**Программный**» уровень предполагает использование языков программирования для создания приложений. Все довольно просто: разработчик пишет команды, ОС обрабатывает их, после чего передает их для выполнения на аппаратный уровень. Каждая команда на языке программирования может подразумевать под собой множество циклов работы с памятью и выполнение сотен инструкций в процессоре.

В общем виде **программирование** — это автоматизация процессов для решения определенных задач с помощью компьютера. В ходе написания программы разработчик манипулирует данными и определяет реакцию программы на них.

Каждая программа взаимодействует с данными, которые могут быть получены из различных источников: загружены из файлов на жест-

ком диске, введены пользователями, получены из Сети и т. д. Часто эти данные нужно не просто получить, обработать и отправить в указанное место, но и обеспечить их хранение с целью использования в будущем.

Так, например, получив ваше имя (предположим, что вы ввели его при открытии приложения), программа запишет его в память и при необходимости будет загружать оттуда, чтобы отобразить в соответствующем месте графического интерфейса.

4.2. Базовые понятия

В предыдущем разделе, когда мы говорили об оперативной памяти, то упоминали понятие «хранилище данных».

Хранилище данных — это виртуальный объект, обладающий некоторым набором свойств, к примеру: записанное значение, количество ячеек, адрес в оперативной памяти, тип информации (числовой, строковый) и т. д.

Практически весь процесс программирования состоит в том, чтобы создавать (определять) объекты, устанавливать (инициализировать) их значения, запрашивать эти значения и производить с ними операции.

В программировании Swift при работе с хранилищами данных выделяют два важнейших понятия: объявление и инициализация.

Объявление — это создание нового объекта (хранилища данных).

Инициализация — это присвоение объявленному объекту определенного значения.

В примере выше были объявлены хранилища данных с именами «Группа 1» и «Группа 2», после чего их проинициализировали значениями `23` и `145`.

Рассмотрим простейший арифметический пример: значение «Группы 1» (`23`) умножить на значение «Группы 2» (`145`) (листинг 4.1).

Листинг 4.1.

```
23 * 145
```

Данный математический пример является выражением, то есть законченной командой на языке математики. В нем можно выделить одну операцию умножения и два операнда, с которыми будут производиться действия (`23` и `145`).

Чтобы произвести данную операцию умножения, нужно выполнить следующую последовательность действий:

- Получить значение «Группы 1».
- Получить значение «Группы 2».
- Произвести операцию умножения значения «Группы 1» на значение «Группы 2».
- Объявить хранилище данных с именем «Группа 3» для хранения результата умножения.
- Проинициализировать хранилищу «Группа 3» результат операции.

В листинге 4.1 указателем на то, что необходимо произвести операцию умножения, служит оператор * (умножение). В результате выполнения выражения будет получено новое значение `3335`, записанное в группу ячеек с именем «Группа 3» (рис. 4.4).

Рис. 4.4. Результат умножения двух чисел помещен в новое хранилище

Помимо этого, хочется обратить внимание на то, что хранилища данных могут содержать в себе не только цифры, но и другие виды информации (текстовую, графическую, логическую и др.). Виды информации в программировании называются типами данных. При объявлении хранилищ вы **всегда** будете сообщать программе, данные какого типа она должна хранить (целые числа, дробные числа, текст, рисунок и т. д.).

Любая программа — это набор выражений или, другими словами, набор четких команд, понятных компьютеру. Например, выражение «Отправь этот документ на печать» укажет компьютеру на необходимость выполнения некоторых действий в определенном порядке: загрузка документа, поиск принтера, отправка документа принтеру

и т. д. Выражения могут состоять как из одной, так и из нескольких команд, как, например: «Отправь этот файл, но перед этим преобразуй его из docx в rtf».

Выражение — завершенная команда, выполняющая одну или несколько операций. Выражение может состоять из множества операторов и операндов.

Оператор — это минимальная автономная функциональная единица (слово или группа слов), выполняющая определенную операцию.

Операнд — это значение, с которым оператор производит операцию.

Все зарезервированные языком программирования наборы символов называются **ключевыми словами**.

Ранее мы рассматривали пример умножения двух чисел `23` и `145` с последующим объявлением нового хранилища. Всего было выделено пять отдельных шагов, которые, в свою очередь, и являлись полноценным выражением.

Если рассмотреть это выражение со стороны синтаксиса языка программирования, то его можно представить в виде следующей строки (листинг 4.2).

Листинг 4.2

```
Создать_хранилище Группа 3 = Группа 1 * Группа 2
```

Обратите внимание, как легко и просто с помощью одного выражения производятся необходимые нам операции.

4.3. Введение в операторы

В Swift выделяют следующие основные виды операторов:

- **Простые операторы**, выполняющие операции с некоторыми значениями (операндами). В их состав входят унарные и бинарные операторы.
  - **Унарные операторы** выполняют операцию с одним операндом (например, `-a`). Они могут находиться перед операндом, то есть быть префиксными (например, `!b`), или после него, то есть быть постфиксными (например, `i?`).
  - **Бинарные операторы** выполняют операцию с двумя операндами (например, `1+6`). Оператор, который располагается между операндами, называется инфиксным.

- **Структурные операторы** влияют на ход выполнения программы. Например, останавливают выполнение программы при определенных условиях или указывают программе, какой блок кода должен быть выполнен при определенных условиях.

В будущем в ходе создания приложений вы будете регулярно применять различные операторы, а также при необходимости создавать собственные.

Обратимся к Swift и Xcode Playground. Перед вами уже должен быть открыт созданный ранее проект.

ПРИМЕЧАНИЕ Если вы закрыли предыдущий playground-проект, то создайте новый. Для этого откройте Xcode, в появившемся меню выберите пункт «Get started with a playground», после чего выберите Blank и укажите произвольное название нового проекта. Перед вами откроется рабочее окно Xcode playground (рис. 4.5).

Рис. 4.5. Окно нового playground

Взгляните на код в Xcode Playground (листинг 4.3).

Листинг 4.3

```
import UIKit
var str = "Hello World"
```

Предназначение первой строки мы подробно рассмотрим позже, а сейчас вкратце: она осуществляет подключение модуля, расширяющего функциональные возможности языка.

Рассмотрим следующую строчку кода (листинг 4.4).

Листинг 4.4

```
var str = "Hello World"
```

В ней объявляется хранилище данных, после чего инициализируется текстовое значение `Hello World`. Данное выражение можно разбить на отдельные шаги (рис. 4.6).

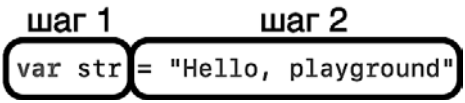

Рис. 4.6. Выражение состоит из отдельных шагов

Шаг 1: С помощью ключевого слова (оператора) `var` объявляется новое хранилище данных с именем `str`.

Шаг 2: В оперативную память записывается текстовое значение `Hello World`, после чего создается указатель в хранилище данных на этот участок памяти. Или, другими словами, в созданное хранилище `str` записывается значение `Hello World`. Этот процесс и называется инициализацией значения с помощью оператора `=` (присваивания).

Если представить текущую схему оперативной памяти, то она выглядит так, как показано на рис. 4.7.

Рис. 4.7. Результат создания нового хранилища данных

В данном выражении используется два оператора. На шаге 1 — это оператор `var`, на шаге 2 — оператор `=`. В двух следующих разделах рассмотрим подробнее их работу, но начнем с оператора инициализации.

4.4. Оператор инициализации

Напомню, что для хранения данных компьютер использует оперативную память, выделяя под каждое значение группу ячеек (хранилище данных), имеющую уникальное имя. Прежде чем говорить о том, каким образом в Swift объявляются хранилища, разберемся, каким образом данные могут быть записаны в эти хранилища. Для этого вернемся к выражению в Xcode Playground (листинг 4.5).

Листинг 4.5

```
var str = "Hello World"
```

Для выполнения операции инициализации значения используется оператор присваивания, или, иначе, — оператор инициализации, обозначаемый как знак равенства (`=`).

Оператор инициализации (присваивания) (`=`) — это особый бинарный оператор. Он используется в типовом выражении `a = b`, инициализируя значение объекта `a` значением объекта `b`. В данном примере объекту `str` инициализируется текстовое значение `"Hello World"`.

ПРИМЕЧАНИЕ В Swift оператор присваивания не возвращает присваиваемое (инициализируемое) значение, он лишь проводит инициализацию (установку значения). Во многих языках программирования данный оператор возвращает присваиваемое значение, и вы можете, например, незамедлительно вывести его на консоль или использовать в качестве условия в операторе условия. В Swift подобный подход вызовет ошибку.

Левая и правая части оператора присваивания должны быть однотипными (то есть иметь одинаковый тип, предназначаться для хранения данных одного и того же типа). В данном случае строка «`Hello World`» — это данные строкового типа. Создаваемая переменная `str` также имеет строковый тип, он определяется автоматически за счет инициализируемого значения. Об определении типа значения поговорим чуть позже.

4.5. Переменные и константы

В предыдущем листинге для объявления хранилища данных `str` используется оператор `var`.

Всего в Swift выделяют два типа хранилищ данных:

- **переменные**, объявляемые с помощью ключевого слова `var`;
- **константы**, объявляемые с помощью ключевого слова `let`.

Любое хранилище, неважно, какого типа, имеет три важнейших свойства:

1) **имя**, по которому можно проинициализировать новое или получить записанное ранее значение;
2) **тип значения**, определяющий, какие данные могут храниться в данном хранилище;
3) **значение**, которое в данный момент находится в хранилище.

ПРИМЕЧАНИЕ Хранилище **всегда** должно иметь значение. Оно должно быть проинициализировано в одном выражении с объявлением этого хранилища (в качестве примера смотрите листинг 4.5). Если вы объявите хранилище, то определите тип его значения, но не передадите само значение, Xcode сообщит об ошибке (рис. 4.8).

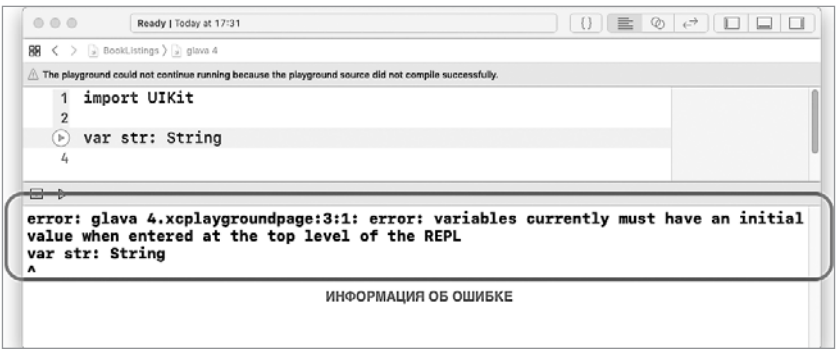

Рис. 4.8. Сообщение об ошибке при создании хранилища без значения

Ключевое слово `String` после имени хранилища указывает на то, что оно предназначено для хранения строковых данных. Позже мы подробно разберем как и зачем определяется тип данных хранилища.

Рассмотрим каждый тип хранилища подробнее.

Переменные

Одно из важнейших базовых понятий в любом языке программирования — переменная.

Переменная — это именованная область памяти (хранилище данных), в которой *должно* быть записано значение определенного типа данных. Значение переменной может быть многократно изменено раз-

работчиком в процессе работы программы. В качестве примера можно привести переменную, которая хранит текущую секунду. Ее значение должно меняться ежесекундно.

Для объявления переменной в Swift используется оператор `var`.

СИНТАКСИС

```
var имяПеременной = значение_переменной
```

После оператора `var` указывается имя объявляемой переменной. После имени указывается оператор присваивания и инициализируемое значение.

С помощью имени переменной будет производиться доступ к значению переменной для его считывания или изменения.

Рассмотрим пример объявления переменной (листинг 4.6).

ПРИМЕЧАНИЕ Вы можете продолжать писать, не убирая предыдущие выражения. В ином случае убедитесь, что выражение с директивой `import` не удалено.

Листинг 4.6

```
var age = 21
```

В данном примере создается переменная с именем `age` и значением `21`. То есть код можно прочитать следующим образом: «Объявить переменную с именем `age` и присвоить ей целочисленное значение `21`».

В ходе работы над программой можно объявлять произвольное количество переменных. Вы ограничены только своими потребностями (ну и конечно, объемом оперативной памяти).

Изменить значение переменной можно, инициализировав ей новое значение. Повторно использовать оператор `var` в этом случае не требуется (листинг 4.7).

ПРИМЕЧАНИЕ Оператор `var` используется единожды для каждой переменной только при ее объявлении.

Будьте внимательны: во время инициализации нового значения старое уничтожается.

Листинг 4.7

```
var age = 21
age = 22
```

В результате выполнения кода в переменной `age` будет храниться значение `22`. Обратите внимание, что в области результатов будет

выведено два значения: старое (21) и новое (22). Старое — напротив первой строки кода, новое — напротив второй (рис. 4.9).

Рис. 4.9. Отображение значения переменной в области результатов

ПРИМЕЧАНИЕ Уже на протяжении нескольких лет при работе в Xcode Playground разработчики встречаются с тем, что написанный код зачастую не выполняется, результаты не отображаются в области результатов Xcode, программа будто зависает. По умолчанию стоит режим автоматического выполнения кода, сразу после его написания. К сожалению, он периодически дает сбой. В этом случае вы можете перейти в режим выполнения кода по запросу. Для этого нажмите и удерживайте кнопку мыши на символе «квадрат» в нижней части Xcode, пока не появится всплывающее окно с двумя вариантами. Выберите Manually Run (рис. 4.10).

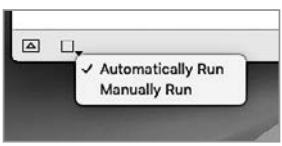

Рис. 4.10. Выбор режима выполнения кода

В результате ваш код начнет выполняться не сразу, как будет введен, а после того, как вы нажмете стрелочку в нижней части окна или слева от строки кода (появляется при наведении указателя мышки) (рис. 4.11).

К сожалению, и такой подход не гарантирует быстрого выполнения кода. В редких случаях вам потребуется либо немного подождать, либо повторно дать команду на выполнение, либо перезапустить Xcode.

Иногда даже после многократных перезапусков среды разработки код все равно не будет выполняться. Причиной этому может быть зависший симулятор. Для решения этой проблемы вы можете попробовать принудительно закрыть его: откройте «Программы — Утилиты — Мониторинг системы», после чего найдите и завершите процесс com.apple.CoreSimulator.CoreSimulatorService.

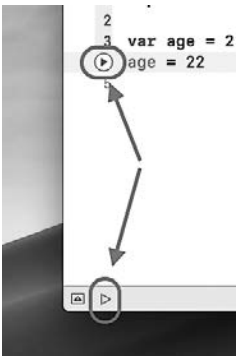

Рис. 4.11. Принудительный запуск кода на выполнение

Помимо этого, вы можете изменить платформу playground-проекта с iOS на macOS. Для этого нажмите кнопку «Hide or Show the Inspector», расположенную в провом верхнем углу, и в поле Platform выберите необходимый пункт (рис. 4.12). В этом случае вам придется заменить строку import UIKit на import Foundation, так как модуль UIKit существует только для iOS-симулятора.

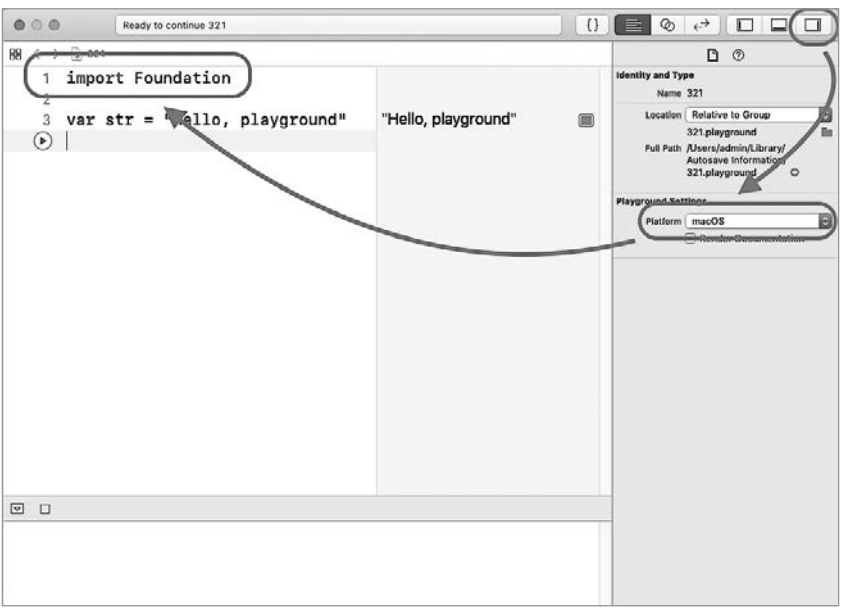

Рис. 4.12. Смена платформы Xcode Playground

Все эти трюки и ухищрения не от хорошей жизни. Для меня до сих пор остается секретом, почему такая крупная корпорация не может решить проблему, существующую уже долгие годы, ведь удобство использования Xcode Playground напрямую влияет на желание юных программистов обучаться. Но это все лирика, вернемся к разработке.

Область результатов Xcode Playground позволяет узнать текущее значение любого параметра. Для этого достаточно лишь написать его имя, после чего актуальное значение появится в области результатов. Определим текущее значение переменной, тем самым убедившись, что оно действительно сменилось. Для этого напишите в следующей строке имя переменной и посмотрите на область результатов (рис. 4.13).

Рис. 4.13. Уточнение значения переменной

Константы

В отличие от переменных, константы позволяют лишь единожды инициализировать свое значение. Все последующие попытки изменить его вызовут ошибку. Константы объявляются с помощью оператора `let`.

СИНТАКСИС

```
let имяКонстанты = значение_константы
```

После оператора `let` указывается имя создаваемой константы, с помощью которого будет производиться обращение к записанному в константе значению. Далее, после имени и оператора присваивания, следует инициализируемое значение.

Рассмотрим пример объявления константы (листинг 4.8).

Листинг 4.8

```
let name = "Vasiliy"
```

В результате выполнения кода будет объявлена константа `name`, содержащая строку «`Vasiliy`». Данное значение невозможно изменить в будущем. При попытке сделать это Xcode сообщит об ошибке (рис. 4.14).

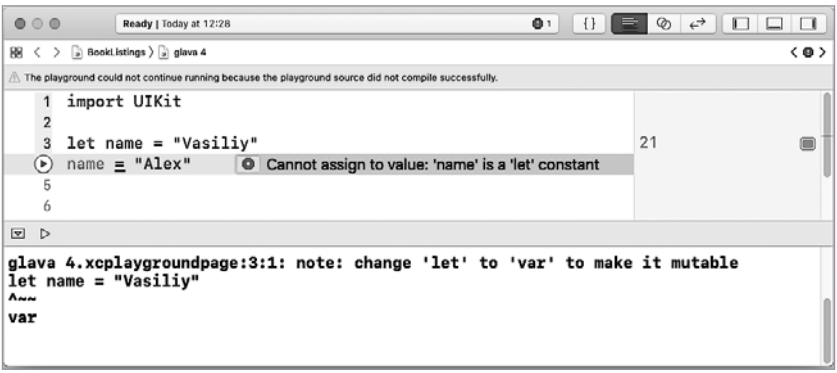

Рис. 4.14. Ошибка при изменении значения константы

ПРИМЕЧАНИЕ Xcode пытается сделать вашу работу максимально продуктивной (если, конечно, забыть о периодически/стабильно отваливающемся симуляторе в Playground). В большинстве случаев он не просто сообщает о возникшей ошибке, а дает рекомендации по ее устранению и даже самостоятельно вносит необходимые правки.

На рис. 4.14 был показан пример ошибки. Для того чтобы ознакомиться с рекомендациями, необходимо нажать на кружок слева от сообщения об ошибке. После нажатия на кнопку Fix ошибка будет исправлена. В данном примере рекомендовано изменить оператор `let` на `var` (рис. 4.5).

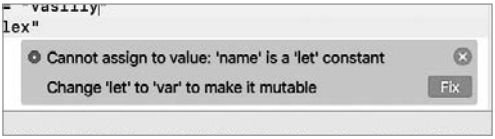

Рис. 4.15. Автоматическое исправление ошибки

Если ранее вы программировали на других языках, то, вполне вероятно, часто избегали использования констант, обходясь переменными. Swift — это язык, который позволяет очень гибко управлять ресур-

сами. Константы — базовое средство оптимизации используемых мощностей. В связи с этим вам следует прибегать к их использованию всегда, когда инициализированное значение не должно и не будет изменяться в ходе работы приложения.

Оператор `let` так же, как и `var`, необходимо использовать единожды для каждого хранилища (только при его объявлении).

При необходимости объявить несколько параметров вы можете использовать один оператор, `var` или `let`, после чего через запятую указать имена и инициализируемые значения (листинг 4.9).

Листинг 4.9

```
let friend1 = "John", friend2 = "Helga"
var age1 = 54, age2 = 25
```

Разумное использование переменных и констант привносит удобство и логичность в Swift.

4.5. Инициализация копированием

Инициализацию значения любых параметров (переменных и констант) можно проводить, указывая в правой части выражения не только конкретное значение, но и имя другого параметра (листинг 4.10).

Листинг 4.10

```
var myAge = 40
var yourAge = myAge
yourAge
```

Переменная `yourAge` имеет значение `40`, что соответствует значению переменной `myAge`.

Стоит отметить, что таким образом вы создаете копию исходного значения, то есть в результате операции будут объявлено две независимые переменные с двумя независимыми значениями. Изменение одного из них не повлияет на другое.

Такой тип параметров, когда передача значения происходит копированием, называется **value type** (значимые типы). Помимо этого, существует передача ссылки на значение (**reference type**, или ссылочные типы), когда все переменные содержат в себе ссылку на одно и то же значение, хранящееся в памяти. При этом изменение значения через любую из переменных отразится и на всех ее копиях-ссылках.

Подробнее об этом поговорим в следующих главах.

4.6. Правила именования переменных и констант

При написании программ на Swift вы можете создавать параметры с любыми именами, при этом использовать произвольные Unicode-символы и эмодзи. Существуют правила, которых необходимо придерживаться при именовании параметров:

1. Переменные и константы следует именовать в *верблюжьем регистре* (camel case). Это значит, что при наименовании используются только латинские символы (без подчеркиваний, тире, математических формул и т. д.) и каждое значимое слово (кроме первого) в имени начинается с прописной (заглавной) буквы. Например: `myBestText`, `theBestCountry`, `highScore`.

   Хотя их имена и могут содержать любые Unicode-символы (листинг 4.11), использование таких имен только мешает читабельности кода.

   Листинг 4.11

   ```
   var d0w = "Значение переменной с невоспроизводимым именем"
   ```

2. Имена должны быть уникальными. Нельзя создавать переменную или константу с именем, уже занятым другой переменной или константой.

ПРИМЕЧАНИЕ У данного правила есть исключения, связанные с областью видимости переменных и констант. Об этом мы поговорим позже.

Xcode отобразит ошибку при попытке дать параметру имя, соответствующее какому-либо ключевому слову. Если вам все же требуется сделать это, то следует написать его в апострофах (`). Я настоятельно рекомендую избегать этого, чтобы не нарушать читабельность кода.

В листинге 4.12 приведен пример создания переменной с именем, соответствующим занятому ключевому слову `var`. Для доступа к значению переменной также необходимо использовать апострофы.

Листинг 4.12

```
var `var` = "Пример переменной с именем var"
```

При росте размеров исходного кода ваших программ вам станет все сложнее и сложнее ориентироваться в них. Для того чтобы в значительной степени облегчить данный процесс, старайтесь осмысленно и максимально кратко именовать параметры, которые объявляете. При

этом нужно найти золотую середину между краткостью и ясностью, чтобы не впадать в крайности.

`var nou` — пример плохого имени переменной.

`var userName` — пример хорошего имени переменной.

`var nameOfUserOfMyBestApplicationInTheWorld` — пример плохого имени переменной.

4.7. Возможности автодополнения и кодовые сниппеты

Уверен, что вы обратили внимание на то, что при вводе кода появляется всплывающее окно, содержащее различные записи (рис. 4.16). Данная функция Xcode называется автодополнением, мы упоминали о ней ранее. С ее помощью вы можете значительно сократить время написания кода, выбирая наиболее подходящий из предложенных вариант.

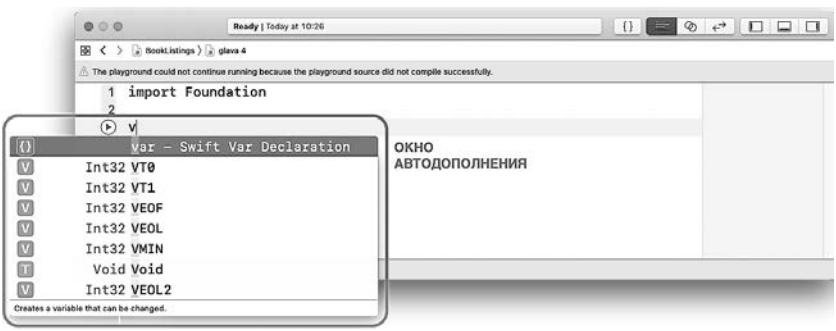

Рис. 4.16. Окно автодополнения в Xcode

Советую вам активно использовать возможности автодополнения, так как иногда оно помогает найти необходимые функциональные элементы без обращения к документации.

Среди доступных элементов окна автодополнения вы найдете имена всех параметров, операторов и ключевых слов, соответствующих введенным символам и данному контексту. Помимо этого, вы сможете найти и кодовые сниппеты, или, другими словами, шаблоны участков кода.

Так, к примеру, если в новой строчке кода ввести ключевое слово `var`, то вы увидите соответствующий сниппет (обозначенный как две фигурные скобки) (рис. 4.17).

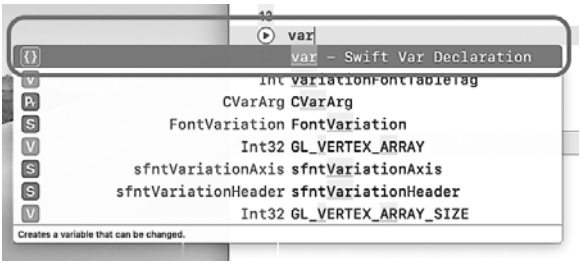

Рис. 4.17. Сниппет оператора var

Щелкнув по нему (или нажав на клавишу Enter), вы сможете с легкостью создать новую переменную, заполняя выделенные серым участки сниппета и перескакивая между ними с помощью клавиши Tab.

Помимо окна автодополнения, кодовые сниппеты можно открыть с помощью кнопки с изображением двух фигурных скобок, расположенной в верхней части Xcode (рис. 4.18). В появившемся окне отображается вся библиотека кодовых шаблонов.

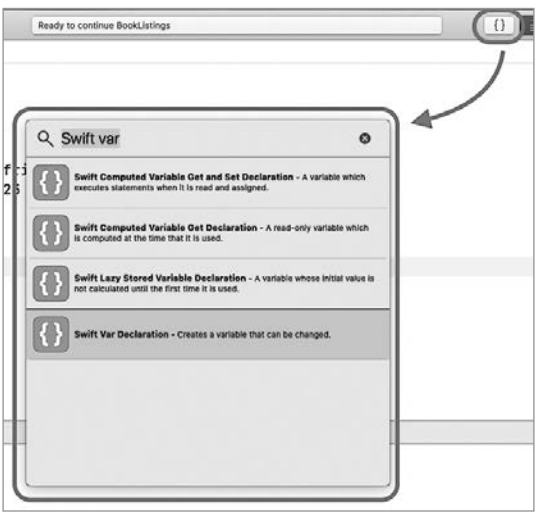

Рис. 4.18. Библиотека кодовых сниппетов

4.8. Глобальные и локальные объекты

Любая программа, написанная в Xcode, содержит большое количество взаимосвязанных объектов, причем одни из них могут входить в состав других.

ПРИМЕЧАНИЕ Удивительно, но переменные и константы могут содержать в своем составе другие переменные и константы! Это более глубокий уровень применения языка Swift, и мы познакомимся с ним в дальнейшем.

У каждого объекта есть так называемая область его применения, или, другими словами, область видимости, которая определяет, где именно данный объект может быть использован (например, область видимости объекта может быть ограничена файлом или отдельным участком кода).

Область видимости делит объекты на два типа.

- *Глобальные объекты* — это объекты, доступные в любой точке программы.
- *Локальные объекты* — это объекты, доступные в пределах родительского объекта.

Рассмотрим следующий пример: у вас есть программа, в которой существуют несколько объектов. У каждого объекта есть собственное имя, по которому он доступен (рис. 4.19).

Объекты Object-1 и Object-2 объявлены непосредственно в корне программы. Такие объекты называются глобальными, и они доступны в любой точке программного кода.

Объект Object-2 имеет вложенный объект Object-3, который является локальным: он объявлен в контексте Object-2 и доступен только в его пределах. Таким образом, попытка получить доступ к Object-3 из Object-1 завершится ошибкой. Но при этом внутри Object-3 будут доступны и Object-1, и Object-2.

Рассмотрим другой вариант организации объектов (рис. 4.20).

Каждый из корневых объектов имеет по одному вложенному. Особенность данной иерархии в том, что локальные объекты имеют точно такие же имена, как и глобальные. В таком случае ошибки не происходит и локальный объект перекрывает глобальный, то есть, обращаясь к Object-2 внутри Object-1, вы обращаетесь к локальному объекту, а не к глобальному.

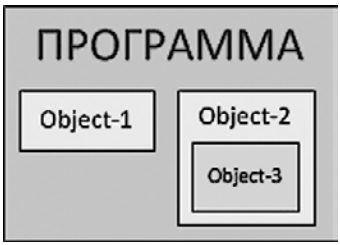

Рис. 4.19. Вариант иерархии объектов внутри программы

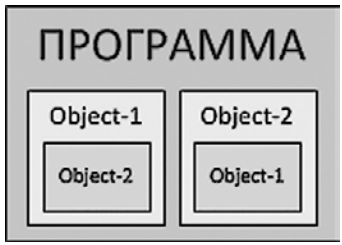

Рис. 4.20. Еще один вариант иерархии объектов

Вот такая запутанная, но очень важная для правильного понимания будущего материала история.

Запомните, что одноименные локальный и глобальный объекты — это разные объекты, которые могут иметь совершенно разные значения и характеристики. Очень важно писать прозрачный и понятный код, который не будет вызывать вопросов о том, к какому именно объекту вы обращаетесь в настоящий момент.

Время жизни локальных объектов равно времени жизни их родительских объектов.

4.9. Комментарии

Классические комментарии

Помимо правильного именования создаваемых объектов, для улучшения читабельности кода вы можете (а скорее, даже вы обязаны) использовать комментарии.

Комментарии в Swift, как и в любом другом языке программирования, представляют собой блоки неисполняемого кода, например пометки или напоминания. Проще говоря, при сборке программ из исходного кода Xcode будет игнорировать участки, помеченные как комментарии, так, будто они вовсе не существуют.

На рис. 4.21 приведен пример комментариев. По умолчанию они выделяются серым цветом.

Правильно написанный код обязательно должен содержать комментарии. Они помогают нам не запутаться в написанном. Если вы не будете их использовать, то рано или поздно попадете в ситуацию, когда навигация по собственному проекту станет невыносимо сложной.

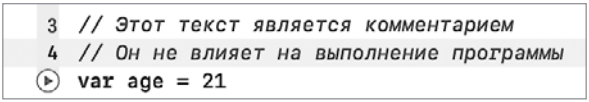

Рис. 4.21. Пример комментариев на странице с кодом

ПРИМЕЧАНИЕ Существуют различные рекомендации относительно того, как много комментариев должен содержать ваш код. Некоторые программисты говорят о том, что «на каждую строку кода должно приходиться две строки комментариев». Другие же рекомендуют не перегружать код комментариями, по максимуму используя другие возможности идентификации написанного (например, с помощью правильного именования переменных, типов данных и т. д.)

Лично я рекомендую писать много комментариев, но при это сам стараюсь максимально сократить их количество, чтобы в первую очередь заниматься созданием кода, а не его документированием. Со временем вы поймете, какой из вариантов наиболее близок и удобен вам.

Swift позволяет использовать однострочные и многострочные комментарии.

Однострочные комментарии пишутся с помощью двойного слеша (`// комментарий`) перед текстом комментария, в то время как многострочные обрамляются звездочками со слешем с обеих сторон (`/* комментарий */`). Пример комментариев приведен в листинге 4.13.

Листинг 4.13

```
// это – однострочный комментарий
/* это -
многострочный
комментарий */
```

Помните, что весь текст, находящийся в комментарии, игнорируется компилятором и совершенно не влияет на выполнение программы.

Markdown-комментарии

Xcode Playground поддерживает markdown-комментарии — особый вид комментариев, позволяющий применять форматирование. Таким образом ваш playground-проект может превратиться в настоящий обучающий материал.

Markdown-комментарии должны начинаться с двоеного слеша и двоеточия (`//:`), после которых следует текст комментария. Несколько примеров неформатированных комментариев приведены на рис. 4.22.

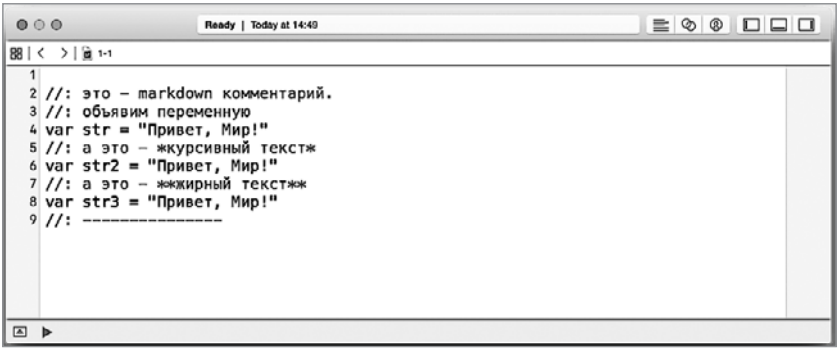

Рис. 4.22. Неформатированные markdown-комментарии

Включить форматирование комментариев, при котором все markdown-комментарии отобразятся в красивом и удобном для чтения стиле, можно, выбрав в меню Xcode пункт Editor Show Rendered Markup. Результат приведен на рис. 4.23.

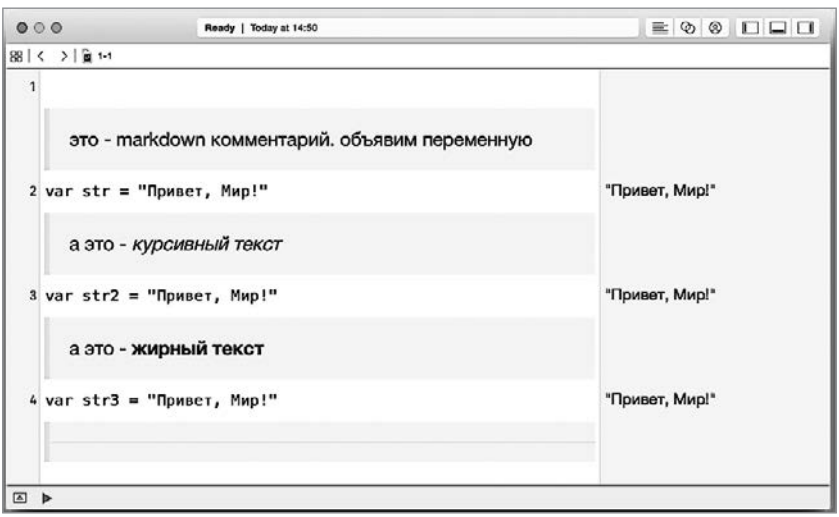

Рис. 4.23. Форматированные markdown-комментарии

Вернуть markdown-комментарии к прежнему неформатированному виду можно, выбрав в меню пункт Editor ▸ Show Raw Markup.

Не стоит думать о форматированных комментариях во время создания программ. Markdown — это лишь дополнительная возможность, о которой стоило упомянуть. Используйте классические комментарии, когда есть необходимость сделать пометку или описание.

4.10. Точка с запятой

Некоторые популярные языки программирования требуют завершать каждую строчку кода символом «точка с запятой» (`;`). Swift в этом отношении пошел по другому пути. Обратите внимание, что ни в одном из предыдущих листингов данный символ не используется. Это связано с тем, что для этого языка нет такой необходимости. Строчка считается завершенной, когда в ее конце присутствует (невидимый) символ переноса, то есть когда вы нажали Enter.

Единственным исключением является ситуация, когда в одной строчке необходимо написать сразу несколько выражений (листинг 4.14).

Листинг 4.14

```
// одно выражение в строке — точка с запятой не нужна
var number = 18
// несколько выражений в строке — точка с запятой нужна
number = 55; var userName = "Alex"
```

Синтаксис Swift очень дружелюбен. Этот замечательный язык программирования не требует писать лишние символы, давая при этом широкие возможности для того, чтобы код был понятным и прозрачным. В дальнейшем вы увидите много интересного — того, чего не встречали в других языках.

4.11. Отладочная консоль и функция print(_:)

Консоль

Консоль — это интерфейс командной строки, который позволяет отображать текстовые данные. Если у вас есть опыт работы с компьютером (в любой из существующих ОС), то вы наверняка неоднократно сталкивались с консольными приложениями (рис. 4.24).

С помощью консоли разработчик приложения может взаимодействовать с пользователем, это своего рода простейший графический интерфейс. Xcode позволяет создавать приложения, имеющие интерфейс консоли, то есть взаимодействие с вами происходит через командную строку.

Рис. 4.24. Пример консоли в различных операционных системах

Одной из составных частей Xcode также является консоль, называемая отладочной консолью. Ее основной функцией является вывод отладочной текстовой информации (к примеру, сообщений об ошибке), а также текстовых данных, указанных вами в процессе разработки программы. Вы уже могли видеть отладочную консоль, когда пытались изменить значение константы.

Вернитесь в Xcode Playground. Обратите внимание на три кнопки, расположенные в ряд в верхнем правом углу окна Xcode (рис. 4.25).

Рис. 4.25. Функциональные кнопки в Xcode Playground

Каждая из кнопок позволяет отобразить дополнительные панели, расширяющие возможности среды разработки. Активные кнопки выделены синим цветом, неактивные — серым. После нажатия центральной кнопки отобразится отладочная консоль Xcode Playground (рис. 4.26).

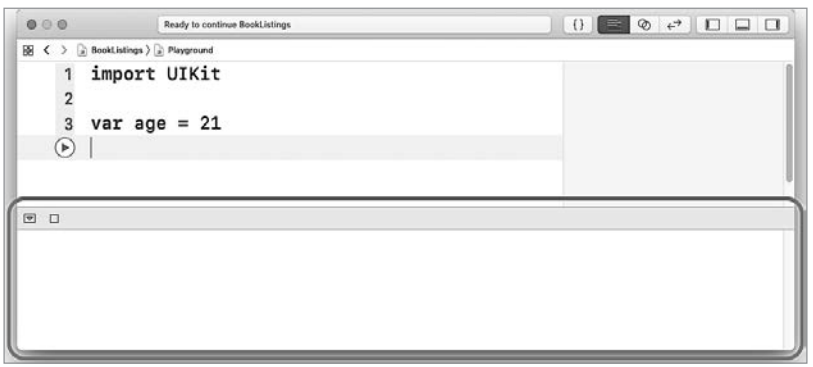

Рис. 4.26. Отладочная консоль Xcode Playground

Отладочная консоль позволяет выводить необходимую разработчику или пользователю информацию и используется в основном в двух случаях: для взаимодействия с пользователем приложения или для поиска ошибок на этапе отладки.

ПРИМЕЧАНИЕ Отладка — это этап разработки программы, на котором обнаруживаются и устраняются ошибки.

Вывод текстовой информации

Как вы могли видеть ранее, в Xcode при работе с Playground значения переменных и констант отображаются в области результатов. Напомню, что для этого необходимо написать любое выражение либо просто имя любого параметра.

В редакторе кода необходимо всего лишь написать имя объявленной и проинициализированной ранее переменной или константы (листинг 4.15).

Листинг 4.15

```
let name = "Dolf" // в области результатов отобразится "Dolf"
var size = 5 // в области результатов отобразится 5
name // в области результатов отобразится "Dolf"
size+3 // в области результатов отобразится 8
```

Данный механизм контроля значений очень полезен, тем не менее не всегда применим (в частности, в реальных Xcode-проектах при создании приложений никакой области результатов нет, интерфейс среды разработки приложений отличается от интерфейса Xcode Playground).

Существует иной способ вывода информации, в том числе значений переменных. Будь то Playground или полноценное приложение для iOS или macOS, произвольные данные можно отобразить на отладочной консоли.

Вывод на консоль осуществляется с помощью глобальной функции `print(_:)`.

СИНТАКСИС

```
print(сообщение)
```

- `сообщение` — текстовое сообщение, выводимое на консоль.

Функция выводит на отладочную консоль текстовое сообщение.

Пример

```
print("Текстовое сообщение")
```

Консоль
Текстовое сообщение

Рис. 4.27. Пример вывода информации на отладочную консоль

В общем случае **функция** — это именованный фрагмент программного кода, к которому можно многократно обращаться. Процесс обращения к функции называется вызовом функции. Для вызова функции необходимо написать ее имя и конструкцию из скобок, например `myFunction()` (такой функции не существует, и попытка вызвать ее завершится ошибкой, это лишь пример).

Функции позволяют избежать дублирования кода: они группируют часто используемый код с целью многократного обращения к нему посредством уникального имени.

Swift имеет большое количество стандартных функций, благодаря которым можно в значительной мере упростить и ускорить процесс разработки. В будущем вы научитесь самостоятельно создавать функции в зависимости от своих потребностей.

Некоторые функции построены таким образом, что при их вызове можно (или даже нужно) передать входные данные, которые будут использованы функцией внутри нее (как у функции `print(_:)`). Такие данные называются **входными параметрами, или аргументами функции**. Они указываются в скобках после имени вызываемой функции. Пример вызовов функций с передачей входных параметров приведен в листинге 4.16.

Листинг 4.16

```
// входной параметр с именем
anotherFunction(name: "Bazil")
// безымянный входной параметр
print("Это тоже входной параметр")
```

В данном коде происходит вызов функции с именем `anotherFunction` (такой функции не существует, это лишь пример), которая принимает входной аргумент `name` с текстовым значением «`Bazil`».

ПРИМЕЧАНИЕ Если вы напишете данный код в своем playground, то Xcode сообщит вам об ошибке, так как такой функции еще не существует, она пока не объявлена (рис. 4.28).

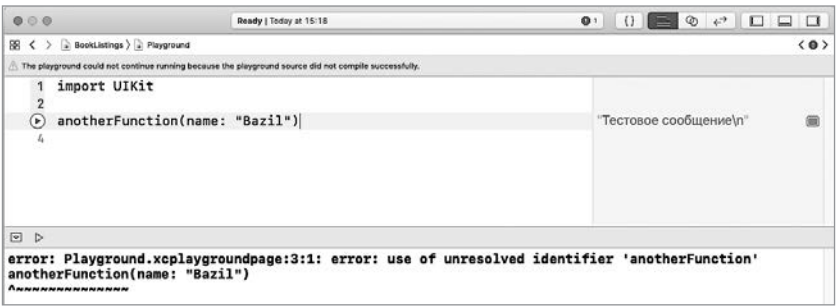

Рис. 4.28. Ошибка при вызове несуществующей функции

При появлении сообщения об ошибке в верхней части рабочего окна в строке статуса появляется красный кружок, сигнализирующий о том, что произошла исключительная ситуация (ошибка). При нажатии на него отобразится панель навигации (в левой части), содержащая информацию обо всех ошибках проекта (рис. 4.29).

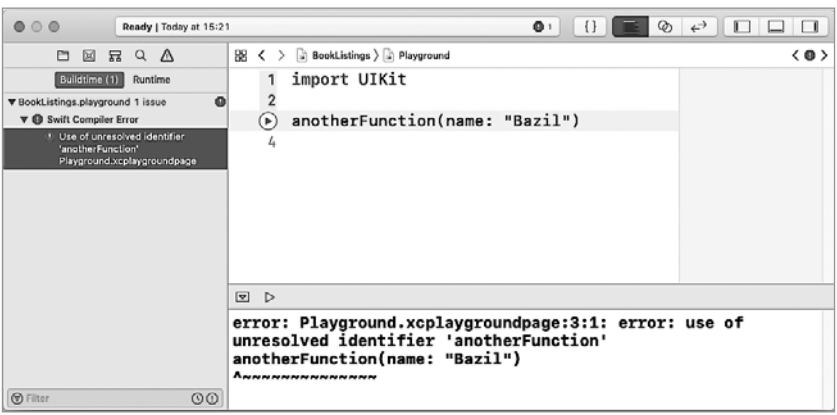

Рис. 4.29. Панель с информацией об ошибках

Вы можете закрыть ее с помощью левой кнопки.

ПРИМЕЧАНИЕ При описании функций в некоторых случаях указывается на необходимость передачи входных параметров путем указания имен аргументов внутри скобок, после каждого из которых пишется двоеточие (например, `anotherFunction(name:)` или `someFunction(a:b:)`).

Если входной параметр не имеет имени, то вместо его имени ставится нижнее подчеркивание (примером может служить упомянутая выше функция `print(_:)`).

Таким образом, указание `goodFunction(_:text:)` говорит о том, что вы можете использовать функцию с именем `goodFunction`, которой необходимо передать два входных параметра: первый не имеет имени, а второй должен быть передан с именем `text`.

Пример вызова функции `goodFunction (_:text:)` приведен ниже.

```
goodFunction(21, text:"Hello!")
```

Вернемся к рассмотрению функции `print(_:)` (листинг 4.17).

Листинг 4.17

```
// вывод информации на отладочную консоль
print("Привет, ученик!")
```

Консоль:

```
Привет, ученик!
```

Приведенный код выводит на отладочную консоль текст, переданный в функцию `print(_:)` (рис. 4.30).

Обратите внимание, что выводимый на консоль текст дублируется и в области вывода результатов, но при этом в конце добавляется символ переноса строки (`\n`).

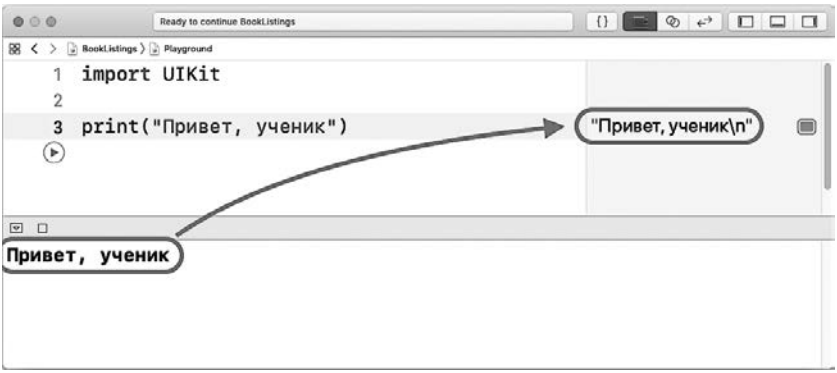

Рис. 4.30. Вывод текста на отладочную консоль

Функция `print(_:)` может принимать на вход не только текст, но и произвольный параметр (переменную или константу), как показано в листинге 4.18.

Листинг 4.18

```
var foo = "Текст для консоли"
print(foo)
```

Консоль:

```
Текст для консоли
```

Созданная переменная `foo` передается в функцию `print(_:)` в качестве входного аргумента (входного параметра), ее значение выводится на консоль.

Помимо этого, существует возможность объединить вывод текстовой информации со значением некоторого параметра (или параметров). Для этого в требуемом месте в тексте необходимо использовать символ обратной косой черты (слеш), после которого в круглых скобках нужно указать имя выводимого параметра (листинг 4.19).

Листинг 4.19

```
var bar = "Swift"
print("Я изучаю \(bar)")
```

Консоль:

```
Я изучаю Swift
```

Вы будете использовать функцию `print(_:)` довольно часто, особенно в ходе обучения разработке на Swift. Это связано с тем, что она предоставляет отличный способ контроля текущего значения параметров, а также один из самых простых способов поиска ошибок в алгоритме работы программы.

5 Фундаментальные типы данных

Переменные и константы — это хранилища данных в памяти компьютера, в которых находятся конкретные значения. Каждая переменная или константа может содержать в себе значение определенного типа: целое число, дробное число, строку, отдельный символ, логическое значение или какой-либо иной объект. Со значениями каждого типа можно совершать только определенные операции, соответствующие типу данных. К примеру, числа можно складывать или вычитать, строки — объединять между собой и т. д.

Тип данных определяет, какая именно информация может храниться в параметре, а также какие операции можно производить. Если вам необходимо хранить данные текстового типа, то и параметр должен иметь текстовый тип данных, для целочисленных значений используют параметры с целочисленным типом данных и т. п.

В Swift, по аналогии с Objective–C, а также с другими языками программирования, есть ряд предустановленных базовых (как их называет Apple, фундаментальных) типов данных.

5.1. Зачем нужны типы данных

При объявлении переменной или константы обязательно должен быть указан тип данных, которые будут храниться в этом параметре. У вас может возникнуть вопрос, каким образом был определен тип данных в предыдущих листингах книги. Переменные и константы создавались, но каких-либо специальных конструкций для этого не использовалось. Swift — умный, если можно так выразиться, язык программирования. В большинстве случаев он определяет тип объявляемого параметра автоматически на основании инициализированного ему значения.

Для всех рассматриваемых в этой главе типов данных (фундаментальных типов) инициализируемые значения называются литералами: строковый литерал, целочисленный литерал и т. д. Теперь поговорим об этом подробнее.

Рассмотрим следующий пример из листинга 5.1.

Листинг 5.1

```
// объявим переменную и присвоим ей строковое значение
var someText = "Я учу Свифт"
```

Данный пример может быть прочитан следующим образом:

Объявить переменную с именем `someText` и присвоить ей значение `«Я учу Свифт»`.

При этом не требуется дополнительно указывать, какие именно данные будут храниться в переменной, то есть не требуется указывать ее тип данных. Это связано с тем, что Swift самостоятельно анализирует инициализируемое значение, после чего принимает решение о типе параметра.

Основываясь на этом, прочитаем код предыдущего листинга следующим образом:

- Объявить переменную строкового типа с именем `someText` и присвоить ей строковое значение `"Я учу Свифт"`.

Или по-другому:

- Объявить переменную типа String с именем `someText` и присвоить ей строковое значение `"Я учу Свифт"`.

`String` — это ключевое слово, используемое в языке Swift для указания на строковый тип данных. Он позволяет хранить в переменных и константах произвольный текст. Подробные приемы работы с данным типом будут рассмотрены далее.

Операция, в которой Swift самостоятельно определяет тип объявляемого параметра, называется **неявным определением типа**.

В противовес неявному определению существует **явное**, когда разработчик сам указывает тип данных объявляемого параметра. При явном (непосредственном) определении типа переменной или константы после ее имени через двоеточие с помощью ключевого слова следует указать требуемый тип данных.

Рассмотрим еще несколько примеров объявления переменных со строковым типом данных (листинг 5.2).

Листинг 5.2

```
// создаем переменную orksName с неявным определением типа String
var orksName = "Рухард"
// создаем переменную elfsName с явным определением типа String
var elfsName: String = "Эанор"
// изменим значения переменных
orksName = "Гомри"
elfsName = "Лиасель"
```

В данном примере все переменные имеют строковый тип данных `String`, то есть могут хранить в себе строковые значения (текст).

У любой объявленной переменной или константы должен быть определен тип и указано первоначальное значение, иначе будет показано сообщение об ошибке (листинг 5.3).

Листинг 5.3

```
var firstHobbitsName // ОШИБКА: Type annotation missing in pattern
(пропущено указание типа данных)
var secondHobbitsName: String //ОШИБКА: Variables currently must have
an initial value (параметр должен иметь инициализируемое значение)
```

ПРИМЕЧАНИЕ Swift — язык со строгой типизацией. Однажды определив тип данных переменной или константы, вы уже не сможете его изменить. В каждый момент времени вы должны иметь четкое представление о типах значений, с которыми работает ваш код.

Все фундаментальные типы данных (строковые, числовые, логические и т. д.) называются *фундаментальными типами*. Они являются значимыми (value type), то есть их значения передаются копированием. Мы уже говорили об этом ранее. Напомню, что помимо значимых типов также существуют ссылочные типы данных (reference type).

При передаче значения переменной или константы значимого типа в другую переменную или константу происходит копирование этого значения, а при передаче значения ссылочного типа передается ссылка на область в памяти, в которой хранится это значение. В первом случае мы получаем два независимых параметра, а во втором — две ссылки на одно и то же значение в памяти.

Рассмотрим еще один пример работы со значимым типом данных (листинг 5.4).

Листинг 5.4

```
// неявно определим целочисленный тип данных
var variableOne = 23
```

```
// явно определим целочисленный типа данных и произведем копирование
// значения
var variableOneCopy: Int = variableOne
print(variableOneCopy)
// изменим значение в первой переменной
variableOne = 25
print(variableOneCopy)
print(variableOne)
```

Консоль:

```
23
23
25
```

ПРИМЕЧАНИЕ `Int` — это числовой тип данных, немного позже мы подробно его рассмотрим.

В данном примере хранилище `variableOne` — значимого типа. При передаче значения, хранящегося в `variableOne`, в новую переменную `variableOneCopy` произойдет создание полной независимой копии. Никакие изменения, вносимые в `variableOne`, не повлияют на значение, хранящееся в `variableOneCopy`.

Со ссылочным типом данных все иначе. Скопировав значение такого типа в другую переменную, вы получите ссылку на исходный объект, и если будете проводить изменения в ней, то изменения отразятся и в исходном объекте. Примеры ссылочных типов мы рассмотрим в следующих главах.

5.2. Числовые типы данных

Работа с числами является неотъемлемой частью практически любой программы, и для этого в Swift есть несколько фундаментальных типов данных. Некоторые из них позволяют хранить целые числа, а некоторые — дробные.

Целочисленные типы данных

Целые числа — это числа, у которых отсутствует дробная часть, например 81 или 18. Целочисленные типы данных бывают знаковыми (ноль, положительные и отрицательные значения) и беззнаковыми (только ноль и положительные значения). Swift поддерживает как знаковые, так и беззнаковые целочисленные типы данных. Для указания значения числовых типов используются числовые литералы.

Числовой литерал — это фиксированная последовательность цифр, начинающаяся либо с цифры, либо с префиксного оператора «минус» или «плюс». Так, например, «2», «–64», «+18» — все это числовые литералы.

Для объявления переменной или константы целочисленного типа необходимо использовать ключевые слова `Int` (для знаковых) и `UInt` (для беззнаковых).

Рассмотрим пример в листинге 5.5.

Листинг 5.5

```
// объявим переменную знакового целочисленного типа и присвоим
// ей значение
var signedNum: Int = -32
// объявим константу беззнакового целочисленного типа
// и проинициализируем ее значением
let unsignedNum: UInt = 128
```

В результате выполнения кода вы получите переменную `signedNum` целочисленного знакового типа `Int` со значением –32, а также константу `unsignedNum` целочисленного беззнакового типа `UInt`.

Разница между знаковыми и беззнаковыми типами заключается в том, что значение знакового типа данных может быть в интервале от –N до +N, а беззнакового — от 0 до +2N, где N определяет разрядность выбранного типа данных.

В Swift существуют дополнительные целочисленные типы данных: `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64` и `UInt64`. Они определяют диапазон возможных хранимых в переменных значениях: 8-, 16-, 32- и 64-битные числа. В табл. 5.1 приведены минимальные и максимальные значения каждого из перечисленных типов.

Таблица 5.1

Тип данных	**Int**	**Int8**	**Int16**	**Int32**
Минимальное значение	–9223372036854775808	–128	–32768	–2147483648
Максимальное значение	9223372036854775807	127	32767	2147483647
Тип данных	**Int64**	**UInt**		
Минимальное значение	–9223372036854775808	0		
Максимальное значение	9223372036854775807	9223372036854775807		

Тип данных	UInt8	UInt16	UInt32	UInt64
Минимальное значение	0	0	0	0
Максимальное значение	255	65535	4294967295	9223372036854775807

ПРИМЕЧАНИЕ Все приведенные выше целочисленные типы данных — это разные типы данных, и значения в этих типах не могут взаимодействовать между собой напрямую.

Все операции в Swift должны происходить между значениями одного и того же типа данных! Это очень важно!

Код, приведенный ниже, вызовет ошибку, так как производится попытка инициализации значения отличающегося типа.

```
var someNum: Int = 12
var anotherNum: Int8 = 14
someNum = anotherNum // ОШИБКА: Cannot assign value of type 'Int8' to
                     // type 'Int' (Нельзя передать значение типа Int8
                     // в Int)
```

Объектные возможности Swift

Swift обладает одной особенностью: *всё в этом языке программирования является объектами*.

Объект — это некоторая сущность, реализованная с помощью программного кода. Например, объектом является цифра «2» или «продуктовый автомат», если, конечно, он описан языком программирования. Каждая сущность может обладать набором характеристик (называемых *свойствами*) и запрограммированных действий (называемых *методами*). Каждое свойство и каждый метод имеют имя, позволяющее использовать их. Так, например, у объекта «продуктовый автомат» могло бы существовать свойство «вместимость» и метод «выдать товар», а у целого числа «2» — свойство «максимально возможное хранимое число» и метод «преобразовать в строку».

Доступ к свойствам и методам в Swift осуществляется с помощью их имен, написанных через точку после имени объекта, к примеру:

```
ПродуктовыйАвтомат.вместимость = 490
ПродуктовыйАвтомат.выдатьТовар()
```

ПРИМЕЧАНИЕ Ранее мы встречались с понятием функции. Метод — это та же функция, но описанная и используемая только в контексте определенного объекта.

Так как «всё — это объект», то и любая переменная числового типа данных также является объектом. Каждый из рассмотренных ранее числовых типов описывает сущность «целое число». Разница между ними состоит в том, какие максимальное и минимальное числа могут хранить переменные этих типов данных. Вы можете получить доступ к этим характеристикам через свойства `min` и `max`. Для примера получим максимально и минимально возможные значения для типов `Int8` и `UInt8`. (листинг 5.6)

Листинг 5.6

```
// минимальное значение параметра типа Int8
var minInt8 = Int8.min // -128
// максимальное значение параметра типа Int8
var maxInt8 = Int8.max // 127
// минимальное значение параметра типа UInt8
var minUInt8 = UInt8.min // 0
// максимальное значение параметра типа UInt8
var maxUInt8 = UInt8.max // 255
```

Так как тип данных `UInt8` является беззнаковым и не предназначен для хранения отрицательных чисел, то и максимально возможное значение будет в два раза больше, чем у знакового «аналога» `Int8`.

Рассматриваемые приемы относятся к объектно-ориентированному программированию (ООП), с которым вы, возможно, встречались в других языках. Подробнее объекты и их возможности будут рассмотрены далее.

ПРИМЕЧАНИЕ Запомните, что среди целочисленных типов данных Apple рекомендует использовать только `Int` и `UInt`, но для тренировки мы поработаем и с остальными.

Даже такой простой тип данных, как целые числа, в Swift наделен широкими возможностями. В дальнейшем вы узнаете о других механизмах, позволяющих обрабатывать различные числовые значения.

Числа с плавающей точкой

Помимо целых чисел, при разработке приложений вы можете использовать числа с плавающей точкой, то есть числа, у которых присутствует дробная часть. Примерами могут служить числа 3.14 и —192.884022.

В данном случае разнообразие типов данных, способных хранить дробные числа, не такое большое, как при работе с целочисленными типами. Для объявления параметров, которые могут хранить числа

с плавающей точкой, используются два ключевых слова: `Float` и `Double`. Оба типа данных являются знаковыми, то есть могут хранить положительные и отрицательные значения.

- `Float` — это 32-битное число с плавающей точкой, содержащее до 6 чисел в дробной части.
- `Double` — это 64-битное число с плавающей точкой, содержащее до 15 чисел в дробной части.

Разница этих типов данных состоит в точности: поскольку `Double` позволяет хранить больше знаков в дробной части, то и используется он лишь при необходимости выполнения расчетов высокой точности.

Пример объявления параметров приведен в листинге 5.7.

Листинг 5.7

```
// дробное число типа Float с явным указанием типа
var numFloat: Float = 104.3
// дробное число типа Double с явным указанием типа
let numDouble: Double = 8.36
// дробное число типа Double с неявным указанием типа
var someNumber = 8.36
```

Обратите внимание, что тип константы `someNumber` задается неявно (с помощью переданного дробного числового значения). В таком случае Swift **всегда** самостоятельно устанавливает тип данных `Double`.

ПРИМЕЧАНИЕ Значения типа дробных чисел не могут начинаться с десятичной точки. Я обращаю на это внимание, потому что вы могли видеть подобный подход в других языках программирования. В связи с этим следующая конструкция вызовет ошибку:

```
var variableErr1 = .12 // ОШИБКА: .12' is not a valid floating point
literal; it must be written '0.12' (0.12 не является корректным числом
с плавающей точкой)
```

Арифметические операторы

Ранее мы познакомились с типами данных, позволяющими хранить числовые значения в переменных и константах. С числами, которые мы храним, можно проводить различные арифметические операции. Swift поддерживает то же множество операций, что и другие языки программирования. Каждая арифметическая операция выполняется с помощью специального оператора. Вот список доступных в Swift операций и операторов:

+ Бинарный оператор сложения складывает первый и второй операнды и возвращает результат операции (`a + b`). Тип результирующего значения соответствует типу операндов.

```
var res = 1 + 2 // 3
```

+ Унарный оператор «плюс» используется в качестве префикса, то есть ставится перед операндом (`+a`). Возвращает значение операнда без каких-либо изменений. На практике данный оператор обычно не используется.

```
var res = +3
```

- Бинарный оператор вычитания вычитает второй операнд из первого и возвращает результат операции (`a – b`). Тип результирующего значения соответствует типу операндов.

```
var res = 4 – 3 // 1
```

- Унарный оператор «минус» используется в качестве префикса, то есть ставится перед операндом (`-a`). Инвертирует операнд и возвращает его новое значение.

```
var res = -5
```

ПРИМЕЧАНИЕ Символы «минус» и «плюс» используются для определения двух операторов (каждый из них). Данная практика должна быть знакома вам еще со времен уроков математики, когда с помощью минуса вы имели возможность определить отрицательное число и выполнить операцию вычитания.

* Бинарный оператор умножения перемножает первый и второй операнды и возвращает результат операции (`a * b`). Тип результирующего значения соответствует типу операндов.

```
var res = 4 * 5 // 20
```

/ Бинарный оператор деления делит первое число на второе и возвращает результат операции (`a / b`). Тип результирующего значения соответствует типу операндов.

```
var res = 20 / 4 // 5
```

% Бинарный оператор вычисления остатка от деления двух целочисленных значений. Тип результирующего значения соответствует типу операндов.

```
var res = 19 % 4 // 3
```

ПРИМЕЧАНИЕ В третьей версии Swift был изменен подход к использованию некоторых операторов. К примеру, довольно часто используемые в программировании операторы инкремента (++) и декремента (–), увеличивающие и уменьшающие значение на единицу соответственно, были удалены из языка.

Перечисленные операторы можно использовать для выполнения математических операций с любыми числовыми типами данных (целочисленными или с плавающей точкой). Важно помнить, что типы значений, участвующих в операции, должны быть одинаковыми.

Чтобы продемонстрировать использование данных операторов, создадим две целочисленные переменные (листинг 5.8).

Листинг 5.8

```
// целочисленные переменные
var numOne = 19
var numTwo = 4
// переменные типа числа с плавающей точкой
var numThree = 3.13
var numFour = 1.1
```

Для первых двух переменных неявно задан целочисленный тип данных `Int`, для вторых двух неявно задан тип `Double`. Рассмотренные ранее операторы позволят выполнить арифметические операции с объявленными переменными (листинг 5.9).

Листинг 5.9

```
// операция сложения
var sum = numOne + numTwo // 23
// операция вычитания
var diff = numOne - numTwo // 15
// операция умножения
var mult = numOne * numTwo // 76
// операция деления
var qo = numOne / numTwo // 4
```

Каждый из операторов производит назначенную ему операцию над переданными ему операндами.

Вероятно, у вас возник вопрос относительно результата операции деления. Подумайте: каким образом при делении переменной `firstNum`, равной `19`, на переменную `secondNum`, равную `4`, могло получиться `4`? Ведь при умножении значения `4` на `secondNum` получается вовсе не `19`. По логике, результат деления должен был получиться равным `4,75`!

Ответ кроется в типе данных. Обе переменные имеют целочисленный тип данных `Int`, а значит, результат любой операции также будет иметь

тип данных `Int`. При этом у результата деления просто отбрасывается дробная часть и никакого округления не происходит.

ПРИМЕЧАНИЕ Повторю, что операции можно проводить только между переменными или константами одного и того же типа данных. При попытке выполнить операцию между разными типами данных Xcode сообщит об ошибке.

Рассмотренные операции будут работать в том числе и с дробными числами (листинг 5.10).

Листинг 5.10

```
// операция сложения
var sumD = numThree + numFour  // 4,23
// операция вычитания
var diffD = numThree – numFour // 2,03
// операция умножения
var multD = numThree * numFour //  3,443
// операция деления
var qoD = numThree / numFour // 2,84545454545455
```

Так как типом данных исходных значений является `Double`, то и результату каждой операции неявно будет определен тот же тип данных.

Выполнение арифметических операций в Swift ничем не отличается от выполнения таких же операций в других языках программирования.

Также в Swift существует функция вычисления остатка от деления, при которой первый операнд делится на второй и возвращается остаток от этого деления. Или, другими словами, программа определяет, как много значений второго операнда поместится в первом, и возвращает значение, которое осталось, — оно называется остатком от деления. Рассмотрим пример в листинге 5.11.

Листинг 5.11

```
// целочисленные переменные
var numOne = 19
var numTwo = 4
// операция получения остатка от деления
var res1 = numOne % numTwo // 3
var res2 = -numOne % numTwo // -3
var res3 = numOne %  -numTwo // 3
```

Распишем подробнее вычисление значения переменной `res1` (остаток от деления `numOne` на `numTwo`).

```
numOne – (numTwo * 4) = 3
// Подставим значения вместо имен переменных
// и произведем расчеты
19 – (4 * 4) = 3
19 – 16 = 3
3 = 3
```

Другими словами, в `numOne` можно поместить 4 значения `numTwo`, а 3 будет результатом операции, так как данное число меньше `numTwo`.

Таким образом, остаток от деления всегда меньше делителя.

В листинге 5.11 при вычислении значений переменных `res2` и `res3` используется оператор унарного минуса. Обратите внимание, что знак результата операции равен знаку делимого, то есть когда делимое меньше нуля, результат также будет меньше нуля.

При использовании оператора вычисления остатка от деления (`%`) есть одно ограничение: он используется только для целочисленных значений.

Для вычисления остатка от деления дробных чисел используется метод `truncatingRemainder(dividingBy:)`, который применяется к делимому (то есть пишется через точку после числа, которое требуется разделить). Имя параметра `dividingBy` внутри скобок указывает на то, что данный метод имеет входной аргумент, то есть значение, которое будет передано ему при его вызове и использовано в ходе работы метода. Пример использования метода приведен в листинге 5.12.

Листинг 5.12

```
// дробные переменные
var numberOne: Float = 3.14
var numberTwo: Float = 1.01
// операция получения остатка от деления
var result1 = numberOne.truncatingRemainder(dividingBy: numberTwo) //
0.1100001
var result2 = -numberOne.truncatingRemainder(dividingBy: numberTwo) //
-0.1100001
var result3 = numberOne.truncatingRemainder(dividingBy: -numberTwo) //
0.1100001
```

Мы применили метод `truncatingRemainder(dividingBy:)` к переменной `numberOne` типа `Float`, а в качестве значения аргумента `dividingBy` передали переменную `numberTwo`.

Операция вычисления остатка от деления очень удобна в тех случаях, когда необходимо проверить, является ли число четным или кратным какому-либо другому числу.

Как определить четность? Очень просто: делите число на 2, и если остаток равен 0, то оно четное. Но для этой задачи можно использовать специальный метод `isMultiple(of:)`, применяемый к анализируемому числу.

Приведение числовых типов данных

Как неоднократно говорилось, при проведении операций (в том числе арифметических) в Swift вы должны следить за тем, чтобы операнды были одного и того же типа. Тем не менее бывают ситуации, когда необходимо провести операцию со значениями, которые имеют разный тип данных. При попытке непосредственного перемножения, например, `Int` и `Double`, Xcode сообщит об ошибке и остановит выполнение программы.

Такая ситуация не осталась вне поля зрения разработчиков Swift, поэтому в языке присутствует специальный механизм, позволяющий преобразовывать одни типы данных в другие. Данный механизм называется **приведением** (от слова «привести»), выполнен он в виде множества глобальных функций.

ПРИМЕЧАНИЕ Справедливости ради стоит отметить, что на самом деле приведенные далее глобальные функции являются специальными методами-инициализаторами типов данных. Ранее мы говорили, что любые значения — это объекты и у них существуют запрограммированные действия, называемые методами. У каждого типа данных есть специальный метод, называемый инициализатором. Он автоматически вызывается при создании нового объекта, а так как в результате вызова объекта «числовой тип данных» создается новый объект — «число», то и метод-инициализатор срабатывает.

Инициализатор имеет собственное фиксированное обозначение — `init()`, и для создания нового объекта определенного типа данных он может быть вызван следующим образом:

```
ИмяТипаДанных.init(_:)
```

например:

```
var numObj = Int.init(2) // 2
```

В результате будет создана переменная numObj целочисленного знакового типа Int со значением 2.

С помощью вызова метода init(_:) создается новый объект, описывающий некую сущность, соответствующую типу данных (число, строка и т. д.). Swift упрощает разработку, позволяя не писать имя метода-инициализатора:

```
ИмяТипаДанных(_:)
```

например:

```
var numObj = Int(2) // 2
```

В результате выполнения данного кода также будет объявлена переменная типа Int со значением 2.

В будущем мы очень подробно разберем, что такое инициализаторы и для чего они нужны.

Имена вызываемых функций в Swift, с помощью которых можно преобразовать типы данных, соответствуют именам типов данных:

- Int(_:) — преобразовывает переданное значение к типу данных Int.
- Double(_:) — преобразовывает переданное значение к типу данных Double.
- Float(_:) — преобразовывает переданное значение к типу данных Float.

ПРИМЕЧАНИЕ Если вы используете типы данных вроде UInt, Int8 и т. д. в своей программе, то для преобразования чисел в эти типы данных также используйте функции, совпадающие по названиям с типами.

Для применения данных функций в скобках после названия требуется передать преобразуемый элемент (переменную, константу, число). Рассмотрим пример, в котором требуется перемножить два числа: целое типа Int и дробное типа Double (листинг 5.13).

Листинг 5.13

```
// переменная типа Int
var numberInt = 19
//переменная типа Double
var numberDouble = 3.13
// операция перемножения чисел
var resD = Double(numberInt) * numberDouble // 59,47
var resI = numberInt * Int(numberDouble)    // 57
```

Существует два подхода к перемножению чисел в переменных numberInt и numberDouble:

- преобразовать значение переменной `numberDouble` в `Int` и перемножить два целых числа;
- преобразовать значение переменной `numberInt` в `Double` и перемножить два дробных числа.

По выводу в области результатов видно, что переменная `resD` имеет более точное значение, чем переменная `resI`. Это говорит о том, что вариант, преобразующий целое число в дробное с помощью функции `Double(_:)`, точнее, чем использование функции `Int(_:)` для переменной типа `Double`, так как во втором случае дробная часть отбрасывается и игнорируется при расчетах.

ПРИМЕЧАНИЕ При преобразовании числа с плавающей точкой в целочисленный тип дробная часть отбрасывается, округление не производится.

Составной оператор присваивания

Swift позволяет максимально сократить объем кода. И чем глубже вы будете постигать этот замечательный язык, тем больше приемов, способных облегчить жизнь, вы узнаете. Одним из них является совмещение оператора арифметической операции (`+`, `-`, `*`, `/`, `%`) и оператора присваивания (`=`). Рассмотрим пример, в котором создадим целочисленную переменную и с помощью составного оператора присваивания будем изменять ее значение, используя минимум кода (листинг 5.14).

Листинг 5.14

```
// переменная типа Int
var someNumInt = 19
// прибавим к ней произвольное число
someNumInt += 5 // 24
/* эта операция аналогична выражению
someNumInt = someNumInt+5 */
// умножим его на число 3
someNumInt *= 3 // 72
/* эта операция аналогична выражению
someNumInt = someNumInt*3 */
// вычтем из него число 3
someNumInt -= 3 // 69
/* эта операция аналогична выражению
someNumInt = someNumInt-3 */
// найдем остаток от деления на 8
someNumInt %= 8 // 5
/* эта операция аналогична выражению
someNumInt = someNumInt%8 */
```

Для использования составного оператора присваивания необходимо после оператора арифметической операции без пробелов написать оператор присваивания. Результат операции возвращается и записывается в параметр, находящийся слева от составного оператора.

Благодаря составным операторам присваивания вы знаете уже минимум два способа проведения арифметических операций, например увеличение значения на единицу (листинг 5.15).

Листинг 5.15

```
// переменная типа Int
var myNumInt = 19
// увеличим ее значение с использованием арифметической операции
сложения
myNumInt = myNumInt + 1 // 20
// увеличим ее значение с использованием составного оператора
присваивания
myNumInt += 1 // 21
```

Каждое выражение увеличивает переменную `myNumInt` ровно на единицу.

ПРИМЕЧАНИЕ Использование составного оператора является той заменой операторам инкремента и декремента, которую предлагает нам Apple.

Способы записи числовых значений

Если в вашей школьной программе присутствовала информатика, то вы, возможно, знаете, что существуют различные системы счисления, например десятичная или двоичная. В реальном мире в большинстве случаев используется десятичная система, в то время как в компьютере все вычисления происходят в двоичной системе.

Swift при разработке программ позволяет задействовать самые популярные системы счисления:

- **Двоичная**. Числа записываются с использованием префикса `0b` перед числом.
- **Восьмеричная**. Числа записываются с использованием префикса `0o` перед числом.
- **Шестнадцатеричная**. Числа записываются с использованием префикса `0x` перед числом.
- **Десятичная**. Числа записываются без использования префикса в привычном и понятном для нас виде.

Целые числа могут быть записаны в любой из приведенных систем счисления. В листинге 5.16 продемонстрирован пример записи числа `17` в различных системах.

Листинг 5.16

```
// 17 в десятичном виде
let decimalInteger = 17
// 17 в двоичном виде
let binaryInteger = 0b10001
// 17 в восьмеричном виде
let octalInteger = 0o21
// 17 в шестнадцатеричном виде
let hexadecimalInteger = 0x11
```

В области результатов отображается, что каждое из приведенных чисел — это `17`.

Числа с плавающей точкой могут быть десятичными (без префикса) или шестнадцатеричными (с префиксом `0x`). Такие числа должны иметь одинаковую форму записи (систему исчисления) по обе стороны от точки.

Помимо этого, Swift позволяет использовать экспоненту. Для этого применяется символ `e` для десятичных чисел и символ `p` для шестнадцатеричных.

Для десятичных чисел экспонента указывает на степень десятки:

- `1.25e2` соответствует $1{,}25 \times 10^2$, или 125,0.

Для шестнадцатеричных чисел экспонента указывает на степень двойки:

- `0xFp-2` соответствует 15×2^{-2}, или 3,75.

В листинге 5.17 приведен пример записи десятичного числа 12,1875 в различных системах счисления и с использованием экспоненты.

Листинг 5.17

```
// десятичное число
let deciamlDouble = 12.1875 // 12,1875
// десятичное число с экспонентой
// соответствует выражению
// exponentDouble = 1.21875*101
let exponentDouble = 1.21875e1 // 12,1875
// шестнадцатеричное число с экспонентой
// соответствует
// выражению hexadecimalDouble = 0xC.3*20
let hexadecimalDouble = 0xC.3p0 // 12,1875
```

Арифметические операции доступны для чисел, записанных в любой из систем счисления. В области результатов вы всегда будете видеть результат выполнения в десятичном виде.

При записи числовых значений можно использовать символ нижнего подчеркивания (андерскор) для визуального отделения порядков числа (листинг 5.18).

Листинг 5.18

```
var number = 1_000_000 // 1000000
var nextNumber = 1000000 // 1000000
```

В обоих случаях, что с символом нижнего подчеркивания, что без него, получилось одно и то же число.

Андерскоры можно использовать для любого числового типа данных и для любой системы счисления.

5.3. Строковые типы данных

Числа очень важны в программировании, но все же они не являются единственным видом данных, с которым вам предстоит взаимодействовать в ходе разработки приложений. Строковые типы данных также очень распространены. Они позволяют работать с текстом. Вы можете представить программу без текста? Это непростая задача!

В качестве примеров текстовых данных в ваших программах могут служить имена людей, адреса их проживания, их логины и многое другое.

В этом разделе вы получите первичное представление о том, что же такое строки и строковые типы данных, из чего они состоят и как могут быть использованы в Swift.

Как компьютер видит строковые данные

Строка в естественном языке — это набор отдельных символов. Любой текст в конечном итоге состоит из символов, на каком бы языке ни писали. Символы могут обозначать один звук, группу звуков, целое слово или даже предложение — это не важно. Важно то, что минимальная единица в естественном языке — это символ.

В разделе про принципы работы компьютера было сказано, что современные устройства представляют всю информацию в виде чисел, состоящих из цифр 0 и 1. Но отдельные символы текстовых данных

и числа из 0 и 1 — это совершенно разные категории информации. Получается, что должны существовать механизмы, используя которые возможно представить любые текстовые данные в виде последовательности цифр.

И такой механизм существует. Имя ему Юникод (Unicode).

Unicode — это международный стандарт, описывающий, каким образом строки преобразуются в числа. До принятия Юникода было множество других стандартов, но главной их проблемой было то, что они в силу своей ограниченности не могли стать международными. Наверняка многие из вас помнят те времена, когда, заходя на сайт в интернете или открывая документ вы видели устрашающие кракозябры и совершенно нечитаемый текст. С того времени, как произошел переход Всемирной паутины на Юникод, такая проблема отпала.

Главная особенность Юникода в том, что для любого существующего символа (практически всех естественных языков) есть однозначно определяющая последовательность чисел. То есть для любого символа существует уникальная кодовая последовательность, называемая **кодовой точкой** (code point). Так, к примеру, маленькая латинская «а» имеет кодовую точку 97 (в десятичной системе счисления) или 61 (в шестнадцатеричной системе счисления).

ПРИМЕЧАНИЕ Если вы не ориентируетесь в существующих системах счисления, то советую познакомиться с ними самостоятельно с помощью дополнительного материала, например статей из Википедии.

Отмечу, что чаще всего вы будете использовать десятичную систему, а также шестнадцатеричную и двоичную.

Юникод содержит кодовые точки для огромного количества символов, включая латиницу, кириллицу, буквы других языков, знаки математических операций, иероглифы и даже эмодзи! Благодаря Юникоду данные, передаваемые между компьютерами, будут корректно раскодированы, то есть переведены из кодовой точки в символ.

Всего в описываемом стандарте есть место для более чем 1 миллиона символов, но на данный момент он содержит лишь около 140 тысяч (плюс некоторые диапазоны отданы для частного использования большим корпорациям). Оставшиеся места зарезервированы для будущего использования.

Приведем пример кодирования (перевода в числовую последовательность) слова «SWIFT» (заглавные буквы, латинский алфавит, кодовые точки в шестнадцатеричной системе счисления) (табл. 5.2).

Таблица 5.2. Соответствие символов и их кодовых точек

S	W	I	F	T
53	57	49	46	54

Все представленные коды довольно просты, так как это латинский алфавит (он располагается в самом начале диапазона). Кодовые точки некоторых менее распространенных символов содержат по пять шестнадцатеричных цифр (например, `100A1`).

Подобным образом может быть закодирован любой набор символов, для каждого из них найдется уникальный числовой идентификатор.

Но хотя мы и перевели слова в шестнадцатеричные числа, на аппаратном уровне должны быть лишь 0 и 1. Поэтому получившуюся числовую последовательность необходимо повторно закодировать. Для этой цели в состав стандарта Unicode входит не только словарь соответствий «символ — кодовая точка», но и описания специальных механизмов, называемых кодировками, к которым относятся UTF-8, UTF-16 и UTF-32. Кодировки определяют, каким образом из `5357494654` может получиться `0011010100111001001100010010111000110110`, а эту последовательность без каких-либо проблем можно передать по каналам связи или обработать на аппаратном уровне.

Таким образом, Юникод содержит исчерпывающую информацию о том, как из текста получить последовательность битов.

Swift имеет полную совместимость со стандартом Unicode, поэтому вы можете использовать совершенно любые символы в работе. Подробному изучению работы со строковыми данными будет посвящен этот раздел и одна из будущих глав книги. Отнеситесь к данному материалу со всей внимательностью, так как он будет очень востребован для работы над вашими приложениями.

Инициализация строковых значений

Немного отойдем от стандарта Unicode и займемся непосредственно строковыми данными в Swift.

Для работы с текстом предназначены два основным типа данных:

- тип `Character` предназначен для хранения отдельных символов;
- тип `String` предназначен для хранения произвольной текстовой информации. С ним мы уже встречались ранее.

Благодаря строковым типам данных вам обеспечена быстрая и корректная работа с текстом в программе.

Для создания строковых данных используются строковые литералы.

Строковый литерал — это фиксированная последовательность текстовых символов, окруженная с обеих сторон двойными кавычками (`""`).

Тип данных Character

Тип данных `Character` позволяет хранить строковый литерал длиной в один символ. Всего один символ! И не символом больше! Пример использования `Character` приведен в листинге 5.19.

Листинг 5.19

```
var char: Character = "a"
print(char)
```

Консоль

```
a
```

В переменной `char` хранится только один символ `a`.

При попытке передать строковый литерал длиной более одного символа в параметр типа `Character` Xcode сообщит об ошибке несоответствия типов записываемого значения и переменной, так как рассматривает его в качестве значения типа данных `String` (неявное определение типа данных).

Тип данных String

С помощью типа данных `String` в параметрах могут храниться строки произвольной длины.

ПРИМЕЧАНИЕ Swift — все еще идет по пути становления, в каждой новой версии в нем происходят значительные изменения. Так, в Swift уже трижды менялся подход к работе со строковыми типами данных.

Для определения данного строкового типа необходимо использовать ключевое слово `String`. Пример приведен в листинге 5.20.

Листинг 5.20

```
// переменная типа String
// тип данных задается явно
var stringOne: String = "Dragon"
```

При объявлении переменной `stringOne` указывается ее тип данных, после чего передается строковый литерал. В качестве строкового литерала может быть передана строка с любым количеством символов.

В большинстве случаев не требуется указывать тип объявляемого строкового параметра, так как Swift неявно задает его при передаче строки любого размера (листинг 5.21).

Листинг 5.21

```
let language = "Swift" // тип данных - String
var version = "5" // тип данных – String
```

Обратите внимание, что в обоих случаях тип объявленного параметра `String`, и задан он неявно. В случае с переменной `version` переданное значение является строковым, а не числовым литералом, так как передано в кавычках.

Запомните, что для строковых значений необходим литерал в один символ, в этом случае неявно определяется тип данных `String`, но не `Character`.

Пустые строковые литералы

Пустой строковый литерал — это строковый литерал, не содержащий символов. Другими словами, это пустая строка (кавычки без содержимого). Она также может быть проинициализирована в качестве значения.

Пустая строка (кавычки без содержимого) также является строковым литералом. Вы можете передать ее в качестве значения параметру типа данных `String` (листинг 5.22).

Листинг 5.22

```
// с помощью пустого строкового литерала
var emptyString = ""
// с помощью инициализатора типа String
var anotherEmptyString = String()
```

Обе строки в результате будут иметь идентичное (пустое) значение.

Напомню, что инициализатор — это специальный метод, встроенный в тип данных, в данном случае в `String`, который позволяет создать хранилище нужного нам типа.

При попытке инициализировать пустой строковый литерал параметру типа `Character` Xcode сообщит об ошибке несоответсвия типов, так

как пустая строка не является *отдельным символом* (рис. 5.1). Она является *пустой строкой*.

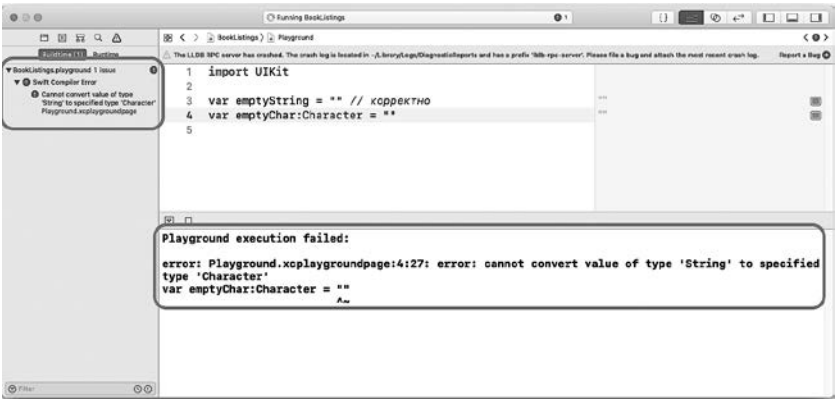

Рис. 5.1. Попытка инициализировать пустой строковый литерал

Многострочные строковые литералы

Swift позволяет писать строковые литералы в несколько строк, разделяя их символом переноса (нажатием клавиши Enter). В этом случае текст обрамляется с обеих сторон тремя двойными кавычками. При этом открывающие и закрывающие тройки кавычек обязательно должны находиться на строке, не содержащей текст литерала:

```
"""
строковый_литерал
"""
```

Пример приведен в листинге 5.23.

Листинг 5.23

```
let longString = """
   Это многострочный
   строковый литерал
"""
```

Приведение к строковому типу данных

Как уже неоднократно говорилось, помимо непосредственной передачи литерала, вы можете использовать специальную функцию,

в данном случае `String(_:)`, для инициализации значения строкового типа (5.24).

Листинг 5.24

```
// инициализация строкового значения
var notEmptyString = String("Hello, Troll!")
```

В переменной `notEmptyString` содержится строковое значение `"Hello, Troll!"`.

С помощью данной функции (по аналогии с числовыми типами данных) можно привести значение другого типа данных к строковому. К примеру, преобразуем дробное число к типу данных `String` (листинг 5.25).

Листинг 5.25

```
// переменная типа Double
var numDouble = 74.22
// строка, созданная на основе переменной типа Double
var numString = String(numDouble) // "74.22"
```

Объединение строк

При необходимости вы можете объединять несколько строк в одну более длинную. Для этого существует два механизма: **интерполяция** и **конкатенация**.

При **интерполяции** происходит объединение строковых литералов, переменных, констант и выражений в едином строковом литерале (листинг 5.26).

Листинг 5.26

```
// переменная типа String
var name = "Дракон"
// константа типа String с использованием интерполяции
let hello = "Привет, меня зовут \(name)!"
// интерполяция с использованием выражения
var meters: Double = 10
let text = "Моя длина \(meters * 3.28) фута"
// выведем значения на консоль
print(hello)
print(text)
```

Консоль

```
Привет, меня зовут Дракон!
Моя длина 32.8 фута
```

При инициализации значения константы `hello` используется переменная `name`. Такой подход мы видели ранее при знакомстве с функцией `print(_:)`.

Самое интересное, что, помимо имен параметров, вы можете использовать любое выражение (например, арифметическую операцию умножения), что и продемонстрировано в предыдущем листинге.

При **конкатенации** происходит объединение различных строковых значений в одно с помощью оператора сложения (+) (листинг 5.27).

Листинг 5.27

```
// константа типа String
let firstText = "Мой вес "
// переменная типа Double
var weight = 12.4
// константа типа String
let secondText = " тонны"
// конкатенация строк при инициализации значения новой переменной
var resultText = firstText + String(weight) + secondText
print(resultText)
```

Консоль

```
Мой вес 12.4 тонны
```

В данном примере используется оператор сложения для объединения различных строковых значений. Тип данных переменной `weight` не строковый, поэтому ее значение приводится к `String` с помощью соответствующей функции.

ПРИМЕЧАНИЕ Значения типа `Character` при конкатенации также должны преобразовываться к типу `String`.

Сравнение строк

Swift позволяет производить сравнение двух различных строк. Для этой цели используется оператор сравнения (==). Рассмотрим пример в листинге 5.28.

Листинг 5.28

```
let firstString = "Swift"
let secondString = "Objective-C"
let anotherString = "Swift"
firstString == secondString // false
firstString == anotherString // true
```

Значения, отображаемые в области результатов (`false`, `true`), определяют результат сравнения строк: `false` соответствует отрицательному результату, а `true` — положительному.

ПРИМЕЧАНИЕ Данная операция сравнения называется логической операцией. Она может быть проведена в том числе и для числовых значений. Подробнее с ней мы познакомимся уже в следующем разделе.

Юникод в строковых типах данных

В строковых литералах для определения символов можно использовать так называемые **юникод-скаляры** — специальные конструкции, состоящие из набора символов `\u{}` и заключенной между фигурными скобками кодовой точки символа в шестнадцатеричной форме.

В листинге 5.29а приведен пример использования юникод-скаляра для инициализации кириллического символа `K` в качестве значения параметра типа `Character`.

Листинг 5.29а

```
var myCharOverUnicode: Character = "\u{041A}"
myCharOverUnicode // K
```

Но не только тип `Character` совместим с Unicode, вы также можете использовать скаляры и для значений типа `String` (листинг 5.29б).

Листинг 5.29б

```
var stringOverUnicode = "\u{41C}\u{438}\u{440}\u{20}\u{412}\u{430}\
u{43C}\u{21}"
stringOverUnicode // "Мир Вам!"
```

Для каждого символа используется собственный юникод-скаляр (в том числе и для пробела). Но вы можете комбинировать, определяя в строке, таким образом, только необходимые символы, а остальные оставляя как есть.

В будущем мы вернемся к Unicode, более подробно изучив принципы его работы и методы взаимодействия со строковыми данными.

5.4. Логический тип данных

Изучение фундаментальных типов данных не завершается на числовых и строковых. В `Swift` существует специальный логический тип

данных, называемый Bool, способный хранить одно из двух значений: «истина» или «ложь». Значение «истина» обозначается как true, а «ложь» — как false. Вспомните, мы видели их, когда сравнивали строки.

Объявим переменную и константу логического типа данных (листинг 5.30).

Листинг 5.30

```
// логическая переменная с неявно заданным типом
var isDragon = true
// логическая константа с явно заданным типом
let isKnight: Bool = false
```

Как и для других типов данных в Swift, для Bool возможно явное и неявное определение типа, что видно из приведенного примера.

ПРИМЕЧАНИЕ Строгая типизация Swift препятствует замене других типов данных на Bool, как вы могли видеть в других языках, где, например, строки i = 1 и i = true обозначали одно и то же. В Xcode подобный подход вызовет ошибку.

Тип данных Bool обычно используется при работе с оператором if–else (познакомимся с ним несколько позже), который в зависимости от значения переданного ему параметра позволяет пускать выполнение программы по различным ветвям (листинг 5. 31).

Листинг 5.31

```
// логическая переменная
var isDragon = true
// конструкция условия
if isDragon {
   print("Привет, Дракон!")
}else{
   print("Привет, Тролль!")
}
```

Консоль:

```
Привет, Дракон!
```

Как вы можете видеть, на консоль выводится фраза «Привет, Дракон!». Оператор условия if-else проверяет, является ли переданное ему выражение истинным, и в зависимости от результата выполняет соответствующую ветвь.

Если бы переменная isDragon содержала значение false, то на консоль была бы выведена фраза «Привет, Тролль!».

Логические операторы

Логические операторы проверяют истинность какого-либо утверждения и возвращают соответствующее логическое значение. Swift поддерживает три стандартных логических оператора:

- логическое НЕ (`!a`);
- логическое И (`a && b`);
- логическое ИЛИ (`a || b`).

Унарный оператор *логического НЕ* является префиксным и записывается символом восклицания. Он возвращает инвертированное логическое значение операнда, то есть если операнд имел значение `true`, то вернется `false`, и наоборот. Для выражения `!a` данный оператор может быть прочитан как «не a» (листинг 5.32)

Листинг 5.32

```
var someBool = true
// инвертируем значение
!someBool // false
someBool // true
```

Переменная `someBool` изначально имеет логическое значение `true`. С помощью оператора логического НЕ возвращается инвертированное значение переменной `someBool`. При этом значение в самой переменной не меняется.

Бинарный оператор *логического И* записывается в виде удвоенного символа «амперсанд» и является инфиксным. Он возвращает `true`, когда оба операнда имеют значение `true`. Если значение хотя бы одного из операндов равно `false`, то возвращается значение `false` (листинг 5.33).

Листинг 5.33

```
let firstBool = true, secondBool = true, thirdBool = false
// группируем различные условия
var one = firstBool && secondBool // true
var two = firstBool && thirdBool // false
var three = firstBool && secondBool && thirdBool // false
```

Оператор логического И позволяет определить, есть ли среди переданных ему операндов ложные значения.

Бинарный оператор *логического ИЛИ* выглядит как удвоенный символ прямой черты и является инфиксным. Он возвращает `true`, когда хотя бы один из операндов имеет значение `true`. Если значения обоих операндов равны `false`, то возвращается значение `false` (листинг 5.34).

Листинг 5.34

```
let firstBool = true, secondBool = false, thirdBool = false
// группируем различные условия
let one = firstBool || secondBool // true
let two = firstBool || thirdBool // true
let three = secondBool || thirdBool // false
```

Оператор логического ИЛИ позволяет определить, есть ли среди значений переданных ему операндов хотя бы одно истинное.

Различные логические операторы можно группировать между собой, создавая сложные логические структуры. Пример показан в листинге 5.35.

Листинг 5.35

```
let firstBool = true, secondBool= false, thirdBool = false
var resultBool = firstBool && secondBool || thirdBool // false
var resultAnotherBool = thirdBool || firstBool && firstBool // true
```

При вычислении результата выражения Swift определяет значение подвыражений последовательно, то есть сначала первого, потом второго и т. д.

Для того чтобы указать порядок вычисления операций, необходимо использовать круглые скобки (точно как в математических примерах). То, что указано в скобках, будет выполняться в первую очередь (листинг 5.36).

Листинг 5.36

```
let firstBool = true, secondBool= false, thirdBool = true
var resultBool = firstBool && (secondBool || thirdBool) // true
var resultAnotherBool = (secondBool || (firstBool && thirdBool)) &&
thirdBool // true
```

Операторы сравнения

Swift позволяет производить сравнение однотипных значений друг с другом. Для этой цели используются операторы сравнения, результатом работы которых является значение типа `Bool`. Всего существует шесть стандартных операторов сравнения:

`==` Бинарный оператор эквивалентности (`a == b`) возвращает `true`, когда значения обоих операндов эквивалентны.

`!=` Бинарный оператор неэквивалентности (`a != b`) возвращает `true`, когда значения операндов различны.

- `>` Бинарный оператор «больше чем» (`a > b`) возвращает `true`, когда значение первого операнда больше значения второго операнда.
- `<` Бинарный оператор «меньше чем» (`a < b`) возвращает `true`, когда значение первого операнда меньше значения второго операнда.
- `>=` Бинарный оператор «больше или равно» (`a >= b`) возвращает `true`, когда значение первого операнда больше значения второго операнда или равно ему.
- `<=` Бинарный оператор «меньше или равно» (`a <= b`) возвращает `true`, когда значение первого операнда меньше значения второго операнда или равно ему.

Каждый из приведенных операторов возвращает значение, указывающее на справедливость некоторого утверждения. Несколько примеров и значений, которые они возвращают, приведены в листинге 5.37.

Листинг 5.37

```
// Утверждение "1 больше 2"
1 > 2 // false
// вернет false, так как оно ложно
// Утверждение "2 не равно 2"
2 != 2 // false
// вернет false, так как оно ложно
// Утверждение "1 плюс 1 меньше 3"
(1+1) < 3 // true
// вернет true, так как оно истинно
// Утверждение "5 больше или равно 1"
5 >= 1 //  true
// вернет true, так как оно истинно
```

Оператор сравнения можно использовать, например, с упомянутой выше конструкцией `if-else`. В следующих главах вы будете часто прибегать к его возможностям.

5.5. Псевдонимы типов

`Swift` предоставляет возможность создания псевдонима для любого типа данных. *Псевдонимом типа* называется дополнительное имя, по которому будет происходить обращение к данному типу. Для создания псевдонима используется оператор `typealias`. Псевдоним необходимо применять тогда, когда существующее имя типа неудобно использовать в контексте программы (листинг 5.38).

Листинг 5.38

```
// определяем псевдоним для типа UInt8
typealias ageType = UInt8
/* создаем переменную типа UInt8,
используя псевдоним */
var myAge: ageType = 29
```

В результате будет создана переменная `myAge`, имеющая значения типа `UInt8`.

ПРИМЕЧАНИЕ При разработке программ вы будете встречаться со сложными типами данных, для которых применение оператора `typealias` значительно улучшает читабельность кода.

У типа может быть произвольное количество псевдонимов. И все псевдонимы вместе с оригинальным названием типа можно использовать в программе (листинг 5.39).

Листинг 5.39

```
// определяем псевдоним для типа String
typealias textType = String
typealias wordType = String
typealias charType = String
//создаем переменные каждого типа
var someText: textType = "Это текст"
var someWord: wordType = "Слово"
var someChar: charType = "Б"
var someString: String = "Строка типа String"
```

В данном примере для типа данных `String` определяется три различных псевдонима. Каждый из них наравне с основным типом может быть использован для объявления параметров.

Созданный псевдоним наделяет параметры теми же возможностями, что и родительский тип данных. Однажды объявив его, вы сможете использовать данный псевдоним для доступа к свойствам и методам типа (листинг 5.40).

Листинг 5.40

```
// объявляем псевдоним
typealias ageType = UInt8
/* используем свойство типа
UInt8 через его псевдоним */
var maxAge = ageType.max //255
```

Для Swift обращение к псевдониму равносильно обращению к самому типу данных. Псевдоним — это ссылка на тип. В данном примере используется псевдоним `maxAge` для доступа к типу данных `UInt8` и свойству `max`.

ПРИМЕЧАНИЕ Запомните, что псевдонимы можно использовать для совершенно любых типов. И если данные примеры недостаточно полно раскрывают необходимость использования оператора `typealias`, то при изучении кортежей (в следующих разделах) вы встретитесь с составными типами, содержащими два и более подтипа. С такими составными типами также можно работать через псевдонимы.

5.6. Дополнительные сведения о типах данных

Осталось лишь несколько вопросов, которые необходимо рассмотреть, прежде чем двигаться к более сложным и не менее интересным темам.

Как узнать тип данных

Xcode и Swift позволяют узнать тип данных любого объявленного параметра. Для этого могут быть использованы два способа: либо с помощью справочного окна, либо с помощью глобальной функции `type(of:)`.

Если на клавиатуре зажать клавишу Option и навести указатель мыши на произвольный параметр в редакторе кода, то он должен принять вид знака вопроса. После щелчка левой кнопкой мыши появится всплывающее окно, содержащее справочную информацию, в которой и будет указан его тип данных (рис. 5.2).

Рис. 5.2. Уточнение типа данных параметра

Обратите внимание, что при объявлении параметра его тип данных был указан неявно, но во всплывающем окне он прописан. Кстати, дан-

ный способ получения справочной информации можно использовать не только с параметрами, но и с любыми другими элементами языка.

Также тип данных может быть определен с помощью глобальной функции `type(of:)`. В качестве входного аргумента необходимо передать имя параметра, тип которого необходимо определить. По возвращенному этой функцией значению можно сделать вывод о типе данных. Пример использования `type(of:)` показан в листинге 5.41.

Листинг 5.41

```
var myVar = 3.54
print( type(of: myVar) )
```

Консоль

```
Double
```

В будущих листингах данная функция будет применяться довольно часто для того, чтобы вы понимали, с данными каких типов работаете.

ПРИМЕЧАНИЕ Если использовать функцию `type(of:)` без вывода на консоль, то в области результатов к имени класса будет добавлено «`.Type`», например `Double.Type`. Для идентификации типа просто отбросьте данное окончание.

Хешируемые и сопоставимые типы данных

Типы данных могут быть классифицированы и сгруппированы по самым разным характеристикам, и чем дальше вы будете изучать материал, тем больше таких характеристик узнаете. Если тип обеспечивает какой-то определенный функционал, то он может быть отнесен к характерной категории, в которую помимо него могут входить и другие типы. Одними из важнейших категорий, с которыми вы будете встречаться в книге, являются *хешируемые* и *сопоставимые* типы данных.

Хешируемым (`Hashable`) называется такой тип данных, для значения которого может быть высчитан специальный цифровой код (называемый хешем). Причем этот код должен быть уникальным для любого значения данного типа.

ПРИМЕЧАНИЕ Хеш — это понятие из области криптографии. Он может принимать не только цифровой, но и буквенно-цифровой вид. Для его получения используются хеш-функции, реализующие алгоритмы шифрования. Их рассмотрение выходит за рамки данной книги, но я настоятельно советую ознакомиться с данным материалом самостоятельно.

Если тип данных является хешируемым, то его значение имеет свойство `hashValue`, к которому вы можете обратиться (листинг 5.42).

Листинг 5.42

```
var stringForHash = "Строка текста"
var intForHash = 23
var boolForHash = false

stringForHash.hashValue // 109231433150392402
intForHash.hashValue // 5900170382727681744
boolForHash.hashValue // 820153108557431465
```

Значения, возвращаемые свойством `hashValue` в вашем случае, будут отличаться (а также будут изменяться при каждом новом исполнении кода). Это связано с тем, что для высчитывания хеша используются переменные параметры вроде текущего времени.

Все изученные вами фундаментальные типы являются хешируемыми.

Сопоставимым (`Comparable`) называется такой тип данных, значения которого могут быть сопоставлены между собой с помощью операторов логического сравнения `<`, `<=`, `>=` и `>`. Другими словами, значения этого типа можно сравнить, чтобы определить, какое из них больше, а какое меньше.

Определить, является ли тип данных сопоставимым, очень просто. Достаточно, используя один из перечисленных логических операторов, сравнить два значения этого типа. Если в результате этого выражения будет возвращено `true` или `false`, то такой тип называется сопоставимым. Все строковые и числовые типы являются сопоставимыми, а вот `Bool` не позволяет сравнивать свои значения.

ПРИМЕЧАНИЕ Обратите внимание, что для сравнения используются логические операторы `<`, `<=`, `>=` и `>`. Операторы эквивалентности `==` и неэквивалентности `!=` отсутствует в этом списке. Если тип данных позволяет использовать `==` и `!=` для сравнения значений, то он относится к категории **эквивалентных** (`Equatable`).

Как вы можете видеть, в Swift просто огромное количество различных категорий, по которым могут быть разделены типы данных.

В листинге 5.43 показан пример проверки типов `String` и `Bool` на предмет возможности сопоставления значений.

Листинг 5.43

```
"string1" < "string2" //true
true < false // error: Binary operator '<' cannot be applied to two
'Bool' operands
```

Как видно из результата, тип данных `String` является сопоставимым, а `Bool` — нет (выражение вызвало ошибку).

Вы стали еще на один шаг ближе к написанию великолепных программ. Понять, что такое типы данных, и начать их использовать — это самый важный шаг в изучении Swift. Важно, чтобы также вы запомнили такие понятия, как хешируемый (`Hashable`) и сопоставимый (`Comparable`), так как уже в скором времени мы вернемся к ним для более подробного изучения материала.

ПРИМЕЧАНИЕ Помните, что все задания для самостоятельного решения представлены на сайте swiftme.ru.

Часть III. Контейнерные типы данных

Во второй части книги вы познакомились с понятием типа данных и получили первые знания о порядке работы с ним. Как было сказано, типы данных обеспечивают хранение различных видов информации: строк, чисел, логических значений. Настало время еще глубже погрузиться в возможности языка по работе с информацией.

Язык программирования Swift содержит специальные механизмы, обеспечивающие группировку множества значений для работы с ними как с одним целым. Представьте, что вы держите коробку с шариками для настольного тенниса. Для вас эта коробка — один объект, удобный для транспортировки и использования. При необходимости вы можете достать один из шариков, после чего использовать его по назначению.

В этой части вы познакомитесь с понятием и возможностями таких контейнерных типов, как кортежи, последовательности и коллекции. Знать и уметь их применять очень важно, так как без них нельзя представить ни одной хорошей программы.

6 Кортежи (Tuple)

Возможно, вы никогда не встречались в программировании с таким понятием, как кортежи, тем не менее это одно из очень полезных функциональных средств, доступных в Swift. Кортежи, к примеру, могут использоваться для работы с координатами. Согласитесь, куда удобнее орудовать конструкцией (x, y, z), записанной в одну переменную, чем создавать по отдельной переменной для каждой оси координат.

6.1. Основные сведения о кортежах

Кортеж (tuple) — это объект, который группирует значения различных типов в пределах одного составного значения.

Кортежи представляют наиболее простой из доступных в Swift способ объединения значений произвольных типов. У каждого отдельного значения в составе кортежа может быть собственный тип данных, который никак не зависит от других.

Литерал кортежа

Кортеж может быть создан с помощью **литерала кортежа**.

СИНТАКСИС

```
(значение_1, значение_2, ..., значение_N)
```

- `значение: Any` — очередное значение произвольного типа данных, включаемое в состав кортежа.

Литерал кортежа возвращает определенное в нем значение в виде кортежа. Литерал обрамляется в круглые скобки и состоит из набора независимых значений, отделяемых друг от друга запятыми. Количество элементов может быть произвольным, но не рекомендуется использовать больше семи.

ПРИМЕЧАНИЕ Далее по тексту будет встречаться большое количество блоков «Синтаксис». В них описывается, каким образом могут быть использованы рассматриваемые конструкции в реальных программах. Прежде чем перейти к данным блокам, необходимо описать структуру этого блока, а также используемые элементы.

Первой строкой всегда будет определяться непосредственно сам синтаксис, описывающий использование конструкции в коде программы. При этом могут использоваться условные элементы (как, например, `значение_1`, `значение_2` и `значение_N` в синтаксисе, приведенном выше). Обычно они написаны в виде текста, позволяющего понять их назначение.

Далее могут следовать подробные описания условных элементов. В некоторых случаях элементы могут группироваться (как в синтаксисе выше, где вместо элементов `значение_1`, `значение_2` и т. д. описывается элемент `значение`, требования которого распространяются на все элементы с данным названием).

После имени условного элемента может быть указан его тип данных. При этом если в качестве элемента должно быть использовано конкретное значение определенного типа (к примеру, строковое, числовое, логическое и т. д.), то тип отделяется двоеточием (`:`). Если же в качестве элемента может быть использовано выражение (например, `a+b` или `r > 100`), то тип будет указан после тире и правой угловой скобки, изображающих стрелку (`->`). Может быть определен как один (например, `Int`) , так и множество типов данных (например, `Int, String`).

В синтаксисе выше используется `Any` в качестве указателя на тип данных. `Any` обозначает любой тип данных. В процессе изучения вы будете встречать все новые и новые типы, которые могут быть указаны в данном блоке синтаксиса.

Далее может идти подробное описание синтаксиса, а также пример его использования.

Советую вам поставить закладку на данной странице, чтобы при необходимости всегда вернуться сюда.

Кортеж хранится в переменных и константах точно так же, как значения фундаментальных типов данных.

СИНТАКСИС

```
let имяКонстанты = (значение_1, значение_2, ..., значение_N)
var имяПеременной = (значение_1, значение_2, ..., значение_N)
```

Объявление переменной и константы и инициализация им литерала кортежа, состоящего из N элементов, в качестве значения. Для записи кортежа в переменную необходимо использовать оператор `var`, а для записи в константу — оператор `let`.

Рассмотрим следующий пример. Объявим константу и проинициализируем ей кортеж, состоящий из трех элементов типов: `Int`, `String` и `Bool` (листинг 6.1).

Листинг 6.1

```
let myProgramStatus = (200, "In Work", true)
myProgramStatus // (.0 200, .1 "In Work", .2 true)
```

В данном примере `myProgramStatus` — константа, содержащая в качестве значения кортеж, описывающий статус работы некоей программы и состоящий их трех элементов:

- `200` — целое число типа `Int`, код состояния программы;
- `"In work"` — строковый литерал типа `String`, текстовое описание состояния программы;
- `true` — логическое значение типа `Bool`, возможность продолжения функционирования программы.

Данный кортеж группирует значения трех различных типов данных в пределах одного, проинициализированного константе.

Тип данных кортежа

Но если кортеж группирует значения различных типов данных в одно, то какой же тогда тип данных у самого кортежа и параметра, хранящего его значение?

Тип данных кортежа — это фиксированная упорядоченная последовательность имен типов данных элементов кортежа.

СИНТАКСИС

```
(имя_типа_данных_элемента_1, имя_типа_данных_элемента_2, ...,
имя_типа_данных_элемента_N)
```

Тип данных обрамляется в круглые скобки, а имена типов элементов отделяются друг от друга запятыми. Порядок указания имен типов **обязательно** должен соответствовать порядку следования элементов в кортеже.

Пример

```
(Int, Int, Double)
```

Типом данных кортежа `myProgramStatus` из листинга 6.1 является (`Int`, `String`, `Bool`). При этом тип данных задан неявно, так как определен автоматически на основании элементов кортежа. Так как порядок указания типов данных должен соответствовать порядку следования элементов в кортеже, тип (`Bool`, `String`, `Int`) является отличным от (`Int`, `String`, `Bool`).

В листинге 6.2 производится сравнение типов данных различных кортежей.

Листинг 6.2

```
var tuple1 = (200, "In Work", true)
var tuple2 = (true, "On Work", 200)
print( type(of:tuple1) == type(of:tuple2) )
```

Консоль

```
false
```

Для сравнения типов данных кортежей используются значения, возвращаемые глобальной функцией `type(of:)`, определяющей тип переданного в него объекта.

Предположим, что в кортеже `myProgramStatus` первым элементом вместо целочисленного должно идти значение типа `Float`. В этом случае можно явно определить тип данных кортежа (через двоеточие после имени параметра) (листинг 6.3).

Листинг 6.3

```
// объявляем кортеж с явно заданным типом
let floatStatus: (Float, String, Bool) = (200.2, "In Work", true)
floatStatus // (.0 200.2, .1 “In Work”, .2 true)
```

Вы не ограничены каким-либо определенным количеством элементов кортежа. Кортеж может содержать столько элементов, сколько потребуется (но помните, что не рекомендуется использовать больше семи элементов). В листинге 6.4 приведен пример создания кортежа из четырех элементов одного типа. При этом используется псевдоним типа данных `Int`, что не запрещается.

Листинг 6.4

```
// объявляем псевдоним для типа Int
typealias numberType = Int
// объявляем кортеж и инициализируем его значение
let numbersTuple: (Int, Int, numberType, numberType) = (0, 1, 2, 3)
numbersTuple // (.0 0, .1 1, .2 2, .3 3)
```

6.2. Взаимодействие с элементами кортежа

Кортеж предназначен не только для установки и хранения некоторого набора значений, но и для взаимодействия с этими значениями. В этом разделе мы разберем способы взаимодействия со значениями элементов кортежа.

Инициализация значений в параметры

Вы можете одним выражением объявить новые параметры и проинициализировать в них значения всех элементов кортежа. Для этого после ключевого слова `var` (или `let`, если объявляете константы) в скобках и через запятую необходимо указать имена новых параметров, а после оператора инициализации передать кортеж. Обратите внимание, что количество объявляемых параметров должно соответствовать количеству элементов кортежа (листинг 6.5).

Листинг 6.5

```
// записываем значения кортежа в переменные
var (statusCode, statusText, statusConnect) = myProgramStatus
// выводим информацию
print("Код ответа – \(statusCode)")
print("Текст ответа – \(statusText)")
print("Связь с сервером – \(statusConnect)")
```

Консоль:

```
Код ответа – 200
Текст ответа – In Work
Связь с сервером – true
```

С помощью данного синтаксиса можно с легкостью инициализировать произвольные значения сразу нескольким параметрам. Для этого в правой части выражения, после оператора инициализации, необходимо передать не параметр, содержащий кортеж, а литерал кортежа (листинг 6.6).

Листинг 6.6

```
/* объявляем две переменные с одновременной
инициализацией им значений */
var (myName, myAge) = ("Тролль", 140)
// выводим их значения
print("Мое имя \(myName), и мне \(myAge) лет")
```

Консоль:

```
Мое имя Тролль, и мне 140 лет
```

Переменные `myName` и `myAge` инициализированы соответствующими значениями элементов кортежа (`"Тролль", 140`).

При использовании данного синтаксиса вы можете игнорировать произвольные элементы кортежа. Для этого в качестве имени переменной, соответствующей элементу, который будет игнорироваться, необходимо указать символ нижнего подчеркивания (листинг 6.7).

Листинг 6.7

```
/* получаем только необходимые
значения кортежа */
var (statusCode, _, _) = myProgramStatus
```

В результате в переменную `statusCode` запишется значение первого элемента кортежа — `myProgramStatus`. Остальные значения будут проигнорированы.

ПРИМЕЧАНИЕ Символ нижнего подчеркивания в Swift обозначает игнорирование параметра. Это не единственный пример, где он может быть использован. В дальнейшем при изучении материала книги вы еще неоднократно с ним встретитесь.

Доступ к элементам кортежа через индексы

Каждый элемент кортежа, помимо значения, содержит целочисленный индекс, который может быть использован для доступа к данному элементу. Индексы всегда расположены по порядку, начиная с нуля. Таким образом, в кортеже из N элементов индекс первого элемента будет 0, а к последнему можно обратиться с помощью индекса N-1.

При доступе к отдельному элементу индекс указывается через точку после имени параметра, в котором хранится кортеж. В листинге 6.8 приведен пример доступа к отдельным элементам кортежа.

Листинг 6.8

```
// выводим информацию с использованием индексов
print(" Код ответа — \(myProgramStatus.0)")
print(" Текст ответа — \(myProgramStatus.1)")
print(" Связь с сервером — \(myProgramStatus.2)")
```

Консоль:

```
Код ответа — 200
Текст ответа — In Work
Связь с сервером — true
```

Доступ к элементам кортежа через имена

Помимо индекса, каждому элементу кортежа может быть присвоено уникальное имя. Имена элементов не являются обязательным параметром и используются только для удобства использования кортежей. Имена могут быть даны как всем, так и части элементов. Имя элемен-

та может быть определено как в литерале кортежа, так и при явном определении его типа.

В листинге 6.9 показан пример определения имен элементов кортежа через литерал.

Листинг 6.9

```
let statusTuple = (statusCode: 200, statusText: "In Work",
statusConnect: true)
```

Указанные имена элементов могут быть использованы при получении значений этих элементов. При этом применяется тот же синтаксис, что и при доступе через индексы, когда индекс указывался через точку после имени параметра. Определение имен не лишает вас возможности использовать индексы. Индексы в кортеже можно задействовать всегда.

В листинге 6.10 показано, как используются имена элементов совместно с индексами.

Листинг 6.10

```
// выводим информацию с использованием индексов
print(" Код ответа – \(statusTuple.statusCode)")
print(" Текст ответа – \(statusTuple.statusText)")
print(" Связь с сервером – \(statusTuple.2)")
```

Консоль:

```
Код ответа  – 200
Текст ответа  – In Work
Связь с сервером  – true
```

Доступ к элементам с использованием имен удобнее и нагляднее, чем доступ через индексы.

Как говорилось ранее, имена элементов могут быть заданы не только в литерале кортежа, но и при явном определении типа данных. В листинге 6.11 показан соответствующий пример.

Листинг 6.11

```
/* объявляем кортеж с
указанием имен элементов
в описании типа */
let anotherStatusTuple: (statusCode: Int, statusText: String,
statusConnect: Bool) = (200, "In Work", true)
// выводим значение элемента
anotherStatusTuple.statusCode // 200
```

Редактирование кортежа

Для однотипных кортежей можно производить операцию инициализации значения одного кортежа в другой. Рассмотрим пример в листинге 6.12.

Листинг 6.12

```
var myFirstTuple: (Int, String) = (0, "0")
var mySecondTuple = (100, "Код")
// копируем значение одного кортежа в другой
myFirstTuple = mySecondTuple
myFirstTuple // (.0 100, .1 "Код")
```

Кортежи `myFirstTuple` и `mySecondTuple` имеют один и тот же тип данных, поэтому значение одного может быть инициализировано в другой. У первого тип задан явно, а у второго через инициализируемое значение.

Индексы и имена могут использоваться для изменения значений отдельных элементов кортежа (листинг 6.13).

Листинг 6.13

```
// объявляем кортеж
var someTuple = (200, true)
// изменяем значение отдельного элемента
someTuple.0 = 404
someTuple.1 = false
someTuple // (.0 404, .1 false)
```

Кортежи очень широко распространены в Swift. К примеру, с их помощью вы можете с легкостью вернуть из вашей функции не одно, а несколько значений (с функциями мы познакомимся несколько позже).

ПРИМЕЧАНИЕ Кортежи не позволяют создавать сложные структуры данных, их единственное назначение — сгруппировать некоторое множество разнотипных или однотипных параметров и передать в требуемое место. Для создания сложных структур необходимо использовать средства объектно-ориентированного программирования (ООП), а точнее, классы или структуры. С ними мы познакомимся в дальнейшем.

6.3. Сравнение кортежей

Сравнение кортежей производится последовательным сравнением элементов кортежей: сперва сравниваются первые элементы обоих кортежей, если они идентичны, то производится сравнение следующих

элементов, и так далее до тех пор, пока не будут обнаружены неидентичные элементы (листинг 6.14).

Листинг 6.14

```
(1, "alpha") < (2, "beta") // true
// истина, так как 1 меньше 2.
// вторая пара элементов не учитывается
(4, "beta") < (4, "gamma") // true
// истина, так как "beta" меньше "gamma".
(3.14, "pi") == (3.14, "pi") // true
//истина, так как все соответствующие элементы идентичны
```

ПРИМЕЧАНИЕ Встроенные механизмы Swift позволяют сравнивать кортежи с количеством элементов менее семи. При необходимости сравнения кортежей с бóльшим количеством элементов вам необходимо реализовать собственные механизмы. Данное ограничение в Apple ввели не от лени: если ваш кортеж имеет большое количество элементов, то есть повод задуматься о том, чтобы заменить его структурой или классом.

7 Последовательности и коллекции

Как и в любом языке программирования, в `Swift` есть механизмы агрегации произвольных значений для манипуляции ими как единым значением. С первым и самым простым из них (кортежи) вы познакомились в предыдущей главе. Помимо кортежей, Swift позволяет использовать последовательности и коллекции, знакомству с которыми будут посвящены несколько последующих глав.

7.1. Классификация понятий

В нашем физическом мире все может быть классифицировано и подробно описано, «разложено по полочкам». Для любого объекта можно выделить свойства и найти признаки, по которым он будет связан с другими объектами или объединен с ними в группы.

Предположим, что мы хотим классифицировать группу физических объектов «автомобили». Автомобиль — это техническое средство, обладающее в самом упрощенном виде двигателем и колесами. Автомобиль — это наивысшее понятие в рассматриваемой области знаний. Любая конкретная модель авто обладает описанными свойствами (двигателем и колесами), то есть относится к категории «Автомобили» (рис. 7.1).

Рис. 7.1. Классификация автомобилей

Но конечно же, на этом классификация не заканчивается. Мы можем выделить еще ряд характеристик, которые позволят разбить главную категорию на подкатегории. К примеру, по используемому источнику энергии их можно разделить на авто с электродвигателями и с двигателями внутреннего сгорания. В этом случае конкретные модели могут быть распределены уже по данным подкатегориям (рис. 7.2).

Рис. 7.2. Классификация автомобилей

Каждая категория может быть описана подкатегориями по различным свойствам, а каждая конкретная реализация может входить в различные категории (рис. 7.3).

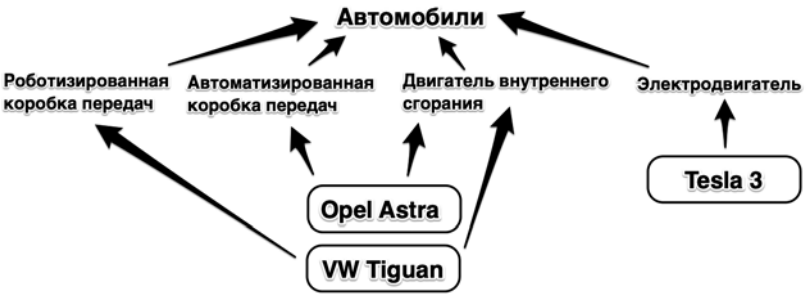

Рис. 7.3. Классификация автомобилей

Категории по своей сути — это всего лишь наборы требований. Они не описывают, как эти требования должны быть реализованы. Категория «Автомобили» требует наличия двигателя и колес, но не дает никаких рекомендаций по их конкретной реализации.

Такие наборы требований (категории) в Swift называются **протоколами** (а в некоторых других языках программирования вы могли встретить понятие **интерфейса**). В будущем мы подробно изучим

все возможности протоколов, а также научимся самостоятельно их создавать и использовать.

Если категория — это набор требований, то порядок их реализации в реальном мире описывается в стандарте. Именно стандарт определяет, что и как должно функционировать. Стандарты в Swift реализованы в виде **типов данных**. Да-да, тех самых типов данных, с которыми мы знакомимся в течение уже нескольких глав. Любой тип данных может реализовывать множество протоколов, а может и не реализовывать их вовсе.

В результате реализации стандарта создается конечное изделие со своими уникальными характеристиками. К примеру, для нескольких одинаковых автомобилей, выполненных по конкретным чертежам (стандартам), уникальным будет VIN.

Конкретная реализация типа данных в Swift называется **объектом, или экземпляром** (это синонимы, но Apple рекомендует использовать второе понятие). В будущем мы еще неоднократно вернемся к данному вопросу, несколько раз подробно разобрав его.

В табл. 7.1 показано соответствие понятий реального мира и мира Swift, а также несколько примеров. Начните ее изучение с двух центральных столбцов.

Таблица 7.1. Соответствие понятий

Пример из реального мира	Понятия реального мира	Понятия в Swift	Пример из Swift
Автомобили с бензиновым двигателем	Категория	Протокол	SignedNumber (требует, чтобы тип данных обеспечивал хранение как положительных, так и отрицательных чисел)
Kia Rio	Стандарт (модель)	Тип данных	Int
Kia Rio VIN XW122FX849	Конкретная реализация	Экземпляр	2 (целое число типа Int)

Протокол ▸ Тип данных ▸ Экземпляр — это типовой вариант реализации функционала в Swift, но это относится к методологиям объектно-ориентированного и протокол-ориентированного программирования, которые будут рассмотрены в дальнейшем.

Кстати, ранее вы уже познакомились с несколькими протоколами, но тогда они были названы категориями. Помните о `Hashable`, `Comparable` и `Equatable`? Все это протоколы, содержащие требования к функционалу типов данных.

ПРИМЕЧАНИЕ Думаю, вы заметили, что материал книги постепенно усложняется. Уверен, что многие из вас не поймут все и сразу, так как для закрепления изученного требуется практика, много практики. Продолжайте обучение, и в нужный момент лампочка над головой загорится, когда все начнет складываться в единую картину.

Но в любом случае все, о чем я говорю, — это очень важно! Вам необходимо читать, перечитывать и вчитываться, стараясь получить из описания максимум полезной информации.

7.2. Последовательности (Sequence)

Представьте, что на вашем столе лежат разноцветные шарики. Так как они мешают изучению Swift (рука постоянно тянется потрогать их), вы решаете убрать надоедливые предметы в специальный цилиндрический кейс, где они располагаются один на другом. При этом вы можете видеть только верхний шарик. Для того чтобы добраться к нижнему, вам потребуется достать все предыдущие. Такой порядок группировки элементов в Swift называется последовательностью.

Последовательность (`Sequence`) — набор элементов, выстроенных в очередь, в котором есть возможность осуществлять последовательный (поочередный) доступ к ним. При этом не предусмотрен механизм, позволяющий обратиться к какому-либо определенному элементу. Вы, как было сказано выше, можете лишь последовательно перебирать элементы последовательности.

`Sequence` — это название протокола в Swift, который определяет требования к последовательностям. Если какой-либо тип данных принимает к реализации протокол (выполняет его требования), то значение этого типа (экземпляр) можно назвать последовательностью. Протокол `Sequence` требует, чтобы тип данных обеспечивал хранение множества однотипных значений и организовывал последовательный доступ к ним с помощью последовательного перебора.

Вернемся к примеру с шариками. Перед вами возникла задача определить, есть ли среди них зеленый. Вы смотрите на верхний шарик и понимаете, что он красный. Достаете его и видите желтый. Достаете его и видите нужный зеленый.

Говоря языком программирования, некий виртуальный указатель в исходном состоянии направлен на первый шарик. Проведя его анализ (определив цвет), указатель переходит к следующему шарику и анализирует. И так далее (рис. 7.4).

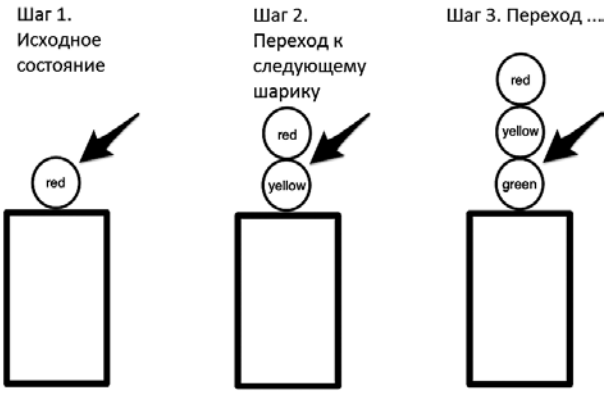

Рис. 7.4. Поиск элемента коллекции

Последовательности могут быть конечными и бесконечными. К примеру, последовательность Фибоначчи (каждое последующее значение — это сумма двух предыдущих) является бесконечной. Программа может перебирать элементы, но ее конец никогда не будет достигнут.

С примерами типов данных, реализующих требования последовательностей, мы познакомимся немного позже.

7.3. Коллекции (Collection)

Последовательности предъявляют довольно ограниченные требования к типам данных, реализующим их. Среди этих требований нет необходимости реализации механизма прямого доступа к произвольному элементу. К примеру, если тип данных является последовательностью, то для доступа к необходимому элементу требуется организовать поочередный доступ ко всем предыдущим элементам. Для решения проблемы прямого доступа к элементу Swift позволяет использовать коллекции.

Коллекция (`Collection`) — это последовательность (`Sequence`), в которой можно обращаться к отдельному элементу напрямую. Други-

ми словами, `Collection` — это протокол, основанный на протоколе `Sequence`, который при этом имеет дополнительное требование по обеспечению прямого доступа к элементам. Помимо этого, коллекция не может быть бесконечной (в отличие от `Sequence`).

Для доступа к элементам коллекции используются **индексы** (indicies). Они могут быть представлены как в виде обычного числового значения (порядковый номер элемента), так и в виде более сложных структур (как, например, в словарях или строках, но об этом поговорим позже).

На рис. 7.5 приведена упрощенная схема классификации рассмотренных протоколов и их требований.

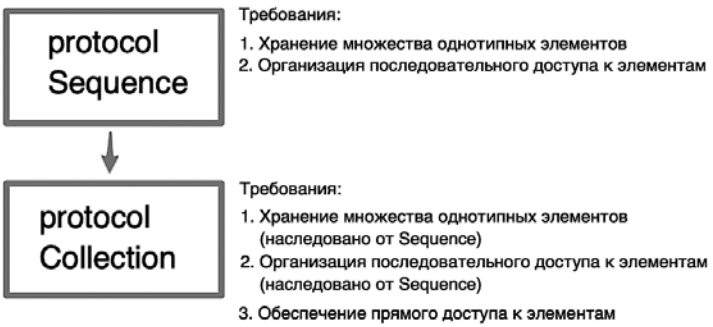

Рис. 7.5. Классификация протоколов и их требования

В будущем вы научитесь создавать собственные типы данных, основанные на последовательностях и коллекциях. А сейчас необходимо уделить особое внимание информации, находящейся в этой части книги, так как приведенный материал достаточно важен и будет регулярно использоваться в будущем.

7.4. Работа с документацией

Если у вас возникают вопросы при изучении материала, это говорит о том, что обучение идет правильно. Если какой-то материал оказывается для вас сложным и необъятным, это также хороший знак, так как у вас появляется стимул самостоятельно искать ответы на поставленные вопросы и разбираться с проблемными моментами. Изучение языка само по себе не дает навыков разработки приложений. Для этого

требуется изучить большое количество дополнительного материала, а самое важное — необходимо научиться работать с официальной документацией, с помощью которой вы сможете ответить на большинство возникающих вопросов.

Swift имеет большое количество разнообразных типов, но все они описаны в документации. Ранее мы говорили с вами, что типы могут быть разделены на категории в соответствии с реализуемым ими функционалом (реализуемыми протоколами).

Предлагаю на примере разобраться с одним утверждением, о котором мы говорили ранее. Это поможет вам лучше воспринимать материал, которому посвящена данная часть книги.

Утверждение: «Все фундаментальные типы являются хешируемыми».

Напомню, что хешируемым называется такой тип данных, для значений которого может быть высчитан специальный числовой хеш, доступ к которому может быть получен через свойство `hashValue`. Если тип называется хешируемым, значит, он соответствует протоколу `Hashable`.

Выделим из данного утверждения частный случай: «Тип данных `Int` является хешируемым, то есть соответствует протоколу `Hashable`».

В Xcode Playground напишите код из листинга 7.1.

Листинг 7.1

```
var intVar = 12
```

В данном примере создается переменная целочисленного типа со значением 12. Зажмите клавишу Opt и щелкните по имени переменной, после чего отобразится информационное окно, содержащее сведения об имени, типе переменной и месте ее объявления (рис. 7.6).

Рис. 7.6. Всплывающее информационное окно

Если щелкнуть по имени типа данных (`Int`), то откроется новое окно, содержащее справочную документацию (рис. 7.7).

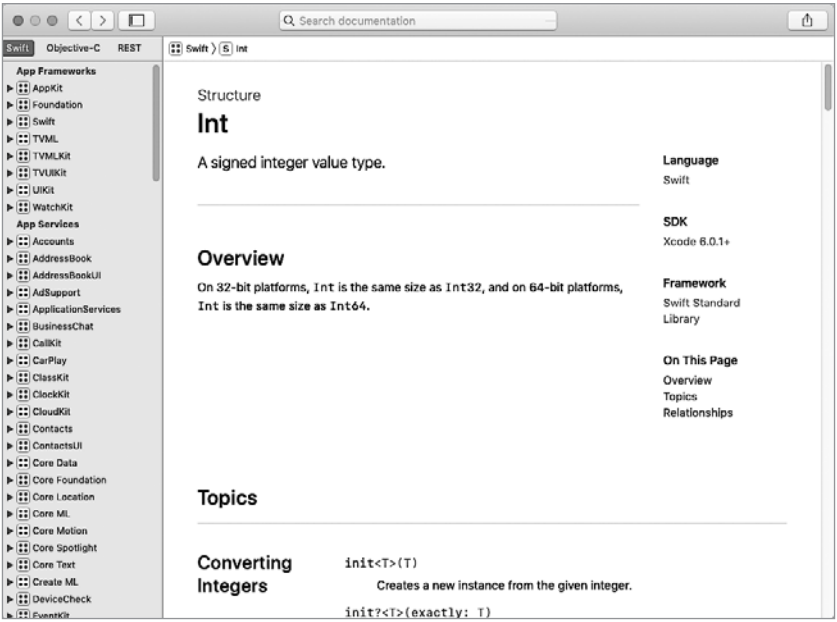

Рис. 7.7. Окно со справочной документацией

Здесь вы можете ознакомиться с описанием типа данных, а также его методов и свойств.

ПРИМЕЧАНИЕ Для работы с документацией необходимо знание английского языка! И к сожалению, ситуация вряд ли изменится. Отечественные разработчики не спешат создавать новые интересные учебные материалы, а зачастую ограничиваются лишь переводом зарубежных статей.

Если у вас возникают трудности с чтением англоязычных статей, то самое время начать исправлять ситуацию. В любом случае ваша карьера во многом зависит от знания языка, каждая вторая фирма «выкатывает» это требование в описании вакансии.

Я считаю, что необязательно тратить тысячи рублей ежемесячно на занятия с преподавателем (особенно если у вас уже есть школьный или институтский опыт изучения). Лично я изучаю и поддерживаю уровень языка с помощью следующих средств:

- Мобильные приложения. Duolingo для словарного запаса, Simpler и «Полиглот» для повторения грамматики (плюс в Simpler потрясающие детективные истории помогают учиться слушать иностранную речь), Lingualeo для чтения небольших несложных рассказов и тематических статей. Так можно с пользой проводить 20–30 минут дороги в общественном транспорте.

- Адаптированные книги. Вы можете купить, к примеру, «Тома Сойера» или «Алису в Стране чудес», но адаптированную для чтения новичками. Самое важное, что есть возможность выбора книг в соответствии с уровнем знания языка. К некоторым изданиям также идут аудиоматериалы (тренировка восприятия речи на слух).
- Учебные видео на Youtube. Тот же «Полиглот» будет очень полезен для начинающих.
- Общение с носителями языка по Skype. Для этого существует большое количество сервисов, где вы можете найти себе партнера для бесед (в большинстве случаев это платные услуги).
- Просмотр фильмов с оригинальной дорожкой. Именно это всегда было целью изучения мной английского языка!

Но возможно, для вас правильным является именно обучение с преподавателем. Самое важное в этом деле — не бояться пробовать, не бояться ошибок! И очень советую — как можно раньше начинайте слушать речь: песни, фильмы с субтитрами и т. д., так как знание языка не должно оканчиваться на умении читать.

Если перейти в самый конец страницы описания типа `Int`, то там вы найдете раздел Comfort To, в котором как раз и описаны протоколы, которым соответствует данный тип (рис. 7.8). Среди них вы найдете и упомянутый выше `Hashable`.

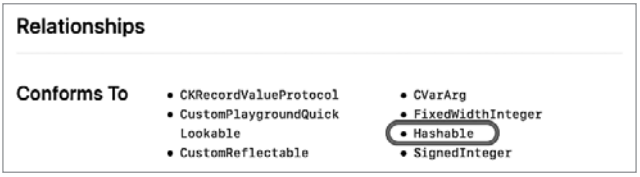

Рис. 7.8. Протоколы типа данных Int

Обратите внимание, что в данном списке отсутствуют протоколы `Comparable` и `Equatable`, хотя ранее говорилось, что `Int`, как и другие фундаментальные типы, соответствует и им. С этим мы разберемся немного позже.

Если щелкнуть по имени протокола `Hashable`, то произойдет переход к его описанию. В данном случае описание куда более подробное, нежели у `Int`. При желании вы можете ознакомиться с ним.

В разделе Adopted By перечислены все типы данных, которые соответствуют данному протоколу. Обратите внимание на раздел Inherits From, в нем указан уже известный нам протокол `Equatable`. В этом разделе перечисляются протоколы, которые являются родительскими по отношению к данному. Таким образом, если какой-то тип данных соответствует требованиям протокола `Hashable`, то это значит, что он

автоматически соответствует требованиям протокола `Equatable`. Так происходит и с типом `Int` (помните, выше у данного типа в разделе Comforts To отсутствовала ссылка на `Equatable`?). Исходя из этого можно сказать, что протоколы могут быть в отношениях «отец–сын».

Но все еще остается неясным, каким образом тип `Int` относится к протоколу `Comparable`. Если провести небольшой анализ и выстроить всю цепочку зависимостей, то все встанет на свои места (рис. 7.9).

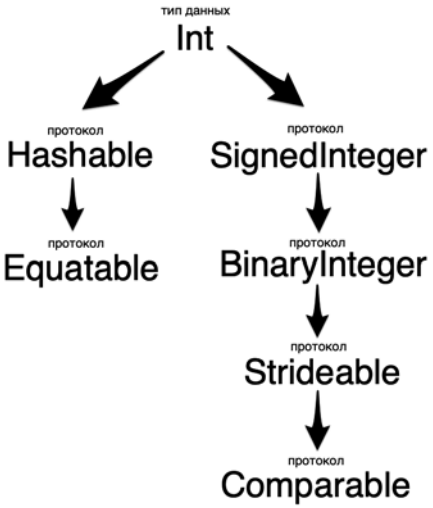

Рис. 7.9. Порядок следования протоколов в типе Int

Таким образом, тип `Int` выполняет требования протокола `Comparable` через линейку протоколов `SignedInteger`, `BinaryInteger` и `Strideable`. Каждый дочерний протокол (сын) выполняет требования отца, но также накладывает дополнительные требования.

Но что такое реализация протокола? Что значит, если тип выполняет его требования? Как выполняются эти требования? Обо всем этом вы узнаете в будущих главах книги.

Работая с документацией, вы сможете найти ответы практически на любые вопросы, связанные с разработкой. Не бойтесь искать необходимые материалы, это может значительно улучшить качество вашего обучения.

ПРИМЕЧАНИЕ Вы можете в любой момент открыть окно документации с помощью пункта Developer Documentation в разделе Help главного меню Xcode.

8 Диапазоны (Range)

Знакомство с последовательностями и коллекциями необходимо начинать с диапазонов — специальных типов данных, позволяющих создавать упорядоченные множества последовательных значений. Диапазоны могут быть конечными и бесконечными, ограниченными слева, справа или с двух сторон. Примером диапазона может служить интервал, включающий целочисленные значения от 1 до 100, идущие по порядку (1, 2, 3, ... 99, 100).

Также они позволяют значительно экономить память компьютера, так как для хранения любого множества (пусть даже с миллионами элементов) необходимо указать лишь начальное и конечное значение. Все промежуточные элементы будут рассчитаны автоматически.

Для создания диапазонов используются два вида операторов: полуоткрытый (`..<`) и закрытый (`...`).

8.1. Оператор полуоткрытого диапазона

Оператор полуоткрытого диапазона обозначается двумя точками и знаком меньше (`..<`). Swift предлагает две формы данного оператора: бинарную (оператор размещен между операндами) и префиксную (оператор размещен перед операндом).

Бинарная форма оператора

Данная форма оператора используется между двумя операндами, определяющими границы создаваемого диапазона.

СИНТАКСИС

```
левая_граница..<правая_граница
```

- `левая_граница: Comparable` — первый элемент диапазона, заданный с помощью сопоставимого типа данных.

- `правая_граница: Comparable` — элемент, следующий за последним элементом диапазона, заданный с помощью сопоставимого типа данных.

Диапазон элементов от левой границы до предшествующего правой границе элемента. В диапазоне `1..<N` первый элемент будет `1`, а последний — `N-1`. Начальное значение должно быть меньше или равно конечному. Попытка создать диапазон, например, от `5` до `2` приведет к ошибке.

Пример

```
var myRange = 1..<500
```

ПРИМЕЧАНИЕ Обратите внимание, что при описании условных элементов синтаксиса могут быть использованы указатели не только на конкретные типы данных, но и на целые группы типов с помощью имен протоколов. Так, `Comparable` говорит о том, что использованное значение должно быть сопоставимого типа данных, а `Collection` — о том, что значение должно быть коллекцией.

Рассмотрим пример использования бинарного оператора полуоткрытого диапазона (листинг 8.1).

Листинг 8.1

```
let rangeInt = 1..<5
```

В данном примере создается диапазон целочисленных значений, включающий `1`, `2`, `3` и `4` (указанное конечное значение `5` исключается), после чего он инициализируется константе `rangeInt`.

Для создания диапазонов используются собственные типы данных (точно так же, как для создания строковых значений используются типы `String` и `Character`). Всего существует 4 типа, описывающих эти упорядоченные множества элементов.

При использовании бинарного оператора полуоткрытого диапазона создается значение типа `Range`, а если быть полностью точным, то `Range<Int>`, где `Int` определяет целочисленный характер элементов диапазона. Так, по аналогии значение типа `Range<Float>` будет описывать диапазон из чисел с плавающей точкой, а `Range<Character>` — из символов.

ПРИМЕЧАНИЕ Ранее мы не встречались с типами данных, содержащих в своих названиях угловые скобки. Данный способ описания типа говорит о том, что он является универсальным шаблоном (Generic). В одной из последних глав книги мы подробно рассмотрим их использование в разработке на Swift.

В общем случае благодаря универсальным шаблонам при создании типа (его реализации на Swift) есть возможность определить требования к типу, указываемому в скобках. Так, для типа `Range<T>` некий тип `T` должен быть `Comparable`, то есть сопоставимым.

Как и для значений других типов данных, при объявлении параметра вы можете явно и неявно задавать его тип (листинг 8.2).

Листинг 8.2

```
// задаем тип данных явно
var someRangeInt: Range<Int> = 1..<10
type(of:someRangeInt) // Range<Int>.Type
// тип данных определен автоматически
// на основании переданного значения (неявно)
var anotherRangeInt = 51..<59
type(of:anotherRangeInt) // Range<Int>.Type
var angeInt: Range<Int> = 1..<10
```

Как говорилось ранее, диапазон может содержать не только целочисленные значения, но и элементы других типов. В листинге 8.3 показаны примеры создания диапазонов с элементами типа `String`, `Character` и `Double`.

Листинг 8.3

```
// диапазон с элементами типа String
var rangeString = «a»..<»z»
type(of:rangeString) // Range<String>.Type

// диапазон с элементами типа Character
var rangeChar: Range<Character> = “a”..<”z”
type(of:rangeChar) // Range<Character>.Type

// диапазон с элементами типа Double
var rangeDouble = 1.0..<5.0
type(of:rangeDouble) // Range<Double>.Typevar rangeChar = "a"..<"z"
var anotherRangeChar: Range<Character> = "b"..<"x"
```

Возможно, вы спросите, с чем связано то, что при передаче `"a"..<"z"` устанавливается тип элементов `String`, хотя в них содержится всего один символ. Логично было бы предположить, что тип данных будет определен как `Character`. В главе про фундаментальные типы данных было рассказано, что при неявном определении типа Swift отдает предпочтение определенным типам (`String` вместо `Character`, `Double` вместо `Float`, `Int` вместо `Int8`). В данном случае происходит точно такая же ситуация: встречая операнд со значением `"a"`, Swift автоматически относит его к строкам, а не к символам.

Кстати, хочу отметить, что вы без каких-либо проблем сможете создать диапазон `"aa"..<"zz"`, где каждый элемент однозначно не `Character`.

В качестве начального и конечного значения в любых диапазонах можно использовать не только конкретные значения, но и параметры

(переменные и константы), которым эти значения инициализированы (листинг 8.3а).

Листинг 8.3а

```
var firstElement = 10
var lastElement = 18
var myBestRange = firstElement..<lastElement
```

Префиксная форма оператора

Данная форма оператора используется перед операндом, позволяющим определить правую границу диапазона.

СИНТАКСИС

```
..<правая_граница
```

- `правая_граница: Comparable` — элемент, следующий за последним элементом диапазона, заданный с помощью сопоставимого типа данных.

Диапазон элементов, определяющий только последний элемент диапазона (предшествует указанной правой границе). Левая граница диапазона заранее неизвестна. Так, в диапазоне `..<N` первый элемент будет не определен, а последний — `N-1`. Данный оператор используется в тех случаях, когда заранее неизвестен первый элемент, но необходимо ограничить диапазон справа.

Пример

```
var myRange = ..<500
```

В листинге 8.4 приведен пример использования данной формы оператора.

Листинг 8.4

```
var oneSideRange = ..<5
type(of: oneSideRange) // PartialRangeUpTo<Int>.Type
```

Тип данных созданного диапазона — `PartialRangeUpTo`, а точнее `PartialRangeUpTo<Int>`, где `Int` указывает на тип значений элементов интервала. Как и в случае с `Range`, данный диапазон может содержать значения и других типов данных. В общем случае тип данных диапазона, создаваемого с помощью постфиксной формы, — `PartialRangeUpTo<T>`, где `T` — *это сопоставимый* (`Comparable`) тип данных.

8.2. Оператор закрытого диапазона

Оператор закрытого диапазона обозначается в виде трех точек (`...`). Swift предлагает три формы: бинарная, префиксная (оператор расположен после операнда) и постфиксная.

Бинарная форма оператора

СИНТАКСИС

```
левая_граница...правая_граница
```

- `левая_граница: Comparable` — первый элемент диапазона, заданный с помощью сопоставимого типа данных.
- `правая_граница: Comparable` — последний элемент диапазона, заданный с помощью сопоставимого типа данных.

Диапазон элементов от левой границы до правой границы, включая концы. В диапазоне `1...N` первый элемент будет `1`, а последний — `N`. Начальное значение должно быть меньше или равно конечному. Попытка создать диапазон, например, от `5` до `2` приведет к ошибке.

Пример

```
var myRange = 1...100
```

В листинге 8.5 приведен пример использования данной формы оператора.

Листинг 8.5

```
var fullRange = 1...10
type(of: fullRange) // ClosedRange<Int>.Type
```

Тип данных диапазона, созданный бинарной формой оператора, — `ClosedRange<Int>`. Помимо `Int`, в качестве значений могут использоваться и другие типы данных. В общем случае тип данных диапазона, создаваемого с помощью бинарной формы, — `ClosedRange<T>`, где `T` — это сопоставимый (`Comparable`) тип данных.

Постфиксная форма оператора

Данная форма позволяет создать, по сути, бесконечный диапазон. Для этого необходимо указать только левую границу, опустив правую.

СИНТАКСИС

```
левая_граница...
```

- `правая_граница: Comparable` — элемент, следующий за последним элементом диапазона, заданный с помощью сопоставимого типа данных.

Диапазон элементов, определяющий только первый элемент диапазона. Правая граница диапазона заранее неизвестна. Таким образом, в диапазоне `1...` первый элемент будет `1`, а последний заранее не определен.

Пример

```
var myRange = 10...
```

Данный тип диапазона может быть использован, например, при работе со строками, когда вы хотите просмотреть все ее символы, начиная с первого, но при этом длина строки заранее неизвестна.

В листинге 8.6 приведен пример использования постфиксной формы оператора для создания диапазона.

Листинг 8.6

```
var infRange = 1...
type(of: infRange) // PartialRangeFrom<Int>.Type
```

Тип данных данного диапазона — `PartialRangeFrom<Int>`, где, как и в случае с предыдущими типами диапазона, вместо `Int` могут быть значения и других типов данных. В общем случае тип данных диапазона, создаваемого с помощью бинарной формы, — `PartialRangeFrom<T>`, где `T` — это сопоставимый (`Comparable`) тип данных.

Префиксная форма оператора

Данная форма, подобно префиксному полуоткрытому оператору, определяет только правую границу, но при этом включает ее в диапазон.

СИНТАКСИС

```
...правая_граница
```

- `правая_граница: Comparable` — последний элемент, заданный с помощью сопоставимого типа данных.

Диапазон элементов, определяющий только последний элемент диапазона. Левая граница диапазона заранее неизвестна. Таким образом, в диапазоне `...N` первый элемент будет не определен, а последний — `N`. Данный оператор используется в тех случаях, когда заранее неизвестен первый элемент, но необходимо ограничить диапазон справа.

Пример

```
var myRange = ...0
```

Тип данных диапазона, созданного с помощью постфиксного оператора, — `PartialRangeThrough<T>`, где `T` — это сопоставимый (`Comparable`) тип данных.

В будущих главах мы подробно разберем примеры использования рассмотренных операторов.

8.3. Базовые свойства и методы

При работе с диапазонами вы можете использовать большое количество встроенных функциональных возможностей, доступных «из

коробки». Разработчики языка позаботились о вас и реализовали все наиболее востребованные механизмы.

При работе с диапазоном, состоящим из целочисленных значений, можно использовать свойство `count` для определения количества элементов (листинг 8.7).

Листинг 8.7

```
var intR = 1...10
intR.count // 10
```

Для определения наличия элемента в диапазоне служит метод `contains(_:)` (листинг 8.8).

Листинг 8.8

```
var floatR: ClosedRange<Float> = 1.0...2.0
floatR.contains(1.4) // true
```

Для определения наличия элементов в диапазоне служит свойство `isEmpty`, возвращающее значение типа `Bool`. При этом обратите внимание, что создать пустой диапазон можно только в случае использования оператора полуоткрытого диапазона (`..<`) с указанием одинаковых операндов (листинг 8.9).

Листинг 8.9

```
// диапазон без элементов
var emptyR = 0..<0
emptyR.count   // 0
emptyR.isEmpty // true

// диапазон с единственным элементом - 0
var notEmptyR = 0...0
notEmptyR.count   // 1
notEmptyR.isEmpty // false
```

Свойства `lowerBound` и `upperBound` позволяют определить значения левой и правой границы, а методы `min()` и `max()` — минимальное и максимальное значения, правда, доступны они только при работе с целочисленными значениями (листинг 8.10).

Листинг 8.10

```
var anotherIntR = 20..<34
anotherIntR.lowerBound // 20
anotherIntR.upperBound // 34
anotherIntR.min()      // 20
anotherIntR.max()      // 33
```

9 Массивы (Array)

Массив — это один из наиболее важных представителей коллекции, который вы с большой долей вероятности будете использовать при реализации функционала любой программы. Уделите особое внимание приведенному в этой главе материалу.

9.1. Введение в массивы

Массив (Array) — это упорядоченная коллекция однотипных элементов, для доступа к которым используются целочисленные индексы. Упорядоченной называется коллекция, в которой элементы располагаются в порядке, определенном разработчиком.

Каждый элемент массива — это пара «индекс—значение».

Индекс элемента массива — это целочисленное значение, используемое для доступа к значениям элемента. Индексы генерируются автоматически при добавлении новых элементов. Индексы в массивах начинаются с нуля (не с единицы!). К примеру, у массива, содержащего 5 элементов, индекс первого равен 0, а последнего — 4. Индексы всегда последовательны и неразрывны. При удалении некоторого элемента, индексы всех последующих уменьшатся на единицу, чтобы обеспечить неразрывность.

Значение элемента массива — это произвольное значение определенного типа данных. Как говорилось ранее, значения доступны по соответствующим им индексам. Значения всех элементов массива должны быть одного и того же типа данных.

Создание массива с помощью литерала

Значение массива задается с помощью *литерала массива*, в котором через запятую перечисляются значения элементов.

СИНТАКСИС

```
[значение_1, значение_2, ..., значение_N]
```

- `значение: Any` — значение очередного элемента массива, может быть произвольного типа данных.

Литерал массива возвращает массив, состоящий из N элементов, значения которых имеют один и тот же тип данных. Литерал обрамляется квадратными скобками, а значения элементов в нем отделяются друг от друга запятыми. Массив может содержать любое количество элементов одного типа. Тип данных значений — произвольный, определяется вами в соответствии с контекстом задачи. Индексы элементов определяются автоматически в зависимости от порядка следования элементов.

Пример

```
[1,1,2,3,5,8,12]
```

В данном примере тип данных значения каждого элемента массива — `Int`. Массив имеет 7 элементов. Первые два (с индексами `0` и `1`) имеют одинаковые значения — `1`. Значение последнего элемента массива с индексом `6` равно `12`.

Массивы, как и любые другие значения, могут быть записаны в параметры. При использовании константы инициализированный массив является неизменяемым. Пример создания изменяемого и неизменяемого массива приведен в листинге 9.1.

Листинг 9.1

```
// неизменяемый массив
// с элементами типа String
let alphabetArray = ["a", "b", "c"]
// изменяемый массив
// с элементами типа Int
var mutableArray = [2, 4, 8]1
```

В данном листинге массивы с помощью литералов проинициализированы переменным `alphabetArray` и `mutableArray`. Неизменяемый массив `alphabetArray` предназначен для хранения значений типа `String`, а изменяемый массив `mutableArray` — для хранения элементов типа `Int`. Оба массива содержат по три элемента. Индексы соответствующих элементов обоих массивов имеют значения `0`, `1` и `2`.

ПРИМЕЧАНИЕ Отмечу, что массивом принято называть как само значение, представленное в виде литерала, так и параметр, в который это значение записано.

Это можно отнести также и к кортежам, словарям, наборам (их нам только предстоит изучить) и другим структурам языка программирования.

Создание массива с помощью Array(arrayLiteral:)

Для создания массива помимо передачи литерала массива можно использовать специальные глобальные функции, одной из которых является `Array(arrayLiteral:)`.

СИНТАКСИС

```
Array(arrayLiteral: значение_1, значение_2, ...,значение_N)
```

- `значение: Any` — значение очередного элемента массива, может быть произвольного типа данных.

Функция возвращает массив, состоящий из N элементов, значения которых имеют один и тот же тип данных. Значения произвольного типа передаются в виде списка в качестве входного параметра `arrayLiteral`. Каждое значение отделяется от последующего запятой.

Пример

```
Array(arrayLiteral: 1, 1, 2, 3, 5, 8, 12)
```

В данном примере в результате выполнения функции будет возвращен массив, состоящий из 7 целочисленных элементов.

Пример создания массива с использованием данной функции приведен в листинге 9.2.

Листинг 9.2

```
// создание массива с помощью передачи списка значений
let newAlphabetArray = Array(arrayLiteral: "a", "b", "c")
newAlphabetArray // ["a", "b", "c"]
```

В результате в переменной `newAlphabetArray` будет находиться массив строковых значений. Индекс первого элемента — `0`, а последнего — `2`.

ПРИМЕЧАНИЕ Возможно, вы обратили внимание на то, что, по логике, мы передали значение входного аргумента `arrayLiteral` как «a» и еще два безымянных аргумента. На самом деле все это значение одного аргумента `arrayLiteral`. О том, как Swift позволяет выполнять такой прием, будет рассказано в одной из следующих глав.

Создание массива с помощью Array(_:)

Также для создания массива можно использовать глобальную функцию `Array(_:)`, которой в качестве входного аргумента необходимо передать произвольную последовательность (`Sequence`).

СИНТАКСИС

```
Array(последовательность_значений)
```

- последовательность_значений: Sequence — последовательность элементов, которая будет преобразована в массив.

Функция возвращает массив, состоящий из элементов, входящих в переданную последовательность. Последовательность значений, переданная в функцию, может быть представлена в виде любой доступной в Swift последовательности (Sequence). Каждому ее элементу будет присвоен уникальный целочисленный индекс.

Пример

```
Array(0...10)
```

Диапазон `0...10` является последовательностью значений, а значит, может быть передан в `Array(_:)` для формирования массива. В результате выполнения функции будет возвращен массив, состоящий из 11 целочисленных элементов, входящих в диапазон `0...10`.

Мы познакомились лишь с двумя видами последовательностей (`Sequence`): диапазоны и массивы (не забывайте, что `Collection` — это расширенный `Sequence`). Рассмотрим пример использования диапазона для создания массива с помощью функции `Array(_:)`. Создадим массив, состоящий из 10 значений от 0 до 9 (листинг 9.3).

Листинг 9.3

```
// создание массива с помощью оператора диапазона
let lineArray = Array(0...9)
lineArray // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Каждый элемент последовательности `0...9` получит целочисленный индекс. В данном примере индекс будет совпадать со значением элемента: так, первый элемент будет иметь индекс `0` и значение `0`, а последний — индекс `9` и значение `9`.

ПРИМЕЧАНИЕ Обратите внимание, что `Array(_:)` в качестве входного параметра может принимать любую последовательность. Но будьте осторожны с бесконечными диапазонами (тип `PartialRangeFrom`), так как в этом случае операция создания массива не завершится никогда, программа займет всю свободную память, после чего зависнет.

Создание массива с помощью Array(repeating:count:)

Помимо рассмотренных способов, существует возможность создания массива с помощью функции `Array(repeating:count:)`, возвращающей массив, состоящий из указанного количества одинаковых (с одним и тем же значением) элементов.

СИНТАКСИС

```
Array(repeating: значение, count: количество)
```

- `значение: Any` — значение произвольного типа, которое будет повторяться в каждом элементе массива.
- `количество: Int` — целое число, определяющее количество повторений произвольного значения.

Функция возвращает массив, состоящий из одинаковых значений, повторяющихся `count` раз. Аргумент `repeating` определяет значение, которое будет присутствовать в массиве столько раз, сколько указано в качестве значения аргумента `count`.

Пример

```
Array(repeating: "Ура", count: 3)
```

В результате выполнения функции будет возвращен массив, состоящий из трех строковых элементов с одинаковым значением `"Ура"`. Индекс первого элемента будет равен `0`, а последнего — `2`.

Рассмотрим пример использования функции `Array(repeating:count:)` для создания массива (листинг 9.4).

Листинг 9.4

```
// создание массива с повторяющимися значениями
let repeatArray = Array(repeating:  "Swift", count: 5)
repeatArray // ["Swift", "Swift", "Swift", "Swift", "Swift"]
```

ПРИМЕЧАНИЕ Наверняка вы обратили внимание на то, что я как будто бы трижды описал одну и ту же функцию с именем `Array`. Да, действительно, имя во всех трех вариантах одно и то же, но оно отличается набором входных параметров — и для Swift это разные значения. Подробнее этот вопрос будет рассмотрен в ходе изучения функций.

Доступ к элементам массива

Синтаксис Swift позволяет использовать индексы для доступа к значениям элементов, которые указываются в квадратных скобках после массива (листинг 9.5).

Листинг 9.5

```
// неизменяемый массив
let alphabetArray = ["a", "b", "c"]
// изменяемый массив
var mutableArray = [2, 4, 8]
// доступ к элементам массивов
alphabetArray[1] // "b"
mutableArray[2] // 8
```

С помощью индексов можно получать доступ к элементам массива не только для чтения, но и для изменения значений (листинг 9.6).

Листинг 9.6

```
// изменяемый массив
var mutableArray = [2, 4, 8]
// изменение элемента массива
mutableArray[1] = 16
// вывод нового массива
mutableArray // [2, 16, 8]
```

ПРИМЕЧАНИЕ Попытка модификации массива, хранящегося в константе, вызовет ошибку.

При использовании оператора диапазона можно получить доступ сразу к множеству элементов в составе массива, то есть к его подмассиву. Данный оператор должен указывать на индексы крайних элементов выделяемого множества. В листинге 9.7 приведен пример замены двух элементов массива на один новый с помощью использования оператора диапазона (листинг 9.7).

Листинг 9.7

```
// изменяемый массив
var stringsArray = ["one", "two", "three", "four"]
// заменим несколько элементов
stringsArray[1...2] = ["five"]
stringsArray // ["one", "five", "four"]
stringsArray[2] // "four"
```

После замены двух элементов с индексами `1` и `2` на один со значением `"five"` индексы всех последующих элементов перестроились. Вследствие этого элемент `"four"`, изначально имевший индекс `3`, получил индекс `2`, так как стал третьим элементом массива.

ПРИМЕЧАНИЕ Индексы элементов массива всегда последовательно идут друг за другом без разрывов в значениях, при необходимости они перестраиваются.

9.2. Тип данных массива

Тип данных массива основан на типе данных значений его элементов. Существует полная и краткая формы записи типа данных массива.

СИНТАКСИС

Полная форма записи:

```
Array<T>
```

Краткая форма записи:

```
[T]
```

- `T: Any` — наименование произвольного типа данных значений элементов массива.

Массив с типом данных `Array<T>` должен состоять из элементов, значения которых имеют тип данных `T`. Обе представленные формы определяют один и тот же тип массива, указывая, что его элементы будут иметь значение типа `T`.

Рассмотрим пример из листинга 9.7а.

Листинг 9.7а

```
// Массив с типом данных [String] или Array<String>
var firstAr = Array(arrayLiteral: «a», «b», «c») // [«a», «b», «c»]
type(of: firstAr) // Array<String>.Type
// Массив с типом данных [Int] или Array<Int>
var secondAr = Array(1..<5) // [1, 2, 3, 4]
type(of: secondAr) // Array<Int>.Type
```

В данном листинге создаются два массива, тип данных которых определен *неявно* (на основании переданного значения).

Также тип массива может быть задан *явно*. В этом случае необходимо указать тип массива через двоеточие после имени параметра.

СИНТАКСИС

Полная форма записи:

```
var имяМассива: Array<T> = литерал_массива
```

Краткая форма записи:

```
var имяМассива: [T] = литерал_массива
```

В обоих случаях объявляется массив, элементы которого должны иметь указанный тип данных. Тип массива в этом случае будет равен `[T]` (с квадратными скобками) или `Array<T>`. Напомню, что оба обозначения эквивалентны. Типом каждого отдельного элемента таких массивов является `T` (без квадратных скобок).

Пример

```
var arrayOne: Array<Character> = ["a", "b", "c"]
var arrayTwo: [Int] = [1, 2, 5, 8, 11]
```

9.3. Массив — это value type

Массив является значимым типом (`value type`), а не ссылочным (`reference type`). Это означает, что при передаче значения массива из одного параметра в другой создается его копия, редактирование которой не влияет на исходную коллекцию.

В листинге 9.7б показан пример создания копии исходного массива с последующей заменой одного из его элементов.

Листинг 9.7б

```
// исходный массив
var parentArray = ["one", "two", "three"]
// создаем копию массива
var copyParentArray = parentArray
copyParentArray // ["one", "two", "three"]
// изменяем значение в копии массива
copyParentArray[1] = "four"
// выводим значение массивов
parentArray // ["one", "two", "three"]
copyParentArray // // ["one", "four", "three"]
```

При передаче исходного массива в новый параметр создается его полная копия. При изменении данной копии значение исходного массива остается прежним.

ПРИМЕЧАНИЕ Будьте внимательны, так как иногда можно случайно создать многочисленные копии одного и того же массива, что приведет к росту бесцельно используемой памяти.

9.4. Пустой массив

Массив может иметь пустое значение, то есть не иметь элементов (это словно строка без символов). Для создания пустого массива можно использовать один из следующих способов:

- явно указать тип создаваемого массива и передать ему значение `[]`;
- использовать специальную функцию `[типДанных]()`, где `типДанных` определяет тип значений элементов массива.

Оба способа создания пустого массива продемонстрированы в листинге 9.8.

Листинг 9.8

```
/* объявляем массив с пустым значением
  с помощью переданного значения */
var emptyArray: [String] = [] // []
 /* объявляем массив с пустым значением
  с помощью специальной функции */
var anotherEmptyArray = [String]() // []
```

В результате создаются два пустых массива, `emptyArray` и `anotherEmptyArray`, уже с инициализированными значениями (хотя и не содержащими элементов).

9.5. Операции с массивами

Сравнение массивов

Массивы, так же как и значения фундаментальных типов данных, можно сравнивать друг с другом. Два массива являются эквивалентными:

- если количество элементов в сравниваемых массивах одинаково;
- если каждая соответствующая пара элементов эквивалентна (имеет одни и те же типы данных и значения).

Рассмотрим пример сравнения двух массивов (листинг 9.9).

Листинг 9.9

```
/* три константы, которые
 станут элементами массива */
let a1 = 1
let a2 = 2
let a3 = 3
var someArray = [1, 2, 3]
someArray == [a1, a2, a3] // true
```

Несмотря на то что в массиве `[a1, a2, a3]` указаны не значения, а константы, содержащие эти значения, условия эквивалентности массивов все равно выполняются.

ПРИМЕЧАНИЕ Если в вашем коде есть строка `import Foundation` или `import UIKit` (одну из них Xcode добавляет автоматически при создании нового playground), то при попытке произвести сравнение массивов с помощью кода

```
someArray == [1,2,3]
```

может быть отражена ошибка. И для этого есть две причины:

- С помощью литерала массива, как вы узнаете далее, также могут быть созданы и другие виды коллекций.
- Директива `import` подключает внешнюю библиотеку функций, в которой существует не один оператор эквивалентности (==), а целое множество. Все они позволяют сравнивать разные виды коллекций.

В результате Swift не может определить, какая именно коллекция передана в правой части, и вызывает ошибку, сообщающую об этом.

Слияние массивов

Со значением массива, как и со значениями фундаментальных типов данных, можно проводить различные операции. Одной из них явля-

ется операция слияния, при которой значения двух массивов сливаются в одно, образуя новый массив. Обратите внимание на несколько моментов:

- результирующий массив будет содержать значения из обоих массивов, но индексы этих значений могут не совпадать с родительскими;
- значения элементов подлежащих слиянию массивов должны иметь один и тот же тип данных.

Операция слияния производится с помощью уже известного оператора сложения (`+`), как показано в листинге 9.10.

Листинг 9.10

```
// создаем три массива
let charsOne = ["a", "b", "c"]
let charsTwo = ["d", "e", "f"]
let charsThree = ["g", "h", "i"]
// создаем новый массив слиянием двух
var alphabet = charsOne + charsTwo
// сливаем новый массив с третьим
alphabet += charsThree
alphabet // ["a", "b", "c", "d", "e", "f", "g", "h", "i"]
```

Полученное в результате значение массива `alphabet` собрано из трех других массивов, причем порядок элементов соответствует порядку элементов в исходных массивах.

9.6. Многомерные массивы

Элементами массива могут быть значения не только фундаментальных типов, но и любых других типов данных, включая сами массивы. Массивы, элементами которых также являются массивы, называются многомерными. Необходимо обеспечить единство типа всех вложенных массивов.

Рассмотрим пример в листинге 9.11.

Листинг 9.11

```
var arrayOfArrays = [[1,2,3], [4,5,6], [7,8,9]]
```

В данном примере создается коллекция, содержащая множество массивов типа `[Int]` в качестве своих элементов. Типом основного массива `arrayOfArrays` является `[[Int]]` (с удвоенными квадратными скобками с каждой стороны) — это значит, что это *массив массивов*.

Для доступа к элементу многомерного массива необходимо указывать несколько индексов (листинг 9.12).

Листинг 9.12

```
arrayOfArrays = [[1,2,3], [4,5,6], [7,8,9]]
// получаем вложенный массив
arrayOfArrays[2] // [7, 8, 9]
// получаем элемент вложенного массива
arrayOfArrays[2][1] // 8
```

Конструкция `arrayOfArrays[2]` возвращает третий вложенный элемент массива `arrayOfArrays`, а `arrayOfArrays[2][1]`, использующая два индекса, возвращает второй элемент подмассива, содержащегося в третьем элементе массива `arrayOfArrays`.

9.7. Базовые свойства и методы массивов

Массивы — очень функциональные элементы языка. Об этом позаботились разработчики Swift, предоставив набор свойств и методов, позволяющих значительно расширить их возможности в сравнении с другими языками.

Свойство `count` возвращает количество элементов в массиве (листинг 9.13).

Листинг 9.13

```
var someArray = [1, 2, 3, 4, 5]
// количество элементов в массиве
someArray.count // 5
```

Если значение свойства `count` равно нулю, то и свойство `isEmpty` возвращает `true` (листинг 9.14).

Листинг 9.14

```
var emptyArray: [Int] = []
emptyArray.count // 0
emptyArray.isEmpty // true
```

Вы можете использовать свойство `count` для того, чтобы получить требуемые элементы массива (листинг 9.15).

Листинг 9.15

```
var numArray = [1, 2, 3, 4, 5]
// количество элементов в массиве
var sliceOfArray = numArray[numArray.count-3...numArray.count-1] //
[3, 4, 5]
```

Другим средством получить множество элементов массива является метод `suffix(_:)` — в качестве входного параметра ему передается количество элементов, которые необходимо получить. Элементы отсчитываются с последнего элемента массива (листинг 9.16).

Листинг 9.16

```
let subArray = numArray.suffix(3) // [3, 4, 5]
```

Свойства `first` и `last` возвращают первый и последний элементы массива (листинг 9.17).

Листинг 9.17

```
// возвращает первый элемент массива
numArray.first // 1
// возвращает последний элемент массива
numArray.last  // 5
```

С помощью метода `append(_:)` можно добавить новый элемент в конец массива (листинг 9.18).

Листинг 9.18

```
numArray // [1, 2, 3, 4, 5]
numArray.append(6) // [1, 2, 3, 4, 5, 6]
```

Если массив хранится в переменной (то есть является изменяемым), то метод `insert(_:at:)` вставляет в массив новый одиночный элемент с указанным индексом (листинг 9.19).

Листинг 9.19

```
numArray // [1, 2, 3, 4, 5, 6]
// вставляем новый элемент в середину массива
numArray.insert(100, at: 2) // [1, 2, 100, 3, 4, 5, 6]
```

При этом индексы массива пересчитываются, чтобы обеспечить их последовательность.

Так же как в случае изменения массива, методы `remove(at:)`, `removeFirst()` и `removeLast()` позволяют удалять требуемые элементы. При этом они возвращают значение удаляемого элемента (листинг 9.20).

Листинг 9.20

```
numArray // [1, 2, 100, 3, 4, 5, 6]
// удаляем третий элемент массива (с индексом 2)
numArray.remove(at: 2) // 100
// удаляем первый элемент массива
```

```
numArray.removeFirst() // 1
// удаляем последний элемент массива
numArray.removeLast() // 6
/* итоговый массив содержит
  всего четыре элемента */
numArray // [2, 3, 4, 5]
```

После удаления индексы оставшихся элементов массива перестраиваются. В данном случае в итоговом массиве `numArray` остается всего четыре элемента с индексами `0`, `1`, `2` и `3`.

Для редактирования массива также можно использовать методы `dropFirst(_:)` и `dropLast(_:)`, возвращающие новый массив, в котором отсутствует несколько первых или последних элементов, но при этом не изменяющие исходную коллекцию. Если в качестве входного аргумента ничего не передавать, то из результата удаляется один элемент, в противном случае — столько элементов, сколько передано (листинг 9.21).

Листинг 9.21

```
numArray // [2, 3, 4, 5]
// удаляем последний элемент
numArray.dropLast() // [2, 3, 4]
// удаляем три первых элемента
var anotherNumArray = numArray.dropFirst(3)
anotherNumArray // [5]
numArray // [2, 3, 4, 5]
```

При использовании данных методов основной массив `numArray`, с которым выполняются операции, не меняется. Они лишь возвращают получившееся значение, которое при необходимости может быть записано в новый параметр.

Метод `contains(_:)` определяет факт наличия некоторого элемента в массиве и возвращает `Bool` в зависимости от результата (листинг 9.22).

Листинг 9.22

```
numArray // [2, 3, 4, 5]
// проверка существования элемента
let resultTrue = numArray.contains(4)     //   true
let resultFalse = numArray.contains(10)   //   false
```

Для поиска минимального или максимального элемента в массиве применяются методы `min()` и `max()`. Данные методы работают только

в том случае, если элементы массива можно сравнить между собой (листинг 9.23).

Листинг 9.23

```
let randomArray = [3, 2, 4, 5, 6, 4, 7, 5, 6]
// поиск минимального элемента
randomArray.min() // 2
// поиск максимального элемента
randomArray.max() // 7
```

Чтобы изменить порядок следования всех элементов массива на противоположный, используйте метод `reverse()`, как показано в листинге 9.24.

Листинг 9.24

```
var myAlphaArray = ["a", "bb", "ccc"]
myAlphaArray.reverse()
myAlphaArray // ["ccc", "bb", "a"]
```

Методы `sort()` и `sorted()` позволяют отсортировать массив по возрастанию. Разница между ними состоит в том, что `sort()` сортирует саму последовательность, для которой он вызван, а `sorted()`, заменяя оригинальный массив, возвращает отсортированную коллекцию (листинг 9.24а).

Листинг 9.24а

```
// исходная неотсортированная коллекция
var unsortedArray = [3, 2, 5, 22, 8, 1, 29]
// метод sorted() возвращает отсортированную последовательность
// при этом исходный массив не изменяется
var sortedArray = unsortedArray.sorted()
unsortedArray // [3, 2, 5, 22, 8, 1, 29]
sortedArray // [1, 2, 3, 5, 8, 22, 29]

//метод sort() изменяет исходный массив
unsortedArray.sort()
unsortedArray // [1, 2, 3, 5, 8, 22, 29]
```

Способ сортировки по убыванию значений будет рассмотрен далее в книге, в главе 16.

ПРИМЕЧАНИЕ Разработчики Swift с умом подошли к именованию доступных методов. В большинстве случаев если какой-либо метод заканчивается на `–ed`, то он, не трогая исходное значение, возвращает его измененную копию. Аналогичный метод без `–ed` на конце модифицирует саму последовательность.

9.8. Срезы массивов (ArraySlice)

При использовании некоторых из описанных ранее свойств и методов возвращается не массив, а значение некоего типа данных `ArraySlice`. Это происходит, к примеру, при получении части массива с помощью оператора диапазона или при использовании методов `dropFirst()` и `dropLast()` (листинг 9.25).

Листинг 9.25

```
// исходный массив
var arrayOfNumbers = Array(1...10)
// его тип данных - Array<Int>
type(of: arrayOfNumbers) // Array<Int>.Type
arrayOfNumbers // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
// получим часть массива (подмассив)
var slice = arrayOfNumbers[4...6]
slice // [5, 6, 7]
// его тип данных отличается от типа исходного массива
type(of: slice) // ArraySlice<Int>.Type
```

Переменная `slice` имеет незнакомый вам тип данных `ArraySlice<Int>`. Если `Array` — это упорядоченное множество элементов, то `ArraySlice` — это его подмножество.

Но в чем необходимость создавать новый тип данных? Почему просто не возвращать значение типа `Array`?

Дело в том, что `ArraySlice` не копирует исходный массив, а ссылается на его подмножество (если быть точным, то ссылается на ту же самую область памяти). Это сделано для экономии ресурсов компьютера, так как не создаются излишние копии одних и тех же данных.

При работе с `ArraySlice` нужно быть предельно внимательными, тем более что Apple рекомендует максимально ограничить его использование. Дело в том, что если имеется `ArraySlice`, а вы удаляете параметр, хранящий исходный массив, то на самом деле незаметно для вас он все еще будет находиться в памяти, так как ссылка на его элементы все еще существует.

При работе с `ArraySlice` вам доступны те же возможности, что и при работе с массивом, но в большинстве случаев все же потребуется преобразовать `ArraySlice` в `Array`. Для этого можно использовать уже знакомую вам функцию `Array(_:)`, где в качестве входного параметра передается коллекция типа `ArraySlice` (листинг 9.26).

Листинг 9.26

```
type(of: slice) // ArraySlice<Int>.Type
var arrayFromSlice = Array(slice)
type(of: arrayFromSlice) // Array<Int>.Type
```

Стоит обратить внимание на то, что индексы ArraySlice соответствуют индексам исходной коллекции, то есть они не обязательно начинаются с 0 (листинг 9.27).

Листинг 9.27

```
// исходный массив
arrayOfNumbers // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
// его срез, полученные в одном из предыдущих листингов
slice // [5, 6, 7]
// отдельный элемент
arrayOfNumbers[5] // 6
slice[5] // 6
```

10 Наборы (Set)

Наборы наряду с массивами являются частью важнейшего механизма, позволяющего работать с множеством однотипных значений как с одним.

10.1. Введение в наборы

Набор (`Set`) — это неупорядоченная коллекция уникальных элементов. В отличие от массивов, у элементов набора нет четкого порядка следования, важен лишь факт наличия некоторого значения в наборе. Определенное значение элемента может существовать в нем лишь единожды, то есть каждое значение в пределах одного набора должно быть уникальным. Возможно, в русскоязычной документации по языку Swift вы встречали другое название наборов — *множества*.

Исходя из определения набора, ясно, что он позволяет собрать множество уникальных значений в пределах одного.

Представьте, что вы предложили друзьям совместный выезд на природу. Каждый из них должен взять с собой одно-два блюда. Вы получаете сообщение от первого товарища, что он возьмет хлеб и овощи. Второй друг готов привезти тушенку и воду. Все поступившие значения вы помещаете в отдельный набор, чтобы избежать дублирования блюд. В вашем наборе уже 4 элемента: «хлеб», «овощи», «тушенка» и «вода». Третий друг чуть позже сообщает, что готов взять мясо и овощи. Однако при попытке поместить в набор элемент «овощи» возникнет исключительная ситуация, поскольку данное значение уже присутствует в наборе. И правильно, зачем вам нужен второй набор овощей!

Варианты создания набора

Набор создается с помощью *литерала набора*. В плане синтаксиса он идентичен литералу массива, но при этом не должен содержать дублирующихся значений.

СИНТАКСИС

```
[значение_1, значение_2, ..., значение_N]
```

- `значение: Hashable` — значение очередного элемента набора, должно иметь хешируемый тип данных.

Литерал набора возвращает набор, состоящий из N элементов, значения которых имеют один и тот же тип данных. Литерал указывается в квадратных скобках, а значения отдельных элементов в нем разделяются запятыми. Литерал может содержать произвольное количество уникальных элементов одного типа.

ПРИМЕЧАНИЕ Несмотря на то что элементы набора являются неупорядоченными, Swift все же проводит внутреннюю работу по их сортировке и выстраиванию в последовательность. В ином случае у разработчика не было бы возможности последовательного перебора элементов набора.

Для упорядочивания у каждого значения высчитывается цифровой хеш (число, вычисляемое по специальному алгоритму). Для каждого возможного значения хеш всегда будет уникальным, так как он выступает своеобразным индексом, но при этом недоступным разработчику. Хеш используется только для внутренней сортировки значений.

В связи с этим не каждый тип данных может использоваться для элементов набора. Для того чтобы тип данных имел такую возможность, он должен соответствовать требованиям протокола `Hashable`. Все фундаментальные типы поддерживают этот протокол.

При создании набора необходимо явно указать, что создается именно набор. Если переменной передать литерал набора, то Swift распознает в нем литерал массива и вместо набора будет создан массив. По этой причине для создания набора необходимо использовать один из следующих способов.

- Явно указать тип данных набора с использованием конструкции `Set<T>`, где `T` указывает на тип значений элементов создаваемого набора, и передать литерал набора в качестве инициализируемого значения.

  Пример

  ```
  var mySet: Set<Int> = [1,5,0]
  ```

- Неявно задать тип данных с помощью конструкции `Set` и передать литерал набора *с элементами* в качестве инициализируемого значения.

  Пример

  ```
  var mySet: Set = [1,5,0]
  ```

- Использовать функцию `Set<T>(arrayLiteral:)`, явно указывающую на тип данных элементов массива, где `T` указывает на тип значений элементов создаваемого набора, а входной аргумент `arrayLiteral` содержит список элементов.

  Пример

  ```
  var mySet = Set<Int>(arrayLiteral: 5,66,12)
  ```

- Использовать функцию `Set(arrayLiteral:)`, где входной аргумент `arrayLiteral` содержит список элементов.

  Пример

  ```
  var mySet = Set(arrayLiteral: 5,66,12)
  ```

Тип данных набора — `Set<T>`, где `T` определяет тип данных элементов набора и должен быть хешируемым типом (`Hashable`).

ПРИМЕЧАНИЕ Для создания неизменяемого набора используйте оператор `let`, в ином случае — оператор `var`.

В листинге 10.1 продемонстрированы все доступные способы создания наборов.

Листинг 10.1

```
var dishes: Set<String> = ["хлеб", "овощи", "тушенка", "вода"]
var dishesTwo: Set = ["хлеб", "овощи", "тушенка", "вода"]
var members = Set<String>(arrayLiteral: "Энакин", "Оби Ван", "Йода")
var membersTwo = Set(arrayLiteral: "Энакин", "Оби Ван", "Йода")
```

В переменных `members`, `membersTwo`, `dishes`, `dishesTwo` хранятся наборы уникальных значений. При этом в области вывода порядок значений может не совпадать с определенным в литерале. Это связано с тем, что наборы — неупорядоченные коллекции элементов.

10.2. Пустой набор

Пустой набор, то есть набор, значение которого не имеет элементов (по аналогии с пустым массивом), создается с помощью пустого литерала набора `[]` либо вызова функции `Set<T>()` без входных параметров, где `T` определяет тип данных элементов массива. Вы также можете передать данный литерал с целью уничтожения всех элементов изменяемого набора (то есть в качестве хранилища используется переменная, а не константа). Пример приведен в листинге 10.2.

Листинг 10.2

```
// создание пустого набора
var emptySet = Set<String>()
// набор со значениями
var setWithValues: Set<String> = ["хлеб", "овощи"]
// удаление всех элементов набора
setWithValues = []
setWithValues // Set([])
```

10.3. Базовые свойства и методы наборов

Набор — это неупорядоченная коллекция, элементы которой не имеют индексов. Для взаимодействия с его элементами используются специальные методы.

Так, для создания нового элемента в наборе применяется метод `insert(_:)`, которому передается создаваемое значение. Обратите внимание, что оно должно соответствовать типу набора (листинг 10.3).

Листинг 10.3

```
// создаем пустой набор
var musicStyleSet: Set<String> = []
// добавляем к нему новый элемент
musicStyleSet.insert("Jazz") // (inserted true, memberAfterInsert
                             // "Jazz")
musicStyleSet  //{"Jazz"}
```

В результате выполнения метода `insert(_:)` возвращается кортеж, первый элемент которого содержит значение типа `Bool`, характеризующее успешность проведенной операции. Если возвращен `true` — элемент успешно добавлен, если `false` — он уже существует в наборе.

Для удаления элемента из набора используется метод `remove(_:)`, который уничтожает элемент с указанным значением и возвращает его значение или ключевое слово `nil`, если такого элемента не существует. Также вы можете задействовать метод `removeAll()` для удаления всех элементов набора (листинг 10.4).

Листинг 10.4

```
// создание набора со значениями
musicStyleSet = ["Jazz", "Hip-Hop", "Rock"]
// удаляем один из элементов
var removeStyleResult = musicStyleSet.remove("Hip-Hop")
removeStyleResult //"Hip-Hop"
musicStyleSet //{"Jazz", "Rock"}
// удаляем несуществующий элемент
```

```
musicStyleSet.remove("Classic") // nil
// удаляем все элементы набора
musicStyleSet.removeAll()
musicStyleSet  // Set([])
```

ПРИМЕЧАНИЕ В предыдущем примере вы впервые встретились с ключевым словом `nil`. С его помощью определяется полное отсутствие какого-либо значения. Его подробному изучению будет посвящена глава 14.

У вас мог возникнуть вопрос, почему в случае, если элемент не был удален, возвращается `nil`, а не `false`. Дело в том, что `false` само по себе является значением хешируемого типа `Bool`. Это значит, что набор вполне может иметь тип `Set<Bool>`. Если бы данный метод вернул `false`, то было бы логично утверждать, что из набора был удален элемент с именно таким значением.

В свою очередь, получив `nil`, можно однозначно сказать, что искомый элемент отсутствует в наборе.

Проверка факта наличия значения в наборе осуществляется методом `contains(_:)`, который возвращает значение типа `Bool` в зависимости от результата проверки (листинг 10.5).

Листинг 10.5

```
musicStyleSet = ["Jazz", "Hip-Hop", "Rock", "Funk"]
// проверка существования значения в наборе
musicStyleSet.contains("Funk") // true
musicStyleSet.contains("Pop") // false
```

Для определения количества элементов в наборе вы можете использовать свойство `count`, возвращающее целое число (листинг 10.5а).

Листинг 10.5а

```
musicStyleSet.count //4
```

Операции с наборами

Наборы подобны математическим множествам. Два или более набора могут содержать пересекающиеся и непересекающиеся между собой значения. `Swift` позволяет получать эти группы значений.

В листинге 10.6 создаются три различных целочисленных набора (рис. 10.1). Один из наборов содержит четные числа, второй — нечетные, третий — те и другие.

Листинг 10.6

```
// набор с четными цифрами
let evenDigits: Set = [0, 2, 4, 6, 8]
// набор с нечетными цифрами
```

```
let oddDigits: Set = [1, 3, 5, 7, 9]
// набор со смешанными цифрами
var differentDigits: Set = [3, 4, 7, 8]
```

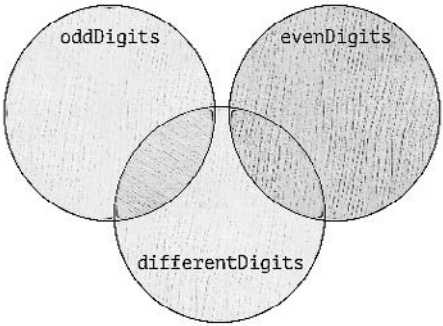

Рис. 10.1. Три набора целочисленных значений

В наборах `evenDigits`, `oddDigits` и `differentDigits` существуют как уникальные для каждого из них, так и общие элементы.

Для каждой пары наборов можно произвести следующие операции (рис. 10.2):

- получить все общие элементы (`intersection`);
- получить все непересекающиеся (не общие) элементы (`symmetricDifference`);
- получить все элементы обоих наборов (`union`);
- получить разницу элементов, то есть элементы, которые входят в первый набор, но не входят во второй (`subtracting`).

При использовании метода `intersection(_:)` возвращается набор, содержащий значения, общие для двух наборов (листинг 10.7).

Листинг 10.7

```
var inter = differentDigits.intersection(oddDigits)
inter // {3, 7}
```

Для получения всех непересекающихся значений служит метод `symmetricDifference(_:)`, представленный в листинге 10.8.

Листинг 10.8

```
var exclusive = differentDigits.symmetricDifference(oddDigits)
exclusive // {4, 8, 1, 5, 9}
```

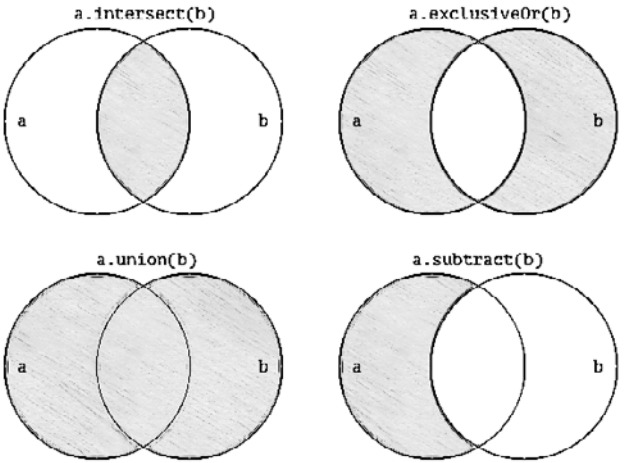

Рис. 10.2. Операции, проводимые с наборами

Для получения всех элементов из обоих наборов (их объединения) применяется объединяющий метод `union(_:)`, как показано в листинге 10.9.

Листинг 10.9

```
var union = evenDigits.union(oddDigits)
union // {8, 4, 2, 7, 3, 0, 6, 5, 9, 1}
```

Метод `subtracting(_:)` возвращает все элементы первого набора, которые не входят во второй набор (листинг 10.10).

Листинг 10.10

```
var subtract = differentDigits.subtracting(evenDigits)
subtract // {3, 7}
```

Отношения наборов

В листинге 10.10а созданы три набора: `aSet`, `bSet` и `cSet`. В них присутствуют как уникальные, так и общие элементы. Их графическое представление показано на рис. 10.3.

Листинг 10.10а

```
var aSet: Set = [1, 2, 3, 4, 5]
var bSet: Set = [1, 3]
var cSet: Set = [6, 7, 8, 9]
```

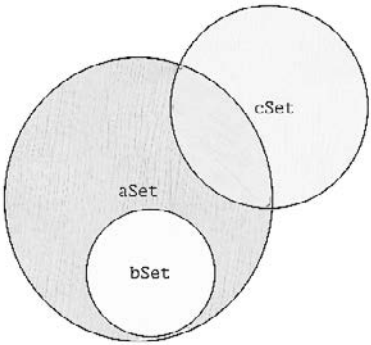

Рис. 10.3. Три набора значений с различными отношениями друг с другом

Набор `aSet` — это супернабор для набора `bSet`, так как включает в себя все элементы из `bSet`. В то же время набор `bSet` — это субнабор (или поднабор) для `aSet`, так как все элементы `bSet` существуют в `aSet`. Наборы `cSet` и `bSet` являются непересекающимися, так как у них нет общих элементов, а наборы `aSet` и `cSet` — пересекающиеся, так как имеют общие элементы.

Два набора считаются эквивалентными, если у них один и тот же набор элементов. Эквивалентность наборов проверяется с помощью оператора эквивалентности (`==`), как показано в листинге 10.11.

Листинг 10.11

```
// создаем копию набора
var copyOfBSet = bSet
/* в наборах bSet и copyOfBSet одинаковый состав
 элементов. Проверим их на эквивалентность */
bSet == copyOfBSet //true
```

Метод `isSubset(of:)` определяет, является ли один набор субнабором другого, как `bSet` для `aSet` (листинг 10.12).

Листинг 10.12

```
var aSet: Set = [1, 2, 3, 4, 5]
var bSet: Set = [1, 3]
bSet.isSubset(of: aSet) //true
```

Метод `isSuperset(of:)` вычисляет, является ли набор супернабором для другого набора (листинг 10.13).

Листинг 10.13

```
var aSet: Set = [1, 2, 3, 4, 5]
var bSet: Set = [1, 3]
aSet.isSuperset(of: bSet)  // true
```

Метод `isDisjoint(with:)` определяет, существуют ли в двух наборах общие элементы, и в случае их отсутствия возвращает `true` (листинг 10.14).

Листинг 10.14

```
bSet.isDisjoint(with: cSet) // true
```

Методы `isStrictSubset(of:)` и `isStrictSuperset(of:)` определяют, является набор субнабором или супернабором, не равным указанному множеству (листинг 10.15).

Листинг 10.15

```
bSet.isStrictSubset(of: aSet) // true
aSet.isStrictSuperset(of: bSet) // true var aSet: Set = [1, 2, 3, 4, 5]
```

С помощью уже знакомого метода `sorted()` вы можете отсортировать набор. При этом будет возвращен массив, в котором все элементы расположены по возрастанию (листинг 10.16).

Листинг 10.16

```
var setOfNums: Set = [1,10,2,5,12,23]
var sortedArray = setOfNums.sorted()
sortedArray // [1, 2, 5, 10, 12, 23]
type(of: sortedArray) // Array<Int>.Type
```

11 Словари (Dictionary)

Продолжая свое знакомство с коллекциями (`Collection`), мы разберем, что такое словари, для чего они предназначены, чем отличаются и что у них общего с другими видами коллекций, а также как их можно использовать при создании программ.

11.1. Введение в словари

Словарь — это неупорядоченная коллекция элементов, для доступа к значениям которых используются специальные индексы, называемые ключами. Каждый элемент словаря состоит из уникального ключа, указывающего на данный элемент, и значения. В качестве ключа выступает не автоматически генерируемый целочисленный индекс (как в массивах), а уникальное для словаря значение произвольного типа, определяемое программистом. Чаще всего в качестве ключей используются строковые или целочисленные значения. Все ключи словаря должны иметь единый тип данных. То же относится и к значениям.

ПРИМЕЧАНИЕ Уникальные ключи словарей не обязаны иметь тип `String` или `Int`. Чтобы какой-либо тип данных мог использоваться для ключей словаря, он должен быть хешируемым, то есть выполнять требования протокола `Hashable` (о нем мы говорили в одном из примечаний предыдущей главы).

Как и наборы, словари — это неупорядоченная коллекция. Это значит, что вы не можете повлиять на то, в каком порядке Swift расположит элементы.

Каждый элемент словаря — это пара «ключ—значение». Идея словарей в том, чтобы использовать уникальные произвольные ключи для доступа к значениям, при этом, как и в наборах, порядок следования элементов неважен.

Создание словаря с помощью литерала словаря

Значение словаря может быть задано с помощью **литерала словаря**.

СИНТАКСИС

```
[ключ_1:значение_1, ключ_2:значение_2, ..., ключ_N:значение_N]
```

- `ключ: Hashable` — ключ очередного элемента словаря, должен иметь хешируемый тип данных.
- `значение: Any` — значение очередного элемента словаря произвольного типа данных.

Литерал словаря возвращает словарь, состоящий из N элементов. Литерал обрамляется квадратными скобками, а указанные в нем элементы разделяются запятыми. Каждый элемент — это пара «ключ—значение», где ключ отделен от значения двоеточием. Все ключи между собой должны иметь один и тот же тип данных. Это относится также и ко всем значениям.

Пример

```
[200:"success",300:"warning",400:"error"]
```

Данный словарь состоит из трех элементов, ключи которых имеют тип `Int`, а значение — `String`. `Int` является хешируемым типом данных, то есть соответствует требованиям протокола `Hashable`.

ПРИМЕЧАНИЕ Для создания неизменяемого словаря используйте оператор `let`, в противном случае — оператор `var`.

Пример создания словаря приведен в листинге 11.1.

Листинг 11.1

```
var dictionary = ["one":"один", "two": "два", "three":"три"]
dictionary //["three": "один", "one": "два", "two": "три"]_
```

Словарь `dictionary` содержит три элемента. Здесь `"one"`, `"two"` и `"three"` — это ключи, которые позволяют получить доступ к значениям элементов словаря. Типом данных ключей, как и типом данных значений элементов словаря, является `String`.

При попытке создания словаря с двумя одинаковыми ключами Xcode сообщит об ошибке.

Создание словаря с помощью Dictionary(dictionaryLiteral:)

Другим вариантом создания словаря служит функция `Dictionary(dictionaryLiteral:)`, принимающая список кортежей, каждый из которых определяет пару «ключ—значение».

СИНТАКСИС

```
Dictionary(dictionaryLiteral: (ключ_1, значение_1), (ключ_2,
значение_2), ..., (ключ_N, значение_N))
```

- `ключ: Hashable` — ключ очередного элемента словаря, должен иметь хешируемый тип данных.
- `значение: Any` — значение очередного элемента словаря произвольного типа данных.

Функция возвращает словарь, состоящий из N элементов. Кортежи, каждый из которых состоит из двух элементов, передаются в виде списка в качестве значения входного параметра `dictionaryLiteral`. Первые элементы каждого кортежа (выше обозначены как `ключ_1`, `ключ_2` и т. д.) должны быть одного и того же типа данных. То же относится и ко вторым элементам каждого кортежа (выше обозначены как `значение_1`, `значение_2` и т. д.).

Каждый кортеж отделяется от последующего запятой. Все первые элементы кортежа (определяющие ключи) должны быть одного и того же типа данных. Это относится и ко вторым элементам (определяющим значения).

Пример

```
Dictionary(dictionaryLiteral: (100, "Сто"), (200, "Двести"),
                              (300, "Триста"))
```

Создание словаря с помощью Dictionary(uniqueKeysWithValues:)

Еще одним способом, который мы рассмотрим, будет использование функции `Dictionary(uniqueKeysWithValues:)`, позволяющей создать словарь на основе коллекции однотипных кортежей.

В листинге 11.2 приведен пример создания словаря из массива кортежей с помощью данной функции.

Листинг 11.2

```
// базовая коллекция кортежей (пар значений)
let baseCollection = [(2, 5), (3, 6), (1, 4)]
// создание словаря на основе базовой коллекции
let newDictionary = Dictionary(uniqueKeysWithValues: baseCollection)
newDictionary //[3: 6, 2: 5, 1: 4]
```

В функции `Dictionary(uniqueKeysWithValues:)` используется входной параметр `uniqueKeysWithValues`, которому передается коллекция пар значений. Результирующий словарь содержит в качестве ключей первый элемент каждой пары значений (каждого кортежа) базовой коллекции, а в качестве значений — второй элемент каждой пары значений.

Вся полезность данного способа проявляется тогда, когда вам необходимо сформировать словарь на основе двух произвольных последовательностей. В этом случае вы можете сформировать из них одну последовательность пар «ключ—значение» с помощью функции `zip(_:_:)` и передать ее в функцию `Dictionary(uniqueKeysWithValues:)` (листинг 11.3).

Листинг 11.3

```
// массив звезд
let nearestStarNames = ["Proxima Centauri", "Alpha Centauri A", "Alpha
Centauri B"]
// массив расстояний до звезд
let nearestStarDistances = [4.24, 4.37, 4.37]
// получение словаря, содержащего пары значений
let starDistanceDict = Dictionary(uniqueKeysWithValues:
zip(nearestStarNames, nearestStarDistances))
starDistanceDict // ["Proxima Centauri": 4.24, "Alpha Centauri B":
4.37, "Alpha Centauri A": 4.37]
```

Функция `zip(_:_:)` пока еще не была нами рассмотрена, мы вернемся к ней в одной из следующих глав. Суть ее работы состоит в том, что она возвращает последовательность пар значений, основанную на двух базовых последовательностях (в данном случае это `nearestStarNames` и `nearestStarDistances`). То есть она берет очередное значение каждой последовательности, объединяет их в кортеж и добавляет в результирующую последовательность в качестве элемента.

После этого сформированная последовательность передается входному аргументу `uniqueKeysWithValues`. В качестве ключей результирующий словарь будет содержать значения первой базовой коллекции (`nearestStarNames`), а в качестве значения — элементы второй базовой коллекции.

В повседневном программировании этот способ используется не так часто, поэтому запоминать его синтаксис не обязательно. Вам нужно лишь знать, что словарь может быть создан таким образом. При необходимости вы всегда сможете вернуться к книге или к официальной документации от Apple.

11.2. Тип данных словаря

Словари, как и другие виды коллекций, имеют обозначение собственного типа данных, у которых есть полная и краткая форма записи.

СИНТАКСИС

Полная форма записи:

```
Dictionary<T1,T2>
```

Краткая форма записи:

```
[T1:T2]
```

- `T1: Hashable` — наименование хешируемого типа данных ключей элементов словаря.
- `T2: Any` — наименование произвольного типа данных значений элементов словаря.

Словарь с типом данных `Dictionary<T1,T2>` или `[T1:T2]` должен состоять из элементов, ключи которых имеют тип данных `T1`, а значения `T2`. Обе представленные формы определяют один и тот же тип словаря.

Рассмотрим пример из листинга 11.4.

Листинг 11.4

```
// Словарь с типом данных [Int:String]
var codeDesc = [200: "success", 300: "warning", 400: "error"]
type(of: codeDesc) //Dictionary<Int, String>.Type
```

Тип данных словаря `codeDesc` задан неявно и определен на основании переданного ему значения. Тип задается единожды, и в будущем при взаимодействии с данной коллекцией необходимо его учитывать.

Помимо неявного определения тип словаря также может быть задан *явно*. В этом случае его необходимо указать через двоеточие после имени параметра.

СИНТАКСИС

Полная форма записи:

```
var имяСловаря: Dictionary<T1,T2> = литерал_словаря
```

Краткая форма записи:

```
var имяСловаря: [T1:T2] = литерал_словаря
```

В обоих случаях объявляется словарь, ключи элементов которого должны иметь тип `T1`, а значения — `T2`.

Пример

```
var dictOne: Dictionary<Int,Bool> = [100: false, 200: true, 400: true]
var dictTwo: [String:String] = ["Jonh":"Dave", "Eleonor":"Green"]
```

11.3. Взаимодействие с элементами словаря

Как отмечалось ранее, доступ к элементам словаря происходит с помощью уникальных ключей. Как и при работе с массивами, ключи предназначены не только для получения значений элементов словаря, но и для их изменения (листинг 11.5).

Листинг 11.5

```
var countryDict = ["RUS":"Россия", "BEL": "Беларусь", "UKR":"Украина"]
// получаем значение элемента
var countryName = countryDict["BEL"]
countryName // "Беларусь"
// изменяем значение элемента
countryDict["RUS"] = "Российская Федерация"
countryDict // ["RUS": "Российская Федерация", "BEL": "Беларусь",
"UKR": "Украина"]
```

В результате исполнения данного кода словарь `countryDict` получает новое значение для элемента с ключом `RUS`.

Изменение значения элемента словаря также может быть произведено с помощью метода `updateValue(_:forKey:)`. В случае, если изменяемый элемент отсутствует, будет возвращен `nil` (с ним мы уже встречались в предыдущей главе). В случае успешного изменения будет возвращено старое значение элемента.

Как показано в листинге 11.6, при установке нового значения данный метод возвращает старое значение или `nil`, если значения по переданному ключу не существует).

Листинг 11.6

```
var oldValueOne = countryDict.updateValue("Республика Белоруссия",
forKey: "BEL")
// в переменной записано старое измененное значение элемента
oldValueOne //"Беларусь"
var oldValueTwo = countryDict.updateValue("Эстония", forKey: "EST")
// в переменной записан nil, так как элемента с таким ключом
// не существует
oldValueTwo //nil
```

Для изменения значения в метод `updateValue` передаются новое значение элемента и параметр `forKey`, содержащий ключ изменяемого элемента.

Для того чтобы создать новый элемент в словаре, достаточно обратиться к несуществующему элементу и передать ему значение (листинг 11.7).

Листинг 11.7

```
countryDict["TUR"] = "Турция"
countryDict //["BEL": "Республика Белоруссия", "TUR": "Турция", "UKR":
"Украина", "EST": "Эстония", "RUS": "Российская Федерация"]
```

Для удаления элемента (пары «ключ—значение») достаточно присвоить удаляемому элементу `nil` или использовать метод `removeValue(forKey:)`, указав ключ элемента (листинг 11.8).

Листинг 11.8

```
countryDict["TUR"] = nil
countryDict.removeValue(forKey: "BEL")
countryDict //["RUS": "Российская Федерация", "UKR": "Украина", "EST":
"Эстония"]
```

При использовании метода `removeValue(forKey:)` возвращается значение удаляемого элемента.

ПРИМЕЧАНИЕ Есть один секрет, к которому вы, вероятно, пока не готовы, но о котором все же нужно сказать. Если вы попытаетесь получить доступ к несуществующему элементу словаря, это не приведет к ошибке. Swift просто вернет `nil`.

А это значит, что любое возвращаемое словарем значение — опционал, но познакомиться с этим понятием нам предстоит лишь в одной из будущих глав.

```
let someDict = [1:"one", 3:"three"]
someDict[2] // nil
type(of: someDict[2]) //Optional<String>.Type
```

11.4. Пустой словарь

Пустой словарь не содержит элементов (как и пустые набор и массив). Для того чтобы создать пустой словарь, необходимо использовать литерал без элементов. Для этого предназначена конструкция `[:]` или функция `Dictionary<типКлючей:типЗначений>()` без входных аргументов (листинг 11.9).

Листинг 11.9

```
var emptyDictionary: [String:Int] = [:]
var anotherEmptyDictionary = Dictionary<String,Int>()
```

С помощью конструкции `[:]` также можно уничтожить все элементы словаря, если проинициализировать ее словарю в качестве значения (листинг 11.10).

Листинг 11.10

```
var birthYears = [1991: ["John", "Ann", "Vasiliy"], 1993: ["Alex",
"Boris"] ]
birthYears = [:]
birthYears  //[:]
```

Обратите внимание, что в качестве значения каждого элемента словаря в данном примере используется массив с типом `[String]`. В результате тип самого словаря `birthYears` будет `[Int:[String]]`.

Вы совершенно не ограничены в том, значения каких типов использовать для вашей коллекции.

11.5. Базовые свойства и методы словарей

Словари, как и массивы с наборами, имеют большое количество свойств и методов, наиболее важные из которых будут рассмотрены в этом разделе.

Свойство `count` возвращает количество элементов в словаре (листинг 11.11).

Листинг 11.11

```
var someDictionary = ["One":1, "Two":2, "Three":3]
// количество элементов в словаре
someDictionary.count // 3
```

Если свойство `count` равно нулю, то свойство `isEmpty` возвращает `true` (листинг 11.12).

Листинг 11.12

```
var emptyDict: [String:Int] = [:]
emptyDict.count //0
emptyDict.isEmpty //true
```

При необходимости вы можете получить все ключи или все значения словаря с помощью свойств `keys` и `values` (листинг 11.13).

Листинг 11.13

```
// все ключи словаря countryDict
var keys = countryDict.keys
type(of: keys) // Dictionary<String, String>.Keys.Type
keys // Dictionary.Keys(["UKR", "RUS", "EST"])

// все значения словаря countryDict
var values = countryDict.values
```

```
type(of: values) //Dictionary<String, String>.Values.Type
values //Dictionary.Values(["Украина", "Эстония", "Российская
Федерация"])
```

При обращении к свойствам `keys` или `values` Swift возвращает не массив или набор, а значение специального типа данных `Dictionary<ТипКлюча,ТипЗначения>.Keys` и `Dictionary<ТипКлюча,ТипЗначения>.Values`.

Не пугайтесь столь сложной записи. Как неоднократно говорилось, Swift используется огромное количество различных типов, у каждого из которых свое предназначение. В данном случае указанные типы служат для доступа к ключам или значениям исходного словаря. При этом они являются полноценными коллекциями (соответствуют требованиям протокола `Collection`), а значит, могут быть преобразованы в массив или набор (листинг 11.14).

Листинг 11.14

```
var keysSet = Set(keys)
keysSet //{"UKR", "RUS", "EST"}
var valuesArray = Array(values)
valuesArray //["Эстония", "Украина", "Российская Федерация"]
```

11.6. О вложенных типах данных

Вернемся еще раз к новому для вас обозначению типов данных `Dictionary<T1,T2>.Keys` и `Dictionary<T1,T2>.Values`. Обратите внимание, что `Keys` и `Values` пишутся через точку после типа словаря. Это говорит о том, что типы данных `Keys` и `Values` реализованы внутри типа `Dictionary<T1,T2>`, то есть они не существуют отдельно от словаря и могут быть использованы только в его контексте. Таким образом, вы не можете создать параметр типа `Values` (`var a: Values = ...` не будет работать), так как глобально такого типа нет. Он существует только в контексте какого-либо словаря.

Это связано с тем, что вам, в принципе, никогда не понадобятся значения этих типов отдельно от родительского словаря. При этом Swift не знает, в каком виде вы хотели бы получить эти значения, а значит, возвращать готовый массив или набор было бы бесцельной тратой ресурсов.

В результате вы получаете значение типа `Dictionary<T1,T2>.Keys` и `Dictionary<T1,T2>.Values` и спокойно обрабатываете его (преобразовываете в другой вид коллекции или проходитесь по его элементам).

Не переживайте, если данный материал оказался для вас сложным. Вероятно, вы еще не готовы полностью воспринять его. В будущем, углубляясь в разработку на Swift и в создание собственных типов, вы сможете вернуться к этому описанию. Сейчас вам важно запомнить лишь то, что свойства `keys` и `values` возвращают коллекции элементов, которые могут быть преобразованы в массив или набор.

12 Строка — коллекция символов (String)

В главе 5 книги мы познакомились со строковыми типами `String` и `Character`, позволяющими работать с текстовыми данными в приложениях, и получили первые и основные знания о стандарте Unicode. Напомню, что строки состоят из отдельных символов, а для каждого символа существует кодовая точка (уникальная последовательность чисел из стандарта Unicode, соответствующая данному символу). Кодовые точки могут быть использованы для инициализации текстовых данных в составе юникод-скаляров (служебных конструкций `\n{}`). В этой главе будет подробно рассказано о том, как функционируют строковые типы в Swift.

12.1. Character в составе String

В самом простом смысле, концептуально, строка в Swift — это коллекция. Да, именно коллекция! `String` соответствует требованиям протокола `Collection` и является коллекцией, подобной массивам, наборам и словарям, но со своими особенностями, которые мы сейчас обсудим.

Во-первых, так как `String` — коллекция, то вам доступно большинство возможностей, характерных для коллекций. К примеру, можно получить количество всех элементов с помощью свойства `count` (листинг 12.1) или осуществить их перебор с помощью оператора `for-in` (с ним мы познакомимся несколько позже).

Листинг 12.1

```
var str = "Hello!"
str.count // 6
```

Переменная `str` имеет строковое значение, состоящее из `6` символов, что видно, в том числе, по выводу свойства `count`. Возможность ис-

пользования данного свойства вы могли видеть и у других видов коллекций.

Во-вторых, раз значение типа `String` — это коллекция, то возникает вопрос: чем являются элементы этой коллекции?

Каждый элемент строкового значения типа `String` представляет собой значение типа `Character`, то есть отдельный символ, который может быть представлен с помощью юникод-скаляра (конструкции `\u{}`, включающей кодовую точку).

В-третьих, значение типа `String` — это упорядоченная коллекция. Элементы в ней находятся именно в том порядке, который определил разработчик при инициализации значения.

На данный момент можно сказать, что строка (значение типа `String`) — это упорядоченная коллекция, каждый элемент которой представляет собой значение типа `Character`.

Так как `String` является упорядоченной коллекцией, было бы правильно предположить, что ее элементы коллекции имеют индексы, по которым они сортируются и выстраиваются в последовательность, а также по которым к этим элементам можно обратиться (для чтения, изменения или удаления). И правда, довольно часто возникает задача получить определенный символ в строке. Если у вас есть опыт разработки на других языках программирования, то, возможно, вам в голову пришла идея попробовать использовать целочисленные индексы для доступа к элементам строки (точно как в массивах). Но, неожиданно, в Swift такой подход приведет к ошибке (листинг 12.2).

Листинг 12.2

```
str[2] // error: 'subscript' is unavailable: cannot subscript String
with an Int
```

Но в чем проблема? Почему такой простой вариант доступа не может быть использован в современном языке программирования? Чтобы ответить на этот вопрос, нам потребуется вновь поговорить о стандарте Unicode и о работе с ним в Swift.

12.2. Графем-кластеры

Значение типа `String` — это коллекция, каждый элемент которой представлен индексом (являющимся значением пока еще не рассмотренного типа данных) и значением типа `Character`. Как мы видели ранее, каждый отдельный символ может быть представлен в виде юникод-

скаляра. В листинге 12.3 параметру типа `Character` инициализируется значение через юникод-скаляр.

Листинг 12.3

```
var char: Character = "\u{E9}"
char // "é"
```

Символ `é` (латинская e со знаком ударения) представлен в данном примере с использованием кодовой точки `E9` (или `233` в десятеричной системе счисления). Но удивительным становится тот факт, что существует и другой способ написания данного символа: с использованием двух юникод-скаляров (а соответственно и двух кодовых точек). Первый будет описывать латинскую букву `e` (`\u{65}`), а второй символ ударения (`\u{301}`) (листинг 12.4).

Листинг 12.4

```
var anotherChar: Character = "\u{65}\u{301}"
anotherChar // "é"
```

И так как тип данных этих параметров — `Character`, ясно, что в обоих случаях для Swift значение состоит из одного символа. Если провести сравнение значений этих переменных, то в результате будет возвращено `true`, что говорит об их идентичности (листинг 12.5).

Листинг 12.5

```
char == anotherChar //true
```

Выглядит очень странно, согласны? Но дело в том, что в переменной `anotherChar` содержится комбинированный символ, состоящий из двух кодовых точек, но по сути являющийся одним полноценным.

Существование таких комбинированных символов становится возможным благодаря специальным символам, модифицирующим предыдущий по отношению к ним символ (как знак ударения в листинге выше, он изменяет отображение латинской буквы `e`).

В связи с этим при работе со строковыми значениями не всегда корректным будет говорить именно о символах, так как мы видели ранее, что символ, по своей сути, сам может состоять из нескольких символов. В этом случае лучше обратиться к понятию графем-кластера.

Графем-кластер — это совокупность юникод-скаляров (или кодовых точек), при визуальном представлении выглядящих как один символ. Графем-кластер может состоять из одного или двух юникод-скаляров.

Таким образом в будущем, говоря о значении типа `Character`, мы будем подразумевать не просто отдельный символ, а графем-кластер.

Графем-кластеры могут определять не только буквы алфавита, но и эмодзи. В листинге 12.6 приведен пример комбинирования символов «Thumbs up sign» (кодовая точка — 1f44d) и «Emoji Modifier Fitzpatrick Type-4» (кодовая точка — 1f3fd) в единый графем-кластер для вывода нового эмодзи (палец вверх со средиземноморским цветом кожи).

Листинг 12.6

```
var thumbsUp = "\u{1f44d}"          // "👍"
var blackSkin = "\u{1f3fd}"         // "🏽"
var combine = "\u{1f44d}\u{1f3fd}" // "👍🏽"
```

ПРИМЕЧАНИЕ Каждый символ помимо кодовой точки также имеет уникальное название. Эти данные при необходимости можно найти в таблицах юникод-символов в Интернете.

Вернемся к примеру с символом `é`. В листинге 12.7 создаются две строки, содержащие данный символ, первая из них содержит непосредственно сам символ `é`, а вторая — комбинацию из латинской `e` и знака ударения.

Листинг 12.7

```
let cafeSimple = "caf\u{E9}" //"café"
let cafeCombine = "cafe\u{301}" //"café"
cafeSimple.count // 4
cafeCombine.count // 4
```

Как видно из данного примера, несмотря на то что в переменной `cafeCombine` пять символов, свойство `count` для обоих вариантов возвращает значение `4`. Это связано с тем, что для Swift строка — это не просто коллекция символов, а коллекция графем-кластеров. Стоит отметить, что время, необходимое для выполнения подсчета количества элементов в строке, растет линейно с увеличением количества этих элементов (букв). Причина заключается в том, что компьютер не может заранее знать, сколько графем-кластеров в коллекции, для этого ему необходимо полностью обойти строку с первого до последнего символа.

Графем-кластеры являются одновременно огромным плюсом стандарта Unicode, а также причиной отсутствия в Swift доступа к отдельным символам через целочисленный индекс. Вы не можете просто взять

третий или десятый символ, так как нет никакой гарантии, что он окажется полноценным графем-кластером, а не отдельным символом в его составе. Для доступа к любому элементу коллекции типа `String` необходимо пройти через все предыдущие элементы. Только в этом случае можно однозначно получить корректный графем-кластер.

Тем не менее строки — это упорядоченные коллекции, а значит, в составе каждого элемента присутствует не только значение типа `Character`, но и индекс, позволяющий однозначно определить положение этого элемента в коллекции и получить к нему доступ. Именно об этом пойдет разговор в следующем разделе.

12.3. Строковые индексы

Почему Swift не позволяет использовать целочисленные индексы для доступа к отдельным графем-кластерам в строке? На самом деле для разработчиков этого языка не составило бы никакого труда слегка расширить тип `String` и добавить соответствующий функционал. Это же программирование, тут возможно все! Но они лишили нас такой возможности сознательно.

Дело в том, что Swift, в лице его разработчиков, а также сообщества, хотел бы, чтобы каждый из нас понимал больше, чем требуется для «тупого» набивания кода, чтобы мы знали, как работает технология «под капотом». Скажите честно, пришло бы вам в голову разбираться со стандартом Юникод, кодовыми точками, кодировками и графем-кластерами, если бы вы могли просто получить символ по его порядковому номеру?

Хотя лучше оставим этот вопрос без ответа и вернемся к строковым индексам.

Значение типа `String` имеет несколько свойств, позволяющих получить индекс определенных элементов. Первым из них является `startIndex`, возвращающий индекс первого элемента строки (листинг 12.8).

Листинг 12.8

```
let name = "e\u{301}lastic" //"élastic"
var index = name.startIndex
```

Обратите внимание, что первый графем-кластер в константе `name` состоит из двух символов. Свойство `startIndex` возвращает индекс, по которому можно получить именно графем-кластер, а не первый сим-

вол в составе графем-кластера. Теперь в переменной `index` хранится индекс первой буквы, и его можно использовать точно так же, как индексы других коллекций (листинг 12.9).

Листинг 12.9

```
let firstChar = name[index]
firstChar // "é"
type(of: firstChar) //Character.Type
```

В данном листинге был получен первый символ строки, хранящейся в константе `name`. Обратите внимание, что тип полученного значения — `Character`, что соответствует определению строки (это коллекция `Character`). Но вопрос в том, а что такое строковый индекс, какой тип данных он имеет (листинг 12.10)?

Листинг 12.10

```
type(of: index) //String.Index.Type
```

Тип строкового индекса — `String.Index`, значит, что это тип данных `Index`, вложенный в `String`. С вложенными типами мы встречались ранее во время изучения словарей (`Dictionary`). Значение типа `String.Index` определяет положение графем-кластера внутри строкового значения, то есть содержит ссылки на область памяти, где он начинается и заканчивается. И это, конечно же, вовсе не значение типа `Int`.

Помимо `startIndex`, вам доступно свойство `endIndex`, позволяющее получить индекс, *который следует за последним символом в строке*. Таким образом, он указывает не на последний символ, а за него, туда, куда будет добавлен новый графем-кластер (то есть добавлена новая буква, если она, конечно, будет добавлена). Если вы попытаетесь использовать значение свойства `endIndex` напрямую, то Swift сообщит о критической ошибке (листинг 12.11).

Листинг 12.11

```
var indexLastChar = name.endIndex
name[indexLastChar] //Fatal error: String index is out of bounds
```

Метод `index(before:)` позволяет получить индекс символа, предшествующего тому, индекс которого передан во входном аргументе `before`. Другими словами, передавая в `before` индекс символа, на выходе вы получите индекс предшествующего ему символа. Вызывая данный метод, в него можно, к примеру, передать значение свойства `endIndex` для получения последнего символа в строке (листинг 12.12).

Листинг 12.12

```
var lastChar = name.index(before: indexLastChar)
name[lastChar] //"c"
```

Метод `index(after:)` позволяет получить индекс последующего символа (листинг 12.13).

Листинг 12.13

```
var secondCharIndex = name.index(after: name.startIndex)
name[secondCharIndex] // "l"
```

Метод `index(_:offsetBy:)` позволяет получить требуемый символ с учетом отступа. В качестве значения первого аргумента передается индекс графем-кластера, от которого будет происходить отсчет, а в качестве значения входного параметра `offsetBy` передается целое число, указывающее на отступ вправо (листинг 12.14).

Листинг 12.14

```
var fourCharIndex = name.index(name.startIndex, offsetBy:3)
name[fourCharIndex] // "s"
```

При изучении возможностей Swift по работе со строками вам очень поможет окно автодополнения, в котором будут показаны все доступные методы и свойства значения типа `String` (рис. 12.1).

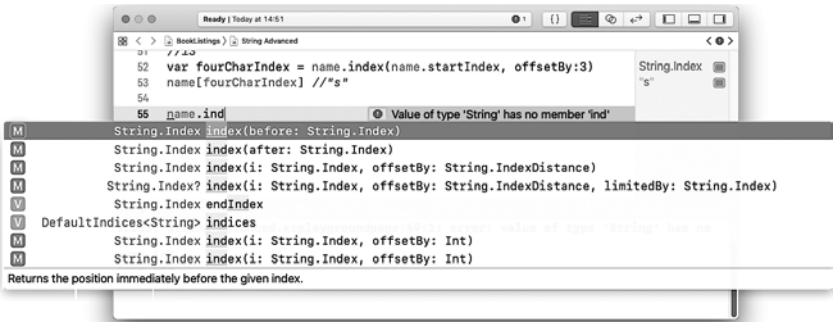

Рис. 12.1. Окно автодополнения в качестве краткой справки

Отмечу еще одно свойство, которое, возможно, понадобится вам в будущем. С помощью `unicodeScalars` можно получить доступ к коллекции юникод-скаляров, из которых состоит строка. Данная коллекция содержит не графем-кластеры, а именно юникод-скаляры с обозначением кодовых точек каждого символа строки. В листинге 12.15 по-

казано, что количество элементов строки и значения, возвращенного свойством `uncodeScalars`, отличается, так как в составе строки есть сложный графем-кластер (состоящий из двух символов).

Листинг 12.15

```
name.count // 7
name.unicodeScalars.count // 8
```

Обратите внимание, что в данном примере впервые в одном выражении использована цепочка вызовов, когда несколько методов или свойств вызываются последовательно в одном выражении (`name.unicodeScalars.count`).

Цепочка вызовов — очень полезный функциональный механизм Swift. С ее помощью можно не записывать возвращаемое значение в параметр для последующего вызова очередного свойства или метода.

В результате вызова свойства `unicodeScalars` возвращается коллекция, а значит, у нее есть свойство `count`, которое тут же может быть вызвано.

Суть работы цепочек вызовов заключается в том, что если какая-либо функция, метод или свойство возвращают объект, у которого есть свои свойства или методы, то их можно вызывать в том же самом выражении. Длина цепочек вызова (количество вызываемых свойств и методов) не ограничена. В следующих главах вы будете все чаще использовать эту прекрасную возможность.

12.4. Подстроки (Substring)

Четвертая версия Swift порадовала разработчиков новым типом данных `Substring`, описывающим подстроку некоторой строки. `Substring` для `String` — это как `ArraySlice` для `Array`. В ранних версиях языка, получая подстроку, возвращалась новая строка (значение типа `String`), для которой выделялась отдельная область памяти. Теперь при получении подстроки возвращается значение типа `Substring`, ссылающееся на ту же область памяти, что и оригинальная строка.

Другими словами, основной целью создания типа `Substring` была оптимизация. Значение типа `Substring` делит одну область памяти с родительской строкой, то есть для нее не выделяется дополнительная память.

Для получения необходимой подстроки можно воспользоваться операторами диапазона (листинг 12.16).

Листинг 12.16

```
var abc = "abcdefghijklmnopqrstuvwxyz"
// индекс первого символа
var firstCharIndex = abc.startIndex
// индекс четвертого символа
var fourthCharIndex = abc.index(firstCharIndex, offsetBy:3)
// получим подстроку
var subAbc = abc[firstCharIndex...fourthCharIndex]
subAbc // "abcd"
type(of: subAbc) // Substring.Type
```

В результате выполнения кода в переменной `subAbc` будет находиться значение типа `Substring`, включающее в себя первые четыре символа строки `abc`.

Подстроки обладают тем же функционалом, что и строки. Но при необходимости вы всегда можете использовать функцию `String(_:)` для преобразования подстроки в строку (листинг 12.17).

Листинг 12.17

```
type( of: String(subAbc) ) //String.Type
```

Также хотелось бы показать пример использования полуоткрытого оператора диапазона для получения подстроки, состоящей из всех символов, начиная с четвертого и до конца строки. При этом совершенно неважно, какого размера строка, вы всегда получите все символы до ее конца (листинг 12.18).

Листинг 12.18

```
var subStr = abc[fourthCharIndex...]
subStr //"defghijklmnopqrstuvwxyz"
```

На этом наше знакомство со строками закончено. Уверен, что оно было интересным и познавательным. Советую самостоятельно познакомиться со стандартом Unicode в отрыве от изучения Swift. Это позволит еще глубже понять принципы его работы и взаимодействия с языком.

Часть IV. Основные возможности Swift

Вы все ближе и ближе к созданию собственных удивительных приложений, которые совершенно точно покорят мир! Вы уже знаете многое о фундаментальных и контейнерных типах данных, а также о типах данных. И конечно же, возможности языка не оканчиваются на этом. Сейчас вы в самом начале долгого и интересного пути. Начиная с этой части книги, вы будете знакомиться с функциональными возможностями, обеспечивающими не просто хранение данных, а их обработку. И уже совсем скоро напишете свое первое приложение.

13 Операторы управления

Любому разработчику требуются механизмы, позволяющие определять логику работы программы в различных ситуациях. К примеру, при использовании калькулятора необходимо выполнять различные арифметические операции в зависимости от нажатых кнопок.

В программировании для этого используются **операторы управления ходом выполнения** программы: представьте — вы можете останавливать работу, многократно выполнять отдельные блоки кода или игнорировать их в зависимости от возникающих условий.

Умение управлять ходом работы программы — очень важный аспект программирования на языке Swift, благодаря которому вы сможете написать сложные функциональные приложения. В данной главе рассмотрены механизмы, присутствующие практически во всех языках и без которых невозможно создание любой, даже самой простой программы.

Операторы управления можно разделить на две группы:

1. **Операторы ветвления**, определяющие порядок и необходимость выполнения блоков кода.
2. **Операторы повторения**, позволяющие многократно выполнять блоки кода.

В качестве входных данных операторам передается выражение, вычисляя значение которого они принимают решение о порядке выполнения блоков кода (игнорировать, однократно или многократно выполнять).

Выделяют следующие конструкции языка, позволяющие управлять ходом выполнения программы:

- Утверждение (глобальная функция `assert(_:_:)`).
- Оператор условия `if`.

- Оператор ветвления `switch`.
- Операторы повторения `while` и `repeat while`.
- Оператор повторения `for`.
- Оператор раннего выхода `guard`.

Помимо этого, существуют специальные операторы, позволяющие влиять на работу описанных выше конструкций, к примеру досрочно прерывать их работу, пропускать итерации и другие функции.

13.1. Утверждения

Swift позволяет прервать выполнение программы в случае, когда некоторое условие не выполняется: к примеру, если значение переменной отличается от требуемого. Для этого предназначен специальный механизм **утверждений** (`assertions`).

Утверждения в Swift реализованы в виде глобальной функции `assert(_:_:)`.

СИНТАКСИС

```
assert(проверяемое_выражение, отладочное_сообщение)
```

- `проверяемое_выражение -> Bool` — вычисляемое выражение, на основании значения которого принимается решение об экстренной остановке программы.
- `отладочное_сообщение -> String` — выражение, текстовое значение которого будет выведено на отладочную консоль при остановке программы. Необязательный параметр.

Функция «утверждает», что переданное ей выражение возвращает логическое значение `true`. В этом случае выполнение программы продолжается, в ином (если возвращен `false`) — выполнение программы завершается, и в отладочную консоль выводится сообщение.

Пример

```
// утверждение с двумя аргументами
assert( someVar > 100, "Данные неверны" )
// утверждение с одним аргументом
assert( anotherVar <= 10 )
```

ПРИМЕЧАНИЕ В синтаксисе выше впервые применена стрелка (`->`) для указания на тип данных условного элемента синтаксиса. Как говорилось ранее, она используется в том случае, когда можно передать не конкретное значение определенного типа, а целое выражение.

В листинге 13.1 показан пример использования утверждений.

Листинг 13.1

```
let strName = "Дракон"
let strYoung = "молод"
let strOld = "стар"
let strEmpty = " "

var dragonAge = 230
assert( dragonAge <= 235, strName+strEmpty+strOld )
assert( dragonAge >= 225, strName+strEmpty+strYoung )
print("Программа упешно завершила свою работу")
```

Консоль

```
Программа упешно завершила свою работу
```

В приведенном примере проверяется возраст дракона, записанный в переменную `dragonAge`. С помощью двух утверждений программа успешно завершит работу, только если возраст находится в интервале от 225 до 235 лет.

Первое утверждение проверяет, является ли возраст менее `235`, и так как выражение `dragonAge <= 235` возвращает `true`, программа продолжает свою работу. То же происходит и во втором утверждении, только проверяется значение выражения `dragonAge >= 235`.

Обратите внимание, что в качестве второго входного аргумента функции `assert(_:_:)` передана не текстовая строка, а выражение, которое возвращает значение типа `String` (происходит конкатенация строк).

Изменим значение переменной `dragonAge` на `220` и посмотрим на результат (листинг 13.2).

Листинг 13.2

```
var dragonAge = 220
assert( dragonAge <= 235, strName+strEmpty+strOld )
assert( dragonAge >= 225, strName+strEmpty+strYoung )
print("Программа упешно завершила свою работу")
```

Консоль

```
Assertion failed: Дракон молод
```

Первое утверждение все так же возвращает `true` при проверке выражения, но второе утверждение экстренно завершает работу программы, так как значение `dragonAge` меньше `225`. В результате на консоль выведено сообщение, переданное в данную функцию.

Данный механизм следует использовать, когда значение переданного условия однозначно должно быть равно `true`, но есть вероятность, что оно вернет `false`.

Утверждения — это простейший механизм управления ходом работы программы. В частности, с его помощью невозможно выполнять различные блоки кода в зависимости от результата переданного выражения, возможно лишь аварийно завершить программу. Для решения этой проблемы в Swift имеются другие механизмы.

ПРИМЕЧАНИЕ Старайтесь не использовать утверждения в готовых к выпуску проектах. В основном они предназначены для отладки разрабатываемых программ или тестирования кода в Playground.

13.2. Оператор условия if

Утверждения, с которыми вы только что познакомились, являются упрощенной формой оператора условия. Они анализируют переданное выражение и позволяют либо продолжить выполнение программы, либо завершить его (возможно, с выводом отладочного сообщения).

Оператор `if` позволяет определить логику вызова блоков кода (исполнять или не исполнять) в зависимости от значения переданного выражения. Данный оператор используется повсеместно, благодаря ему можно совершать именно те действия, которые необходимы для получения корректного результата. К примеру, если пользователь нажал кнопку «Умножить», то необходимо именно перемножить, а не сложить числа.

Оператор условия `if` имеет четыре формы записи, различающихся по синтаксису и функциональным возможностям:

- сокращенная;
- стандартная;
- расширенная;
- тернарная.

Сокращенный синтаксис оператора if

СИНТАКСИС

```
if проверяемое_выражение {
    // тело оператора
}
```

`проверяемое_выражение -> Bool` — вычисляемое выражение, на основании значения которого принимается решение об исполнении кода, находящегося в теле оператора.

Оператор условия начинается с ключевого слова `if`, за которым следует проверяемое выражение. Если оно возвращает `true`, то выполняется код из тела оператора. В противном случае код в теле игнорируется и выполнение программы продолжается кодом, следующим за оператором условия.

Как и в утверждениях, проверяемое выражение должно возвращать значение типа `Bool`.

Пример

```
if userName == "Alex" {
    print("Привет, администратор")
}
```

В зависимости от значения параметра `userName`, проверяемое выражение вернет `true` или `false`, после чего будет принято решение о вызове функции `print(_:)`, находящейся в теле оператора.

В листинге 13.3 приведен пример использования оператора условия. В нем проверяется значение переменной `logicVar`.

Листинг 13.3

```
// переменная типа Bool
var logicVar = true
// проверка значения переменной
if logicVar == true {
   print("Переменная logicVar истинна")
}
```

Консоль

```
Переменная logicVar истинна
```

В качестве условия для оператора `if` используется выражение сравнения переменной `logicVar` с логическим значением `true`. Так как результатом этого выражения является «истина» (`true`), код, находящийся в теле оператора, будет исполнен, о чем свидетельствует сообщение на консоли.

Левая часть выражения `logicVar == true` сама по себе возвращает значение типа `Bool`, после чего сравнивается с `true`. По этой причине данное выражение является избыточным, а значит, его правая часть может быть опущена (листинг 13.3а).

Листинг 13.3а

```
if logicVar {
   print("Переменная logicVar истинна")
}
```

Консоль

```
Переменная logicVar истинна
```

Если изменить значение `logicVar` на `false`, то проверка не пройдет и функция `print(_:)` не будет вызвана (листинг 13.4).

Листинг 13.4

```
logicVar = false
if logicVar {
   print("Переменная logicVar истинна")
}
// вывод на консоли пуст
```

Swift — это язык со строгой типизацией. Любое проверяемое выражение **обязательно** должно возвращать либо `true`, либо `false`, и никак иначе. По этой причине, если оно возвращает значение другого типа, Xcode сообщит об ошибке (листинг 13.5).

Листинг 13.5

```
var intVar = 1
if intVar { // ОШИБКА
}
```

Если требуется проверить выражение не на истинность, а на ложность, то для этого достаточно сравнить его с `false` **или** добавить знак логического отрицания перед выражением (листинг 13.6).

Листинг 13.6

```
logicVar = false
// полная форма проверки на отрицание
if logicVar == false {
    print("Переменная logicVar ложна")
}
// сокращенная форма проверки на отрицание
if !logicVar {
    print("Переменная logicVar вновь ложна")
}
```

Консоль

```
Переменная logicVar ложна
Переменная logicVar вновь ложна
```

Обе формы записи из предыдущего листинга являются синонимами. В обоих случаях результатом выражения будет значение `true` (именно по этой причине код в теле оператора выполняется).

В первом варианте порядок вычисления выражения следующий:

1. `logicVar == false`.

2. `false == false` — заменили переменную ее значением.
3. `true` — два отрицательных значения идентичны друг другу.

Во втором варианте он отличается:

1. `!logicVar`.
2. `!false` — заменили переменную ее значением.
3. `True` — *НЕ ложь* является *истиной*.

ПРИМЕЧАНИЕ Если данный материал вам непонятен или кажется ошибочным, то предлагаю вновь познакомиться с описанием логического типа `Bool` и внимательно выполнить домашнее задание.

Стандартный синтаксис оператора if

Сокращенный синтаксис оператора `if` позволяет выполнить блок кода только в случае истинности переданного выражения. С помощью стандартного синтаксиса оператора условия `if` вы можете включить в рамки одного оператора блоки кода, как для истинного, так и для ложного результата проверки выражения.

СИНТАКСИС

```
if проверяемое_выражение {
    // первый блок исполняемого кода (при истинности проверяемого
    // условия)
} else {
    // второй блок исполняемого кода (при ложности проверяемого
    // условия)
}
```

- `проверяемое_выражение -> Bool` — вычисляемое выражение, на основании значения которого принимается решение об исполнении кода, находящегося в соответствующем блоке.

Если проверяемое выражение возвращает `true`, то выполняется код из первого блока. В ином случае выполняется блок кода, который следует после ключевого слова `else`.

Пример

```
if userName == "Alex" {
    print("Привет, администратор")
}else{
    print("Привет, пользователь")
}
```

В зависимости от значения в переменной `userName` будет выполнен первый или второй блок кода.

ПРИМЕЧАНИЕ В тексте книги вы будете встречаться с различными обозначениями данного оператора: оператор `if`, конструкция `if-else` и т. д. В большинстве случаев это синонимы, указывающие на использование данного оператора без привязки к какому-либо синтаксису (сокращенному или стандартному).

Рассмотрим пример использования стандартного синтаксиса конструкции `if-else` (листинг 13.7).

Листинг 13.7

```
// переменная типа Bool
var logicVar = false
// проверка значения переменной
if logicVar {
    print("Переменная logicVar истинна")
} else {
    print("Переменная logicVar ложна")
}
```

Консоль

```
Переменная logicVar ложна
```

В приведенном примере предусмотрен блок кода для любого варианта значения переменной `logicVar`.

С помощью оператора условия можно осуществить любую проверку, которая вернет логическое значение. В качестве примера определим, в какой диапазон попадает сумма двух чисел (листинг 13.8).

Листинг 13.8

```
var a = 1054
var b = 952
if a+b > 1000 {
    print( "Сумма больше 1000" )
}else{
    print( "Сумма меньше или равна 1000" )
}
```

Консоль

```
Сумма больше 1000
```

Левая часть проверяемого выражения `a+b>1000` возвращает значение типа `Int`, после чего оно сравнивается с помощью оператора `>` с правой частью (которая также имеет тип `Int`), в результате чего получается логическое значение.

Ранее, при рассмотрении типа данных `Bool`, мы познакомились с операторами `И` (`&&`) и `ИЛИ` (`||`), позволяющими усложнять вычисляемое

логическое выражение. Они также могут быть применены и при использовании конструкции `if-else` для проверки нескольких условий. В листинге 13.9 проверяется истинность значений двух переменных.

Листинг 13.9

```
// переменные типа Bool
var firstLogicVar = true
var secondLogicVar = false
// проверка значения переменных
if firstLogicVar || secondLogicVar {
    print("Одна или две переменные истинны")
} else {
    print("Обе переменные ложны")
}
```

Консоль

```
Одна из переменных истинна
```

Так как проверяемое выражение возвращает `true`, в случае, если хотя бы один из операндов равен `true`, в данном примере будет выполнен первый блок кода.

В приведенном выше примере есть один недостаток: невозможно отличить, когда одна, а когда две переменные имеют значение `true`. Для решения этой проблемы можно вкладывать операторы условия друг в друга (листинг 13.10).

Листинг 13.10

```
if firstLogicVar || secondLogicVar {
    if firstLogicVar && secondLogicVar {
        print("Обе переменные истинны")
    }else{
        print("Только одна из переменных истинна")
    }
} else {
    print("Обе переменные ложны")
}
```

Внутри тела первого оператора условия используется дополнительная конструкция `if-else`.

Стоит отметить, что наиболее однозначные результаты лучше проверять в первую очередь. По этой причине выражение `firstLogicVar && secondLogicVar` стоит вынести в первый оператор, так как оно вернет `true`, только если обе переменные имеют значение `true`. При этом

вложенный оператор с выражением `firstLogicVar || secondLogicVar` потребуется перенести в блок `else` (листинг 13.10а).

Листинг 13.10а

```
if firstLogicVar && secondLogicVar {
    print("Обе переменные истинны")
} else {
    if firstLogicVar || secondLogicVar {
        print("Только одна из переменных истинна ")
    }else{
        print("Обе переменные ложны")
    }
}
```

Конструкции `if-else` могут вкладываться друг в друга без каких-либо ограничений, но помните, что такие «башенные» конструкции могут плохо сказываться на читабельности вашего кода.

Расширенный синтаксис оператора if

Вместо использования вложенных друг в друга операторов `if` можно использовать расширенный синтаксис оператора условия, позволяющий объединить несколько вариантов результатов проверок в рамках одной конструкции.

СИНТАКСИС

```
if проверяемое_выражение_1 {
   // первый блок кода
} else if проверяемое_выражение_2 {
   // второй блок кода
}...
} else {
    // последний блок кода
 }
```

- `проверяемое_выражение -> Bool` — вычисляемое выражение, на основании значения которого принимается решение об исполнении кода, находящегося в блоке.

Если первое проверяемое условие возвращает `true`, то исполняется первый блок кода.

В ином случае проверяется второе выражение. Если оно возвращает `true`, то исполняется второй блок кода. И так далее.

Если ни одно из условий не вернуло истинный результат, то выполняется последний блок кода, расположенный после ключевого слова `else`.

Количество блоков `else if` в рамках одного оператора условия может быть произвольным, а наличие оператора `else` не является обязательным.

После нахождения первого выражения, которое вернет `true`, дальнейшие проверки не проводятся.

Пример

```
if userName == "Alex" {
    print("Привет, администратор")
}else if userName == "Bazil"{
    print("Привет, модератор")
}else if userName == "Helga"{
    print("Привет, редактор")
}else{
    print("Привет, пользователь")
}
```

Изменим код проверки значения переменных `firstLogicVar` и `secondLogicVar` с учетом возможностей расширенного синтаксиса оператора `if` (листинг 13.11).

Листинг 13.11

```
// проверка значения переменных
if firstLogicVar && secondLogicVar {
    print("Обе переменные истинны")
} else if firstLogicVar || secondLogicVar {
    print("Одна из переменных истинна")
} else {
    print("Обе переменные ложны")
}
```

Консоль

```
Обе переменные истинны
```

Запомните, что, найдя первое совпадение, оператор выполняет соответствующий блок кода и прекращает свою работу. Проверки последующих условий **не проводятся**.

При использовании данного оператора будьте внимательны. Помните, ранее говорилось о том, что наиболее однозначные выражения необходимо располагать первыми? Если отойти от этого правила и установить значения переменных `firstLogicVar` и `secondLogicVar` в `true`, но выражение из `else if` блока (с логическим `ИЛИ`) расположить первым, то вывод на консоль окажется ошибочным (листинг 13.12).

Листинг 13.12

```
firstLogicVar = true
secondLogicVar = true
```

```
if firstLogicVar || secondLogicVar {
    print("Одна из переменных истинна")
} else if firstLogicVar && secondLogicVar {
    print("Обе переменные истинны")
} else {
    print("Обе переменные ложны")
}
```

Консоль

```
Одна из переменных истинна
```

Выведенное на консоль сообщение («Одна из переменных истинна») не является достоверным, так как на самом деле обе переменные имеют значение `true`.

Рассмотрим еще один пример использования конструкции `if-else` .

Предположим, что вы сдаете в аренду квартиры в жилом доме. Стоимость аренды, которую платит каждый жилец, зависит от общего количества жильцов:

- Если жильцов менее 5 — стоимость аренды жилья равна 1000 рублей с человека в день.
- Если жильцов от 5 до 7 — стоимость аренды равна 800 рублям с человека в день.
- Если жильцов более 7 — стоимость аренды равна 500 рублям с человека в день.

Тарифы, конечно, довольно странные, но, тем не менее, перед вами стоит задача подсчитать общий дневной доход.

Для реализации этого алгоритма используем оператор if-else. Программа получает количество жильцов в доме и в зависимости от стоимости аренды для одного жильца возвращает общую сумму средств, которая будет получена (листинг 13.13).

Листинг 13.13

```
// количество жильцов в доме
var tenantCount = 6
// стоимость аренды на человека
var rentPrice = 0
/* определение цены на одного
человека в соответствии с условием */
if tenantCount < 5 {
    rentPrice = 1000
} else if tenantCount >= 5 && tenantCount <= 7 {
   rentPrice = 800
```

```
} else {
   rentPrice = 500
}
// вычисление общей суммы средств
var allPrice = rentPrice * tenantCount // 4800
```

Так как общее количество жильцов попадает во второй блок конструкции `if-else`, переменная `rentPrice` (сумма аренды для одного человека) принимает значение `800`. Итоговая сумма равна `4800`.

Кстати, данный алгоритм может быть реализован с использованием операторов диапазона и метода `contains(_:)`, позволяющего определить, попадает ли значение в требуемый диапазон (листинг 13.13а).

Листинг 13.13а

```
if (..<5).contains(tenantCount) {
    rentPrice = 1000
} else if (5...7).contains(tenantCount) {
    rentPrice = 800
} else if (8...).contains(tenantCount) {
    rentPrice = 500
}
```

Последний `else if` блок можно заменить на `else` без указания проверяемого выражения, но показанный пример нагляднее.

Тернарный оператор условия

Swift позволяет значительно упростить стандартную форму записи оператора `if` всего до нескольких символов. Данная форма называется **тернарным оператором условия**. Его отличительной особенностью является то, что он не просто выполняет соответствующее выражение, но и возвращает результат его работы.

СИНТАКСИС

```
проверяемое_выражение ? выражение_1 : выражение_2
```

- `проверяемое_выражение -> Bool` — вычисляемое выражение, на основании значения которого принимается решение об исполнении кода, находящегося в блоке.
- `выражение_1 -> Any` — выражение, значение которого будет возвращено, если проверяемое выражение вернет `true`.
- `выражение_2 -> Any` — выражение, значение которого будет возвращено, если проверяемое выражение вернет `false`.

При истинности проверяемого выражения выполняется первый блок кода. В ином случае выполняется второй блок. При этом тернарный оператор не просто выполняет код в блоке, но возвращает значение из него.

Пример

```
var y = ( x > 100 ? 100 : 50 )
```

В данном примере в переменной y будет инициализировано значение одного из выражений (100 или 50).

В листинге 13.14 показан пример использования тернарного оператора условия. В нем сопоставляются значения констант для того, чтобы вывести на отладочную консоль соответствующее сообщение.

Листинг 13.14

```
let a = 1
let b = 2
// сравнение значений констант
a <= b ? print("A меньше или равно B"):print("A больше B")
```

Консоль

```
A меньше или равно B
```

Так как значение константы a меньше значения b, то проверяемое выражение a <= b вернет true, а значит, будет выполнено первое выражение. Обратите внимание, что в данном примере тернарный оператор ничего не возвращает, а просто выполняет соответствующее выражение (так как возвращать, собственно, и нечего). Но вы можете использовать его и иначе. В листинге 13.15 показан пример того, что возвращенное оператором значение может быть использовано в составе другого выражения.

Листинг 13.15

```
// переменная типа Int
var height = 180
// переменная типа Bool
var isHeader = true
// вычисление значения константы
let rowHeight = height + (isHeader ? 20 : 10 )
// вывод значения переменной
rowHeight // 200
```

В данном примере тернарный оператор условия возвращает одно из двух целочисленных значений типа Int (20 или 10) в зависимости от логического значения переменной isHeader.

13.3. Оператор ветвления switch

Нередки случаи, когда приходится работать с большим количеством вариантов значений вычисляемого выражения, и для каждого из возможных значений необходимо выполнить определенный код. Для этого можно использовать расширенный синтаксис оператора `if`, многократно повторяя блоки `else if`.

Предположим, что в зависимости от полученной пользователем оценки стоит задача вывести определенный текст. Реализуем логику с использованием оператора `if` (листинг 13.16).

Листинг 13.16

```
// оценка
var userMark = 4
if userMark == 1 {
   print("Единица на экзамене! Это ужасно!")
} else if userMark == 2 {
   print("С двойкой ты останешься на второй год!")
} else if userMark == 3 {
   print("Ты плохо учил материал в этом году!")
} else if userMark == 4 {
   print("Неплохо, но могло быть и лучше")
} else if userMark == 5 {
   print("Бесплатное место в университете тебе обеспечено!")
} else {
   print("Переданы некорректные данные об оценке")
}
```

Консоль

```
Неплохо, но могло бы быть и лучше
```

Для вывода сообщения, соответствующего оценке, оператор `if` последовательно проходит по каждому из указанных условий, пока не найдет то, которое вернет `true`. Хотя программа работает корректно, но с ростом количества возможных вариантов (предположим, у вас десятибалльная шкала оценок) ориентироваться в коде будет все сложнее.

Эту задачу можно решить с помощью оператора ветвления `switch`, позволяющего с легкостью обрабатывать большое количество возможных вариантов.

СИНТАКСИС

```
switch проверяемое_выражение {
   case значение_1:
      // первый блок кода
   case значение_2, значение_3:
```

```
        // второй блок кода
    ...
    case значение_N:
        // N-й блок кода
    default:
        // блок кода по умолчанию
}
```

- проверяемое_выражение -> Any — вычисляемое выражение, значение которого Swift будет искать среди case-блоков. Может возвращать значение любого типа.
- значение:Any — значение, которое может вернуть проверяемое выражение. Должно иметь тот же тип данных.

После ключевого слова switch указывается выражение, значение которого вычисляется. После получения значения выражения производится поиск совпадающего значения среди указанных в case-блоках (после ключевых слов case).

Если совпадение найдено, то выполняется код тела соответствующего case-блока.

Если совпадение не найдено, то выполняется код из default-блока, который должен всегда располагаться последним в конструкции switch-case.

Количество блоков case может быть произвольным и определяется программистом в соответствии с контекстом задачи. После каждого ключевого слова case может быть указано любое количество значений, которые отделяются друг от друга запятыми.

Конструкция switch-case должна быть исчерпывающей, то есть содержать информацию обо всех возможных значениях проверяемого выражения. Это обеспечивается в том числе наличием блока default.

Код, выполненный в любом блоке case, приводит к завершению работы оператора switch.

Пример

```
switch userMark {
    case 1,2:
        print("Экзамен не сдан")
    case 3:
    print("Необходимо выполнить дополнительное задание")
    case 4,5:
        print("Экзамен сдан")
    default:
        print("Указана некорректная оценка")
}
```

В зависимости от значения в переменной userMark, будет выполнен соответствующий блок кода. Если ни одно из значений, указанных после case, не подошло, то выполняется код в default-блоке.

ПРИМЕЧАНИЕ Далее в тексте книги вы будете встречать различные наименования данного оператора: оператор switch, конструкция switch-case и другие термины. Все они являются синонимами.

Перепишем программу проверки оценки, полученную учеником, с использованием конструкции `switch-case`. При этом вынесем вызов функции `print(_:)` за пределы оператора. Таким образом, если вам в будущем потребуется изменить принцип вывода сообщения (например, вы станете использовать другую функцию), то достаточно будет внести правку в одном месте кода вместо шести (листинг 13.17).

Листинг 13.17

```
var userMark = 4
// переменная для хранения сообщения
var message = ""
switch userMark {
case 1:
    message = "Единица на экзамене! Это ужасно!"
case 2:
    message = "С двойкой ты останешься на второй год!"
case 3:
    message = "Ты плохо учил материал в этом году!"
case 4:
    message = "Неплохо, но могло быть и лучше"
case 5:
    message = "Бесплатное место в университете тебе обеспечено!"
default:
    message = "Переданы некорректные данные об оценке"
}
// вывод сообщения на консоль
print(message)
```

Консоль

```
Неплохо, но могло быть и лучше
```

Оператор `switch` вычисляет значение переданного в него выражения (состоящего всего лишь из одной переменной `userMark`) и последовательно проверяет каждый блок `case` в поисках совпадения.

Если `userMark` будет иметь значение менее 1 или более 5, то будет выполнен код в блоке `default`.

Самое важное отличие конструкции `switch-case` от `if-else` заключается в том, что проверяемое выражение может возвращать значение совершенно любого типа, включая строки, числа, диапазоны и даже кортежи.

ПРИМЕЧАНИЕ Обратите внимание, что переменная `message` объявляется вне конструкции `switch-case`. Это связано с тем, что если бы это делалось в каждом case-блоке, то ее область видимости была бы ограничена оператором ветвления и выражение `print(message)` вызвало бы ошибку.

Диапазоны в операторе switch

Одной из интересных возможностей конструкции `switch-case` является возможность работы с диапазонами. В листинге 13.18 приведен пример определения множества, в которое попадает заданное число. При этом используются операторы диапазона без использования метода `contains(_:)`, как было у оператора `if`.

Листинг 13.18

```
userMark = 4
switch userMark {
case 1..<3:
    print("Экзамен не сдан")
case 3:
    print("Требуется решение дополнительного задания")
case 4...5:
    print("Экзамен сдан")
default:
    assert( false, "Указана некорректная оценка")
}
```

Консоль

```
Экзамен сдан!
```

Строка `"Экзамен не сдан"` выводится на консоль при условии, что проверяемое значение в переменной `userMark` попадает в диапазон `1..<3`, то есть равняется `1` или `2`. При значении `userMark` равном `3` будет выведено «`Требуется решение дополнительного задания`». При `userMark` равном `4` или `5` выводится строка « `Экзамен сдан!`».

Если значение `userMark` не попадает в диапазон `1...5`, то управление переходит default-блоку, в котором используется утверждение, прерывающее программу. С помощью передачи логического `false` функция `assert(_:_:)` прекратит работу приложения в любом случае, если она была вызвана.

Другим вариантом реализации данного алгоритма может быть использование уже знакомой конструкции `if-else` с диапазонами и методом `contains(_:)`.

Кортежи в операторе switch

Конструкция `switch-case` оказывается очень удобной при работе с кортежами.

Рассмотрим следующий пример. Происходит взаимодействие с удаленным сервером. После каждого запроса вы получаете ответ, состоящий из цифрового кода и текстового сообщения, после чего записываете эти данные в кортеж (листинг 13.19).

Листинг 13.19

```
var answer: (code: Int, message: String) = (code: 404, message: "Page
not found")
```

По коду сообщения можно определить результат взаимодействия с сервером.

Если код находится в интервале от 100 до 399, то сообщение необходимо вывести на консоль.

Если код — от 400 до 499, то это говорит об ошибке, вследствие которой работа приложения должна быть аварийно завершена с выводом сообщения на отладочную консоль.

Во всех остальных сообщениях необходимо отобразить сообщение о некорректном ответе сервера.

Реализуем данный алгоритм с использованием оператора `switch`, передав кортеж в качестве входного выражения (листинг 13.20).

Листинг 13.20

```
switch answer {
case (100..<400, _):
    print( answer.message )
case (400..<500, _):
    assert( false, answer.message )
default:
    print( "Получен некорректный ответ" )
}
```

Для первого элемента кортежа в заголовках case-блоков используются диапазоны. С их помощью значения первых элементов будут отнесены к определенному множеству. При этом второй элемент кортежа игнорируется с помощью уже известного вам нижнего подчеркивания. Таким образом, получается, что в конструкции `switch-case` используется значение только первого элемента.

Рассмотрим другой пример.

Вы — владелец трех вольеров для драконов. Каждый вольер предназначен для содержания драконов с определенными характеристиками:

- Вольер 1 для зеленых драконов с массой менее двух тонн.
- Вольер 2 для красных драконов с массой менее двух тонн.

- Вольер 3 для зеленых и красных драконов с массой более двух тонн.

При поступлении нового дракона нужно определить, в какой вольер его поместить.

В условии задачи используются две характеристики драконов, поэтому имеет смысл обратиться к кортежам для их объединения в единое значение (листинг 13.21).

Листинг 13.21

```
var dragonCharacteristics: (color: String, weight: Float) = ("красный",
1.4)
switch dragonCharacteristics {
    case ("зеленый", 0..<2 ):
        print("Вольер № 1")
    case ("красный", 0..<2 ):
        print("Вольер № 2")
    case ("зеленый", 2...), ("красный", 2...):
        print("Вольер № 3")
    default:
        print("Дракон не может быть принят в стаю")
}
```

Консоль

```
Вольер № 2
```

Задача выполнена, при этом код получился довольно простым и изящным.

Ключевое слово where в операторе switch

Представьте, что в задаче про вольеры для драконов появилось дополнительное условие: поместить дракона в вольер № 3 можно только при условии, что в нем находятся менее пяти особей. Для хранения количества драконов будет использоваться дополнительная переменная.

Включать в кортеж третий элемент и проверять его значение было бы неправильно с точки зрения логики, так как количество драконов не имеет ничего общего с характеристиками конкретного дракона.

С одной стороны, данный функционал можно реализовать с помощью конструкции `if-else` внутри последнего case-блока, но наиболее правильным вариантом станет использование ключевого слова `where`, позволяющего указать дополнительные требования, в том числе к значениям внешних параметров (листинг 13.22).

Листинг 13.22

```
var dragonsCount = 3
switch dragonCharacteristics {
case ("зеленый", 0..<2 ):
    print("Вольер № 1")
case ("красный", 0..<2 ):
    print("Вольер № 2")
case ("зеленый", 2...) where dragonsCount < 5,
     ("красный", 2...) where dragonsCount < 5:
    print("Вольер № 3")
default:
    print("Дракон не может быть принят в стаю")
}
```

Консоль

```
Вольер № 3
```

Ключевое слово `where` и дополнительные условия указываются в case-блоке для каждого значения отдельно. После `where` следует выражение, которое должно вернуть `true` или `false`. Тело блока будет выполнено, когда одновременно совпадет значение, указанное после `case`, и условие после `where` вернет `true`.

Связывание значений

Представьте ситуацию, что ваше руководство окончательно сошло с ума (думаю, многим и представлять не требуется ☺) и выставило новое условие: вольер № 3 может принять только тех драконов, чей вес без остатка делится на единицу. Конечно, глупая ситуация, да и вряд ли ваше начальство станет отсеивать столь редких рептилий, но гипотетически такое возможно, и мы должны быть к этому готовы.

Подумайте, как бы вы решили эту задачу? Есть два подхода: используя оператор условия `if` внутри тела case-блока или используя `where` с дополнительным условием. Второй подход наиболее верный, так как не имеет смысла переходить к выполнению кода, если условия не выполняются (листинг 13.23).

Листинг 13.23

```
switch dragonCharacteristics {
case ("зеленый", 0..<2 ):
    print("Вольер № 1")
case ("красный", 0..<2 ):
    print("Вольер № 2")
case ("зеленый", 2...) where
```

```
        dragonCharacteristics.weight.truncatingRemainder(dividingBy:
        1) == 0
        && dragonsCount < 5,
     ("красный", 2...) where
        dragonCharacteristics.weight.truncatingRemainder(dividingBy:
        1) == 0
        && dragonsCount < 5:
    print("Вольер № 3")
default:
    print("Дракон не может быть принят в стаю")
}
```

Выражение `dragonCharacteristics.weight.truncatingRemainder (dividingBy: 1) == 0` позволяет определить, делится ли вес дракона, указанный в кортеже, без остатка на единицу.

Хотя данный подход и является полностью рабочим, он содержит два значительных недостатка:

1. Вследствие длинного имени кортежа выражение не очень удобно читать и воспринимать.
2. Данное выражение привязывается к имени кортежа вне конструкции `switch-case`. Таким образом, если вы решите поменять имя с `dragonCharacteristics`, предположим, на `dragonValues`, то необходимо не только соответствующим образом изменить входное выражение, но и внести правки внутри `case`.

Для решения данной проблемы можно использовать прием, называемый **связыванием значений**. При связывании в заголовке case-блока используется ключевое слово `let` или `var` для объявления локального параметра, с которым будет связано значение. Затем данный параметр можно использовать после `where` и в теле `case`-блока.

В листинге 13.24 происходит связывание значения второго элемента кортежа `dragonCharacteristics`.

Листинг 13.24

```
switch dragonCharacteristics {
case ("зеленый", 0..<2 ):
    print("Вольер № 1")
case ("красный", 0..<2 ):
    print("Вольер № 2")
case ("зеленый", let weight) where
        weight > 2
        && dragonsCount < 5,
     ("красный", let weight) where
        weight > 2
```

```
        && dragonsCount < 5:
    print("Вольер № 3")
default:
    print("Дракон не может быть принят в стаю")
}
```

В заголовке case-блока для второго элемента кортежа вместо указания диапазона используется связывание его значения с локальной константой `weight`. Данная константа называется **связанным параметром**, она содержит **связанное значение**. Обратите внимание, что проверка веса дракона (более двух тонн) перенесена в where-условие.

После связывания значения данный параметр можно использовать не только в where-условии, но и внутри тела case-блока (листинг 13.25).

Листинг 13.25

```
switch dragonCharacteristics {
case ("зеленый", 0..<2 ):
    print("Вольер № 1")
case ("красный", 0..<2 ):
    print("Вольер № 2")
case ("зеленый", let weight) where
        weight > 2
        && weight.truncatingRemainder(dividingBy: 1) == 0
        && dragonsCount < 5,
     ("красный", let weight) where
        weight > 2
        && weight.truncatingRemainder(dividingBy: 1) == 0
        && dragonsCount < 5:
    print("Вольер № 3. Вес дракона \(weight) тонны")
default:
    print("Дракон не может быть принят в стаю")
}
```

Можно сократить и упростить код, если объявить сразу два связанных параметра, по одному для каждого элемента кортежа (листинг 13.25а).

Листинг 13.25а

```
switch dragonCharacteristics {
case ("зеленый", 0..<2 ):
    print("Вольер № 1")
case ("красный", 0..<2 ):
    print("Вольер № 2")
case let (color, weight) where
    (color == "зеленый" || color == "красный")
    && weight.truncatingRemainder(dividingBy: 1) == 0
    && dragonsCount < 5:
        print("Вольер № 3. Вес дракона \(weight) тонны")
```

```
default:
    print("Дракон не может быть принят в стаю")
}
```

Оператор break в конструкции switch-case

Если требуется, чтобы case-блок не имел исполняемого кода, то необходимо указать ключевое слово `break`, с помощью которого работа оператора `switch` будет принудительно завершена.

Одним из типовых вариантов использования `break` является его определение в блоке `default`, когда в нем не должно быть другого кода. Таким образом достигается избыточность — даже если в `case` не будет найдено необходимое значение, то конструкция завершит свою работу без ошибок, просто передав управление в default-блок, где находится `break`.

В листинге 13.26 показан пример, в котором на консоль выводятся сообщения в зависимости от того, является целое число положительным или отрицательным. При этом в случае, когда оно равняется нулю, оператор `switch` просто завершает работу.

Листинг 13.26

```
var someInt = 12
switch someInt {
case 1...:
    print( "Больше 0" )
case ..<0:
    print( "Меньше 0" )
default:
    break
}
```

Ключевое слово fallthrough

С помощью ключевого слова `fallthrough` можно изменить логику функционирования оператора `switch` и не прерывать его работу после выполнения кода в `case`-блоке. Данное ключевое слово позволяет перейти к телу последующего `case`-блока.

Рассмотрим следующий пример: представьте, что существует три уровня готовности к чрезвычайным ситуациям — А, Б и В. Каждая степень предусматривает выполнение ряда мероприятий, причем каждый последующий уровень включает в себя мероприятия предыдущих уровней. Минимальный уровень — это В, максимальный — А (включает в себя мероприятия уровней В и Б).

Реализуем программу, выводящую на консоль все мероприятия, соответствующие текущему уровню готовности к ЧС. Так как мероприятия повторяются от уровня к уровню, то можно реализовать задуманное с помощью оператора `switch` с использованием ключевого слова `fallthrough` (листинг 13.27).

Листинг 13.27

```
var level: Character = "Б"
// определение уровня готовности
switch level {
    case "А":
        print("Выключить все электрические приборы ")
        fallthrough
    case "Б":
        print("Закрыть входные двери и окна ")
        fallthrough
    case "В":
        print("Соблюдать спокойствие")
    default:
        break
}
```

Консоль

```
Закрыть входные двери и окна
Соблюдать спокойствие
```

При значении `"Б"` переменной `level` на консоль выводятся строки, соответствующие значениям `"Б"` и `"В"`. Когда программа встречает ключевое слово `fallthrough`, она переходит к выполнению кода следующего `case`-блока.

13.4. Операторы повторения while и repeat while

Ранее мы рассмотрели операторы, позволяющие выполнять различные участки кода в зависимости от условий, возникших в ходе работы программы. Далее будут рассмотрены конструкции языка, позволяющие циклично (многократно) выполнять блоки кода и управлять данными повторениями. Другими словами, их основная идея состоит в том, чтобы программа многократно выполняла одни и те же выражения, но с различными входными данными.

ПРИМЕЧАНИЕ Конструкции, позволяющие в зависимости от условий многократно выполнять код, называются циклами.

Операторы `while` и `repeat while` позволяют выполнять блок кода до тех пор, пока проверяемое выражение возвращает `true`.

Оператор while

СИНТАКСИС

```
while проверяемое_выражение {
   //тело оператора
}
```

- `проверяемое_выражение -> Bool` — выражение, при истинности которого выполняется код из тела оператора.

Одно выполнение кода тела оператора называется итерацией. Итерации повторяются, пока выражение возвращает `true`. Его значение проверяется перед каждой итерацией.

Рассмотрим пример использования оператора `while`. Произведем с его помощью сложение всех чисел от 1 до 10 (листинг 13.28).

Листинг 13.28

```
// начальное значение
var i = 1
// хранилище результата сложения
var resultSum = 0
// цикл для подсчета суммы
while i <= 10 {
    resultSum += i
    i  += 1
}
resultSum // 55
```

Переменная `i` является счетчиком в данном цикле. Основываясь на ее значении, оператором определяется необходимость выполнения кода в теле цикла. На каждой итерации значение `i` увеличивается на единицу, и как только оно достигает `10`, то условие, проверяемое оператором, возвращает `false`, после чего происходит выход из цикла.

Оператор `while` — это цикл с предварительной проверкой условия, то есть вначале проверяется условие, а уже потом выполняется или не выполняется код тела оператора. Если условие вернет `false` уже при первой проверке, то код внутри оператора проигнорируется и не будет выполнен ни одного раза.

ВНИМАНИЕ Будьте осторожны при задании условия, поскольку по невнимательности можно создать такую ситуацию, при которой проверяемое условие всегда будет возвращать `true`. В этом случае цикл будет выполняться бесконечно, что, вероятно, приведет к зависанию программы.

Оператор repeat while

В отличие от `while` оператор `repeat while` является циклом с постпроверкой условия. В таком цикле проверка значения выражения происходит в конце итерации.

СИНТАКСИС

```
repeat {
     // тело оператора
} while проверяемое_выражение
```

- `проверяемое_выражение -> Bool` — выражение, при истинности которого выполняется код из тела оператора.

Одно выполнение кода тела оператора называется итерацией. Итерации повторяются, пока выражение возвращает `true`. Его значение проверяется после каждой итерации, таким образом, тело оператора будет выполнено не менее одного раза.

Реализуем с помощью данного оператора рассмотренную ранее задачу сложения чисел от 1 до 10 (листинг 13.29).

Листинг 13.29

```
// начальное значение
var y = 1
// хранилище результата сложения
var result = 0
// цикл для подсчета суммы
repeat{
    result += y
    y += 1
} while y <= 10
result // 55
```

Разница между операторами `while` и `repeat while` заключается в том, что код тела оператора `repeat while` выполняется не менее одного раза. То есть даже если условие при первой итерации вернет `false`, код тела цикла к этому моменту уже будет выполнен.

Использование оператора continue

Оператор `continue` предназначен для перехода к очередной итерации, игнорируя следующий за ним код. Если программа встречает `continue`, то она незамедлительно переходит к новой итерации.

Код в листинге 13.30 производит сложение всех четных чисел в интервале от 1 до 10. Для этого в каждой итерации производится проверка на четность (по значению остатка от деления на 2).

Листинг 13.30

```
var x = 0
var sum = 0
while x <= 10 {
    x += 1
    if x % 2 == 1 {
        continue
    }
    sum += x
}
sum // 30
```

Использование оператора break

Оператор `break` предназначен для досрочного завершения работы цикла, с ним мы уже встречались ранее. При этом весь последующий код в теле цикла игнорируется. В листинге 13.31 показано, как производится подсчет суммы всех чисел от 1 до 54. При этом если сумма достигает 450, то происходит выход из оператора и выводится соответствующее сообщение.

Листинг 13.31

```
var lastNum = 54
var currentNum = 1
var sumOfInts = 0
while currentNum <= lastNum {
    sumOfInts += currentNum
    if sumOfInts > 450 {
        print("Хранилище заполнено. Последнее обработанное число -
        \(currentNum)")
        break
    }
    currentNum += 1
}
```

Консоль

```
Хранилище заполнено. Последнее обработанное число — 30
```

13.5. Оператор повторения for

Оператор `for` предназначен для цикличного выполнения блока кода для каждого элемента некоторой последовательности (`Sequence`). Другими словами, для каждого элемента будет выполнен один и тот же блок кода. Данный оператор принимает на вход любую последователь-

ность (включая коллекции). К примеру, с его помощью можно вывести все символы строки (`String` — это `Collection`) по одному на консоль, вызывая функцию `print(_:)` для каждого символа. И для реализации этой задачи потребуется всего несколько строк кода.

СИНТАКСИС

```
for связанный_параметр in последовательность {
    // тело оператора
}
```

- `связанный_параметр` — локальный параметр, которому будет инициализироваться значение очередного элемента последовательности.
- `последовательность:Sequence` — итерируемая последовательность, элементы которой по одному будут доступны в теле оператора через связанный параметр.

Цикл `for-in` выполняет код, расположенный в теле оператора, для каждого элемента в переданной последовательности. При этом перебор элементов происходит последовательно и по порядку (от первого к последнему).

Перед каждой итерацией происходит связывание значения очередного элемента последовательности с параметром, объявленным после ключевого слова `for`. После этого в коде тела оператора это значение доступно через имя связанного параметра. Данный (связанный) параметр является локальным для конструкции `for-in` и недоступен за ее пределами.

После всех итераций (перебора всех элементов последовательности) цикл завершает свою работу, а связанный параметр уничтожается.

В качестве входной последовательности может быть передана любая `Sequence` (в том числе `Collection`): массив (`Array`), словарь (`Dictionary`), множество (`Set`), диапазон (`Range`) и т. д.

Пример

```
for oneElementOfArray in [1,3,5,7,9]{
    // тело оператора
}
```

В листинге 13.32 показан пример использования оператора `for`, в котором производится подсчет суммы всех элементов массива, состоящего из целочисленных значений.

Листинг 13.32

```
// массив целых чисел
let numArray: Array<Int> = [1, 2, 3, 4, 5]
// в данной переменной будет храниться
// сумма элементов массива numArray
var result: Int = 0
```

```
// цикл подсчета суммы
for number in numArray {
    result += number
}
result //15
```

В данном примере с помощью `for-in` вычисляется сумма значений всех элементов `numArray`. Связанный параметр `number` последовательно получает значение каждого элемента массива и суммирует его с переменной `result`. Данная переменная объявлена вне цикла (до него), поэтому она не уничтожается после завершения работы оператора `for` и сохраняет свое состояние.

В ходе первой итерации переменной `number` присваивается значение первого элемента массива, то есть `1`, после чего выполняется код тела цикла. В следующей итерации переменной `number` присваивается значение второго элемента (`2`) и повторно выполняется код тела цикла. И так далее, пока не будет выполнена последняя итерация тела оператора со значением `5` в переменной `number`.

В качестве входной последовательности также можно передать диапазон (листинг 13.33) или строку (листинг 13.33а).

Листинг 13.33

```
for number in 1...5 {
    print(number)
}
```

Консоль

```
1
2
3
4
5
```

Листинг 13.33а

```
for number in "Swift" {
    print(number)
}
```

Консоль

```
S
w
i
f
t
```

Связанный параметр и все объявленные в теле цикла переменные и константы — локальные, то есть недоступны вне оператора `for`. Если существуют внешние (относительно тела оператора) одноименные переменные или константы, то их значение не будет пересекаться с локальными (листинг 13.34).

Листинг 13.34

```
// внешняя переменная
var myChar = "a"
// внешняя константа
let myString = "Swift"
// цикл использует связанный параметр с именем,
// уже используемым глобальной переменной
for myChar in myString {
    // локальная константа
    // вне цикла уже существует константа с таким именем
    let myString = "Char is"
    print("\(myString) \(myChar)")
}
myChar //"a"
myString //Swift
```

Консоль

```
Char is S
Char is w
Char is i
Char is f
Char is t
```

Вне оператора `for` объявлены два параметра: переменная `myChar` и константа `myString`. В теле оператора объявляются параметры с теми же именами (`myChar` — связанный параметр, а `myString` — локальная константа). Несмотря на то что внутри цикла локальным параметрам инициализируются значения, глобальные `myChar` и `myString` не изменились.

Данный пример может показаться вам слегка запутанным, поэтому уделите ему особое внимание и самостоятельно построчно разберите его код.

ПРИМЕЧАНИЕ Хочу обратить ваше внимание еще на одну замечательную возможность Xcode playground — отображение кривой изменения значения. Если щелкнуть на темно-сером прямоугольнике в области результатов, расположенном напротив тела оператора `if` (из предыдущего листинга), то в редакторе кода отобразится график, позволяющий оценить, как менялось значение параметра `result` (рис. 13.1).

Рис. 13.1. График изменения значения параметра

Порой может возникнуть задача, когда нет необходимости использовать значения элементов коллекции, — к примеру, нужно просто трижды вывести некоторое сообщение на консоль. В этом случае для экономии памяти имя связанного параметра можно заменить на нижнее подчеркивание (листинг 13.34а).

Листинг 13.34а

```
for _ in 1...3 {
    print("Повторяющаяся строка")
}
```

Консоль

```
Повторяющаяся строка
Повторяющаяся строка
Повторяющаяся строка
```

При итерации по элементам словаря (`Dictionary`) можно создать отдельные связанные параметры для ключей и значений элементов (листинг 13.35).

Листинг 13.35

```
var countriesAndBlocks = ["Россия": "СНГ", "Франция":"ЕС"]
for (countryName, orgName) in countriesAndBlocks {
    print("\(countryName) вступила в \(orgName)")
}
```

Консоль

```
Франция вступила в ЕС
Россия вступила в СНГ
```

ПРИМЕЧАНИЕ Как говорилось ранее, `Dictionary` — это неупорядоченная коллекция. Поэтому порядок следования элементов при инициализации значения отличается от того, как выводятся данные на консоль (сперва Франция, а потом Россия, хотя при инициализации было наоборот).

Если требуется получать только ключи или только значения элементов, то можно вновь воспользоваться нижним подчеркиванием (листинг 13.36).

Листинг 13.36

```
var countriesAndBlocks = ["Россия": "СНГ", "Франция":"ЕС"]
for (countryName, _) in countriesAndBlocks {
    print("страна – \(countryName)")
}
for (_, orgName) in countriesAndBlocks{
    print("ораганизация – \( orgName)")
}
```

Помимо этого, в случае если требуется получить последовательность, состоящую только из ключей или значений словаря, можно воспользоваться свойствами `keys` и `value` и передать результат их работы в оператор `for` (листинг 13.37).

Листинг 13.37

```
countriesAndBlocks = ["Россия": "ЕАЭС", "Франция":"ЕС"]
for countryName in countriesAndBlocks.keys {
    print("страна – \(countryName)")
}
for countryName in countriesAndBlocks.values {
    print("организация – \(countryName)")
}
```

Если при работе с массивом для каждого элемента помимо значения требуется получить и индекс, то можно воспользоваться методом `enumerated()`, возвращающим последовательность кортежей, где первый элемент — индекс, а второй — значение (листинг 13.38).

Листинг 13.38

```
print("Несколько фактов обо мне:")
var myMusicStyles = ["Rock", "Jazz", "Pop"]
for (index, musicName) in myMusicStyles.enumerated() {
    print("\(index+1). Я люблю \(musicName)")
}
```

Консоль

```
Несколько фактов обо мне:
1. Я люблю Rock
2. Я люблю Jazz
3. Я люблю Pop
```

Вновь вернемся к работе с последовательностями, состоящими из чисел.

Предположим, что перед вами стоит задача обработать все числа от 1 до 10, идущих с шагом 3 (массив значений 1, 4, 7, 10). В этом случае вы можете «руками» создать коллекцию с необходимыми элементами и передать ее в конструкцию `for-in` (листинг 13.39).

Листинг 13.39

```
// коллекция элементов от 1 до 10 с шагом 3
var intNumbers: Array = [1, 4, 7, 10]
for element in intNumbers{
    // код, обрабатывающий очередной элемент
}
```

Если диапазон чисел будет не таким маленьким, а значительно шире (например: от 1 до 1000 с шагом 5), то самостоятельно описать его будет затруднительно. Также возможна ситуация, когда характеристики множества (начальное и конечное значение, а также шаг) заранее могут быть неизвестны. В этом случае удобнее всего воспользоваться специальными функциями `stride(from:through:by:)` или `stride(from:to:by:)`, формирующими последовательность (`Sequence`) элементов на основе указанных правил.

Функция `stride(from:through:by:)` возвращает последовательность числовых элементов, начиная с `from` до `through` с шагом `by` (листинг 13.40).

Листинг 13.40

```
for i in stride( from:1, through: 10, by: 3 ) {
    // тело оператора
}
```

Параметр `i` будет последовательно принимать значения `1`, `4`, `7`, `10`.

Функция `stride(from:to:by:)` имеет лишь одно отличие — вместо входного параметра `through` используется `to`, который исключает указанное в нем значение из последовательности (листинг 13.41).

Листинг 13.41

```
for i in stride( from:1, to: 10, by:3 ) {
    // тело оператора
}
```

Параметр `i` будет получать значения `1`, `4` и `7`.

В листинге 13.42 приведен пример вычисления суммы всех нечетных чисел от 1 до 1000 с помощью функции `stride(from:through:by:)`.

Листинг 13.42

```
var result = 0
for i in stride( from:1, through: 1000, by:2 ) {
    result += i
}
result // 250000
```

ПРИМЕЧАНИЕ Коллекции элементов, возвращаемые функциями `stride(from:through:by:)` и `stride(from:to:by:)`, представлены в виде значений специальных типов данных `StrideThrough` и `StrideTo` (если точнее, то `StrideTo<T>` и `StrideThrough<T>`, где `T` — это тип данных элементов коллекции).

Уверен, что вы обратили внимание, что по ходу изучения материала появляется все больше новых и порой непонятных типов данных. Дело в том, что архитектура Swift создана так, что для каждой цели используется свой тип данных, и в этом нет ничего страшного. Со временем, активно создавая приложения и изучая справочную документацию, вы будете без каких-либо проблем ориентироваться во всем многообразии доступных возможностей. И уже скоро начнете создавать собственные типы данных.

Использование where в конструкции for-in

Одной из замечательных возможностей оператора `for` является использование ключевого слова `where` для указания дополнительных условий итерации элементов последовательности.

Вернемся к примеру подсчета суммы всех четных чисел от 1 до 10. Для этой цели можно использовать оператор `for` совместно с `where`-условием (листинг 13.43).

Листинг 13.43

```
var result = 0
for i in 1...10 where i % 2 == 0 {
    result += i
}
result // 30
```

После where указано условие, в котором определено, что код в теле оператора будет исполняться только в случае, когда остаток от деления связанного параметра i на 2 равен 0 (то есть число четное).

Также с помощью where можно уйти от использования вложенных друг в друга операторов (например, for в if). В листинге 13.44 показаны два блока: первый с использованием двух операторов, а второй с использованием for и where. При этом обе конструкции функционально идентичны.

Листинг 13.44

```
var isRun = true

// вариант 1
if isRun {
    for i in 1...10 {
        // тело оператора
    }
}
// вариант 2
for i in 1...10 where isRun {
    // тело оператора
}
```

В обоих вариантах код тела оператора будет выполнен 10 раз при условии, что isRun истинно, но при этом вариант № 2 более читабельный.

ПРИМЕЧАНИЕ В данном примере вариант 2 пусть и является более читабельным, но потенциально создает большую нагрузку на компьютер. Если значение isRun будет ложным (false), то вариант 1 единожды проверит его и пропустит вложенный оператор for, в то время как вариант 2 предусматривает выполнение проверки 10 раз (при каждой итерации).

Программирование должно заставлять вас задумываться об оптимизации того, что вы пишете, находя наиболее экономные пути решения поставленных задач.

Многомерные коллекции в конструкции for-in

Если перед вами стоит задача обработки многомерных коллекций, то вы можете с легкость организовать это, а именно вкладывать конструкции for-in друг в друга.

В листинге 13.45 объявляется словарь, содержащий результаты игр одной хоккейной команды в чемпионате. Ключ каждого элемента — это название команды соперника, а значение каждого элемента — массив

результатов ее игр. На консоль выводятся результаты всех игр с указанием команд и итогового счета.

Листинг 13.45

```
// словарь с результатами игр
var resultsOfGames = ["Red Wings":["2:1","2:3"], "Capitals":["3:6","5:
5"],"Penguins":["3:3","1:2"]]
// обработка словаря
for (teamName, results) in resultsOfGames {
    // обработка массива результатов игр
    for oneResult in results {
        print("Игра с \(teamName)  – \(oneResult)")
    }
}
Консоль:
Игра с Capitals – 3:6
Игра с Capitals – 5:5
Игра с Red Wings – 2:1
Игра с Red Wings – 2:3
Игра с Penguins – 3:3
Игра с Penguins – 1:2
```

Параметр `resultsOfGames` — «словарь массивов» с типом `[String: [String]]`. Переменная `teamName` — локальная для родительского оператора `for`, но в ее область видимости попадает вложенный `for-in`, поэтому она может быть использована при выводе значения на консоль.

Использование continue в конструкции for-in

Оператор `continue` предназначен для перехода к очередной итерации, игнорируя следующий за ним код тела оператора.

В листинге 13.46 представлена программа, в которой переменная поочередно принимает значения от 1 до 10, причем когда значение нечетное, оно выводится на консоль.

Листинг 13.46

```
for i in 1...10 {
    if i % 2 == 0 {
     continue
    } else {
     print(i)
    }
}
```

Консоль:

```
1
3
5
7
9
```

Проверка четности значения вновь происходит с помощью операции вычисления остатка от деления на два. Если остаток от деления равен нулю, значит, число четное, и происходит переход к следующей итерации. Для этого используется оператор `continue`.

Использование break в конструкции for-in

Оператор `break` предназначен для досрочного завершения работы цикла. При этом весь последующий код в теле цикла игнорируется.

В листинге 13.47 многократно (потенциально бесконечно) случайным образом вычисляется число в пределах от 1 до 10. Если это число равно 5, то на консоль выводится сообщение с номером итерации и выполнение цикла завершается.

Листинг 13.47

```
import Foundation
for i in 1... {
    let randNum = Int(arc4random_uniform(100))
    if randNum == 5 {
        print("Итерация номер \(i)")
        break
    }
}
```

Консоль:

```
Итерация номер 140
```

Обратите внимание, что в качестве входного множества используется оператор диапазона `1...`. С его помощью цикл будет выполняться до тех пор, пока не будет выполнено условие достижения `break`, то есть пока `randNum` не станет равно `5`.

Вывод в консоль в вашем случае может отличаться от того, что приведено в примере, так как используется генератор случайных чисел.

Функция `arc4random_uniform()` принимает на вход параметр типа `Uint32` и возвращает случайное число в диапазоне от `0` до переданного

значения типа `UInt32`. Возвращаемое случайное число также имеет тип данных `UInt32`, поэтому его необходимо привести к `Int`.

ПРИМЕЧАНИЕ Сейчас мы не будем подробно рассматривать директиву `import`. Пока вам необходимо запомнить лишь то, что она подгружает в программу внешнюю библиотеку, благодаря чему обеспечивается доступ к ее ресурсам. Подробнее об использовании команды import и существующих библиотеках функции будет рассказано в одной из следующих глав.

В данном примере подгружается библиотека Foundation для обеспечения доступа к функции `arc4random_uniform()`, которая предназначена для генерации случайного числа. Если убрать строку `import Foundation`, то Xcode сообщит о том, что функции `arc4random_uniform()` не существует.

Может возникнуть ситуация, когда из внутреннего цикла необходимо прервать выполнение внешнего, для этого в Swift используются метки (листинг 13.48).

Листинг 13.48

```
mainLoop: for i in 1...5 {
    for y in i...5 {
        if y == 4 && i == 2{
            break mainLoop
        }
        print("\(i) – \(y)")
    }
}
```

Консоль:

```
1 – 1
1 – 2
1 – 3
1 – 4
1 – 5
2 – 2
2 – 3
```

Метка представляет собой произвольный набор символов, который ставится перед оператором повторения и отделяется от него двоеточием.

Чтобы изменить ход работы внешнего цикла, после оператора `break` или `continue` необходимо указать имя метки.

13.6. Оператор досрочного выхода guard

Оператор `guard` называется *оператором досрочного выхода*. Подобно оператору `if` он проверяет истинность переданного ему условия. Отличие его в том, что он выполняет код в теле оператора только в том случае, если условие вернуло значение `false`.

СИНТАКСИС

```
guard проверяемое_условие else {
   // тело оператора
}
```

После ключевого слова `guard` следует некоторое проверяемое утверждение. Если утверждение возвращает `true`, то тело оператора игнорируется и управление переходит следующему за `guard` коду.

Если утверждение возвращает `false`, то выполняется код внутри тела оператора.

Для данного оператора существует ограничение: его тело должно содержать один из следующих операторов — `return`, `break`, `continue`, `throw`.

В дальнейшем мы более подробно познакомимся с `guard` и рассмотрим примеры его использования.

14 Опциональные типы данных

Как вы знаете, типы данных обеспечивают хранение информации различных видов: строковой, логической, числовой и т. д. Помимо фундаментальных типов, изученных ранее, в Swift присутствуют также опциональные типы данных. Это одно из важных нововведений языка, которое, вероятно, вы еще не встречали.

Все переменные и константы, которые мы определяли ранее, всегда имели некоторое конкретное значение.

Когда вы объявляете некоторый параметр, например `var name:String`, то с ним обязательного ассоциируется определенное значение, например `«Владимир Высоцкий»`, которое **всегда** возвращается по имени данного параметра. Значение всегда имеет определенный тип, даже если это пустая строка, пустой массив и т. д. Это одна из функций безопасного программирования в Swift: если объявлен параметр определенного типа, то при обращении к нему вы гарантированно получите значение этого типа. Без каких-либо исключений!

В этой главе вы познакомитесь с концепцией опционалов, позволяющих представлять не только значение определенного типа, но и полное отсутствие какого-либо значения.

14.1. Введение в опционалы

Опциональные типы данных, также называемые **опционалами**, — это особый тип, который говорит о том, что параметр либо имеет значение определенного типа, либо вообще не имеет никакого значения. Иногда очень полезно оперировать отсутствием значения. Рассмотрим два примера.

Пример 1

Представьте, что перед вами бесконечная двумерная плоскость (с двумя осями координат). В ходе эксперимента на ней устанавливают точку с координатами `x=0` и `y=0`, которые в коде могут быть представлены либо как два целочисленных параметра (`x:Int` и `y:Int`), либо как кортеж типа (`Int, Int`). В зависимости от ваших потребностей вы можете передвигать точку, изменяя ее координаты. В любой момент времени вы можете говорить об этой точке и получать конкретные значения `x` и `y`.

Что произойдет, если убрать точку с плоскости? Она все еще будет существовать в вашей программе, но при этом не будет иметь координат. Совершенно никаких координат. В данном случае `x` и `y` не могут быть установлены, в том числе и в `0`, так как `0` — это точно такая же координата, как и любая другая.

Данная проблема может быть решена с помощью введения дополнительного параметра (например, `isSet: Bool`), определяющего, установлена ли точка на плоскости. Если `isSet = true`, то можно производить операции с координатами точки, в ином случае считается, что точка не установлена на плоскости. При таком подходе велика вероятность ошибки, то есть необходимо контролировать значение `isSet` и проверять его перед каждой операцией с точкой.

В такой ситуации наиболее верным решением станет использование опционального типа данных в качестве типа значения, определяющего координаты точки. В случае, когда точка находится на плоскости, будут возвращаться конкретные целочисленные значения, а когда она убрана — специальное ключевое слово, определяющее отсутствие координат (а значит, и точки на плоскости).

Пример 2

Ваша программа запрашивает у пользователя его имя, возраст и место работы. Если первые два параметра могут быть определены для любого пользователя, то конкретное рабочее место может отсутствовать. Конечно же, чтобы указать на то, что работы нет, можно использовать пустую строку, но опционалы, позволяющие определить отсутствие значения, являются наиболее правильным решением. Таким образом, обращаясь к переменной, содержащей место работы, вы будете получать либо конкретное строковое значение, либо специальное ключевое слово, сигнализирующее об отсутствии работы.

Самое важное, чего позволяют достичь опционалы, — это исключение неоднозначности. Если значение есть, то оно есть, если его нет, то оно не сравнивается с нулем или пустой строкой, его просто нет.

ПРИМЕЧАНИЕ Важно не путать отсутствие какого-либо значения в опциональном типе данных с пустой строкой или нулем. Пустая строка — это обычный строковый литерал, то есть вполне конкретное значение переменной типа `String`, а ноль — вполне конкретное значение числового типа данных. То же относится и к пустым коллекциям.

У вас мог возникнуть вопрос: как Swift показывает, что в параметре опционального типа отсутствует значение? Для этого используется ключевое слово `nil`. С ним мы, кстати, уже встречались ранее в ходе изучения коллекций.

Рассмотрим практический пример использования опционалов.

Ранее мы неоднократно использовали функцию `Int(_:)` для создания и приведения целочисленных значений. Но не каждый переданный в нее литерал может быть преобразован к целочисленному типу данных: к примеру, строку `"1945"` можно конвертировать в число, а `"Одна тысяча сто десять"` вернуть в виде числа не получится (листинг 14.1).

Листинг 14.1

```
let possibleString = "1945"
let convertPossibleString = Int(possibleString) // 1945

let unpossibleString = "Одна тысяча сто десять"
let convertUnpossibleString = Int(unpossibleString) // nil
```

При конвертации строкового значения `"1945"`, состоящего только из цифр, возвращается число. А во втором случае возвращается ключевое слово `nil`, сообщающее о том, что в результате конвертации не получено никакого целочисленного значения. То есть это не ноль, это не пустая строка, это именно отсутствие значения как такового.

Самое интересное, что в обоих случаях (и при числовом, и при строковом значении переданного аргумента) возвращается значение опционального типа данных. То есть `1945` — это значение не целочисленного, а опционального целочисленного типа данных. Также и `nil` — **в данном примере** это указатель на отсутствие значения в хранилище опционального целочисленного типа.

В этом примере функция `Int(_:)` возвращает опционал, то есть значение такого типа данных, который может либо содержать конкретное значение (целое число), либо не содержать совершенно ничего (`nil`).

Опционалы — это отдельная самостоятельная группа типов данных. Целочисленный тип и опциональный целочисленный тип — это два совершенно разных типа данных. По этой причине опционалы должны иметь собственное обозначение типа. И они его имеют. Убедимся в этом, определив тип данных констант из предыдущего листинга (листинг 14.2).

Листинг 14.2

```
type(of: convertPossibleString) // Optional<Int>.Type

type(of: convertUnpossibleString) // Optional<Int>.Type
```

`Optional<Int>` — это идентификатор **опционального целочисленного типа данных**, то есть значение такого типа может быть либо целым числом, либо отсутствовать полностью. Тип `Int` является базовым для этого опционала, то есть он основан на типе `Int`.

Более того, опциональные типы данных всегда строятся на основе базовых неопциональных. Они могут брать за основу совершенно любой тип данных, включая `Bool`, `String`, `Float` и `Double`, а также типы данных кортежей, ваши собственные типы, типы коллекций и т. д.

Напомню, что опционалы являются самостоятельными типами, отличными от базовых, то есть тип `Int` и тип `Optional<Int>` — это два разных типа данных.

ПРИМЕЧАНИЕ Функция `Int(_:)` не всегда возвращает опционал, а лишь в том случае, если в нее передано не числовое значение. Так, если в `Int(_:)` передается значение типа `Double`, то нет никакой необходимости возвращать опционал, так как при любом значении `Double` оно сможет быть преобразовано в `Int` (что нельзя сказать про преобразование `String` в `Int`).

Далее показано, что приведение `String` и `Double` к `Int` дает значения различных типов данных (`Optional<Int>` и `Int`).

```
var x1 = Int("12")
type(of: x1) // Optional<Int>.Type
var x2 = Int(43.2)
type(of: x2) // Int.Type
```

В общем случае тип данных опционала имеет две формы записи.

СИНТАКСИС

Полная форма записи:

```
Optional<T>
```

Краткая форма записи:

```
T?
```

- `T: Any` — наименование типа данных, на котором основан опционал.

При объявлении параметра, имеющего опциональный тип, **необходимо явно указать его тип данных**. Для этого можно использовать полную форму записи. В листинге 14.3 приведен пример объявления переменной опционального типа, основанного на `Character`.

Листинг 14.3

```
var optionalChar: Optional<Character> = "a"
```

При объявлении опционала Swift также позволяет использовать сокращенный синтаксис. Для этого в конце базового типа необходимо добавить знак вопроса, никаких других элементов не требуется. Таким образом тип `Optional<Int>` может быть переписан в `Int?`, `Optional<String>` в `String?` и в любой другой тип. В листинге 14.4 показан пример объявления опционала с использованием сокращенного синтаксиса.

Листинг 14.4

```
var xCoordinate: Int? = 12
```

В любой момент значение опционала может быть изменено на `nil`. Это можно сделать как при объявлении параметра, так и потом (листинг 14.5).

Листинг 14.5

```
xCoordinate //12
xCoordinate = nil
xCoordinate //nil
```

Переменная `xCoordinate` является переменной опционального целочисленного типа данных `Int?`. Изначально ей было присвоено значение, соответствующее базовому для опционала типу данных, которое позже было заменено на `nil` (то есть значение переменной было уничтожено).

Если объявить переменную опционального типа, но не проинициализировать ее значение, Swift по умолчанию сочтет ее равной `nil` (листинг 14.6).

Листинг 14.6

```
var someOptional: Bool? // nil
```

Для создания опционала помимо явного указания типа также можно использовать функцию `Optional(_:)` , в которую необходимо передать инициализируемое значение требуемого базового типа (листинг 14.7).

Листинг 14.7

```
// опциональная переменная с установленным значением
var optionalVar = Optional ("stringValue")
optionalVar // "stringValue"
print( // уничтожаем значение опциональной переменной
optionalVar = nil // nil
type(of: optionalVar) // Optional<String>.Type
```

Так как в функцию `Optional(_:)` в качестве входного аргумента передано значение типа `String`, то возвращаемое ею значение имеет опциональный строковый тип данных `String?` (или `Optional<String>`, что является синонимом).

Опционалы в кортежах

Так как в качестве базового для опционалов может выступать любой тип данных, вы можете использовать в том числе и кортежи. В листинге 14.8 приведен пример объявления опционального кортежа.

Листинг 14.8

```
var tuple: (code: Int, message: String)? = nil
tuple = (404, "Page not found") // (code 404, message "Page
                                // not found")
```

В этом примере опциональный тип основан на типе кортежа `(Int, String)`.

При необходимости вы можете использовать опционал для отдельных элементов кортежей (листинг 14.9).

Листинг 14.9

```
var tupleWithoptelements: (Int?, Int) = (nil, 100)
tupleWithoptelements.0 // nil
tupleWithoptelements.1 // 100
```

14.2. Извлечение опционального значения

Важно отметить, что нельзя производить прямые операции между значениями опционального и базового типов данных, так как это приведет к ошибке.

Листинг 14.10

```
var a: Int = 4
var b: Int? = 5
a+b // ОШИБКА. Несоответствие типов
```

В переменной `a` хранится значение неопционального типа `Int`, в то время как значение `b` является опциональным (`Int?`).

Типы `Int?` и `Int`, `String?` и `String`, `Bool?` и `Bool` — все это разные типы данных. Для решения проблемы их взаимодействия можно применить прием, называемый **извлечением опционального значения**, или, другими словами, преобразовать опционал в соответствующий ему базовый тип.

Выделяют три способа извлечения опционального значения:

- принудительное извлечение;
- косвенное извлечение;
- операция объединения с `nil` (рассматривается в конце главы).

После извлечения значение опционального типа приводится к базовому, а значит, может взаимодействовать с другими значениями базового типа. Рассмотрим каждый из указанных способов подробнее.

Принудительное извлечение значения

Принудительное извлечение (force unwrapping) преобразует значение опционального типа в значение базового (например, `Int?` в `Int`) с помощью знака восклицания (`!`), указываемого после имени параметра с опциональным значением. Пример принудительного извлечения приведен в листинге 14.11.

Листинг 14.11

```
var optVar: Int? = 12
var intVar = 34
var result = optVar! + 34 // 46

// проверяем тип данных извлеченного значения
type(of: optVar!) // Int.Type
```

Переменная `optVar` является опционалом. Для проведения арифметической операции с целочисленным значением используется принудительное извлечение (после имени переменной указан восклицательный знак). Таким образом, операция сложения производится между двумя неопциональными целочисленными значениями.

Точно такой же подход используется и при работе с типами, отличными от `Int` (листинг 14.12).

Листинг 14.12

```
var optString: String? = "Vasiliy Usov"
var unwrapperString = optString!
print( "My name is \(unwrapperString)" )
```

Консоль

```
My name is Vasiliy Usov
```

При всем удобстве этого способа вам нужно быть крайне осторожными. На рис. 14.1 показано, что происходит, когда производится попытка извлечения несуществующего значения.

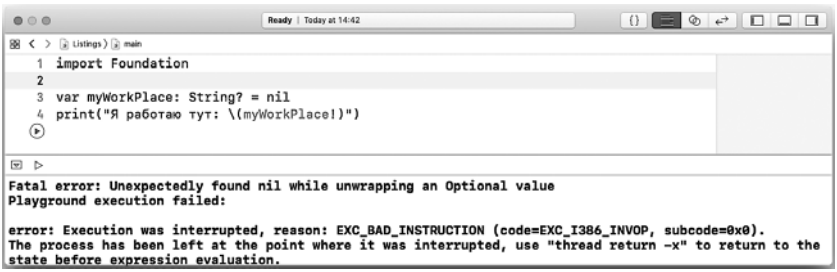

Рис. 14.1. Ошибка при извлечении несуществующего опционального значения

Ошибка возникает из-за того, что переменная не содержит значения (оно соответствует `nil`). Эта досадная неприятность способна привести к тому, что приложение аварийно завершит работу, что, без сомнений, приведет к отрицательным отзывам в AppStore.

При использовании принудительного связывания вы должны быть уверены, что опционал имеет значение.

ПРИМЕЧАНИЕ Далее мы познакомимся с тем, как проверять наличие значения в опционале, прежде чем производить его извлечение.

Косвенное извлечение значения

В противовес принудительному извлечению опционального значения Swift предлагает использовать **косвенное извлечение опционального значения** (`implicitly unwrapping`).

Если вы уверены, что в момент проведения операции с опционалом в нем **всегда** будет значение (не `nil`), то при явном указании типа

данных знак вопроса может быть заменен на знак восклицания. При этом все последующие обращения к параметру необходимо производить без принудительного извлечения, так как оно будет происходить автоматически (листинг 14.13).

Листинг 14.13

```
var wrapInt: Double! = 3.14
// сложение со значением базового типа не вызовет ошибок
// при этом не требуется использовать принудительное извлечение
wrapInt + 0.12 // 3.26
```

Запомните, что отсутствие значения в опционале приведет к ошибке приложения (а это, напоминаю, плохие отзывы пользователей).

14.3. Проверка наличия значения в опционале

Для осуществления проверки наличия значения в опционале его можно сравнить с `nil`. При этом будет возвращено логическое `true` или `false` в зависимости от наличия значения (листинг 14.14).

Листинг 14.14

```
var optOne: UInt? = nil
var optTwo: UInt? = 32

optOne != nil // false
optTwo != nil // true
```

Подобное выражение можно использовать совместно с оператором условия `if`. Если в опционале имеется значение, то в теле оператора оно может быть извлечено без ошибок.

В листинге 14.15 приведен пример, в котором определяется количество положительных оценок, а точнее — пятерок. Если пятерки есть, то вычисляется количество пирожных, которые необходимо приобрести в награду за старания.

Листинг 14.15

```
var fiveMarkCount: Int? = 8
var allCakesCount = 0;
// определение наличия значения
if fiveMarkCount != nil {
    // количество пирожных за каждую пятерку
    let  cakeForEachFiveMark = 2
```

```
    // общее количество пирожных
    allCakesCount = cakeForEachFiveMark * fiveMarkCount!
}
allCakesCount //16
```

Обратите внимание на то, что при вычислении значения `allCakesCount` в теле конструкции `if` используется принудительное извлечение опционального значения переменной `fiveMarkCount`.

ПРИМЕЧАНИЕ Данный способ проверки существования значения опционала работает исключительно при принудительном извлечении опционального значения, так как косвенно извлекаемое значение не может быть равно `nil`, а значит, и сравнивать его с `nil` не имеет смысла.

14.4. Опциональное связывание

В ходе проверки наличия значения в опционале существует возможность одновременного извлечения значения (если оно не `nil`) и инициализации его во временный параметр. Этот способ носит название **опционального связывания** (`optional binding`) и является наиболее корректным способом работы с опционалами.

СИНТАКСИС

```
if let связываемый_параметр = опционал {
    // тело оператора
}
```

В результате опционального связывания создается связанный параметр, в котором при возможности извлекается значение опционала. Если опционал не равен `nil`, то будет выполнен код в теле оператора, в котором значение опционала будет доступно через связанный параметр.

Пример

```
if let userName = userLogin {
    print("Имя: \(userName)")
}else {
    print("Имя не введено")
}
// userLogin - опционал
type(of: userLogin) // Optional<String>.Type
```

ПРИМЕЧАНИЕ Напомню, что область видимости определяет, где в коде доступен некоторый объект. Если этот объект является глобальным, то он доступен в любой точке программы (его область видимости не ограничена). Если объект является локальным, то он доступен только в том блоке кода (и во всех вложенных в него блоках), для которого он является локальным. Вне этого блока объект просто не виден.

В листинге 14.16 показан пример использования опционального связывания.

Листинг 14.16

```
var markCount: Int? = 8
// определение наличия значения
if let marks = markCount {
    print("Всего \(marks) оценок")
}
```

Консоль

```
Всего 8 оценок
```

Так как опционал `markCount` не `nil`, в ходе опционального связывания происходит автоматическое извлечение его значения с последующей инициализацией в локальную константу `marks`.

Переменная, создаваемая при опциональном связывании, локальна для оператора условия, поэтому использовать ее можно только внутри данного оператора. Если бы в переменной `markCount` не существовало значения, то тело оператора условия было бы проигнорировано.

Вы можете не ограничиваться одним опциональным связыванием в рамках одного оператора `if` (листинг 14.17).

Листинг 14.17

```
var pointX: Int? = 10
var pointY: Int? = 3

if let x = pointX, let y = pointY {
   print("Точка установлена на плоскости")
}
```

Консоль

```
Точка установлена на плоскости
```

В этом примере проверяется наличие значения в обеих переменных. Если бы хоть одна из переменных соответствовала `nil`, то вывод на консоль оказался бы пуст.

Во время написания последнего листинга вы получили от Xcode уведомление (желтого цвета) о том, что объявленные в ходе опционального связывания константы не используются в теле оператора, вследствие чего они могут быть заменены нижним подчеркиванием _ (рис. 14.2).

Ранее мы уже неоднократно встречались с нижним подчеркиванием, позволяющим игнорировать определенные элементы или значения.

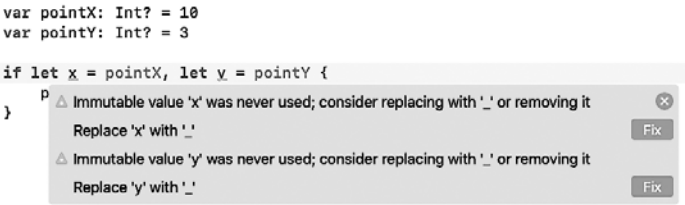

Рис. 14.2. Предупреждение от Xcode

Напомню, что оно может заменять имена параметров в тех случаях, когда в их объявлении нет необходимости. В данном примере опциональное связывание требуется лишь с целью определения наличия значений в опционалах, при этом внутри блока кода оператора условия созданные параметры не используются. Поэтому можно последовать совету среды разработки и заменить имена констант на нижнее подчеркивание. Код в этом случае будет работать без ошибок (листинг 14.18).

Листинг 14.18

```
if let _ = pointX, let _ = pointY {
   print("Точка установлена на плоскости")
}
```

При необходимости вы можете использовать параметры, объявленные с помощью опционального связывания, для указания условий в рамках того же условия оператора `if`, где они были определены (листинг 14.19).

Листинг 14.19

```
if let x = pointX, x > 5 {
    print("Точка очень далеко от вас ")
}
```

Консоль:

```
Точка очень далеко от вас
```

14.5. Опциональное связывание как часть оптимизации кода

Рассмотрим еще один вариант грамотного применения опционального связывания на примере уже полюбившейся нам функции `Int(_:)`.

Все любят драконов! А все драконы любят золото! Представьте, что у вас есть группа драконов, у большинства из которых есть свой сундук с золотом, а количество золотых монет в каждом из этих сундуков разное. В любой момент времени может потребоваться знать общее количество монет во всех сундуках. Внезапно к вам поступает новый дракон, его золото тоже должно быть учтено.

Напишем код, в котором определяется количество монет в сундуке нового дракона (если, конечно, у него есть сундук), после чего оно суммируется с общим количеством золота (листинг 14.20).

Листинг 14.20

```
/* переменная типа String,
содержащая количество золотых монет
в сундуке нового дракона */
var coinsInNewChest = "140"
/* переменная типа Int,
в которой будет храниться общее
количество монет у всех драконов */
var allCoinsCount = 1301
// проверяем существование значения
if Int(coinsInNewChest) != nil{
    allCoinsCount += Int(coinsInNewChest)!
} else {
    print("У нового дракона отсутствует золото")
}
```

ПРИМЕЧАНИЕ У вас мог возникнуть вопрос, почему в качестве количества монет в сундуке не используется значение целочисленного типа. Причин тому три:

- это пример, который позволяет вам подробнее рассмотреть работу опционалов;
- в интерфейсе мнимой программы, вполне вероятно, будет находиться текстовое поле, в котором будет вводиться **строковое** значение, содержащее количество монет;
- монеты могут отсутствовать по причине отсутствия сундука, а 0 в качестве значения говорит о том, что сундук есть, но монет в нем нет.

На первый взгляд все очень просто и логично, и в результате значение переменной `allCoinsCount` станет равно `1441`. Но обратите внимание, что `Int(coinsInNewChest)` используется дважды:

- при сравнении с `nil`;
- при сложении с переменной `allCoinsCount`.

В результате происходит бесцельная трата процессорного времени, так как одна и та же функция выполняется дважды. Можно избежать данной ситуации, заранее создав переменную `coins`, в которую будет

извлечено значение опционала. Данную переменную необходимо использовать в обоих случаях вместо вызова функции `Int(_:)` (листинг 14.21).

Листинг 14.21

```
var coinsInNewChest = "140"
var allCoinsCount = 1301
/* извлекаем значение опционала
в новую переменную */
var coins = Int(coinsInNewChest)
/* проверяем существование значения
с использованием созданной переменной */
if coins != nil{
    allCoinsCount += coins!
} else {
    print("У нового дракона отсутствует золото")
}
```

Несмотря на то что приведенный код в полной мере решает поставленную задачу, у него есть один недостаток: созданная переменная `coins` будет существовать (а значит, и занимать оперативную память) даже после завершения работы условного оператора, хотя в ней нет необходимости.

Необходимо всеми доступными способами избегать бесполезного расходования ресурсов компьютера, к которым относится и процессорное время, и оперативная память.

Чтобы избежать расходования памяти, можно использовать опциональное связывание, так как после выполнения оператора условия созданная при связывании переменная автоматически удалится (листинг 14.22).

Листинг 14.22

```
var coinsInNewChest = "140"
var allCoinsCount = 1301
/* проверяем существование значения
с использованием опционального связывания */
if let coins = Int(coinsInNewChest){
    allCoinsCount += coins
} else {
    print("У нового дракона отсутствует золото")
}
```

Мы избавились от повторного вызова функций `Int(_:)` и расходования оперативной памяти, получив красивый и оптимизированный код. В данном примере вы, вероятно, не ощутите увеличения скорости

работы программы, но при разработке на языке Swift более сложных приложений для мобильных или стационарных устройств данный подход позволит получать вполне ощутимые результаты.

14.6. Оператор объединения с nil

Говоря об опционалах, осталось рассмотреть еще один способ извлечения значения, известный как операция объединения с `nil` (**nil coalescing**). С помощью оператора `??` (называемого **оператором объединения с nil**) возвращается либо значение опционала, либо значение по умолчанию (если опционал равен `nil`).

СИНТАКСИС

```
let имя_параметра = имя_опционала ?? значение_по умолчанию
```

- `имя_параметра:T` — имя нового параметра, в который будет извлекаться значение опционала.
- `имя_опционала:Optional<T>` — имя параметра опционального типа, из которого извлекается значение.
- `значение_по умолчанию:T` — значение, инициализируемое новому параметру в случае, если опционал равен `nil`.

Если опционал не равен `nil`, то опциональное значение извлекается и инициализируется объявленному параметру.

Если опционал равен `nil`, то в параметре инициализируется значение, расположенное справа от оператора `??`.

Базовый тип опционала и тип значения по умолчанию должны быть одним и тем же типом данных.

Вместо оператора `let` может быть использован оператор `var`.

В листинге 14.23 показан пример использования оператора объединения с `nil` для извлечения значения.

Листинг 14.23

```
var optionalInt: Int? = 20
var mustHaveResult = optionalInt ?? 0 // 20
```

Таким образом, константе `mustHaveResult` будет проинициализировано целочисленное значение. Так как в `optionalInt` есть значение, оно будет извлечено и присвоено константе `mustHaveResult`. Если бы `optionalInt` был равен `nil`, то `mustHaveResult` принял бы значение `0`.

Код из предыдущего листинга эквивалентен приведенному в листинге 14.24.

Листинг 14.24

```
var optionalInt: Int? = 20
var mustHaveResult: Int = 0
if let unwrapped = optionalInt {
    mustHaveResult = unwrapped
} else {
    mustHaveResult = 0
}
```

Наиболее безопасными способами извлечения значений из опционалов являются опциональное связывание и `nil coalescing`. Старайтесь использовать именно их в своих приложениях.

15 Функции

Мы уже неоднократно встречались с функциями, использовали предлагаемые ими возможности. Одной из часто используемых нами функций была `print(_:)`, с помощью которой осуществляется вывод информации на отладочную консоль.

В этой главе мы глубже рассмотрим понятие функции, принципы ее работы, а также научимся создавать собственные, благодаря которым программный код заиграет новыми красками.

15.1. Введение в функции

Знакомство с функциями мы начнем с описания их свойств и характеристик.

Функция:

- группирует исполняемый программный код в единый контейнер;
- имеет собственное имя;
- может быть многократно вызвана с помощью имени;
- может принимать входные аргументы;
- может возвращать значение как результат исполнения сгруппированного в ней кода;
- имеет собственный функциональный тип данных;
- может быть записана в параметр (переменную или константу) и таким образом передана;
- объявляется с помощью специального синтаксиса.

Сложно? Это только кажется! И вы убедитесь в этом, когда начнете создавать функции самостоятельно.

На вопрос о том, когда необходимо создавать функции, есть замечательный ответ: «Функцию стоит объявлять тогда, когда некоторый

программный код может быть многократно использован. С ее помощью исключается дублирование кода, так как она позволяет не писать его дважды».

ПРИМЕЧАНИЕ У программистов существует шутка, гласящая, что любой код мечтает стать функцией. Она хотя и «бородатая», но все еще актуальная.

Готовая функция — это своеобразный черный ящик, у которого скрыта внутренняя реализация. Вам важно лишь то, для чего она предназначена, что принимает на вход и что возвращает.

Рассмотрим пример из реального физического мира. В качестве функции может выступать соковыжималка. Вы, вероятно, ничего не знаете о ее внутреннем устройстве, но обладаете информацией о ее предназначении (выжимать сок), входных аргументах (свежие фрукты и овощи) и возвращаемом значении (свежевыжатый сок). Если говорить языком Swift, то использование соковыжималки могло бы выглядеть следующим образом:

```
// создаем яблоко
var apple = Apple()
// используем соковыжималку
var juice: AppleJuice = juicer( apple, apple, apple )
```

ПРИМЕЧАНИЕ Имена типов данных, функции и остальные элементы в данном примере являются абстрактными, предназначенными лишь для демонстрации идеи использования функций. Попытка использования данного кода в Xcode приведет к ошибке.

В данном примере вызывается функция `juicer(_:)`, принимающая на вход три яблока (значения типа `Apple`). В результате своей работы она возвращает яблочный сок (значение типа `AppleJuice`).

Перейдем непосредственно к созданию (объявлению) функций. Как говорилось ранее, для этого используется специальный синтаксис.

СИНТАКСИС

```
func имяФункции (входные_параметры) -> тип {
    // тело функции
}
```

- `имяФункции` — имя объявляемой функции, по которому она сможет быть вызвана.
- `входные параметры` — список аргументов функции с указанием их имен и типов.
- `Тип` — тип данных значения, возвращаемого функцией. Если функция ничего не возвращает, то данный элемент может быть опущен.

Объявление функции начинается с ключевого слова `func`.

За `func` следует имя создаваемой функции. Оно используется при каждом ее вызове в вашем коде и должно быть записано в нижнем верблюжьем регистре. Например:

```
func myFirstFunc
```

Далее в скобках указываются входные аргументы (также называемые входными параметрами). Список входных параметров заключается в круглые скобки и состоит из разделенных запятыми элементов. Каждый отдельный элемент описывает один входной параметр и состоит из имени и типа этого параметра, разделенных двоеточием. Входные параметры позволяют передать в функцию значения, которые ей требуются для корректного выполнения возложенных на нее задач. Количество входных параметров может быть произвольным (также они могут вовсе отсутствовать). Например:

```
func myFirstFunc
 (a: Int, b: String)
```

Указанные параметры являются локальными для функции, таким образом, `a` и `b` будут существовать только в пределах ее тела. По окончании ее работы данные параметры будут уничтожены и станут недоступными.

Далее, после списка параметров, может быть указан тип возвращаемого значения. Для этого используется стрелка (`->`), после которой следует имя конкретного типа данных. В качестве типа можно задать любой фундаментальный тип, тип массива или кортежа или любой другой. Например:

```
func myFirstFunc
 (a: Int, b: String)
 -> String
```

или

```
func myFirstFunc
 (a: Int, b: String)
 -> [(String,Int)?]
```

Если функция не должна возвращать никакого значения, то на это можно указать тремя способами:

- с помощью пустых скобок `()`, например:

```
func myFirstFunc
(a: Int, b: String
-> ()
```

- с помощью ключевого слова `Void`, например:

```
func myFirstFunc
(a: Int, b: String
-> Void
```

- не указывать тип вовсе, например:

```
func myFirstFunc
(a: Int, b: String
```

Тело функции содержит весь исполняемый код и заключается в фигурные скобки. Оно содержит в себе всю логику работы.

Если функция возвращает какое-либо значение, то в ее теле должен присутствовать оператор `return`, за которым следует возвращаемое значение. После выполнения программой оператора `return` работа функции завершается и происходит выход из нее. Например:

```
func myFirstFunc
 (a: Int, b: String)
–> String {
 return String(someValue) + anotherValue
}
```

В данном случае тело функции состоит всего из одного выражения, в котором содержится оператор `return`. После его выполнения функция вернет сформированное значение и завершит свою работу.

Функция может содержать произвольное количество операторов `return`. При достижении первого из них будет произведено завершение ее работы.

В представленных ранее примерах объявление функции разнесено на разные строки для удобства восприятия кода. Вам не обязательно делать это, можете писать элементы объявления функции в одну строку. Например:

```
func myFirstFunc(a: Int, b: String) –> String {
 return String(someValue) + anotherValue
}
```

Процесс обращения к объявленной функции по ее имени называется *вызовом функции*.

Рассмотрим пример создания функции, которая не имеет входных и выходных данных.

Предположим, что вам требуется многократно выводить на консоль один и тот же текст. Для реализации этого можно объявить новую функцию, которая при обращении к ней будет производить необходимую операцию (листинг 15.1).

Листинг 15.1

```
func printMessage(){
    print("Сообщение принято")
}
// вызываем функцию по ее имени
printMessage()
printMessage()
```

Консоль

```
Сообщение принято
Сообщение принято
```

Для вывода текста на консоль вы вызываете функцию `printMessage()`, просто написав ее имя с круглыми скобками. Данная функция не имеет каких-либо входных параметров или возвращаемого значения. Она всего лишь выводит на консоль необходимый текст.

ПРИМЕЧАНИЕ Вывод информации на консоль с помощью `print(_:)` не является возвращаемым функцией значением. Возвращаемое значение может быть проинициализировано параметру.

15.2. Входные аргументы и возвращаемое значение

Функция может принимать аргументы (параметры) в качестве входных значений и возвращать результат своей работы (возвращаемое значение). И для входных, и для возвращаемого значений должны быть определены типы данных.

Рассмотрим пример. Требуется многократно производить сложение двух целочисленных чисел и возвращать полученный результат в виде значения типа `Int`. Правильным подходом будет объявление функции, производящее данные действия. В качестве входного аргумента будут служить складываемые числа, а результат операции будет возвращаемым значением.

Конечно, это очень простой пример, и куда лучше написать `a+b` для сложения двух операндов, а не городить функцию. Но для рассмотрения учебного материала он подходит как нельзя лучше. Но если вычисляемое выражение значительно сложнее, например `a+b*b+a*(a+b)*(a+b)`, то создание функции будет оправданно.

Входные аргументы

Реализуем описанную выше задачу, но при этом исключим из нее требование возвращать результат сложения. Пусть результат операции выводится на отладочную консоль (листинг 15.2).

Листинг 15.2

```
func sumTwoInt(a: Int, b: Int){
    print("Результат операции - \(a+b)")
}
sumTwoInt(a: 10, b: 12)
```

Консоль

```
Результат операции - 22
```

Функция `sumTwoInt(a:b:)` имеет два входных параметра типа `Int` — `a` и `b`.

Обратите внимание, что все входные аргументы должны иметь значения, поэтому попытка вызвать функцию, передав в нее лишь один параметр или не передав их вовсе, завершится ошибкой.

ПРИМЕЧАНИЕ Типы и имена входных аргументов вам подскажет Xcode, а точнее — механизм автодополнения кода.

Внешние имена входных аргументов

Аргументы `a` и `b` функции `sumTwoInt(a:b:)` используются как при вызове функции, так и в ее теле. Swift позволяет указать внешние имена параметров, которые будут использоваться при вызове функции (листинг 15.3).

Листинг 15.3

```
func sumTwoInt(num1 a: Int, num2 b: Int){
    print("Результат операции - \(a+b)")
}
sumTwoInt(num1: 10, num2: 12)
```

Теперь при вызове функции `sumTwoInt(num1:num2:)` необходимо указывать значения не для безликих `a` и `b`, а для более-менее осмысленных `num1` и `num2`. Данный прием очень полезен, так как позволяет задать понятные и соответствующие контексту названия входных аргументов, но при этом сократить количества кода в теле, используя краткие внутренние имена.

Если внешнее имя заменить на символ нижнего подчеркивания (`_`), то при вызове функции имя параметра вообще не потребуется указывать (листинг 15.4).

Листинг 15.4

```
func sumTwoInt(_ a: Int, _ b: Int){
    print("Результат операции - \(a+b)")
}
sumTwoInt(10, 12)
```

Примечание Внешние имена могут быть заданы для произвольных аргументов, не обязательно указывать их для всех.

Возвращаемое значение

Доработаем функцию `sumTwoInt(_:_:)` таким образом, чтобы она не только выводила сообщение на консоль, но и возвращала результат сложения. Для этого необходимо выполнить два требования:

1. Должен быть указан тип возвращаемого значения.
2. Должен быть использован оператор `return` в теле функции с возвращаемым значением в качестве операнда.

Так как результат операции сложения — целое число, в качестве типа данных выходного значения необходимо указать `Int` (листинг 15.5).

Листинг 15.5

```
func sumTwoInt(_ a: Int, _ b: Int) -> Int{
    let result = a + b
    print("Результат операции - \(result)")
    return result
}
var result = sumTwoInt(10, 12) // 22
```

Консоль

```
Результат операции - 22
```

Возвращенное с помощью оператора `return` значение может быть записано в произвольный параметр вне функции.

Обратите особое внимание на то, что в теле функции объявляется константа `result`, а после функции — переменная с таким же именем. Это два разных и независимых параметра! Все, что объявляется в теле функции, является локальным для нее и уничтожается после завершения ее работы. Таким образом, в теле функции константа используется для вывода информации на консоль и совместно с оператором `return`, а вне функции в переменную `result` записывается возвращенное функцией значение.

Изменяемые копии входных аргументов

Все входные параметры функции — константы. При попытке изменения их значения внутри тела функции происходит ошибка. При необходимости изменения переданного входного значения внутри функции потребуется создать новую переменную и присвоить переданное значение ей (листинг 15.6).

Листинг 15.6

```
func returnMessage(code: Int, message: String) -> String {
    var mutableMessage = message
    mutableMessage += String(code)
    return mutableMessage
}
var myMessage = returnMessage(code: 200, message: "Код сообщения - ")
```

Функция `returnMessage(code:message:)` получает на вход два аргумента: `code` и `message`. В ее теле создается изменяемая копия `message`, которая без каких-либо ошибок модифицируется, после чего возвращается.

Сквозные параметры

Приведенный способ модификации значений аргументов позволяет получать доступ к изменяемому значению только в пределах тела самой функции. Для того чтобы была возможность модификации входных аргументов с сохранением измененных значений после окончания работы функции, необходимо использовать *сквозные параметры*.

Чтобы преобразовать входной параметр в сквозной, перед описанием его типа необходимо указать модификатор `inout`. Сквозной параметр передается в функцию, изменяется в ней и сохраняет свое значение при завершении работы функции, заменяя собой исходное значение. При вызове функции перед передаваемым значением аргумента необходимо ставить символ амперсанд (`&`), указывающий на то, что параметр передается по ссылке.

Функция в листинге 15.7 обеспечивает обмен значениями двух внешних параметров.

Листинг 15.7

```
func changeValues(_  a: inout Int,  _ b: inout Int) -> () {
    let tmp = a
    a = b
    b = tmp
}
var x = 150, y = 45
changeValues(&x, &y)
x // 45
y // 150
```

Функция принимает на входе две переменные, `a` и `b`. Эти переменные передаются в функцию как сквозные параметры, что позволяет изме-

нить их значения внутри функции и сохранить эти изменения после завершения ее работы.

ПРИМЕЧАНИЕ В качестве сквозного параметра может выступать только переменная. Константы или литералы нельзя передавать, так как они являются неизменяемыми.

Функция в качестве входного аргумента

Вы можете использовать возвращаемое некоторой функцией значение в качестве значения входного аргумента другой функции. Самое важное, чтобы тип возвращаемого значения функции совпадал с типом входного параметра.

В листинге 15.8 используется объявленная ранее функция `returnMessage(code:message:)`, возвращающая значение типа `String`.

Листинг 15.8

```
// используем функцию в качестве значения
print( returnMessage(code: 400, message: "Сервер недоступен. Код
сообщения - ") )
```

Консоль

```
Сервер недоступен. Код сообщения - 400
```

Уже известная нам функция `print(_:)` принимает на входе строковый литерал типа `String`. Так как функция `returnMessage(code:message:)` возвращает значение этого типа, она может быть указана в качестве входного аргумента для `print(_:)`.

Входной параметр с переменным числом аргументов

В некоторых ситуациях необходимо, чтобы функция получала неизвестное заранее число однотипных аргументов. Мы уже встречались с таким подходом при использовании `Array(arrayLiteral:)`, когда заранее неизвестно, сколько элементов будет содержать аргумент `arrayLiteral`. Такой тип входного аргумента называется *вариативным*.

Вариативный аргумент обозначается в списке входящих параметров с указанием оператора диапазона `...` сразу после типа. Значения для этого аргумента при вызове функции задаются через запятую.

Рассмотрим пример из листинга 15.9. Представьте, что удаленный сервер на каждый запрос отправляет вам несколько ответов. Каждый ответ — это целое число, но их количество может быть различным.

Вам необходимо написать функцию, которая принимает на входе все полученные ответы и выводит их на консоль.

Листинг 15.9

```
func printRequestString(codes: Int...) -> () {
    var codesString = ""
    for oneCode in codes {
        codesString += String(oneCode) + " "
    }
    print("Получены ответы  — \(codesString)")
}
printRequestString(codes: 600, 800, 301)
printRequestString(codes: 101, 200)
```

Консоль

```
Получены ответы  — 600 800 301
Получены ответы  — 101 200
```

Параметр `codes` может содержать произвольное количество значений указанного типа. Внутри функции он трактуется как последовательность (`Sequence`), поэтому его можно обработать с помощью конструкции `for-in`.

У одной функции может быть только один вариативный параметр, и он должен находиться в самом конце списка входных аргументов.

Кортеж в качестве возвращаемого значения

Функция может возвращать значения любого типа данных. Отдельно отмечу, что и кортежи могут быть использованы для этого, так как с их помощью можно с легкостью вернуть сразу несколько значений (возможно, именно этого вам не хватало в других языках программирования, лично мне не хватало ☺).

Представленная в листинге 15.10 функция принимает на вход код ответа сервера и, в зависимости от того, к какому диапазону относится переданный код, возвращает кортеж с его описанием.

Листинг 15.10

```
func getCodeDescription(code: Int) -> (Int, String){
    let description: String
    switch code {
    case 1...100:
        description = "Error"
    case 101...200:
        description = "Correct"
```

```
    default:
        description = "Unknown"
    }
    return (code, description)
}
getCodeDescription(code: 150) // (150, "Correct")
```

В качестве типа возвращаемого значения функции `getCodeDescription(code:)` указан тип кортежа, содержащего два значения: код и его описание.

Функцию `getCodeDescription(code:)` можно улучшить, если указать не просто тип возвращаемого кортежа, а названия его элементов (прямо в типе возвращаемого функцией значения) (листинг 15.11).

Листинг 15.11

```
func getCodeDescription(code: Int)
    -> (code: Int, description: String){
        let description: String
        switch code {
        case 1...100:
            description = "Error"
        case 101...200:
            description = "Correct"
        default:
            description = "Unknown"
        }
        return (code, description)
}
let request = getCodeDescription(code: 45)
request.description // "Error"
request.code // 45
```

Полученное в ходе работы `getCodeDescription(code:)` значение записывается в константу `request`, у которой появляются свойства `description` и `code`, что соответствует именам элементов возвращаемого кортежа.

Значение по умолчанию для входного аргумента

Напомню, что все входные аргументы должны обязательно иметь значения. Ранее для этого мы указывали их при вызове функции. Но существует возможность определить значения по умолчанию, которые позволяют не указывать значения при вызове.

Другими словами: если вы передали значение входного аргумента, то оно будет использовано в теле функции; если вы не передали значение

аргумента, для него будет использовано значение по умолчанию. Значение по умолчанию указывается при объявлении функции в списке входных аргументов для каждого параметра отдельно.

Доработаем объявленную ранее функцию `returnMessage(code:message:)` таким образом, чтобы была возможность не передавать значение аргумента `message`. Для этого укажем значение по умолчанию (листинг 15.12).

Листинг 15.12

```
func returnMessage(code: Int, message: String = "Код - ") -> String {
    var mutableMessage = message
    mutableMessage += String(code)
    return mutableMessage
}
returnMessage(code: 300) //"Код - 300"
```

Как вы можете видеть, при вызове `returnMessage(code:message:)` не передается значение для аргумента `message`. Это стало возможным благодаря установке значения по умолчанию `"Код - "` в списке входных параметров.

15.3. Функциональный тип

Одно из свойств функции заключается в том, что она может быть записана в параметр и с его помощью передана. Но как такое возможно, если у любого параметра, как мы знаем, должен быть определенный тип данных? Возможно, я вас удивлю, но любая функция имеет свой функциональный тип данных, который указывает на типы входных и выходного значений.

Просто функциональный тип

Если функция ничего не принимает и не возвращает, то ее тип указывается двумя парами круглых скобок, разделенных стрелкой:

```
()->()
```

В листинге 15.13 приведен пример функции с типом `()->()`, то есть не имеющей ни входных, ни выходных параметров.

Листинг 15.13

```
func printErrorMessage(){
    print("Произошла ошибка")
}
```

В первых скобках функционального типа всегда описываются типы данных входных параметров, а **вместо** вторых указывается тип данных выходного значения (если, конечно, оно существует). Если функция принимает на вход массив целочисленных значений, а возвращает опциональное строковое значение, то ее тип данных будет выглядеть следующим образом:

```
([Int]) -> String?
```

ПРИМЕЧАНИЕ Обратите внимание, что при наличии возвращаемого значения оно указывается вместо круглых скобок, а не в них.

В левой части функционального типа указываются типы входных параметров, в правой — тип выходного значения.

Сложный функциональный тип

В некоторых случаях выходное значение функции также является функцией, которая, в свою очередь, может возвращать значение. В результате этого функциональный тип становится сложным, то есть содержащим несколько указателей на возвращаемое значение (несколько стрелок `->`).

В самом простом варианте функция, возвращающая другую функцию, которая ничего не возвращает, будет иметь функциональный тип, состоящий из трех пар круглых скобок:

```
() -> () -> ()
```

Представим, что некоторая функция принимает на вход значение типа `Int` и возвращает функцию, которая принимает на вход значение типа `String`, и возвращает значение типа `Bool`. Ее функциональный тип будет выглядеть следующим образом:

```
(Int) -> (String) -> Bool
```

Каждый блок, описывающий типы данных входных аргументов, заключается в круглые скобки. Таким образом можно определить, где начинается функциональный тип очередной функции.

Но функция может не только возвращаться другой функцией, но и передаваться в качестве входного аргумента. Далее приведен пример, в котором функция принимает на вход целое число и другую функцию, а возвращает логическое значение.

```
(Int, (Int)->()) -> Bool
```

Функция, которая передается в качестве входного параметра, имеет тип `(Int)->()`, то есть она сама принимает целочисленное значение, но не возвращает ничего.

15.4. Функция в качестве входного и возвращаемого значений

Возвращаемое значение функционального типа

Так как функция имеет определенный функциональный тип, его можно использовать для того, чтобы указать возвращаемое функцией значение. Так функция может вернуть другую функцию.

В листинге 15.14 объявлена функция `returnPrintTextFunction()`, которая возвращает значение функционального типа `()->()`, то есть другую функцию.

Листинг 15.14

```
// функция вывода текста на консоль
func printText() {
    print("Функция вызвана")
}
// функция, которая возвращает функцию
func returnPrintTextFunction() -> () -> () {
        return printText
}
print("шаг 1")
let newFunctionInLet = returnPrintTextFunction()
print("шаг 2")
newFunctionInLet()
print("шаг 3")
```

Консоль

```
шаг 1
шаг 2
Функция вызвана
шаг 3
```

Для возвращения функции другой функцией достаточно указать ее имя (без скобок) после оператора `return`. Тип возвращаемого значения `returnPrintTextFunction()` соответствует собственному типу `printText()`.

В результате инициализации значения константе `newFunctionInLet` ее тип данных неявно определяется как `()->()`, а сама она хранит в себе

функцию, которую можно вызывать, указав имя хранилища с круглыми скобками после него. На рис. 15.1 отображено справочное окно, описывающее параметр `newFunctionInLet`, в котором хранится функция `printText()`.

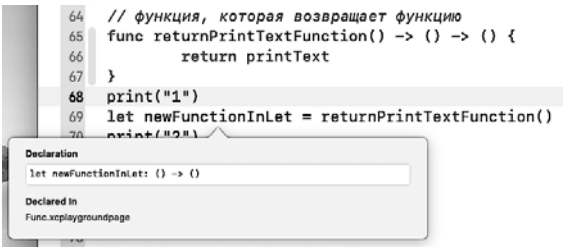

Рис. 15.1. Справочное окно для константы, хранящей функцию в качестве значения

Обратите внимание на вывод на отладочной консоли. Так как строка `"Функция вызвана"` находится между шагами 2 и 3, а не между шагами 1 и 2, можно говорить о том, что функция вызывается не в ходе инициализации значения константе `newFunctionInLet`, а именно в результате выражения `newFunctionInLet()`.

Входное значение функционального типа

Как уже говорилось, функции могут выступать в качестве входных аргументов. Переданную таким образом функцию можно будет использовать в теле той функции, в которую она была передана. Для этого необходимо указать корректный функциональный тип входного аргумента и в качестве его значения указать имя передаваемой функции.

Напишем функцию `generateWallet(walletLength:)`, которая случайным образом генерирует массив банкнот, каждая из которых представлена целым числом разрешенного номинала. Функция должна принимать на вход требуемое количество банкнот в кошельке.

Также реализуем функцию с именем `sumWallet`, которая может принять на вход `generateWallet(walletLength:)`, после чего высчитывает и возвращает сумму всех купюр в кошельке (листинг 15.15).

Листинг 15.15

```
import Foundation
// функция генерации случайного массива банкнот
```

```
func generateWallet(walletLength: Int) -> [Int] {
        // существующие типы купюр
        let typesOfBanknotes = [50, 100, 500, 1000, 5000]
        // массив купюр
        var wallet: [Int] = []
        // цикл генерации массива случайных купюр
        for _ in 1...walletLength {
            let randomIndex = Int( arc4random_uniform( UInt32(
typesOfBanknotes.count-1 ) ) )
            wallet.append( typesOfBanknotes[randomIndex] )
        }
        return wallet
}
// функция подсчета денег в кошельке
func sumWallet(banknotsFunction wallet: (Int)->[Int],
               walletLength: Int )
    -> Int? {
        // вызов переданной функции
        let myWalletArray = wallet( walletLength )
        var sum: Int = 0
        for oneBanknote in myWalletArray {
            sum += oneBanknote
        }
        return sum
}
// передача функции в функцию
sumWallet(banknotsFunction: generateWallet, walletLength: 20) // 6900
```

Значение в области результатов, вероятно, будет отличаться от того, что показано в примере. Это связано с использованием глобальной функции `arc4random_uniform()`, генерирующей и возвращающей случайное число.

Функция `generateWallet(walletLength:)` создает массив купюр такой длины, которая передана ей в качестве входного параметра. В массиве `typesOfBanknotes` содержатся все доступные (разрешенные) номиналы купюр. Суть работы функции такова: купюра случайным образом изымается из массива `typesOfBanknotes`, после чего она помещается в массив-кошелек `wallet`, который является возвращаемым значением. Обратите внимание, что в цикле `for` вместо переменной используется символ нижнего подчеркивания. С этим замечательным заменителем переменных мы уже встречались не раз. В данном случае он заменяет собой создаваемый в цикле параметр, так как внутри цикла он не используется. В результате не выделяется дополнительная память, что благоприятно влияет на расходуемые ресурсы компьютера.

В качестве типа входного параметра `banknotsFunction` функции `sumWallet(banknotsFunction:walletLength:)` указан функциональный тип `(Int)->[Int]`. Он соответствует типу функции `generateWallet(walletLength:)`.

При вызове `sumWallet(banknotsFunction:walletLength:)` необходимо указать лишь имя передаваемой функции без фигурных скобок.

Чего мы добились таким образом? Того, что функция `sumWallet(banknotsFunction:walletLength:)` может принять на вход не только `generateWallet(walletLength:)`, но и любую другую функцию с соответствующим типом. К примеру, можно реализовать функцию `get1000wallet(walletLength:)`, возвращающую массив указанной длины из тысячных купюр, после чего передать ее в качестве входного аргумента в `sumWallet(banknotsFunction:walletLength:)`.

В следующей главе вы увидите всю мощь, которую дают нам входные и выходные аргументы функционального типа.

Параметры функционального типа для ленивых вычислений

Использование значений функционального типа является одним из простейших примеров ленивых вычислений (с ними довольно подробно мы познакомимся в одной из будущих глав). Если кратко, то ленивые вычисления позволяют получить некоторое значение не в момент передачи параметра, а при попытке доступа к хранящемуся в нем значению. Для того чтобы лучше понять это, рассмотрим следующий пример.

У вас есть две функции, каждая из которых предназначена для вычисления некоторой математической величины. Обе функции являются ресурсозатратными, то есть при работе занимают продолжительное процессорное время и значительный объем памяти. Первая из них высчитывает указанный знак после запятой в числе π, а вторая — первую цифру числа с указанным порядковым номером в последовательности Фибоначчи. Помимо этого, существует третья функция, которая принимает на вход оба числа и в зависимости от внутренней логики использует только одно из переданных значений.

В самом общем виде вы могли бы использовать функции примерно следующим образом:

```
// порядковый номер числа, которое нужно получить
let n = 1000000
//передаем значения в главную функцию
returnSomeNum( getPiNum(n), getFibNum(n) )
```

Функция `returnSomeNum(_:_:)` имеет функциональный тип `(Int, Int)->Int`. В ней два входных целочисленных параметра, но внутри своей реализации она использует только один из них (об этом сказано в условии выше), получается, что ресурсы, использованные на получение второго числа, потрачены впустую. Но мы вынуждены делать это, так как невозможно заранее сказать, какое из чисел будет использовано.

Выходом из этой ситуации может стать применение входных аргументов с функциональным типом. То есть если преобразовать функцию `returnSomeNum(_:_:)` к типу `((Int)->Int, (Int)->Int))->Int`, то в нее можно будет передать не результаты работы функций `getPiNum(_:)` и `getFibNum(_:)`, а сами функции. Далее в ее теле будет применена именно та функция, которая требуется, а ресурсы на подсчет второй использоваться не будут. То есть необходимое значение будет высчитано именно в тот момент, когда к нему произойдет обращение, а не в тот момент, когда они переданы в виде входного аргумента.

ПРИМЕЧАНИЕ Как бы это странно ни звучало, но в некотором роде вам нужно развивать в себе лень… много лени! Но не ту, которая говорит: «Оставлю задачу на завтра, лучше полежу немного еще!»; а ту, которая ищет максимально простой и незатратный способ быстрее выполнить поставленную задачу.

15.5. Возможности функций

Вложенные функции

Функции могут входить в состав друг друга, то есть они могут быть вложенными. Вложенные функции обладают ограниченной областью видимости, то есть напрямую доступны только в теле родительской функции.

Представьте бесконечную плоскость и точку на этой плоскости. Точка имеет некоторые координаты. Она может перемещаться по плоскости. Создадим функцию, которая принимает на входе координаты точки и направление перемещения, после чего передвигает точку и фиксирует ее новые координаты (листинг 15.16).

Листинг 15.16

```
func oneStep( coordinates: inout (Int, Int), stepType: String ) {
    func up( coords: inout (Int, Int)) {
        coords = (coords.0+1, coords.1)
    }
    func right( coords: inout (Int, Int)) {
```

```
        coords = (coords.0, coords.1+1)
    }
    func down( coords: inout (Int, Int)) {
        coords = (coords.0-1, coords.1)
    }
    func left( coords: inout (Int, Int)) {
        coords = (coords.0, coords.1-1)
    }

    switch stepType {
    case "up":
        up(coords: &coordinates)
    case "right":
        right(coords: &coordinates)
    case "down":
        down(coords: &coordinates)
    case "left":
        left(coords: &coordinates)
    default:
        break;
    }
}
var coordinates = (10, -5)
oneStep(coordinates: &coordinates, stepType: "up")
oneStep(coordinates: &coordinates, stepType: "right")
coordinates //(.0 11, .1 -4)
```

Функция `oneStep(coordinates:stepType:)` осуществляет перемещение точки по плоскости. В ней определено несколько вложенных функций, которые вызываются в зависимости от значения аргумента `stepType`. Данный набор функций доступен только внутри родительской функции `oneStep(coordinates:stepType:)`.

Входной параметр `coordinates` является сквозным, поэтому все изменения, производимые в нем, сохраняются и после окончания работы функции.

Перегрузка функций

`Swift` позволяет *перегружать функции* (`overloading`), то есть в одной и той же области видимости создавать функции с одинаковыми именами. Различия функций должны заключаться в их функциональных типах.

В листинге 15.17 представлены функции, которые могут сосуществовать одновременно в одной области видимости.

Листинг 15.17

```
func say(what: String){}
func say(what: Int){}
```

У данных функций одно и то же имя `say(what:)`, но разные типы входных аргументов. В результате Swift определяет обе функции как различные и позволяет им сосуществовать одновременно. Это связано с тем, что функциональный тип первой функции `(String)->()`, а второй — `(Int)->()`. Если вы имеете один и тот же список входных параметров (их имена и типы идентичны), то для перегрузки необходимо, чтобы функции имели разные типы выходных значений. Рассмотрим пример из листинга 15.18. Представленные в нем функции также могут сосуществовать одновременно.

Листинг 15.18

```
func cry() -> String {
    return "one"
}
func cry() -> Int {
    return 1
}
```

В данном случае можно сделать важное замечание: возвращаемое значение функции не может быть передано переменной или константе без явного указания типа объявляемого параметра (листинг 15.19).

Листинг 15.19

```
let resultOfFunc = say() // ошибка
```

В данном случае `Swift` просто не знает, какой тип данных у константы, поэтому не может определить, какую функцию вызвать. В результате Xcode сообщит об ошибке.

Если каким-либо образом указать тип данных параметра, согласуемый с типом возвращаемого значения одной из функций, то код отработает корректно (листинг 15.20).

Листинг 15.20

```
let resultString: String = cry() // "one"
var resultInt = cry() + 100 // 101
```

Рекурсивный вызов функций

Функция может вызывать саму себя. Этот механизм называется *рекурсией*. Очень многие алгоритмы могут быть реализованы с помощью

данной техники. Не бойтесь пользоваться рекурсией. Однако нужно быть крайне осторожными с этим механизмом, так как по невнимательности можно создать «бесконечную петлю», в которой функция будет постоянно вызывать саму себя. При корректном использовании рекурсий функция всегда будет завершать свою работу.

Пример рекурсии приведен в листинге 15.21.

Листинг 15.21

```
func countdown(firstNum num: Int) {
    print(num)
    if num > 0 {
        // рекурсивный вызов функции
        countdown(firstNum:num-1)
    }
}
countdown(firstNum: 20)
```

Функция `countdown(firstNum:)` отсчитывает числа в сторону понижения, начиная от переданного параметра `firstNum` и заканчивая нулем. Этот алгоритм реализуется рекурсивным вызовом функции.

16 Замыкания (closure)

Как объясняется в документации к языку `Swift`, *замыкания* (`closures`) — это организованные блоки с определенным функционалом, которые могут быть переданы и использованы в коде.

Согласитесь, не очень доступное объяснение. Попробуем иначе.

Замыкания (closure), или **замыкающие выражения**, — это сгруппированный программный код, который может быть передан в виде параметра и многократно использован. Ничего не напоминает? Если вы скажете, что в этом определении узнали функции, то будете полностью правы. Поговорим об этом подробнее.

16.1. Виды замыканий

Как вы знаете, параметры предназначены для хранения информации, а функции могут выполнять определенные задачи. Говоря простым языком, с помощью замыканий вы можете поместить блок исполняемого кода в переменную или константу, свободно передавать ее и при необходимости вызывать хранящийся в ней код. Вы уже видели подобный подход при изучении функций, и в этом нет ничего странного. Дело в том, что функции — это частный случай замыканий.

В общем случае замыкание (closure) может принять две формы:

- именованная функция;
- безымянная функция, определенная с помощью облегченного синтаксиса.

Знакомству с именованными функциями была посвящена вся предыдущая глава. Уверен, что вы уже довольно неплохо знакомы с их возможностями. Далее рассмотрим безымянные функции как один из способов представления замыканий.

ПРИМЕЧАНИЕ В дальнейшем безымянные функции будут именоваться замыканиями, или замыкающими выражениями. Говоря о функции, мы будем иметь в виду именно функции, а говоря о замыканиях — о безымянных функциях.

16.2. Введение в безымянные функции

Как вы уже знаете, переменная и константа могут хранить в себе ссылку на функцию. Но для того чтобы организовать это, не обязательно возвращать одну функцию из другой. Вы можете использовать специальный облегченный синтаксис, создав *безымянную функцию*, после чего передать ее в качестве значения в требуемый параметр. Безымянные функции не имеют имен. Они состоят только из тела, заключенного в фигурные скобки.

СИНТАКСИС

```
{ (входные_параметры) -> тип in
    // тело замыкающего выражения
}
```

- `входные_параметры` — список аргументов замыкания с указанием их имен и типов.
- `Тип` — тип данных значения, возвращаемого замыканием.

Замыкающее выражение пишется в фигурных скобках. После указания перечня входных аргументов и типа возвращаемого значения ставится ключевое слово `in`, после которого следует тело замыкания.

В самом простом случае можно опустить указание входных параметров и тип выходного значения, оставив лишь тело замыкания.

Пример

```
// безымянная функция в качестве значения константы
let functionInLet = {return true}
// вызываем безымянную функцию
functionInLet() // true
```

Константа `functionInLet` имеет функциональный тип `() -> Bool` (ничего не принимает на вход, но возвращает логическое значение) и хранит в себе тело функции.

Обратите внимание, что при инициализации безымянной функции в параметр для ее вызова используется имя параметра с круглыми скобками.

Рассмотрим пример, в котором наглядно продемонстрированы все плюсы от использования безымянных функций (замыканий).

В нашей программе объявлена переменная `wallet`, хранящая в себе программный аналог кошелька с купюрами (в предыдущей главе мы

уже использовали подобный массив-кошелек). Каждый элемент этой коллекции представляет собой одну купюру определенного номинала. Перед нами стоит задача отбора купюр в кошельке по различным условиям. Для каждого условия может быть создана отдельная функция, принимающая на вход массив `wallet` и возвращающая отфильтрованную коллекцию.

В листинге 16.1 показано, каким образом может быть реализована функция отбора всех сторублевых купюр.

Листинг 16.1

```
// массив с купюрами
var wallet = [10,50,100,100,5000,100,50,50,500,100]

// функция отбора купюр
func handle100(wallet: [Int]) -> [Int] {
    var returnWallet = [Int]()
    for banknot in wallet {
        if banknot==100{
            returnWallet.append(banknot)
        }
    }
    return returnWallet
}
// вызов функции отбора купюр достоинством 100
handle100(wallet: wallet) // [100, 100, 100, 100]
```

При каждом вызове функция `handle100(wallet:)` будет возвращать массив сторублевых купюр переданного массива-кошелька.

Но условия отбора не ограничиваются данной функцией. Расширим функционал нашей программы, написав дополнительную функцию для отбора купюр достоинством 1000 рублей и более (листинг 16.2).

Листинг 16.2

```
func handleMore1000(wallet: [Int]) -> [Int] {
    var returnWallet = [Int]()
    for banknot in wallet {
        if banknot>=1000{
            returnWallet.append(banknot)
        }
    }
    return returnWallet
}
// вызов функции отбора купюр достоинством более или равным 1000
handleMore1000(wallet: wallet) // [5000]
```

В результате для отбора купюр по требуемым условиям реализовано уже две функции: `handle100(wallet:)` и `handleMore1000(wallet:)`. При этом тела обеих функций очень похожи (практически дублируют друг друга), разница лишь в проверяемом условии, остальной код в функциях один и тот же. В случае дальнейшего расширения программы будут появляться все новые и новые функции, также повторяющие один и тот же код.

Для решения проблемы дублирования можно пойти двумя путями:

1) реализовать весь функционал отбора купюр в пределах одной функции, а в качестве входного аргумента передавать условие;
2) реализовать весь функционал в виде трех функций. Первая будет группировать повторяющийся код и принимать в виде аргумента одну из двух других функций. Переданная функция будет производить проверку условия в теле главной функции.

Если выбрать первый путь, то при увеличении количества условий отбора единая функция будет разрастаться и в конце концов станет нечитабельной и слишком сложной. Плюс к этому необходимо придумать, каким образом передавать указатель на проверяемое условие, а значит, потребуется вести документацию к данной функции.

По этой причине воспользуемся вторым вариантом, реализуем функционал в виде трех функций:

1. Функция с именем `handle`, принимающая массив-кошелек и условие отбора (в виде имени функции) в качестве входных аргументов и возвращающая массив отобранных купюр. В теле функции будут поочередно проверяться элементы входного массива на соответствие переданному условию.
2. Функция с именем `compare100`, принимающая на вход значение очередного элемента массива-кошелька, производящая сравнение с целым числом 100 и возвращающая логический результат этой проверки.
3. Функция с именем `compareMore1000`, аналогичная `compare100`, но производящая проверку на соответствие целому числу 1000.

В листинге 16.3 показана реализация описанного алгоритма.

Листинг 16.3

```
// единая функция формирования результирующего массива
func handle(wallet: [Int], closure: (Int) -> Bool) -> [Int] {
    var returnWallet = [Int]()
    for banknot in wallet {
        if closure(banknot) {
```

```
            returnWallet.append(banknot)
        }
    }
    return returnWallet
}
// функция сравнения с числом 100
func compare100(banknot: Int) ->Bool {
    return banknot==100
}
// функция сравнения с числом 1000
func compareMore1000(banknot:Int) -> Bool {
    return banknot>=1000
}
// отбор
let resultWalletOne = handle(wallet: wallet, closure: compare100)
let resultWalletTwo = handle(wallet: wallet, closure: compareMore1000)
```

Функция `handle(wallet:closure:)` получает в качестве входного параметра `closure` одну из функций проверки условия и в операторе `if` вызывает переданную функцию. Функции проверки принимают на вход анализируемую купюру и возвращают `Bool` в зависимости от результата сравнения. Чтобы получить купюры определенного достоинства, необходимо вызвать функцию `handle(wallet:closure:)` и передать в нее имя одной из функций проверки.

В итоге мы получили очень качественный и легкочитаемый код.

Представим, что возникла необходимость написать функции для отбора купюр по множеству условий (найти все полтинники; все купюры достоинством менее 1000 рублей; все купюры, которые без остатка делятся на 200, и т. д.). В определенный момент писать отдельную функцию проверки для каждого из них станет довольно тяжелой задачей, так как для того, чтобы использовать единую функцию проверки, необходимо знать имя проверяющей функции, а их могут быть десятки.

В подобной ситуации можно отказаться от создания отдельных функций и передавать в `handle(wallet:closure:)` условие отбора в виде безымянной функции. В листинге 16.4 показано, каким образом это может быть реализовано.

Листинг 16.4

```
// отбор купюр достоинством выше 1000 рублей
// аналог передачи compare100
handle(wallet: wallet, closure: {(banknot: Int) -> Bool in
    return banknot>=1000
})
```

```
// отбор купюр достоинством 100 рублей
// аналог передачи compareMore1000
handle(wallet: wallet, closure: {(banknot: Int) -> Bool in
    return banknot==100
})
```

Входной аргумент `closure` имеет функциональный тип `(Int)->Bool`, а значит, передаваемая безымянная функция должна иметь тот же тип данных, что мы и видим в коде.

Для переданного замыкания указан входной параметр типа `Int` и определен тип возвращаемого значения (`Bool`). После ключевого слова `in` следует тело функции, в котором с помощью оператора `return` возвращает логическое значение — результат проверки очередного элемента кошелька. Таким образом, в теле функции `handle(wallet:closure:)` будет вызываться не какая-то внешняя функция, имя которой передано, а безымянная функция, переданная в виде входного аргумента.

В результате такого подхода необходимость в существовании функций `compare100(banknot:)` и `compareMore1000(banknot:)` отпадает, так как код условия передается напрямую в качестве замыкания в аргумент `closure`.

ПРИМЕЧАНИЕ Далее в качестве примера будет производиться работа только с функцией отбора купюр достоинством больше или равным 1000 рублей.

16.3. Возможности замыканий

Замыкающие выражения позволяют в значительной мере упрощать ваши программы. Это лишь одна из многих возможностей `Swift`, обеспечивающих красивый и понятный исходный код проектов. Приступим к оптимизации замыкающих выражений из примера выше и параллельно рассмотрим возможности, которые доступны нам при их использовании.

Пропуск указания типов

При объявлении входного параметра `closure` в функции `handle(wallet:closure:)` указывается его функциональный тип `(Int)->Bool`, поэтому при передаче замыкающего выражения можно опустить данную информацию, оставив лишь имя входного аргумента (листинг 16.5).

Листинг 16.5

```
// отбор купюр достоинством выше 1000 рублей
handle(wallet: wallet, closure: {banknot in
    return banknot>=1000
})
```

В замыкающем выражении перед ключевым словом `in` необходимо указать только имя параметра без входных и выходных типов.

Неявное возвращение значения

Если тело замыкающего выражения содержит всего одно выражение, которое возвращает некоторое значение (с использованием оператора `return`), то такие замыкания могут неявно возвращать выходное значение. Неявно — значит без использования оператора `return` (листинг 16.6).

Листинг 16.6

```
handle(wallet: wallet,
    closure: {banknot in banknot>=1000})
```

Сокращенные имена параметров

В случае, когда замыкание состоит из одного выражения, можно опустить входные параметры (все до ключевого слова `in`, включая само слово). При этом доступ к аргументам внутри тела замыкания необходимо осуществлять через сокращенные имена в форме `$номер_параметра`. Номера входных параметров начинаются с нуля.

ПРИМЕЧАНИЕ В сокращенной форме записи имен входных параметров обозначение $0 указывает на первый передаваемый аргумент. Для доступа ко второму аргументу необходимо использовать обозначение $1, к третьему — $2 и т. д.

Перепишем вызов функции `handle(wallet:closure:)` с использованием сокращенных имен (листинг 16.7).

Листинг 16.7

```
handle(wallet: wallet,
    closure: {$0>=1000})
```

Здесь `$0` — это аргумент `banknot` входного аргумента замыкания `closure` в функции `handle(wallet:closure:)`.

Вынос замыкания за скобки

Если входной параметр-функция расположен последним в списке входных параметров функции (как в данном случае в функции `handle(wallet:closure:)`, где параметр `closure` является последним), Swift позволяет вынести его значение (тело замыкающего выражения) за круглые скобки (листинг 16.8).

Листинг 16.8

```
handle(wallet: wallet){$0>=1000}
```

Эта возможность особенно полезна, когда замыкание, передаваемое в качестве входного аргумента функции, является многострочным. В листинге 16.9 показан пример выноса замыкания, состоящего из нескольких выражений. С его помощью производится сравнение элементов с массивом «разрешенных» купюр. В результирующей коллекции будут находиться только те купюры, которые являются «разрешенными».

Листинг 16.9

```
handle(wallet: wallet){
banknot in
    for number in Array(arrayLiteral: 100,500) {
        if number == banknot {
            return true
        }
    }
    return false
}
```

ПРИМЕЧАНИЕ Существует и другой способ реализовать проверку из предыдущего листинга. Для этого можно использовать метод `contains(_:)`, передавая в него очередную купюру:

```
var successBanknots = handle(wallet: wallet)
    { Array(arrayLiteral: 100,500).contains($0) }
successBanknots //[100, 100, 100, 500, 100]
```

16.4. Безымянные функции в параметрах

В листинге 16.10 показан пример инициализации замыкания в параметр `closure`. При этом у параметра явно указан функциональный тип (ранее в примерах он определялся неявно).

Листинг 16.10

```
var closure: () -> () = {
    print("Замыкающее выражение")
}
closure()
```

Консоль

```
Замыкающее выражение
```

Так как данное замыкающее выражение не имеет входных параметров и возвращаемого значения, то его тип равен `()->()`. Для вызова записанного в константу замыкающего выражения необходимо написать имя константы с круглыми скобками, точно так же, как мы делали это ранее.

Явное указание функционального типа позволяет определить входные аргументы и тип выходного значения (листинг 16.11).

Листинг 16.11

```
// передача в функцию строкового значения
var closurePrint: (String) -> () = {text in
    print(text)
}
closurePrint("Text")

// передача в функцию целочисленных значений
// с осуществлением доступа через краткий синтаксис $0 и $1
var sum: (_ numOne: Int, _ numTwo: Int) -> Int = {
    return $0 + $1
}
sum(10, 34) // 44
```

Консоль

```
Text
```

Входные аргументы замыкания не должны иметь внешних имен. По этой причине в первом случае указание имени вообще отсутствует, а во втором используется знак нижнего подчеркивания.

16.5. Пример использования замыканий при сортировке массива

Swift предлагает большое количество функций и методов, позволяющих значительно упростить разработку приложений. Одним из таких

методов является `sorted(by:)`, предназначенный для сортировки массивов, как строковых, так и числовых. Он принимает на входе массив, который необходимо отсортировать, и условие сортировки.

Принимаемое условие сортировки — это обыкновенное замыкающее выражение, которое вызывается внутри метода `sorted(by:)`, принимает на входе два очередных элемента сортируемого массива и возвращает значение Bool в зависимости от результата их сравнения.

В листинге 16.12 отсортируется массив `array` таким образом, чтобы элементы были расположены по возрастанию. Для этого в метод `sorted(by:)` передается замыкающее выражение, которое возвращает `true`, когда второе из сравниваемых чисел больше.

Листинг 16.12

```
var array = [1,44,81,4,277,50,101,51,8]
var sortedArray = array.sorted(by: { (first: Int, second: Int) -> Bool
in
    return first < second
})
sortedArray //[1, 4, 8, 44, 50, 51, 81, 101, 277]
```

Теперь применим все рассмотренные ранее способы оптимизации замыкающих выражений:

- уберем функциональный тип замыкания;
- уберем оператор `return`;
- заменим имена переменных сокращенной формой.

В результате получится выражение, приведенное в листинге 16.13. Как и в предыдущем примере, здесь тоже необходимо отсортировать массив `array` таким образом, чтобы элементы были расположены по возрастанию. Для этого в метод `sorted(by:)` передается такое замыкающее выражение, которое возвращает `true`, когда второе из сравниваемых чисел больше.

Листинг 16.13

```
sortedArray = array.sorted(by: {$0<$1})
sortedArray //[1, 4, 8, 44, 50, 51, 81, 101, 277]
```

В результате код получается более читабельным и красивым.

Но и это еще не все. Так как выражение в замыкании состоит всего из одного бинарного оператора, то можно убрать даже имена параметров, оставив лишь оператор сравнения (листинг 16.14).

Листинг 16.14

```
sortedArray = array.sorted(by: <)
sortedArray //[1, 4, 8, 44, 50, 51, 81, 101, 277]
```

Надеюсь, вы приятно удивлены потрясающими возможностями Swift!

16.6. Захват переменных

Swift позволяет зафиксировать значения внешних по отношению к замыканию параметров, которые они имели на момент его определения.

Синтаксис захвата переменных

Обратимся к примеру в листинге 16.15. Существуют два параметра, `a` и `b`, которые не передаются в качестве входных аргументов в замыкание, но используются им в вычислениях. При каждом вызове такого замыкания оно будет определять значения данных параметров, прежде чем приступить к выполнению операции с их участием.

Листинг 16.15

```
var a = 1
var b = 2
let closureSum : () -> Int = {
    return a+b
}
closureSum() // 3
a = 3
b = 4
closureSum() // 7
```

Замыкание, хранящееся в константе `closureSum`, складывает значения переменных `a` и `b`. При изменении их значений возвращаемое замыканием значение меняется.

Существует способ «захватить» значения параметров, то есть зафиксировать те значения, которые имеют эти параметры на момент объявления замыкающего выражения. Для этого в начале замыкания в квадратных скобках необходимо перечислить захватываемые переменные, разделив их запятой, после чего указать ключевое слово `in`.

Перепишем инициализированное переменной `closureSum` замыкание таким образом, чтобы оно захватывало первоначальные значения переменных `a` и `b` (листинг 16.16).

Листинг 16.16

```
var a = 1
var b = 2
let closureSum : () -> Int = {
    [a,b] in
    return a+b
}
closureSum() // 3
a = 3
b = 4
closureSum() // 3
```

Замыкание, хранящееся в константе `closureSum`, складывает значения перемененных `a` и `b`. При изменении этих значений возвращаемое замыканием значение не меняется.

Захват вложенной функцией

Другим способом захвата значения внешнего параметра является вложенная функция, написанная в теле другой функции. Вложенная функция может захватывать произвольные переменные, константы и даже входные аргументы, объявленные в родительской функции.

Рассмотрим пример из листинга 16.17.

Листинг 16.17

```
func makeIncrement(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func increment() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return increment
}
```

Функция `makeIncrement(forIncrement:)` возвращает значение с функциональным типом `()->Int`. Это значит, что вернется замыкание, не имеющее входных аргументов и возвращающее целочисленное значение.

Функция `makeIncrement(forIncrement:)` использует два параметра:

- `runningTotal` — переменную типа `Int`, объявляемую в теле функции. Именно ее значение является результатом работы всей конструкции;
- `amount` — входной аргумент, имеющий тип `Int`. Он определяет, насколько увеличится значение `runningTotal` при очередном обращении.

Вложенная функция `increment()` не имеет входных или объявляемых параметров, но при этом обращается к `runningTotal` и `amount` внутри своей реализации. Она делает это в автоматическом режиме путем захвата значений обоих параметров по ссылке. Захват значений по ссылке гарантирует, что измененные значения параметров не исчезнут после окончания работы функции `makeIncrement(forIncrement:)` и будут доступны при повторном вызове функции `increment()`.

Теперь обратимся к листингу 16.18.

Листинг 16.18

```
var incrementByTen = makeIncrement(forIncrement: 10)
var incrementBySeven = makeIncrement(forIncrement: 7)
incrementByTen()   // 10
incrementByTen()   // 20
incrementByTen()   // 30
incrementBySeven() // 7
incrementBySeven() // 14
incrementBySeven() // 21
```

В переменных `incrementByTen` и `incrementBySeven` хранятся возвращаемые функцией `makeIncrement(forIncrement:)` замыкания. В первом случае значение `runningTotal` увеличивается на 10, а во втором — на 7. Каждая из переменных хранит свою копию захваченного значения `runningTotal`, именно поэтому при их использовании увеличиваемые значения не пересекаются и увеличиваются независимо друг от друга.

ВНИМАНИЕ Так как в переменных `incrementByTen` и `incrementBySeven` хранятся замыкания, то при доступе к ним после их имени необходимо использовать скобки (по аналогии с доступом к функциям).

16.7. Замыкания передаются по ссылке

Функциональный тип данных — это ссылочный тип (reference type). Это значит, что замыкания передаются не копированием, а с помощью ссылки на область памяти, где хранится это замыкание.

Рассмотрим пример, описанный в листинге 16.19.

Листинг 16.19

```
var incrementByFive = makeIncrement(forIncrement: 5)
var copyIncrementByFive = incrementByFive
```

В данном примере используется функция `makeIncrement(forIncrement:)`, объявленная ранее. Напомню, она возвращает замыка-

ние типа `()->Int`, которое в данном случае предназначено для увеличения значения на 5. Возвращаемое замыкание записывается в переменную `incrementByFive`, после чего копируется в переменную `copyIncrementByFive`. В результате можно обратиться к одному и тому же замыканию, используя как `copyIncrementByFive`, так и `incrementByFive` (листинг 16.20).

Листинг 16.20

```
incrementByFive() // 5
copyIncrementByFive() // 10
incrementByFive() // 15
```

Как видите, какую бы функцию мы ни использовали, происходит модификация одного и того же значения (каждое последующее значение больше предыдущего на 5). Это обусловлено тем, что замыкания передаются по ссылке.

16.8. Автозамыкания

Автозамыкания — это замыкания, которые автоматически создаются из переданного выражения. Иными словами, может существовать функция, имеющая один или несколько входных параметров, которые при ее вызове передаются как значения, но во внутренней реализации функции используются как самостоятельные замыкания. Рассмотрим пример из листинга 16.21.

Листинг 16.21

```
var arrayOfNames = ["Helga", "Bazil", "Alex"]
func printName(nextName: String ) {
    print(nextName)
}
printName(nextName: arrayOfNames.remove(at: 0))
```

Консоль

```
Helga
```

При каждом вызове функции `printName(nextName:)` в качестве входного значения ей передается результат вызова метода `remove(at:)` массива `arrayOfNames`.

Независимо от того, в какой части функции будет использоваться переданный параметр (или не будет использоваться вовсе), значение, возвращаемое методом `remove(at:)`, будет вычислено в момент вызова

функции `printName(nextName:)`. Получается, что передаваемое значение вычисляется независимо от того, нужно ли оно в ходе выполнения функции.

Отличным решением данной проблемы станет использование ленивых вычислений, то есть таких вычислений, которые будут выполняться лишь в тот момент, когда это понадобится. Для того чтобы реализовать этот подход, можно передавать в функцию `printName(nextName:)` замыкание, которое будет вычисляться в тот момент, когда к нему обратятся (листинг 16.22).

Листинг 16.22

```
func printName(nextName: ()->String) {
    // какой-либо код
    print(nextName())
}
printName(nextName: {arrayOfNames.remove(at: 0)})
```

Консоль

```
Helga
```

Для решения этой задачи потребовалось изменить тип входного параметра `nextName` на `()->String` и заключить передаваемый метод `remove(at:)` в фигурные скобки. Теперь внутри реализации функции `printName(nextName:)` к входному аргументу `nextName` необходимо обращаться как к самостоятельной функции (с использованием круглых скобок после имени параметра). Таким образом, значение метода `remove(at:)` будет вычислено именно в тот момент, когда оно понадобится, а не в тот момент, когда оно будет передано. Единственным недостатком данного подхода является то, что входной параметр должен быть заключен в фигурные скобки, а это несколько усложняет использование функции и чтение кода.

С помощью автозамыканий можно использовать положительные стороны обоих рассмотренных примеров: отложить вычисление переданного значения и передавать значение без фигурных скобок.

Для реализации автозамыкания необходимо следующее:

- Входной аргумент должен иметь функциональный тип.

  В примере, приведенном ранее, аргумент `nextName` уже имеет функциональный тип `()->String`.

- Функциональный тип не должен определять типы входных параметров.

В примере типы входных параметров не определены (пустые скобки).

- Функциональный тип должен определять тип возвращаемого значения.

  В примере тип возвращаемого значения определен как `String`.

- Переданное выражение должно возвращать значение того же типа, которое определено в функциональном типе замыкания.

  В примере передаваемая в качестве входного аргумента функция возвращает значение типа `String` точно так же, как определено функциональным типом входного аргумента.

- Перед функциональным типом необходимо использовать атрибут `@autoclosure`.
- Передаваемое значение должно указываться без фигурных скобок.

Перепишем код из предыдущего листинга в соответствии с указанными требованиями (листинг 16.23).

Листинг 16.23

```
func printName(nextName: @autoclosure ()->String) {
    print(nextName())
}
printName(nextName: arrayOfNames.remove(at: 0))
```

Консоль

```
Helga
```

Теперь метод `remove(at:)` передается в функцию `printName(nextName:)` как обычный аргумент, без использования фигурных скобок, но внутри тела используется как самостоятельная функция.

Ярким примером глобальной функции, входящей в стандартные возможности `Swift` и использующей механизм автозамыканий, является функция `assert(condition:message)`. Аргументы `condition` и `message` — это автозамыкания, первое из которых вычисляется только в случае активного debug-режима, а второе — только в случае, когда `condition` соответствует `false`.

ПРИМЕЧАНИЕ Это еще одна встреча с так называемыми ленивыми вычислениями, о которых мы начали говорить в предыдущей главе.

16.9. Выходящие замыкания

Как вы уже неоднократно убеждались и убедитесь еще не раз, Swift — очень умный язык программирования. Он старается экономить ваши ресурсы там, где вы можете об этом даже не догадываться.

По умолчанию все переданные в функцию замыкания имеют ограниченную этой функцией область видимости, то есть если вы решите сохранить замыкание для дальнейшего использования, то встретитесь с определенными трудностями. Другими словами, все переданные в функцию замыкания называются не выходящими за пределы ее тела. Если Swift видит, что область, где замыкание доступно, ограничена, он при первой же возможности удалит его, чтобы освободить и не расходовать оперативную память.

Для того чтобы позволить замыканию выйти за пределы области видимости функции, необходимо указать атрибут `@escaping` перед функциональным типом при описании входных параметров функции.

Рассмотрим пример. Предположим, что в программе есть специальная переменная, предназначенная для хранения замыканий типа `()->Int`, то есть являющаяся коллекцией замыканий (листинг 16.24).

Листинг 16.24

```
var arrayOfClosures: [()->Int] = []
```

Пока еще пустой массив `arrayOfClosures` может хранить в себе замыкания с функциональным типом `()->Int`. Реализуем функцию, добавляющую в этот массив переданные ей в качестве аргументов замыкания (листинг 16.25).

Листинг 16.25

```
func addNewClosureInArray(_ newClosure: ()->Int){
    arrayOfClosures.append(newClosure) // ошибка
}
```

Xcode сообщит вам об ошибке. И тому есть две причины:

- Замыкание — это тип-ссылка, то есть оно передается по ссылке, но не копированием.
- Замыкание, которое будет храниться в параметре `newClosure`, будет иметь ограниченную телом функции область видимости, а значит, не может быть добавлено в глобальную (по отношению к телу функции) переменную `arrayOfClosures`.

Для решения этой проблемы необходимо указать, что замыкание, хранящееся в переменной `newClosure`, является выходящим. Для этого перед описанием функционального типа данного аргумента укажите атрибут `@escaping`, после чего вы сможете передать в функцию `addNewClosureInArray(_:)` произвольное замыкание (листинг 16.26).

Листинг 16.26

```
func addNewClosureInArray(_ newClosure: @escaping ()->Int){
    arrayOfClosures.append(newClosure)
}
addNewClosureInArray({return 100})
addNewClosureInArray{return 1000}
arrayOfClosures[0]() // 100
arrayOfClosures[1]() // 1000
```

Обратите внимание на то, что в одном случае замыкание передается с круглыми скобками, а в другом — без них. Так как функция `addNewClosureInArray(_:)` имеет один входной аргумент, то допускаются оба варианта.

ПРИМЕЧАНИЕ Если вы передаете замыкание в виде параметра, то можете использовать модификатор `inout` вместо `@escaping`.

16.10. Каррирование функций

Каррирование — это процесс, при котором функция от нескольких аргументов преобразуется в функцию (или набор функций) от одного аргумента. Это становится возможным благодаря тому, что в качестве выходного значения функции может выступать другая функция.

Предположим, что у нас есть функция, которая принимает на вход три параметра и возвращает некоторый результат. При каррировании мы получим функцию, которая принимает на вход один аргумент, а возвращает функцию, которая также принимает на вход один аргумент и в свою очередь также возвращает функцию, которая принимает на вход один аргумент и возвращает некоторый результат. То есть получается своеобразная лесенка из функций.

Разложим все по полочкам.

У нас есть функция с типом `(Int, Int, Int)->Int`, которая зависит от трех входных аргументов типа `Int` и возвращает значение типа `Int`.

При каррировании мы получим функцию типа `(Int)->(Int)->(Int)->Int`. Для этого выполним следующие шаги:

- обозначим функциональный тип `(Int)->Int` последней функции в цепочке как `A`. Тогда каррированная функция будет выглядеть как `(Int)->(Int)->A`;
- обозначим функциональный тип `(Int)->A` как `B`. Тогда каррированная функция будет выглядеть как `(Int)->B`.

Получается, что наша функция принимает на вход одно целое число и возвращает значение типа `B`.

Значение типа `B`, в свою очередь, также является функцией, которая принимает на вход одно целое число и возвращает значение типа `A`.

Значение типа `A` также является функцией, которая принимает на вход одно целое число и возвращает одно целое число.

В результате одну функцию, зависящую от трех параметров, мы реорганизовали (каррировали) к трем взаимозависимым функциям, каждая из которых зависит всего от одного значения.

Рассмотрим пример каррирования. Существует функция с типом `(Int, Int)->Int`, которая получает на вход два целочисленных значения, производит сложение и возвращает его результат в виде целого числа (листинг 16.27).

Листинг 16.27

```
func sum(x: Int, y: Int) -> Int {
    return x + y
}
sum(x: 1,y: 4) // 5
```

С целью каррирования напишем новую функцию, которая принимает на вход всего один целочисленный параметр, а возвращает функцию типа `(Int)->Int` (листинг 16.28).

Листинг 16.28

```
func sumTwo(_ x: Int) -> (Int) -> Int {
    return {  return $0+x }
}
var anotherClosure = sumTwo(1)
anotherClosure(12) // 13
```

Переменная `anotherClosure` получает в качестве значения, которому мы можем передать входной параметр, безымянную функцию. Прелесть каррирования в том, что мы можем объединить вызов функции `sum2(x:)` и передачу значения в возвращаемое ею замыкание (листинг 16.29).

Листинг 16.29

```
sumTwo(5)(12) // 17
```

В результате мы получаем прекрасно читаемую и удобную в использовании функцию.

Если говорить о пользе, то каррирование хорошо именно тем, что устраняет зависимость функции от нескольких параметров. Мы можем вызвать функцию, как только получим первое нужное значение, и далее при необходимости многократно вызывать возвращенное ею замыкание при поступлении второго требуемого параметра (листинг 16.30).

Листинг 16.30

```
var sumClosure = sumTwo(1)
sumClosure(12) // 13
sumClosure(19) // 20
```

17 Дополнительные возможности

Целью книги является не только изучение синтаксиса и основ разработки на «яблочном» языке программирования. Мне бы хотелось, чтобы вы начали лучше и глубже понимать принципы разработки в Xcode на Swift. В этой главе приведены различные вспомогательные функциональные элементы, которыми так богат этот язык программирования. Вы также узнаете о важных функциональных элементах, которые смогут значительно облегчить практику программирования.

Описанные в главе методы помогут успешно работать с коллекциями различных типов. Если у вас уже есть опыт программирования, то, вероятно, вы привыкли заниматься обработкой этих типов с помощью циклов, однако сейчас это не всегда оправданно. Советую вам активно использовать описанные функции.

17.1. Метод map(_:)

Метод `map(_:)` позволяет применить переданное в него замыкание для каждого элемента коллекции. В результате его выполнения возвращается новая последовательность, тип элементов которой может отличаться от типа исходных элементов (рис. 17.1).

Рассмотрим пример, описанный в листинге 17.1.

Листинг 17.1

```
var myArray = [2, 4, 5, 7]
var newArray = myArray.map{$0}
newArray // [2, 4, 5, 7]
```

Метод `map(_:)` принимает замыкание и применяет его к каждому элементу массива `myArray`. Переданное замыкание `{$0}` не производит каких-либо действий над элементами, поэтому результат, содержащийся в переменной `newArray`, не отличается от исходного.

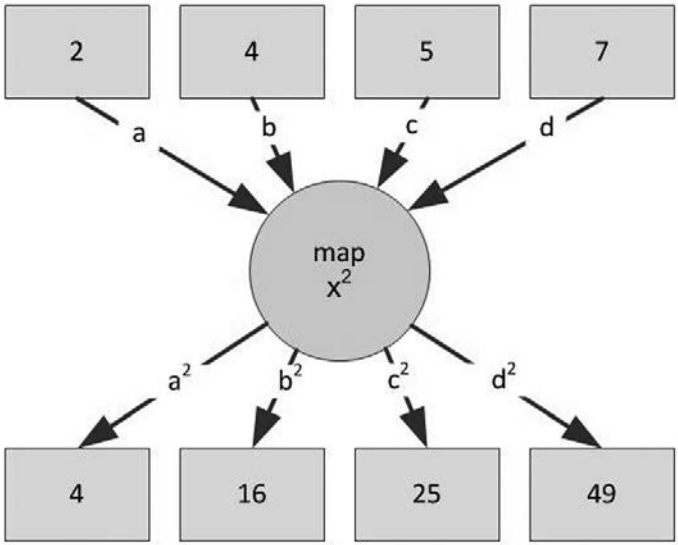

Рис. 17.1. Принцип работы метода map

ПРИМЕЧАНИЕ В данном примере используется сокращенное имя параметра, если точнее, то $0. Данная тема была изучена в главе 16. Давайте повторим, каким образом функция `map(_:)` лишилась круглых скобок и приобрела вид `map{$0}`.

Метод `map(_:)` позволяет передать в него замыкание, которое имеет один входной аргумент того же типа, что и элементы обрабатываемой коллекции, и один выходной параметр. Если не использовать сокращенный синтаксис, то вызов метода будет выглядеть следующим образом:

```
var array = [2, 4, 5, 7]
var newArray = array.map({
    (value: Int) -> Int in
    return value
})
```

Замыкание никак не изменяет входной аргумент — просто возвращает его.

Оптимизируем замыкание.

Сократим код перед ключевым словом `in`, так как передаваемое замыкание имеет всего один входной аргумент.

Уберем круглые скобки, так как метод `map(_:)` имеет один входной аргумент.

Уберем оператор `return`, так как тело замыкания помещается в одно выражение.

В результате получим следующий код:

```
var newArray = array.map{value in value}
```

Теперь можно убрать ключевое слово `in` и заменить `value` на сокращенное имя `$0`:

```
var newArray = array.map{$0}
```

Изменим замыкание таким образом, чтобы `map(_:)` возводил каждый элемент в квадрат (листинг 17.2).

Листинг 17.2

```
newArray = newArray.map{$0*$0}
newArray // [4, 16, 25, 49]
```

Как говорилось ранее, тип значений результирующей последовательности может отличаться от типа элементов исходной последовательности. Так, например, в листинге 17.3 количество элементов массивов `intArray` и `boolArray` одинаково, но тип элементов различается (`Int` и `Bool` соответственно).

Листинг 17.3

```
var intArray = [1, 2, 3, 4]
var boolArray = intArray.map{$0 > 2}
boolArray // [false, false, true, true]
```

Каждый элемент последовательности сравнивается с двойкой, в результате чего возвращается логическое значение.

Вы можете обрабатывать элементы коллекции с помощью метода `map(_:)` произвольным образом. В листинге 17.4 показан пример создания многомерного массива на основе базового.

Листинг 17.4

```
let numbers = [1, 2, 3, 4]
let mapped = numbers.map { Array(repeating: $0, count: $0) }
mapped // [[1], [2, 2], [3, 3, 3], [4, 4, 4, 4]]
```

Метод `map(_:)` позволяет обрабатывать элементы любой коллекции, в том числе и словаря. В листинге 17.5 показан пример перевода расстояния, указанного в милях, в километры.

Листинг 17.5

```
let milesToDest = ["Moscow":120.0,"Dubai":50.0,"Paris":70.0]
var kmToDest = milesToDest.map { name,miles in [name:miles * 1.6093] }
kmToDest // [["Dubai": 80.465], ["Paris": 112.651],
             ["Moscow": 193.116]]
```

17.2. Метод mapValues(_:)

Метод `mapValues(_:)` позволяет обработать значения каждого элемента словаря, при этом ключи элементов останутся в исходном состоянии (листинг 17.6).

Листинг 17.6

```
var mappedCloseStars = ["Proxima Centauri": 4.24, "Alpha Centauri A":
4.37]
var newCollection = mappedCloseStars.mapValues{ $0+1 }
newCollection  // ["Proxima Centauri": "5.24", "Alpha Centauri A":
"5.37]
```

В результате вы получаете все тот же словарь, но с обработанными значениями.

17.3. Метод filter(_:)

Метод `filter(_:)` используется, когда требуется отфильтровать элементы коллекции по определенному правилу (рис. 17.2).

В листинге 17.7 показана фильтрация всех целочисленных элементов исходного массива, которые делятся на 2 без остатка, то есть всех четных чисел.

Листинг 17.7

```
let numArray = [1, 4, 10, 15]
let even = numArray.filter{ $0 % 2 == 0 }
even // [4, 10]
```

Помимо массивов, можно производить фильтрацию других типов коллекций. В листинге 17.8 показана фильтрация элементов словаря `starDistanceDict`.

Листинг 17.8

```
var starDistanceDict = ["Wolf 359": 7.78, "Alpha Centauri B": 4.37,
"Barnard's Star": 5.96]
let closeStars = starDistanceDict.filter { $0.value < 5.0 }
closeStars // ["Alpha Centauri B": 4.37]
```

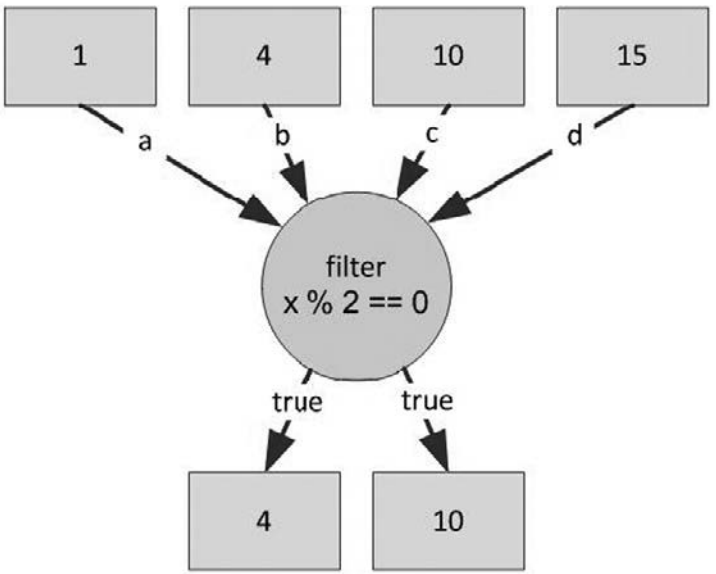

Рис. 17.2. Принцип работы метода filter

17.4. Метод reduce(_:_:)

Метод `reduce(_:_:)` позволяет объединить все элементы коллекции в одно значение в соответствии с переданным замыканием. Помимо самих элементов, метод принимает первоначальное значение, которое служит для выполнения операции с первым элементом коллекции.

Предположим, необходимо определить общее количество имеющихся у вас денег. На вашей карте 210 рублей, а в кошельке 4 купюры разного достоинства. Эта задача легко решается с помощью метода `reduce(_:_:)` (рис. 17.3 и листинг 17.9).

Листинг 17.9

```
let cash = [10, 50, 100, 500]
let total = cash.reduce(210, +) // 870
```

Первый аргумент — это начальное значение, второй — замыкание, обрабатывающее каждую пару элементов. Первая операция сложения производится между начальным значением и первым элементом массива `cash`.

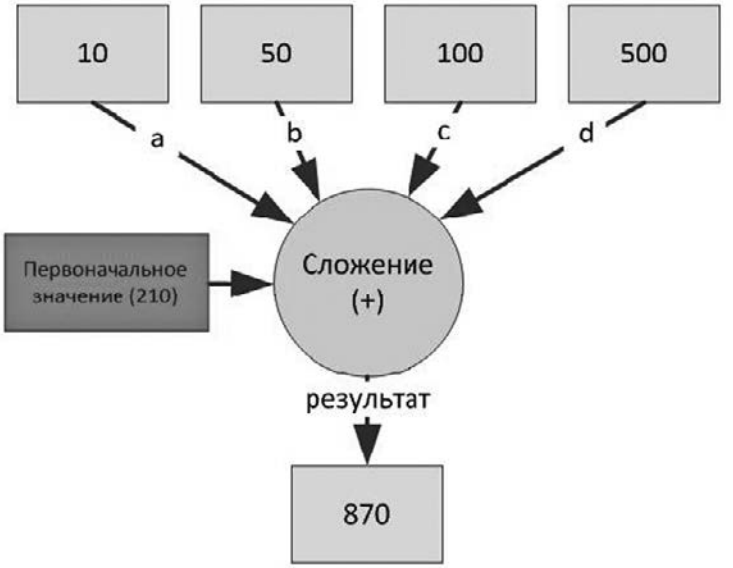

Рис. 17.3. Принцип работы метода reduce

Результат этой операции складывается со вторым элементом массива и т. д.

Замыкание, производящее операцию, может быть произвольным — главное, чтобы оно обрабатывало операцию для двух входящих аргументов (листинг 17.10).

Листинг 17.10

```
let multiTotal = cash.reduce(210, {$0*$1})
multiTotal // 5250000000
let totalThree = cash.reduce(210, {a,b in a-b})
totalThree // -450
```

17.5. Метод flatMap(_:)

Метод `flatMap(_:)` отличается от `map(_:)` тем, что всегда возвращает плоский одномерный массив. Так, пример, приведенный в листинге 17.4, но с использованием `flatMap(_:)`, вернет одномерный массив (листинг 17.11).

Листинг 17.11

```
let numbersArray = [1, 2, 3, 4]
let flatMapped = numbersArray.flatMap { Array(repeating: $0,
                 count: $0) }
flatMapped // [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
```

Вся мощь `flatMap(_:)` проявляется тогда, когда в многомерном массиве требуется найти все подпадающие под некоторое условие значения (листинг 17.12).

Листинг 17.12

```
let someArray = [[1, 2, 3, 4, 5], [11, 44, 1, 6], [16, 403, 321, 10]]
let filterSomeArray = someArray.flatMap{$0.filter{ $0 % 2 == 0}}
filterSomeArray // [2, 4, 44, 6, 16, 10]
```

17.6. Метод zip(_:_:)

Функция `zip(_:_:)` предназначена для формирования последовательности пар значений, каждая из которых составлена из элементов двух базовых последовательностей. Другими словами, если у вас есть две последовательности и вам нужно попарно брать их элементы, группировать и складывать в новую последовательность, то эта функция как раз то, что нужно. Сначала вы берете первые элементы каждой последовательности, группируете их, потом берете вторые элементы, и т. д.

Пример использования функции `zip(_:_:)` приведен в листинге 17.13.

Листинг 17.13

```
let collectionOne = [1, 2, 3]
let collectionTwo = [4, 5, 6]
var zipSequence = zip(collectionOne ,collectionTwo)
type(of: zipSequence) // Zip2Sequence<Array<Int>, Array<Int>>.Type
// генерация массива из сформированной последовательности
Array(zipSequence) // [(.0 1, .1 4), (.0 2, .1 5), (.0 3, .1 6)]
// генерация словаря на основе последовательности пар значений
let newDictionary = Dictionary(uniqueKeysWithValues: zipSequence)
newDictionary // [1: 4, 3: 6, 2: 5]
```

Обратите внимание на еще один новый для вас тип данных `Zip2Sequence<Array<Int>, Array<Int>>`.

Со многими новыми типами данных вы познакомитесь в будущих главах, а со временем даже научитесь создавать собственные. Но настоящая магия начнется тогда, когда вы перестанете бояться таких

конструкций и поймете, что они значат и откуда появляются. Это неминуемо, если вы будете старательно продолжать обучение и пытаться делать чуть больше, чем сказано в книге.

17.7. Оператор guard для опционалов

Рассмотрим пример использования оператора `guard` при работе с опционалами.

Предположим, что название некоторой геометрической фигуры хранится в константе. Вам потребовалось реализовать механизм вывода на консоль сообщения, содержащего информацию о количестве сторон в данной фигуре. Для реализации задуманного напишем две функции:

- Первая — `countSidesOfShape` — возвращает количество сторон фигуры по ее названию.
- Вторая — `maybePrintCountSides` — выводит необходимое сообщение на консоль.

Почему лучше написать две функции вместо одной? Так как ваша программа предназначена для работы с геометрическими фигурами, то функция `countSidesOfShape` может потребоваться вам и для других целей. По этой причине имеет смысл разбить функционал.

Так как вы не можете заранее предусмотреть все варианты геометрических фигур, то в случае обработки неизвестной фигуры программа должна выводить сообщение о том, что ей неизвестно количество сторон для нее.

Реализуем функцию с именем `countSidesOfShape` (листинг 17.14).

Листинг 17.14

```
func countSidesOfShape(shape: String) -> Int? {
    switch shape {
    case "треугольник":
        return 3;
    case "квадрат":
        return 4;
    case "прямоугольник":
        return 4;
    default:
        return nil;
    }
}
```

Данная функция принимает имя фигуры на вход и возвращает количество ее сторон либо `nil`, если фигура неизвестна.

Далее реализуем функцию `maybePrintCountSides(shape:)`, принимающую на вход название фигуры (листинг 7.15).

Листинг 17.15

```
func maybePrintCountSides(shape: String) {
    if let sides = countSidesOfShape(shape: shape) {
        print("Фигура \(shape) имеет \(sides) стороны")
    }else{
        print("Неизвестно количество сторон фигуры \(shape)")
    }
}
```

Для получения количества сторон используется оператор условия `if`, осуществляющий проверку операции опционального связывания. Логика работы функции состоит в том, что, если фигура отсутствует в базе, не имеет смысла выполнять функцию, можно вывести информационное сообщение и досрочно завершить ее работу. Для этого можно использовать оператор раннего выхода `guard` (листинг 17.16).

Листинг 17.16

```
func maybePrintCountSides (shape: String)  {
    guard let sides = countSidesOfShape(shape: shape) else {
        print("Неизвестно количество сторон фигуры \(shape)")
        return
    }
    print("Фигура \(shape) имеет \(sides) стороны")
}
```

Оператор `guard` проверяет, возможно ли провести операцию опционального связывания, и в случае отрицательного результата выполняет код тела оператора, где с помощью `return` досрочно завершается работа функции.

Если опциональное связывание успешно завершается, то тело `guard` игнорируется и выполняется следующий за ним код.

С помощью `guard` код функции стал значительно более читабельным. Особенно это заметно, если код, следующий за оператором, будет занимать больше одной строки. С его помощью вы проверяете возможность получения всех необходимых параметров прежде, чем перейдете к выполнению кода функции.

18 Ленивые вычисления

Мы уже встречались с понятием ленивых вычислений и даже немного «пощупали» их руками. В этой главе пойдем дальше и углубим свои знания в данной теме.

18.1. Понятие ленивых вычислений

«Ленивый» в Swift звучит как lazy. Можно сказать, что lazy — синоним производительности. Хорошо оптимизированные программы практически всегда используют ленивые вычисления. Возможно, вы работали с ними и в других языках. В любом случае, внимательно изучите приведенный далее материал.

В программировании ленивыми называются такие элементы, вычисление значений которых откладывается до момента обращения к ним. Таким образом, пока значение не потребуется и не будет использовано, оно будет храниться в виде сырых исходных данных. С помощью ленивых вычислений достигается экономия процессорного времени, то есть компьютер не занимается ненужными в данный момент вычислениями.

Существует два типа ленивых элементов:

- *lazy-by-name* — значение элемента вычисляется при каждом доступе к нему;
- *lazy-by-need* — элемент вычисляется один раз при первом обращении к нему, после чего фиксируется и больше не изменяется.

Swift позволяет работать с обоими типами ленивых элементов, но в строгом соответствии с правилами.

18.2. Замыкания в ленивых вычислениях

С помощью замыканий мы можем создавать ленивые конструкции типа `lazy-by-name`, значение которых высчитывается при каждом обращении к ним.

Рассмотрим пример из листинга 18.1.

Листинг 18.1

```
var arrayOfNames = ["Helga", "Bazil", "Alex"]
print(arrayOfNames.count)
let nextName = { arrayOfNames.remove(at: 0) }
arrayOfNames.count // 3
nextName()
arrayOfNames.count // 2
```

В константе `nextName` хранится замыкание, удаляющее первый элемент массива `arrayOfNames`. Несмотря на то что константа объявлена, а ее значение проинициализировано, количество элементов массива не уменьшается до тех пор, пока не произойдет обращение к хранящемуся в ней замыканию.

Если пойти дальше, то можно сказать, что любая функция или метод являются `lazy-by-name`, так как их значение высчитывается при каждом обращении.

18.3. Свойство lazy

Некоторые конструкции языка Swift (например, массивы и словари) имеют свойство `lazy`, позволяющее преобразовать их в ленивые. Наиболее часто это происходит, когда существуют цепочки вызова свойств или методов и выделение памяти и вычисление промежуточных значений является бесполезной тратой ресурсов, так как эти значения никогда не будут использованы.

Рассмотрим следующий пример: существует массив целых чисел, значения которого непосредственно не используются в работе программы. Вам требуется лишь результат его обработки методом `map(_:)`, и то не в данный момент, а позже (листинг 18.2).

Листинг 18.2

```
var baseCollection = [1,2,3,4,5,6]
var myLazyCollection = baseCollection.lazy
```

```
type(of:myLazyCollection) // LazyCollection<Array<Int>>.Type
var collection = myLazyCollection.map{$0 + 1}
type(of:collection) // LazyMapCollection<Array<Int>, Int>.Type
```

В результате выполнения возвращается ленивая коллекция. При этом память под отдельный массив целочисленных значений не выделяется, а вычисления метода `map(_:)` не производятся до тех пор, пока не произойдет обращение к переменной `collection`.

Вся прелесть такого подхода в том, что вы можете увеличивать цепочки вызовов, но при этом лишнего расхода ресурсов не будет (листинг 18.3).

Листинг 18.3

```
var resultCollection = [1,2,3,4,5,6].lazy.map{$0 + 1}.filter{$0 % 2
                    == 0}
Array(resultCollection) // [2, 4, 6]
```

Часть V.
Введение в разработку приложений

Позади большое количество материала, и нам пора сделать небольшой перерыв. В данной части книги вы создадите свои первые полноценные приложения в Xcode.

Не так важно, помните ли вы все, что изучали ранее, или нет, так как в процессе разработки вы будете возвращаться к материалу, повторяя и закрепляя его. Со временем, при определенном упорстве, вы будете чувствовать себя в среде разработки на Swift словно рыба в воде.

Надеюсь, что эта часть станет толчком к дальнейшему самостоятельному изучению материала.

- ✓ Глава 19. Консольное приложение «Сумма двух чисел»
- ✓ Глава 20. Консольная игра «Отгадай число»

19 Консольное приложение «Сумма двух чисел»

Процесс обучения разработке приложений в Xcode на Swift увлекателен и интересен. Многие из современных учебных материалов направлены не на глубокое поэтапное изучение, а на поверхностное и узкое решение определенных задач. Например, обучение решению задачи «Как написать калькулятор в Xcode на Swift» само по себе не научит вас разработке, но если к этому вопросу подойти с умом, комплексно, по ходу объясняя весь новый и повторяя весь прошедший материал, то результат гарантирован! Именно такого подхода мне и хочется придерживаться. Из-за ограничений объема книги я, конечно, не смогу дать вам все знания, которыми обладаю, но все необходимые начальные навыки вы получите.

В этой части книги мы вновь вернемся к изучению интерфейса Xcode, после чего создадим первое приложение для операционной системы macOS: программу «Сумма двух чисел». Несмотря на то что основной целью книги является изучение Swift, я считаю разработку реальных проектов отличным способом усвоить изученный материал, это наиболее интересный и простой способ сделать первые шаги в освоении Xcode. Обратите внимание, что мы будем взаимодействовать не с Xcode Playground, а с полноценным Xcode, позволяющим воплощать в программах любые ваши идеи.

19.1. Обзор интерфейса Xcode

Прежде чем переходить к созданию полноценного приложения, рассмотрим основные элементы Xcode, которые будут использованы при создании практически любого приложения.

Создание Xcode-проекта

На протяжении книги вам предстоит создать несколько проектов, поэтому отнеситесь к данному материалу со всей серьезностью и возвращайтесь к нему в случае возникновения трудностей.

Откройте стартовое окно Xcode (рис. 19.1). Как уже говорилось, стартовое окно служит для создания новых проектов и доступа к созданным ранее. Во время изучения Swift мы использовали исключительно кнопку Get started with a playground. Теперь вам необходимо начать процесс создания нового Xcode-проекта, нажав Create a new Xcode project.

Рис. 19.1. Стартовое окно Xcode

Перед вами появится окно выбора шаблона приложения (рис. 19.2).

В верхней части окна вам доступен выбор платформы, под которой будет функционировать приложение (рис. 19.3).

В настоящий момент список состоит из следующих пунктов:

- iOS — разработка приложения под iPhone и iPad.
- watchOS — разработка приложения под смарт-часы Apple Watch.
- tvOS — разработка приложения под телевизионную приставку Apple TV.
- macOS — разработка приложения под настольные и мобильные персональные компьютеры iMac и Macbook.
- Cross-platform — разработка кросс-платформенного приложения.

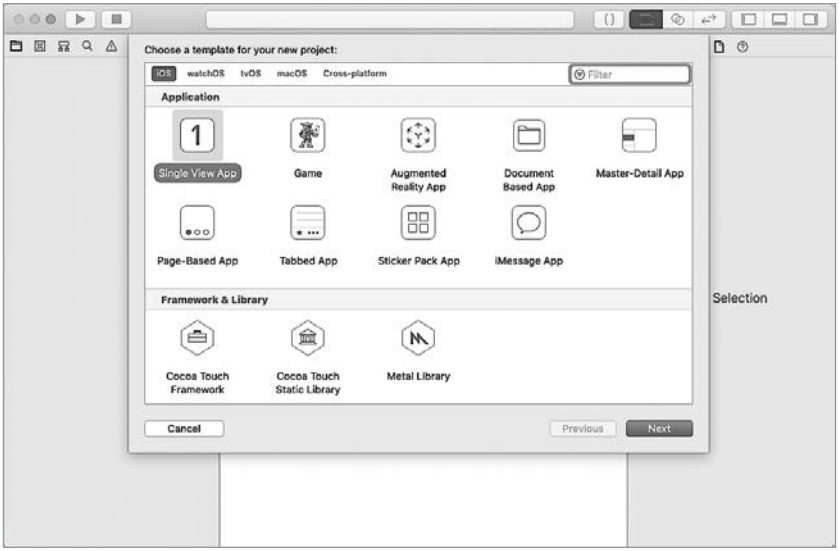

Рис. 19.2. Окно выбора шаблона проекта

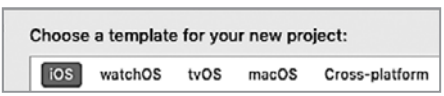

Рис. 19.3. Выбор платформы

В этой главе мы создадим приложение для настольной операционной системы, поэтому вам необходимо выбрать пункт macOS.

Ниже списка выбора системы обновится список доступных шаблонов. Выбор шаблона не ограничивает функциональность будущей программы — это всего лишь набор предустановленных настроек (например, подключенных библиотек).

В процессе разработки вы сможете поработать со многими из них. Частота их использования зависит от того, какие задачи вы будете перед собой ставить. Сейчас нас интересует шаблон Command Line Tool. Выделите его, после чего нажмите кнопку Next. Далее потребуется ввести идентификационные данные и первичные настройки будущей программы (рис. 19.4).

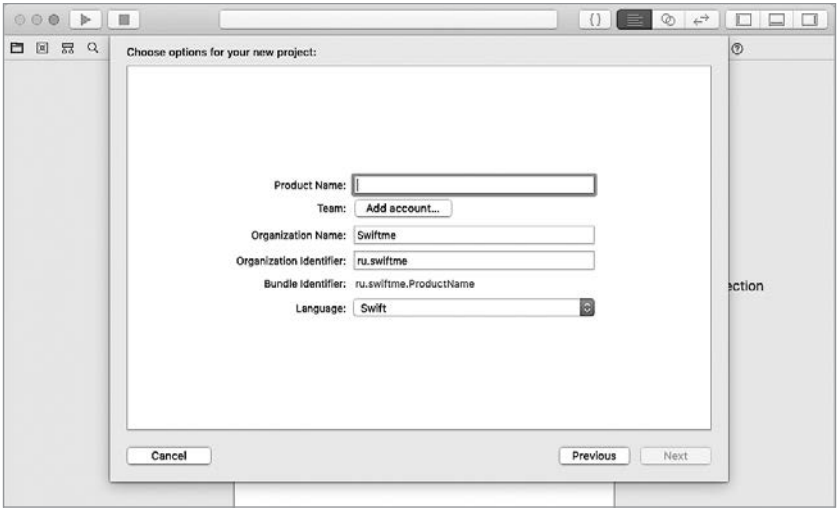

Рис. 19.4. Окно первичной настройки приложений

В ходе настройки вы сможете определить следующие параметры:

- Product Name — название будущего проекта. Введите `"Swiftme-FirstApp"`.
- Team — так как мы не планируем размещать программу в магазине приложений, данный пункт оставим без изменений.
- Organization Name — название разработчика проекта. Введите название вашей фирмы или свои имя и фамилию.
- Organization Identifier — идентификатор вашего разработчика. Обычно в качестве идентификатора используют перевернутое доменное имя вашей организации, например `"com.myorganization"`. При этом Xcode не связывается с каким-либо доменом в Сети, его цель состоит лишь в том, чтобы однозначно идентифицировать разработчика. Organization Identifier должен быть уникальным среди всех разработчиков, размещающих свои приложения в AppStore. Введите произвольное значение.
- Bundle Identifier — данная строка генерируется автоматически и состоит из имени вашей организации и названия вашего проекта. Это уникальный идентификатор вашего приложения. Он чувствителен к регистру.

- Language — язык программирования, на котором будет написано ваше приложение. Вам следует выбрать Swift.

Подтвердим создание Xcode-проекта, нажав кнопку Next и указав место для его сохранения.

Интерфейс Xcode-проекта

Перед вами откроется рабочая среда Xcode (рис. 19.5).

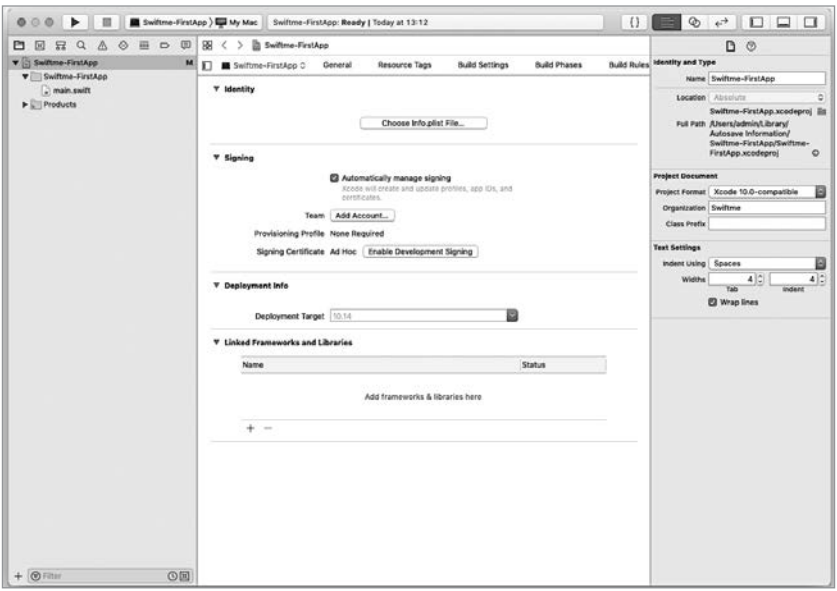

Рис. 19.5. Рабочая среда Xcode

ПРИМЕЧАНИЕ Xcode просто создан для того, чтобы эффективно программировать. Каждый его элемент предназначен для быстрого и удобного размещения, а также настройки необходимого функционала ваших приложений.

Интерфейс Xcode-проекта отличается от уже знакомого вам Xcode Playground. Его рабочая среда предоставляет доступ ко всему функционалу, с помощью которого создаются и управляются проекты. Она автоматически подстраивается под решаемую задачу. На рис. 19.5 изображен один из множества вариантов внешнего вида рабочей среды. Со временем вы узнаете все возможности этой программы.

Рабочая среда Xcode состоит из пяти основных рабочих областей (рис. 19.6): Navigator Area (панель навигации), Toolbar (панель инструментов), Utilities Area (панель утилит), Debug Area (панель отладки, пока что скрыта) и Project Editor (редактор проекта).

ПРИМЕЧАНИЕ Значения некоторых полей вашей рабочей среды Xcode и среды, изображенной на рис. 19.6, могут отличаться.

Панель инструментов (Toolbar) находится в верхней части рабочей среды Xcode. В ее левой части находятся кнопки запуска и остановки проекта.

Большой прямоугольник посередине панели инструментов носит название панели статуса (Statusbar). В ней отображаются все действия и процессы, которые происходят с проектом. Например, когда вы запускаете проект, в данной области отображается каждый шаг сборки вашего приложения, а также возникающие ошибки.

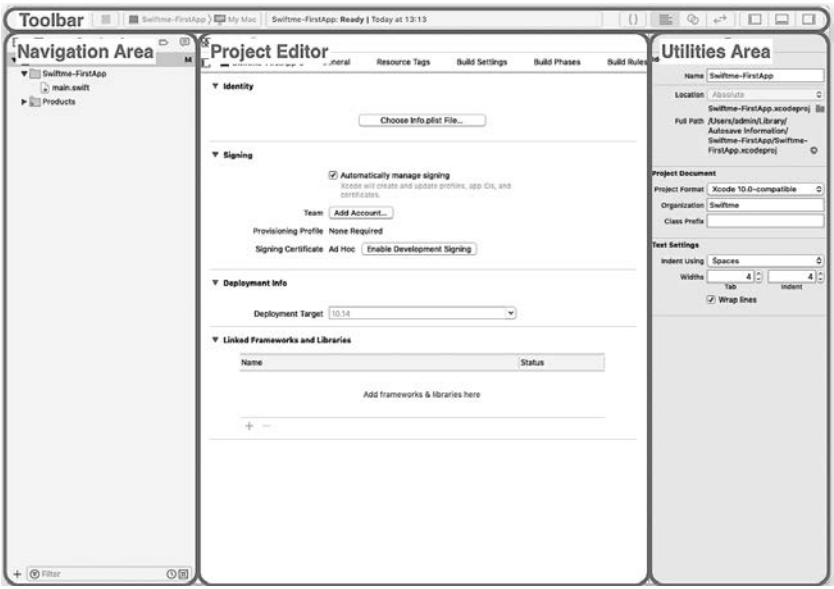

Рис. 19.6. Основные области рабочей среды Xcode

С правой стороны панели инструментов располагаются три набора кнопок: одна отдельная и две группы по три кнопки (рис. 19.7).

Рис. 19.7. Кнопки в правой части Toolbar

С помощью первой кнопки можно получить доступ к библиотекам сниппетов, объектов и медиаресурсов. Мы воспользуемся ее возможностями позже.

Кнопки из первой трехкнопочной группы позволяют переключаться между различными конфигурациями редактора проекта:

- Standart Editor (активен по умолчанию) отображает редактор проекта в виде одной рабочей зоны.
- Assistant Editor делит редактор на несколько рабочих зон. Например, вы можете одновременно редактировать несколько файлов со сходным кодом или применить схему, при которой будет одновременно отображаться код проекта и визуальный интерфейс приложения.
- Version Editor превращает вашу рабочую зону в своеобразную машину времени и позволяет работать с предыдущими версиями файлов проекта. Эта функция доступна при наличии настроенной системы контроля версий.

С помощью правого набора кнопок вы можете управлять отображением областей рабочей среды Xcode-проекта. При этом для активированных кнопок используется синий цвет, для деактивированных — серый. Щелкните по серой центральной кнопке, после чего в рабочей среде Xcode появится область отладки (Debug Area) (рис. 19.8).

Рис. 19.8. Область отладки Xcode-проекта

Щелкните несколько раз по каждой из кнопок панели отображения областей и посмотрите, что изменится в рабочей среде Xcode.

- При нажатии на кнопку  будет скрываться/отображаться панель навигации (Navigator Area), с помощью которой можно переходить между файлами, а также другими категориями вашего проекта.

- Кнопка  позволяет скрыть/отобразить область отладки (Debug Area), которую вы будете использовать в процессе поиска и устранения ошибок.
- С помощью кнопки  будет скрываться/отображаться панель утилит (Utilities Area). Ее основным назначением является добавление в проект функциональных и графических элементов, просмотр и модификация различных атрибутов, а также доступ к справочной информации.

В процессе обучения мы поработаем с каждой из описанных выше панелей.

19.2. Подготовка к разработке приложения

В дальнейшем мы будем подробнее изучать каждую область и ее функциональные элементы. Сейчас перейдем к разработке консольного приложения «Сумма двух чисел».

ПРИМЕЧАНИЕ Консольные приложения в macOS запускаются и работают в среде программы Терминал, которую вы можете найти в macOS в папке Программы ▸ Утилиты.

Наличие удобного интерфейса — это обязательное требование практически к любой программе. В некоторых случаях нет смысла нанимать профессионального дизайнера или думать о том, как правильно расположить элементы. Все зависит от решаемой задачи, которая и определяет способ доведения информации до пользователя. Возможно, вашему приложению будет достаточно интерфейса командной строки или оно должно будет работать только на планшетах, а быть может, для его использования потребуется дисплей с диагональю от 21 дюйма. Повторюсь, все зависит от поставленной задачи. Но в любом случае всегда старайтесь найти наиболее приемлемый для программы интерфейс.

Ваши программы всегда должны обладать достаточным уровнем удобства при использовании, что в первую очередь и обеспечивается способом отображения информации и взаимодействия с приложением. Приложение, которое мы разрабатываем, не будет иметь каких-либо графических элементов, кроме командной строки. Оно будет запрашивать у пользователя два числа, а затем выводить их сумму.

Прежде чем приступить к разработке, обратимся к области Navigator Area, расположенной в левой части рабочей среды (рис. 19.9).

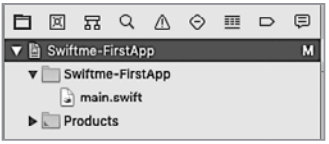

Рис. 19.9. Панель Navigator Area

С помощью пиктограмм, расположенных в верхней части области навигации, вы можете переключаться между девятью различными разделами навигации вашего проекта. По умолчанию Navigator Area отображает Project Navigator (навигатор проекта). Неважно, какие ресурсы входят в состав вашего проекта: файлы с исходным кодом, модели данных, списки настроек, файлы раскадровок (что это такое, вы узнаете позже), — все будет отображено в навигаторе проекта. Сейчас ваш проект состоит только из набора папок и одного файла main.swift.

Верхний элемент представленного дерева в Project Navigator имеет название, совпадающее с названием проекта Swiftme-FirstApp. Он определяет ваш проект в целом. Выделив его, вы можете перейти к конфигурированию приложения или, другими словами, перейти к его настройкам. В будущих главах мы познакомимся с некоторыми из них.

Помимо файла, в состав проекта входят две папки:

- Swiftme-FirstApp — группа ресурсов, всегда имеющая такое же имя, как и сам проект. В данной папке группируются все файлы и ресурсы, которые используются в вашей программе. С ее помощью вы легко можете организовать структуру всех входящих в состав проекта ресурсов, в том числе с помощью создания собственных папок.
- Products — папка, содержащая приложение, которое будет сформировано после запуска проекта.

В Project Navigator раскройте содержимое папки Products. Вы увидите, что название файла Swiftme-FirstApp (со значком консольного приложения слева от него) написано красным цветом. Это связано с тем, что проект еще не был запущен, а его исходный код не откомпилирован.

Теперь щелкните по файлу main.swift. Вы увидите, что Project Editor изменится: в нем отобразится редактор кода. Также изменится и панель Utilities Area.

ПРИМЕЧАНИЕ Файлы с расширением swift содержат исходный код приложения.

ПРИМЕЧАНИЕ В зависимости от того, какой объект в панели навигации является активным, содержимое Project Editor и Utilities будет соответствующим образом изменяться.

В случае, когда в панели навигации выбран файл с исходным кодом, которым и является main.swift, в редакторе проекта появляется возможность изменять его содержимое.

19.3. Запуск приложения

Сейчас мы рассмотрим запуск создаваемого приложения. Этот процесс всегда включает одни и те же шаги и не зависит от платформы, под которую вы разрабатываете приложение.

Так как разрабатываемое приложение является консольным, для отображения результатов его работы и взаимодействия с ним, конечно же, используется консоль. Она находится в правой части Debug Area.

Отобразите (если скрыта) область Debug Area среды Xcode-проекта, а в качестве активного файла в области навигации выберите main.swift.

В данный момент вывод на консоли Xcode пуст, так как приложение еще ни разу не было запущено и не производило каких-либо выводов. Если взглянуть на код, написанный в редакторе (листинг 19.1), то вы увидите директиву `import` и функцию `print(_:)`.

Листинг 19.1

```
import Foundation
print("Hello, World!")
```

Напомню, что функция `print(_:)` предназначена для вывода текстовой информации на консоль. В качестве входного аргумента она принимает данные, которые необходимо вывести.

Чтобы приложение осуществило вывод информации на консоль, оно должно быть запущено. Для управления работой проекта в левой части Toolbar расположены специальные кнопки (рис. 19.10).

Рис. 19.10. Кнопки управления работой приложения

Кнопка с изображением стрелки, называющаяся Build and run, активирует сборку приложения с его последующим запуском.

Кнопка с квадратом позволяет досрочно прекратить процесс сборки или завершить работу запущенного приложения.

Теперь запустим наше приложение. Для этого нажмите кнопку Build and run. Обратите внимание, что текущее состояние процесса отображается в Statusbar (рис. 19.11). Спустя несколько секунд на консоли (в области отладки) вы увидите вывод вашего консольного приложения, который осуществила функция `print(_:)` (рис. 19.12).

Рис. 19.11. Statusbar отображает текущее состояние приложения

Рис. 19.12. Вывод на консоль

Теперь, если обратиться к Project Navigator, то можно увидеть, что в папке Products файл Swiftme-FirstApp изменил свой цвет с красного на черный. Это говорит о том, что он был автоматически создан и сохранен на диске вашего компьютера. Выделите файл щелчком левой кнопки мыши и посмотрите на Utilities Area. Убедитесь, что активирован раздел File Inspector ( ). Атрибут Full Path указывает путь на расположение активного ресурса, в данном случае это скомпилированное консольное приложение (рис. 19.13). Чтобы открыть его в Finder, достаточно щелкнуть на кнопке с изображением стрелки ( ), расположенной справа от пути. Нажмите на нее, и перед вами откроется файловый менеджер Finder с файлом консольного приложения (рис. 19.14).

Рис. 19.13. Панель атрибутов файла SumOfNumbers

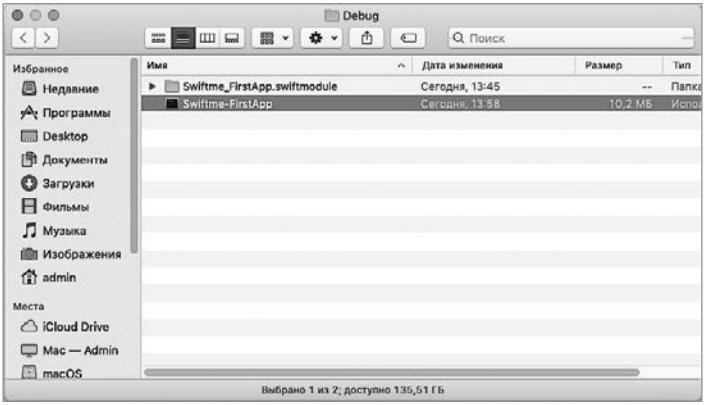

Рис. 19.14. Расположение программы SumOfNumbers

Запустите приложение, дважды щелкнув по нему в `Finder`, и вы увидите результат его выполнения в программе Терминал (рис. 19.15).

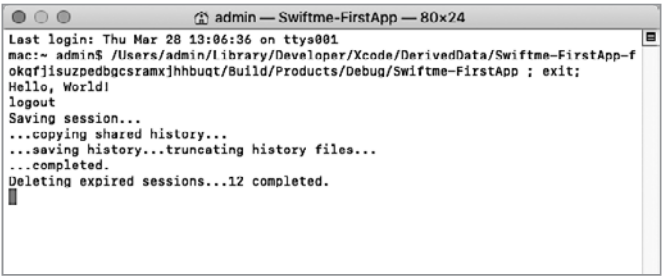

Рис. 19.15. Результат работы программы SumOfNumbers

19.4. Код приложения «Сумма двух чисел»

Разрабатываемое приложение пока еще не способно решать возложенную на него задачу по сложению двух введенных пользователем чисел. Для этого потребуется написать программный код, реализующий данный функционал. В Project Navigator щелкните по файлу main.swift и удалите из него все содержимое (комментарии удалять не обязательно).

Напомню, что разрабатываемая программа должна запрашивать у пользователя значения двух аргументов, производить их сложение и выводить результат на консоль. Для получения значений, вводимых

с клавиатуры, в консольных приложениях служит функция `readLine()`. Она ожидает ввода данных с клавиатуры с последующим нажатием кнопки Enter, после чего возвращает значение типа `String?` (опциональный строковый тип данных).

Напишем код, который будет запрашивать значения параметров `a` и `b` (листинг 19.2).

Листинг 19.2

```
print("Введите значение первого аргумента")
// получение первого аргумента
var a = readLine()
print("Введите значение второго аргумента")
// получение второго аргумента
var b = readLine()
```

Следующей задачей станет подсчет суммы введенных значений. Для этого требуется создать специальную функцию `sum(_:_:)`, принимающую на вход два значения типа `String?` в качестве операндов операции сложения. Несмотря на то что наше приложение требует довольно мало кода, немного оптимизируем его структуру, написав функцию `sum(_:_:)` в отдельном файле.

Щелкните правой кнопкой мыши на папке Swiftme-FirstApp в Navigation Area и выберите пункт New File.... В появившемся окне выберите Swift File (рис. 19.16), нажмите Next и введите имя `func` для создаваемого файла. После нажатия Create файл появится в Navigation Area.

Щелкните по файлу func.swift, в котором мы реализуем функцию `sum(_:_:)`, предназначенную для сложения двух чисел. Входные аргументы функции будут иметь тип `String?` (соответствует типу возвращаемого функцией `readLine()` значения), а внутри ее реализации перед операцией сложения — преобразовываться к `Int`.

Упростить процесс создания функции помогут кодовые сниппеты. На панели Toolbar нажмите кнопку Library с изображением двух фигурных скобок, после чего отобразится список кодовых сниппетов. Далее с помощью поля поиска отфильтруйте их по фразе «`func`». Среди отобразившихся элементов дважды щелкните по Swift Function Statement. В результате этих действий в области редактора появится шаблон функции, в котором потребуется заполнить несколько полей (рис. 19.17).

Выделив поле name в шаблоне, вы сможете ввести имя функции, а с помощью клавиши Tab на клавиатуре — перескакивать к следующему требуемому полю. Используя созданный шаблон, напишите функцию `sum(_:_:)` (листинг 19.3).

Рис. 19.16. Создание файла func.swift

```
func name(parameters) -> return type {
    function body
}
```

Рис. 19.17. Кодовый сниппет функции

Листинг 19.3

```
func sum(_ a: String?, _ b: String?) -> Int {
    return Int(a!)! + Int(b!)!
}
```

ЗАДАНИЕ

Ответьте, с чем связано то, что при вычислении значения параметра `result` используется по две пары знаков восклицания для каждого аргумента операции сложения, например `"Int(a!)!"`?

РЕШЕНИЕ

Значение, переданное на вход функции `Int(_:)`, не должно быть опциональным. С этим связан знак принудительного извлечения опционального значения внутри `Int(_:)`, то есть первый знак восклицания.

Оператор сложения внутри функции `sum(_:_:)` может производить операцию сложения только с неопциональными значениями, в то время как функция `Int(_:)` в качестве результата своей работы возвращает значение типа `Int?`. Именно по этой причине ставится второй знак восклицания.

Несмотря на то что функция `sum(_:_:)` описана в файле func.swift, она может использоваться без каких-либо ограничений и в других файлах проекта. Добавьте в конец файла main.swift код из листинга 19.4.

Листинг 19.4

```
let result = sum(a, b)
print("Результат сложения - \(result)")
```

Поздравляю, ваша программа готова! Запустите ее и с помощью консоли в панели отладки попробуйте произвести сложение двух чисел (рис. 19.18).

Осталось лишь сохранить программу в виде автономного приложения. Вы уже знаете один способ получить к нему доступ, открыв его в Finder с помощью атрибута Full Path. Но есть еще один способ: в Project Navigator в папке Product выберите файл Swiftme-FirstApp, зажмите Opt (Alt) и перетащите его в произвольное место на вашем компьютере (например, на рабочий стол или в программы).

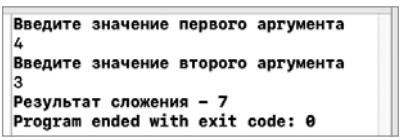

Рис. 19.18. Работа консольного приложения в области отладки

Написанная программа неидеальна, она будет вести себя некорректно, например, при вводе букв в качестве запрашиваемых значений. Да и вообще то, что наличие значений в опционалах не проверяется, — это ужасно, но вы исправите это самостоятельно при выполнении домашней работы.

20 Консольная игра «Отгадай число»

Консоль замечательно подходит для создания простых в реализации приложений с минимальным функционалом. В этой главе мы создадим игру «Отгадай число», в которой пользователь должен отгадать случайное число, загаданное компьютером.

Суть игры в следующем: компьютер загадывает целое число в диапазоне от 1 до 50, а игрок пытается угадать его за минимальное количество ходов. После каждой попытки приложение должно сообщать, как введенное число соотносится с загаданным: оно больше, меньше или равно ему. Игровой процесс продемонстрирован на рис. 20.1.

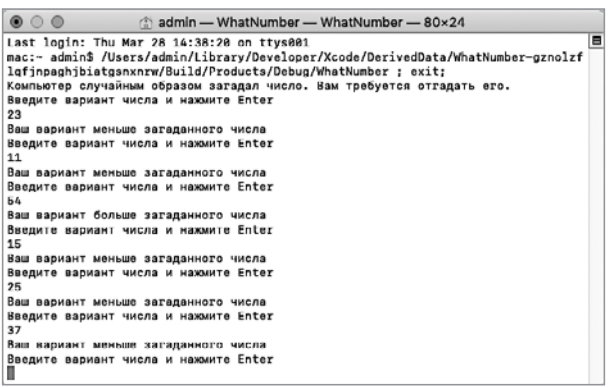

Рис. 20.1. Игра «Отгадай число»

Наша будущая игра будет функционировать по следующему алгоритму:

- Генерация случайного числа.
- Запрос числа у пользователя.

- Сопоставление сгенерированного числа с запрошенным.
- Вывод результата сопоставления.
- Если числа одинаковые, то работа программы завершается.
- Если числа разные, то происходит переход к шагу 2.

20.1. Код приложения «Угадай число»

Создайте в Xcode новый консольный проект с именем «UnknownNumber». После этого удалите весь код, находящийся в файле main.swift.

ПРИМЕЧАНИЕ Если вы забыли, как создается консольное приложение, то вернитесь к предыдущей главе.

Для начала в файле main.swift реализуем механизм генерации случайного числа (листинг 20.1).

Листинг 20.1

```
import Foundation
// генерация случайного числа
let randomNumber = UInt8(arc4random_uniform(50))
```

Теперь константа `randomNumber` содержит случайно сгенерированное число.

ПРИМЕЧАНИЕ Напомню, что для генерации случайного числа используется функция `arc4random_uniform(_:)`, которая входит в состав библиотеки Foundation.

Пара слов про оптимизацию приложения. В общем случае любая оптимизация — это поиск компромисса, обычно между памятью и процессорным временем устройства. Обратите внимание, что в листинге 20.1 в качестве типа данных константы `randomNumber` используется `Int8`. Если не определить тип данных самостоятельно, то Swift автоматически определит его как `Int`, а это 32 (или 64) бита памяти вместо 8. Да, конечно, в данном случае экономия не имеет какой-либо практической пользы, но я настоятельно советую вам привыкать к процессу оптимизации с самого начала вашего пути разработчика. Не бойтесь ошибок, ведь автодополнение и справка в Xcode всегда подскажут вам правильный путь.

Тема оптимизации очень широка и интересна. Порой вы даже можете пойти на некоторые траты ресурсов в угоду читабельности кода, но, тем не менее, я советую вам искать пути оптимизации любого кода, который вы создаете.

ПРИМЕЧАНИЕ Случайные числа, генерируемые большинством языков программирования (в том числе и Swift), в действительности являются псевдослучайными! Для их вычисления используются алгоритмы, не использующие какие-либо непредсказуемые составляющие. Да и вообще, ничего случайного внутри вашего компьютера нет, все подчиняется заранее запрограммированным алгоритмам. Для создания по-настоящему случайных чисел используются специальные аппаратные решения, обычно измеряющие значения внешних физических явлений. Но вам не стоит задумываться об этом, так как функции `arc4random_uniform(_:)` вполне достаточно для программы практически любого уровня сложности.

Шаги 2–6 описанного выше алгоритма — это цикл. Ваша программа не должна завершать работу до тех пор, пока число не будет отгадано. Для реализации этого условия лучше всего использовать конструкцию `repeat while` (листинг 20.2).

Листинг 20.2

```
import Foundation
// генерация случайного числа
let randomNumber = UInt8(arc4random_uniform(50))
print("Компьютер случайным образом загадал число. Вам требуется
отгадать его.")
// в переменную будет записываться число с консоли
var myNumber: String?
// цикл с постпроверкой условия
repeat {
    print("Введите ваш вариант и нажмите Enter")
    // получение значения с клавиатуры пользователя
    myNumber = readLine()
    // сравнение введенного и сгенерированного чисел
    if (UInt8(myNumber!) == randomNumber){
        print("Вы угадали!")
    }else if (UInt8(myNumber!)! < randomNumber){
        print("Ваш вариант меньше загаданного числа")
    }else if (UInt8(myNumber!)! > randomNumber){
        print("Ваш вариант больше загаданного числа")
    }
} while randomNumber != UInt8(myNumber!)
```

Ваша программа готова. Запустите ее и попробуйте себя в борьбе с искусственным интеллектом.

20.2. Устраняем ошибки приложения

Если бы вы сдали это приложение на `code rewiev` (проверка кода) или отдали его тестировщикам, то, скорее всего, лишились бы премии, так как, реализуя его код, мы пошли по пути наименьшего сопро-

тивления и реализовали программу по принципу «авось прокатит». Выделим основные проблемы вашей/нашей программы:

- Многократное использование `UInt8`.
- Аварийное завершение работы приложения при вводе нецифровых символов.
- Аварийное завершение работы приложения при вводе числа больше 255 (верхняя граница типа `UInt8`).
- При доступе к значению опционала используется принудительное извлечение значения.

Если вы чувствуете в себе силы самостоятельно исправить эти ошибки, то дерзайте. В любом случае ознакомьтесь с решением, которое предлагаю вам я (листинг 20.3).

Листинг 20.3

```
import Foundation
print("Компьютер случайным образом загадал число. Вам требуется
отгадать его.")

// словарь сообщений
let message = [
    "start":"Введите вариант числа и нажмите Enter",
    "more":"Ваш вариант больше загаданного числа",
    "less":"Ваш вариант меньше загаданного числа",
    "win":"Вы угадали число!"]
// храним загаданное число в виде строки, чтобы избежать
// тройного преобразования:
// 1) результат функции readLine(_:) из String? в String
// 2) полученный String в UInt8? с помощью UInt8(_:)
// 3) полученный UInt8? в UInt8 для сравнения с randomNumber
let randomNumber = String(arc4random_uniform(50))
// введенное пользователем число
var userNumber: String = ""
// цикл проверки
repeat {
    print(message["start"]!)
    //получение числа
    let myNumber = readLine()
    userNumber = myNumber ?? ""
    if userNumber < randomNumber{
        print(message["less"]!)
    }else if userNumber > randomNumber{
        print(message["more"]!)
    }
} while userNumber != randomNumber

print(message["win"]!)
```

Вот она, оптимизация! Решение сменить тип переменной `randomNumber` с `UInt8` на `String` позволило убрать все описанные проблемы, при этом доступ к опционалу, возвращаемому функцией `readLine(_:)`, теперь происходит с помощью безопасного извлечения значения с использованием оператора `??`.

Обратите внимание, что все информационные сообщения были вынесены в словарь message. Такой подход позволяет, с одной стороны, с легкостью менять эти сообщения, даже если они были использованы в нескольких местах в коде программы, а с другой — открывает простейший путь к локализации (вы можете, к примеру, определить активный язык системы и подгрузить необходимые значения в данный словарь).

Часть VI. Нетривиальные возможности Swift

В предыдущих частях книги вы плавно погружались в основы разработки на Swift и даже написали свои первые приложения. Но все, что вы сейчас знаете, — это вершина айсберга, самое интересное еще впереди.

Так как Swift придерживается парадигмы «всё — это объект», то любой параметр (переменная или константа) с определенным типом данных — это объект. Для реализации новых объектов вы уже изучили множество различных типов данных, но, как отмечалось ранее, Swift обладает функционалом создания собственных объектных типов. Для этого существуют три механизма: перечисления (`enum`), структуры (`struct`) и классы (`class`). В чем разница между ними? Как их создавать и использовать? Все это будет рассказано в данной части книги. Мы обсудим, что такое объектные типы в общем и в чем разница между ними. Следующим шагом станет изучение механизмов, позволяющих расширить возможности объектных типов, включая протоколы, расширения, универсальные шаблоны и т. д.

- ✓ Глава 21. Введение в объектно-ориентированное программирование
- ✓ Глава 22. Перечисления
- ✓ Глава 23. Структуры
- ✓ Глава 24. Классы
- ✓ Глава 25. Свойства
- ✓ Глава 26. Сабскрипты
- ✓ Глава 27. Наследование
- ✓ Глава 28. Псевдонимы Any и AnyObject
- ✓ Глава 29. Инициализаторы и деинициализаторы

- ✓ Глава 30. Удаление экземпляров и ARC
- ✓ Глава 31. Опциональные цепочки
- ✓ Глава 32. Расширения
- ✓ Глава 33. Протоколы
- ✓ Глава 34. Разработка приложения в Xcode Playground
- ✓ Глава 35. Универсальные шаблоны
- ✓ Глава 36. Обработка ошибок
- ✓ Глава 37. Нетривиальное использование операторов

21 Введение в объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) является основой разработки программ на языке Swift. Мы уже неоднократно встречались с одним фундаментальным для данного языка правилом «всё — это объект». Пришло время приступить к изучению наиболее интересных и сложных механизмов, доступных в Swift. Данная глава расскажет вам о наиболее важных терминах и понятиях объектно-ориентированного программирования.

Возможно, вы изучали ООП в прошлом на уроках информатики в школе или в институте, а может, самостоятельно. В таком случае вы, конечно же, знаете три постулата ООП: инкапсуляция, наследование и полиморфизм. Я настоятельно советую вам потратить свободное время и дополнительно разобраться с ними так как в связи с ограничением на объем книги в основном будет показана практическая сторона ООП. Но в данном случае общая теоретическая база крайне важна.

21.1. Экземпляры

Основные конструкции, с которыми мы будем работать в этой части, — перечисления, структуры и классы. Открою вам один удивительный секрет: вы уже давно работаете с некоторыми из них! Например, тот же целочисленный тип данных `Int` — это структура.

Перечисления, структуры и классы имеют свои особенности и возможности, и именно они делают Swift таким, какой он есть. Тем не менее у них есть одно общее базовое свойство — для них могут быть созданы *экземпляры*. Когда вы самостоятельно определяете объектный

тип (перечисление, структуру или класс), вы лишь создаете новый тип данных. В свою очередь, создание экземпляра объектного типа — это создание хранилища (переменной или константы) для данных определенного типа. Так, например, упомянутый ранее тип `Int` — это структура, а переменная `myInteger`, хранящая в себе значение этого типа, — экземпляр данной структуры.

ПРИМЕЧАНИЕ Несмотря на то что мы много раз говорили об объектах, правильнее называть их экземплярами.

Объектами в других языках программирования назывались экземпляры классов, а экземпляры структур и перечислений — просто экземплярами. Так как функционал структур, перечислений и классов очень близок по своим возможностям, в Swift соответствующие объекты называются просто экземплярами.

Ранее в книге мы говорили об экземплярах и даже приводили несколько примеров их реализации. Помните проведенные ранее аналогии между категорией и протоколом, стандартом и типом данных, конкретной реализацией и экземпляром (табл. 21.1)?

Таблица 21.1. Соответствия между понятиями

Пример из реального мира	Понятия реального мира	Понятия в Swift	Пример из Swift
Автомобили с бензиновым двигателем	Категория	Протокол	SignedNumber (требует, чтобы тип данных обеспечивал хранение как положительных, так и отрицательных чисел)
Kia Rio	Стандарт (модель)	Тип данных	Int
Kia Rio VIN XW122FX849	Конкретная реализация	Экземпляр	2 (целое число типа Int)

ПРИМЕЧАНИЕ Данная таблица уже рассматривалась в книге ранее в главе 7.

Вы уже умеете создавать и использовать экземпляры конкретных типов данных. В этой части вы научитесь создавать сперва сами типы данных, а потом и протоколы. Вы можете использовать все свое воображение, создавая типы данных. А используются для этого именно классы, структуры и перечисления.

ПРИМЕЧАНИЕ Приведенная выше схема «Протокол — Тип данных — Экземпляр» (см. табл. 21.1) является «идеальной», но при этом совершенно не обязательно реализовывать протоколы, так как типы данных могут быть созданы напрямую. Обо всем этом вы узнаете из книги.

Рассмотрим простой пример.

Представьте, что определен класс Automobile (автомобиль). Этот класс является типом данных. Данный «Автомобиль» является не каким-то конкретным автомобилем, а лишь конструкцией, с помощью которой можно определить этот конкретный автомобиль. Если создать переменную типа `Automobile`, то в результате мы получим экземпляр этого класса (рис. 21.1).

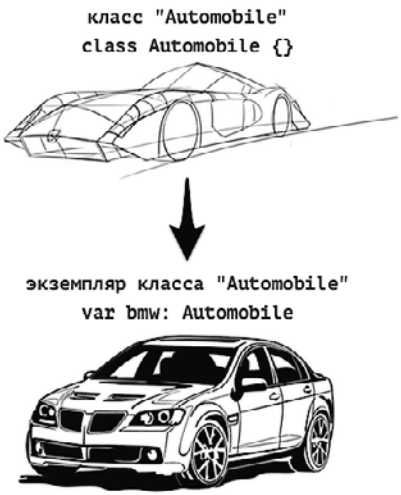

Рис. 21.1. Класс и его экземпляр

Сам класс на рисунке не имеет выраженных черт, так как еще неизвестно, какой же определенный объект реального (или нереального) мира он будет определять. Но когда создана переменная `bmw` типа `Automobile`, мы уже знаем, что с экземпляром в этой переменной мы будем работать как с реальным авто марки BMW:

```
bmw: Automobile = Automobile()
```

Такой тип данных может наделять экземпляр какими-либо характеристиками. Для класса `Automobile` это могли бы быть марка, модель,

цвет, максимальная скорость, объем двигателя и т. д. Характеристики объектных типов называются *свойствами*, с ними мы встречались неоднократно. Для переменной `bmw` значения этих свойств могли бы быть следующими:

```
bmw.brand = "BMW"
bmw.type = "X3"
bmw.maxSpeed = 210
bmw.engineCapacity = 1499
```

Свойства представляют собой хранилища данных, то есть это те же самые переменные и константы, но с ограниченным доступом: они доступны только через экземпляр. Иначе говоря, вначале вы получаете доступ к экземпляру, а уже потом (через точку) — к его свойству.

Помимо свойств, у экземпляра могут быть определены методы. Методы, а с ними мы также неоднократно встречались, — это функции, которые определены внутри объектных типов. Класс `Automobile` мог бы иметь следующие методы: завестись, ускориться, посигналить:

```
bmw.startEngine()
bmw.accelerate()
bmw.beep()
```

Таким образом, создавая экземпляр, мы можем наполнять его свойства информацией и использовать его методы. И свойства, и методы определяются типом данных.

Способ разработки программ с использованием объектных типов называется объектно-ориентированным программированием. Этот стиль программирования позволяет достичь очень многого при разработке программ. Несмотря на то что вместо термина «объект» используется термин «экземпляр», аббревиатура ООП является устоявшейся в программировании, и в Swift она не трансформируется в ЭОП.

21.2. Пространства имен

Пространства имен (`namespaces`) — это именованные фрагменты программ. Пространства имен имеют одно очень важное свойство — они скрывают свою внутреннюю реализацию и не позволяют получить доступ к объектам внутри пространства имен без доступа к самому пространству имен. Это замечательная черта, благодаря которой вы можете иметь объекты с одним и тем же именем в различных пространствах имен.

Мы уже неоднократно говорили об областях видимости переменных и функций. Пространства имен как раз и реализуют в приложении различные области видимости.

Простейшим примером ограничения области видимости может служить функция. Все переменные, объявленные в ней, вне функции — недоступны. Но при этом функция не является пространством имен, так как не позволяет получить доступ к объектам внутри себя извне.

К пространствам имен относятся *перечисления*, *структуры* и *классы*, о которых мы уже упоминали. Именно их изучением мы и займемся в этой главе. Также к пространствам имен относятся модули, но их рассмотрение не является темой данной книги, мы познакомимся с ними лишь поверхностно. Вообще, модули — это верхний уровень пространств имен. В простейшем варианте ваша программа — это модуль, а значит, это отдельное пространство имен, именем которого является название вашего приложения. Также модулями являются различные фреймворки. С одним из них, кстати, мы уже работали, когда выполняли операцию импорта: `import Foundation`.

Этот фреймворк называется `Cocoa's Foundation Framework` и содержит большое количество функциональных механизмов, позволяющих расширить возможности ваших программ.

Одни пространства имен могут включать другие: так, модуль `UIKit`, ориентированный на разработку iOS-приложений, в своем коде выполняет импорт модуля `Cocoa's Foundation Framework`.

21.3. API Design Guidelines

Одной из главных проблем предыдущих версий Swift была нестандартизированность и неоднозначность написания имен функциональных элементов. Каждый разработчик сам определял, как он хочет называть создаваемые структуры, классы, перечисления, переменные и т. д. С одной стороны, это, конечно, хорошо, но если в своем проекте вы использовали библиотеки сторонних производителей, то синтаксис вашей программы мог превратиться в невнятное месиво. А если еще библиотеки были написаны на `Objective-C`, то разработку вообще хотелось забросить, настолько неудобным могло стать использование Swift.

Но вместе с выходом Swift 3 был разработан документ, определяющий правила именования любых элементов, будь то переменная, константа, функция, класс, перечисление, структура или что-то иное. Он получил название Swift API Design Guidelines.

Swift ADG — это своеобразная дорожная карта, собравшая правила, благодаря которым синтаксис языка стал четким, понятным и приятным. Когда вы достигнете определенного уровня в программировании и приступите к разработке собственных API-библиотек, то изучение приведенного в API Design Guidelines станет отличной базой для создания удобного и функционального продукта.

До разработки Apple Design Guidelines язык Swift был очень изменчив. Далее приведены некоторые наиболее важные правила.

- Комментарии необходимо писать для каждого объявляемого экземпляра в вашем коде. Комментарии должны быть максимально полными.
- Имена всех экземпляров должны быть ясными и краткими. Избегайте дублирования и избыточности. Избегайте пустых слов, не несущих смысловой нагрузки.
- Четкость и ясность именования экземпляров важнее краткости.
- Имена экземпляров должны исключать неоднозначность.
- Именуйте экземпляры в соответствии с их ролью и предназначением в программе.
- Именуйте экземпляры с использованием понятных и максимально простых фраз на английском языке.
- Названия типов данных указывайте в верхнем верблюжьем регистре (`UpperCamelCase`).
- Названия свойств, методов, переменных и констант указывайте в нижнем верблюжьем регистре (`camelCase`).

Используя этот небольшой набор правил, вы уже можете создавать приятные для чтения программы. Старайтесь! Включайте фантазию, и всё получится!

22 Перечисления

Перейдем к изучению механизмов создания объектных типов данных. Начнем с простейшего из них — перечисления.

22.1. Синтаксис перечислений

Перечисление — это объектный тип данных, который предоставляет доступ к различным предопределенным значениям. Рассматривайте его как перечень возможных значений, то есть набор констант, значения которых являются альтернативами друг другу.

Рассмотрим хранилище, которое описывает некоторую денежную единицу (листинг 22.1). Для того чтобы решить поставленную задачу с помощью изученных ранее типов данных, можно использовать тип `String`. В этом случае потребуется вести учет всех возможных значений для описания денежных единиц.

Листинг 22.1

```
var russianCurrency: String = "Rouble"
```

Подобный подход создает проблем больше, чем позволяет решить, поскольку не исключает влияния «человеческого фактора», из-за которого случайное изменение всего лишь одной буквы приведет к тому, что программа не сможет корректно обработать поступившее значение. А что делать, если потребуется добавить обработку нового значения денежной единицы?

Альтернативой этому способу может служить создание коллекции, например массива (листинг 22.2). Массив содержит все возможные значения, которые доступны в программе. При необходимости происходит получение требуемого элемента массива.

Листинг 22.2

```
var currencyUnit: [String] = ["Rouble", "Euro"]
var euroCurrency = currencyUnit[1]
```

В действительности это очень хороший способ ведения списков возможных значений. И его положительные свойства заканчиваются ровно там, где они начинаются у перечислений.

Для того чтобы ввести дополнительную вспомогательную информацию для элементов массива (например, страну, доступные купюры и монеты определенного достоинства), потребуется создать множество дополнительных массивов и словарей. Идеальным было бы иметь отдельный тип данных, который позволил бы описать денежную единицу, но специалисты Apple не предусмотрели его в Swift. Значительно улучшить ситуацию позволяет использование перечислений вместо массивов или фундаментальных типов данных.

Перечисление — это набор значений определенного типа данных, позволяющий взаимодействовать с этими значениями. Так, с помощью перечислений можно создать набор доступных значений и одно из значений присвоить некоторому параметру.

Синтаксис

```
enum ИмяПеречисления {
    case значение1
    case значение2
...
}
```

- `Значение` — значение очередного члена перечисления, может быть произвольного типа данных.

Перечисление объявляется с помощью ключевого слова `enum`, за которым следует имя перечисления. Имя должно определять предназначение создаваемого перечисления и, как название любого типа данных, быть написано в верхнем верблюжьем регистре.

Тело перечисления заключается в фигурные скобки и содержит перечень доступных значений. Эти значения называются членами перечисления. Каждый член определяется с использованием ключевого слова case, после которого без кавычек указывается само значение. Эти значения необходимо начинать с прописной буквы. Их количество в перечислении может быть произвольным.

ПРИМЕЧАНИЕ Объявляя перечисление, вы создаете новый тип данных.

Решим задачу указания типа денежной единицы с использованием перечислений (листинг 22.3). Перечисление подобно массиву. Оно

содержит список значений, одно из которых мы можем присвоить некоторому параметру.

Листинг 22.3

```
enum CurrencyUnit {
    case rouble
    case euro
}
```

Несколько членов перечисления можно писать в одну строку через запятую (листинг 22.4).

Листинг 22.4

```
enum CurrencyUnit {
    case rouble, euro
}
```

Несмотря на то что перечисление `CurrencyUnit` создано и его члены определены, ни одно из значений не присвоено какому-либо параметру. Для того чтобы инициализировать некоторый параметр некоторым членом перечисления, используется специальный синтаксис.

СИНТАКСИС

Для того чтобы инициализировать параметру один из членов перечисления, можно использовать два способа:

- Краткий синтаксис (точка и имя члена перечисления). При этом требуется явно указать тип данных.

  ```
  var имяПараметра: ИмяПеречисления =.значение
  ```

- Полный синтаксис. При этом тип определяется неявно

  ```
  var имяПараметра = ИмяПеречисления.значение
  ```

В дальнейшем для изменения значения переменной, указывающей на член перечисления, можно использовать сокращенный синтаксис.

```
имяПараметра = .значение
```

Имя перечисления выступает в качестве типа данных параметра. Далее доступ к значениям происходит уже без указания его имени.

В листинге 22.5 показаны примеры создания параметров и инициализации им членов перечислений.

Листинг 22.5

```
var roubleCurrency: CurrencyUnit = .rouble
var otherCurrency = CurrencyUnit.euro
```

```
// сменим значение одного параметра
otherCurrency = .rouble
```

В результате создаются две константы типа `CurrencyUnit`, каждая из которых в качестве значения содержит определенный член перечисления `CurrencyUnit`.

ПРИМЕЧАНИЕ Члены перечисления не являются значениями какого-либо типа данных, например `String` или `Int`. Поэтому значения в следующих переменных `currency1` и `currency2` не эквивалентны:

```
var currency1 = CurrencyUnit.rouble
var currency2 = "rouble"
```

22.2. Ассоциированные параметры

У каждого из членов перечисления могут быть ассоциированные с ним значения, то есть его характеристики. Они указываются для каждого члена точно так же, как входящие аргументы функции, то есть в круглых скобках с заданием имен и типов, разделенных двоеточием. Набор ассоциированных параметров может быть произвольным для каждого отдельного члена.

Создадим новое усовершенствованное перечисление `AdvancedCurrencyUnit`, основанное на `CurrencyUnit`, но имеющее ассоциированные параметры, с помощью которых можно указать список стран, в которых данная валюта используется, а также кратких наименований валюты (листинг 22.6).

Листинг 22.6

```
enum AdvancedCurrencyUnit {
    case rouble(countries: [String], shortName: String)
    case euro(countries: [String], shortName: String)
}
```

Параметр `countries` является массивом, так как валюта может использоваться не в одной, а в нескольких странах: например, евро используется на территории Европейского союза.

Теперь для того, чтобы создать переменную или константу типа `AdvancedCurrencyUnit`, необходимо указать значения для всех ассоциированных параметров (листинг 22.7).

Листинг 22.7

```
var euroCurrency: AdvancedCurrencyUnit =  .euro(countries: ["German",
"France"], shortName: "EUR")
```

Теперь в переменной `euroCurrency` хранится член `euro` со значениями двух ассоциированных параметров. При описании ассоциированных параметров в перечислении указывать их имена не обязательно. При необходимости можно указывать лишь их типы.

Ассоциированные параметры могут различаться для каждого члена перечисления. Для демонстрации этого расширим возможности перечисления `AdvancedCurrencyUnit`, добавив в него новый член, описывающий доллар. При этом ассоциированные с ним параметры будут отличаться от параметров уже имеющихся членов. Как известно, доллар является национальной валютой большого количества стран: США, Австралии, Канады и т. д. По этой причине создадим еще одно перечисление, `DollarCountries`, и укажем его в качестве типа данных ассоциированного параметра нового члена перечисления `AdvancedCurrencyUnit` (листинг 22.8).

Листинг 22.8

```
// страны, использующие доллар
enum DollarCountries {
    case usa
    case canada
    case australia
}
// дополненное перечисление
enum AdvancedCurrencyUnit {
    case rouble(countries: [String], shortName: String)
    case euro(countries: [String], shortName: String)
    case dollar(nation: DollarCountries, shortName: String)
}
var dollarCurrency: AdvancedCurrencyUnit = .dollar( nation: .usa,
shortName: "USD" )
```

Для параметра `nation` члена `dollar` перечисления `AdvancedCurrencyUnit` используется тип данных `DollarCountries`. Обратите внимание, что при инициализации значения этого параметра используется сокращенный синтаксис (`.usa`). Это связано с тем, что его тип данных уже задан при определении перечисления `AdvancedCurrencyUnit`.

22.3. Вложенные перечисления

Перечисления могут быть частью других перечислений, то есть могут быть определены в области видимости родительских перечислений.

Так как перечисление `DollarCountries` используется исключительно в перечислении `AdvancedCurrencyUnit` и создано для него, его можно перенести внутрь этого перечисления (листинг 22.9).

Листинг 22.9

```
enum AdvancedCurrencyUnit {
    enum DollarCountries {
        case usa
        case canada
        case australia
    }
    case rouble(countries: [String], shortName: String)
    case euro(countries: [String], shortName: String)
    case dollar(nation: DollarCountries, shortName: String)
}
```

Теперь перечисление `DollarCountries` обладает ограниченной областью видимости и доступно только через родительское перечисление. Можно сказать, что это подтип типа, или вложенный тип. Тем не менее при необходимости вы можете создать параметр, содержащий значение этого перечисления, и вне перечисления `AdvancedCurrencyUnit` (листинг 22.10).

Листинг 22.10

```
var australia: AdvancedCurrencyUnit.DollarCountries = .australia
```

Так как перечисление `DollarCountries` находится в пределах перечисления `AdvancedCurrencyUnit`, обращаться к нему необходимо как к свойству этого типа, то есть через точку.

ПРИМЕЧАНИЕ Мы уже встречались с вложенными типами при изучении словарей (`Dictionary`). Помните `Dictionary<T1,T2>.Keys` и `Dictionary<T1,T2>.Values`?

В очередной раз отмечу, насколько язык Swift удобен в использовании. После перемещения перечисления `DollarCountries` в `AdvancedCurrencyUnit` код продолжает работать, а Xcode дает корректные подсказки в окне автодополнения.

22.4. Оператор switch для перечислений

Для анализа и разбора значений перечислений можно использовать оператор `switch`.

Рассмотрим пример из листинга 22.11, в котором анализируется значение переменной типа `AdvancedCurrencyUnit`.

Листинг 22.11

```
switch dollarCurrency {
    case .rouble:
        print("Рубль")
    case let .euro(countries, shortname):
        print("Евро. Страны: \(countries). Краткое наименование: \
(shortname)")
    case .dollar(let nation, let shortname):
        print("Доллар \(nation). Краткое наименование: \(shortname) ")
}
```

Консоль

```
Доллар usa. Краткое наименование: USD
```

В операторе `switch` описан каждый элемент перечисления `AdvancedCurrencyUnit`, поэтому использовать оператор `default` не обязательно. Доступ к ассоциированным параметрам реализуется связыванием значений: после ключевого слова `case` и указания значения в скобках объявляются константы, которым будут присвоены ассоциированные с членом перечисления значения. Так как для всех ассоциированных параметров создаются константы со связываемым значением, оператор `let` можно ставить сразу после ключевого слова `case` (это продемонстрировано для члена `euro`).

22.5. Связанные значения членов перечисления

Как альтернативу ассоциированным параметрам для членов перечислений им можно задать связанные значения некоторого типа данных (например, `String`, `Character` или `Int`). В результате вы получаете член перечисления и постоянно привязанное к нему значение.

ПРИМЕЧАНИЕ Связанные значения также называют исходными, или сырыми. Но в данном случае термин «связанные» значительно лучше отражает их предназначение.

Указание связанных значений

Для задания связанных значений членов перечисления необходимо указать тип данных самого перечисления, соответствующий значе-

ниям членов, и определить значения для каждого отдельного члена перечисления (листинг 22.12).

Листинг 22.12

```
enum Smile: String {
    case joy = ":)"
    case laugh = ":D"
    case sorrow = ":("
    case surprise = "o_O"
}
```

Перечисление `Smiles` содержит набор смайликов. В качестве связанных значений членов этого перечисления указаны значения типа `String`.

Связанные значения и ассоциированные параметры — не одно и то же. Первые устанавливаются при определении перечисления, причем обязательно для всех его членов и в одинаковом типе данных. Ассоциированные параметры могут быть разными для каждого перечисления и устанавливаются лишь при инициализации члена перечисления в качестве значения.

ВНИМАНИЕ Одновременное определение исходных значений и ассоциированных параметров запрещено.

Если в качестве типа данных перечисления указать `Int`, то исходные значения создаются автоматически путем увеличения значения на единицу для каждого последующего члена (значение первого члена равно 0). Тем не менее, конечно же, можно указать эти значения самостоятельно. Например, в листинге 22.13 представлено перечисление, содержащее список планет Солнечной системы в порядке удаленности от Солнца.

Листинг 22.13

```
enum Planet: Int {
      case mercury = 1, venus, earth, mars, jupiter, saturn, uranus,
                        neptune, pluton = 999
}
```

Для первого члена перечисления в качестве исходного значения указано целое число `1`. Для каждого следующего члена значение увеличивается на единицу, так как не указано иное: для `venus` — это `2`, для `earth` — `3` и т. д.

Для члена `pluton` связанное значение указано конкретно, поэтому оно равно `999`.

Доступ к связанным значениям

При создании экземпляра перечисления можно получить доступ к исходному значению члена этого экземпляра перечисления. Для этого используется свойство `rawValue`. Создадим экземпляр объявленного ранее перечисления `Smile` и получим исходное значение установленного в этом экземпляре члена (листинг 22.14).

Листинг 22.14

```
var iAmHappy = Smile.joy
iAmHappy.rawValue // ":)"
```

В результате использования свойства `rawValue` мы получаем исходное значение члена `joy` типа `String`.

22.6. Инициализатор

При объявлении структуры в ее состав обязательно входит специальный метод-инициализатор. Более того, вам даже не требуется его объявлять, так как эта возможность заложена в Swift изначально. Как мы говорили ранее, при изучении фундаментальных типов, инициализаторы всегда имеют имя `init`. Другими словами, инициализатор — это метод в составе объектного типа (перечисления, или, как вы узнаете далее, структуры или класса), имеющий имя `init`.

Перечисления имеют всего один инициализатор `init(rawValue:)`. Он позволяет передать связанное значение, соответствующее требуемому члену перечисления. Таким образом, у нас есть возможность инициализировать параметру конкретный член перечисления по связанному с ним значению.

В листинге 22.15 показан пример использования инициализатора перечисления.

Листинг 22.15

```
var myPlanet = Planet.init(rawValue: 3) // earth
var anotherPlanet = Planet.init(rawValue: 11) // nil
```

Повторю:

- Инициализатор перечисления `Planet` — это метод `init(rawValue:)`. Ему передается указатель на исходное значение, связанное с искомым членом этого перечисления.

- Данный метод не описан в теле перечисления, — он существует там всегда по умолчанию и закреплен в исходном коде языка Swift.

Инициализатор `init(rawValue:)` возвращает опционал, поэтому если вы укажете несуществующее связанное значение, возвратится `nil`.

ПРИМЕЧАНИЕ Инициализаторы вызываются каждый раз при создании нового экземпляра какого-либо перечисления, структуры или класса. Для некоторых конструкций их можно и нужно создавать самостоятельно, а для некоторых, вроде перечислений, они существуют по умолчанию.

Инициализатор проводит процесс инициализации, то есть выполняет установку всех требуемых значений для параметров с непосредственным созданием экземпляра и помещением его в хранилище.

Инициализатор — это всегда метод с именем `init`.

С инициализаторами мы познакомимся подробнее в следующих главах книги.

22.7. Свойства в перечислениях

Благодаря перечислениям можно смоделировать ситуацию, в которой существует ограниченное количество исходов. У таких ситуаций помимо возможных результатов (членов перечисления) могут существовать и некоторые свойства.

Свойства позволяют хранить в перечислении вспомогательную информацию. Мы уже неоднократно встречались со свойствами в процессе изучения Swift.

Свойство в перечислении — это хранилище, аналогичное переменной или константе, объявленное в пределах перечисления и доступное через его экземпляр. В Swift существует определенное ограничение для свойств в перечислениях: в качестве их значений не могут выступать фиксированные значения-литералы, а лишь замыкания. Такие свойства называются *вычисляемыми*. При каждом обращении к ним происходит вычисление присвоенного замыкания с возвращением получившегося значения.

Для вычисляемого свойства после имени через двоеточие указывается тип возвращаемого значения и далее без оператора присваивания в фигурных скобках — тело замыкающего выражения, генерирующего возвращаемое значение.

Объявим вычисляемое свойство для разработанного ранее перечисления (листинг 22.16). За основу возьмем перечисление `Smile` и создадим вычисляемое перечисление, которое возвращает связанное с текущим членом перечисления значение.

Листинг 22.16

```
enum Smile: String {
    case joy = ":)"
    case laugh = ":D"
    case sorrow = ":("
    case surprise = "o_O"
    // вычисляемое свойство
    var description: String {return self.rawValue}
}
var mySmile: Smile = .sorrow
mySmile.description // ":("
```

Вычисляемое свойство должно быть объявлено как переменная (`var`). В противном случае, если вы используете оператор `let`, то получите сообщение об ошибке.

С помощью оператора `self` вы получаете доступ к текущему члену перечисления. Данный оператор будет очень активно использоваться вами при разработке приложений. С ним мы познакомимся подробнее уже в ближайших разделах.

22.8. Методы в перечислениях

Перечисления могут группировать в себе не только свойства, члены и другие перечисления, но и методы. Ранее мы говорили об инициализаторах `init()`, которые являются встроенными в перечисления методами. *Методы* — это функции внутри перечислений, поэтому их синтаксис и возможности идентичны синтаксису и возможностям изученных ранее функций.

Вернемся к примеру с перечислением `Smile` и создадим метод, который выводит на консоль справочную информацию о предназначении перечисления (листинг 22.17).

Листинг 22.17

```
enum Smile: String {
    case joy = ":)"
    case laugh = ":D"
    case sorrow = ":("
    case surprise = "o_O"
```

```
    var description: String {return self.rawValue}
    func about(){
        print("Перечисление содержит список смайликов ")
    }
}
var otherSmile = Smile.joy
otherSmile.about()
```

Консоль

```
Перечисление содержит список смайликов
```

В этом перечислении объявлен метод `about()`. После создания экземпляра метода и помещения его в переменную данный метод может быть вызван.

22.9. Оператор self

Для организации доступа к текущему значению перечисления внутри этого перечисления используется оператор `self` (в одном из предыдущих листингов мы уже использовали его). Данный оператор возвращает указатель на конкретный член перечисления, инициализированный параметру.

Рассмотрим пример.

Требуется написать два метода, один будет возвращать сам член перечисления, а второй — его связанное значение. Используем для этого уже знакомое перечисление `Smile` (листинг 22.18).

Листинг 22.18

```
enum Smile: String {
    case joy = ":)"
    case laugh = ":D"
    case sorrow = ":("
    case surprise = "o_O"
    var description: String {return self.rawValue}
    func about(){
        print("Перечисление содержит список  смайликов")
    }
    func descriptionValue() -> Smile{
        return self
    }
    func descriptionRawValue() -> String{
        return self.rawValue
    }
}
var otherSmile = Smile.joy
```

```
otherSmile.descriptionValue() // joy
otherSmile.descriptionRawValue() // ":)"
```

При вызове метода `descriptionValue()` происходит возврат `self`, то есть самого экземпляра. Именно поэтому тип возвращаемого значения данного метода — `Smile`, он соответствует типу экземпляра перечисления.

Метод `descriptionRawValue()` возвращает связанное значение члена данного экземпляра также с использованием оператора `self`.

При необходимости вы даже можете выполнить анализ перечисления внутри самого перечисления с помощью конструкции `switch self {}`, где значениями являются члены перечисления.

Оператор `self` можно использовать не только для перечислений, но и для структур и классов. Об этом мы поговорим позже.

22.10. Рекурсивные перечисления

Перечисления отлично справляются с моделированием ситуации, когда есть всего несколько вариантов ее развития. Но вы можете использовать их не только для того, чтобы хранить некоторые связанные и ассоциированные значения. Вы можете пойти дальше и наделить перечисление функционалом анализа собственного значения и вычисления на его основе выражений.

Возьмем, к примеру, простейшие арифметические операции: сложение, вычитание, умножение и деление. Все они заранее известны, поэтому могут быть помещены в перечисление в качестве его членов (листинг 22.19). Для простоты оставим только две операции: сложение и вычитание.

Листинг 22.19

```
enum ArithmeticExpression{
    // операция сложения
    case addition(Int, Int)
    // операция вычитания
    case substraction(Int, Int)
}
var expr = ArithmeticExpression.addition(10, 14)
```

Каждый из членов перечисления соответствует операции с двумя операндами. В связи с этим они имеют по два ассоциированных параметра.

В результате выполнения кода в переменной `expr` будет храниться член перечисления ArithmeticExpression, определяющий арифметическую операцию сложения.

Объявленное перечисление не несет какой-либо функциональной нагрузки в вашем приложении. Но вы можете создать в его пределах метод, который определяет наименование члена и возвращает результат данной операции (листинг 22.20).

Листинг 22.20

```
enum ArithmeticExpression{
    case addition(Int, Int)
    case substraction(Int, Int)
    func evaluate() -> Int {
        switch self{
        case .addition(let left, let right):
            return left+right
        case .substraction(let left, let right):
            return left-right
        }
    }
}
var expr = ArithmeticExpression.addition(10, 14)
expr.evaluate() // 24
```

При вызове метода `evaluate()` происходит поиск определенного в данном экземпляре члена перечисления. Для этого используются операторы `switch` и `self`. Далее, после того как член определен, путем связывания значений возвращается результат требуемой арифметической операции.

Данный способ работает просто замечательно, но имеет серьезное ограничение: он способен моделировать только одноуровневые арифметические выражения: `1 + 5`, `6 + 19` и т. д. В ситуации, когда выражение имеет вложенные выражения: `1 + (5 – 7)`, `6 – 5 + 4` и т. д., нам придется вычислять каждое отдельное действие с использованием собственного экземпляра типа `ArithmeticExpression`.

Для решения этой проблемы необходимо доработать перечисление `ArithmeticExpression` таким образом, чтобы оно давало возможность складывать не только значения типа `Int`, но и значения типа `ArithmeticExpression`.

Получается, что перечисление, описывающее выражение, должно давать возможность выполнять операции само с собой. Данный механизм реализуется в *рекурсивном перечислении*. Для того чтобы разрешить членам перечисления обращаться к этому перечислению, используется ключевое слово `indirect`, которое ставится:

- либо перед оператором `enum` — в этом случае каждый член перечисления может обратиться к данному перечислению;

- либо перед оператором `case` того члена, в котором необходимо обратиться к перечислению.

Если в качестве ассоциированных параметров перечисления указывать значения типа самого перечисления `ArithmeticExpression`, то возникает вопрос: а где же хранить числа, над которыми совершаются операции? Такие числа также необходимо хранить в самом перечислении, в его отдельном члене.

Рассмотрим пример из листинга 22.21. В данном примере вычисляется значение выражения `20 + 10 – 34`.

Листинг 22.21

```
enum ArithmeticExpression {
    // указатель на конкретное значение
    case number( Int )
    // указатель на операцию сложения
    indirect case addition( ArithmeticExpression, ArithmeticExpression )
    // указатель на операцию вычитания
    indirect case subtraction( ArithmeticExpression,
ArithmeticExpression )
    // метод, проводящий операцию
    func evaluate( _ expression: ArithmeticExpression? = nil ) -> Int{
        // определение типа операнда (значение или операция)
        switch expression ?? self{
        case let .number( value ):
            return value
        case let .addition( valueLeft, valueRight ):
            return self.evaluate( valueLeft )+self.evaluate
                                ( valueRight )
        case .subtraction( let valueLeft, let valueRight ):
            return self.evaluate( valueLeft )-self.evaluate
                                ( valueRight )
        }
    }
}
var hardExpr = ArithmeticExpression.addition( .number(20),
.subtraction( .number(10), .number(34) ) )
hardExpr.evaluate() // -4
```

У перечисления появился новый член `number`, который определяет целое число — операнд для проведения очередной операции. Для членов арифметических операций использовано ключевое слово `indirect`, позволяющее передать значение типа `ArithmeticExpression` в качестве ассоциированного параметра.

Метод `evaluate(expression:)` принимает на входе опциональное значение типа `ArithmeticExpression?`. Опционал в данном случае позволяет

вызвать метод, не передавая ему экземпляр, из которого этот метод был вызван. В противном случае последняя строка листинга выглядела бы следующим образом:

```
expr.evaluate(expression: expr)
```

Согласитесь, что существующий вариант значительно удобнее.

Оператор `switch`, используя принудительное извлечение, определяет, какой член перечисления передан, и возвращает соответствующее значение.

В результате данное перечисление позволяет смоделировать любую операцию, в которой присутствуют операторы сложения и вычитания.

Перечисления в `Swift` мощнее, чем аналогичные механизмы в других языках программирования. Вы можете создавать свойства и методы, применять к ним расширения и протоколы, а также делать многое другое. Обо всем этом мы вскоре поговорим.

23 Структуры

Перечисления — это входной билет в объектно-ориентированное программирование. Теперь пришло время познакомиться с еще более функциональными и интересными конструкциями — структурами.

Вы уже давно используете структуры при написании кода. Все фундаментальные типы данных — это структуры. Они в некоторой степени похожи на перечисления и во многом сходны с классами (с ними мы познакомимся в скором времени).

23.1. Синтаксис объявления структур

Знакомство со структурами начнем с рассмотрения примера. Перед вами стоит задача описать в вашей программе сущность «игрок в шахматы», включая характеристики: имя и количество побед. Для решения этой задачи можно использовать кортежи и хранить в переменной имя и количество побед игрока (листинг 23.1).

Листинг 23.1

```
var playerInChess = (name: "Василий", wins: 10)
```

Таким образом мы, конечно же, решим поставленную задачу, но если количество характеристик будет увеличиваться, то кортеж станет чересчур сложным.

Соответственно, нам требуется механизм, позволяющий гибко описывать даже самые сложные сущности, учитывая все возможные параметры. Структуры как раз и являются таким механизмом. Они позволяют создать «скелет» сущности. Например, структура `Int` описывает сущность «целое число».

СИНТАКСИС

```
struct ИмяСтруктуры {
// свойства и методы структуры
}
```

- Структуры объявляются с помощью ключевого слова `struct`, за которым следует имя создаваемой конструкции. Требования к имени предъявляются точно такие же, как и к имени перечислений: оно должно писаться в верхнем верблюжьем регистре.
- Тело структуры заключается в фигурные скобки и может содержать свойства и методы, подобные тем, что используются в перечислениях, но более функциональные.

ПРИМЕЧАНИЕ Объявляя структуру, вы определяете новый тип данных.

Объявим структуру, которая будет описывать сущность «игрок в шахматы» (листинг 23.2).

Листинг 23.2

```
struct PlayerInChess {}
var oleg = PlayerInChess()
type(of:oleg) //PlayerInChess.Type
```

Так как структура `PlayerInChess` является типом данных, то можно объявить новое хранилище (переменную) данного типа.

ПРИМЕЧАНИЕ Обратите внимание на то, что напротив функции `type(of:)` в области результатов в указании на тип присутствует префикс вида `__lldb_expr_141` (число может отличаться). Это особенность Xcode Playground, данный префикс определяет модуль, в котором структура определена, то есть к какому пространству имен она относится. Вы можете не волноваться по этому поводу и не обращать на префикс никакого внимания.

23.2. Свойства в структурах

Объявление свойств

Созданная структура `PlayerInChess` пуста — она не описывает какие-либо характеристики игрока. Для их реализации можно использовать свойства, с которыми мы уже встречались при изучении перечислений.

СИНТАКСИС

```
struct ИмяСтруктуры{
    var свойство1: ТипДанных
    let свойство2: ТипДанных
    // остальные свойства и методы
}
```

- `свойство: Any` — свойство структуры может быть любого типа данных.

Свойство может быть представлено как в виде переменной, так и в виде константы. Количество свойств в структуре не ограничено.

В структуру `PlayerInChess` добавим два свойства, описывающие имя и количество побед (`name` и `wins`) (листинг 23.3).

Листинг 23.3

```
struct PlayerInChess {
    var name: String
    var wins: UInt
}
```

Встроенный инициализатор

Структуры, так же как и перечисления, имеют встроенный инициализатор (метод с именем `init`), который не требуется объявлять. Данный инициализатор принимает на входе значения всех свойств структуры, производит их инициализацию и возвращает экземпляр данной структуры (листинг 23.4).

Листинг 23.4

```
var harry = PlayerInChess.init(name: "Гарри", wins: 32)
```

В результате будет создан новый экземпляр структуры `PlayerInChess`, содержащий свойства с определенными в инициализаторе значениями.

ВНИМАНИЕ При создании экземпляра структуры всем свойствам обязательно должны быть инициализированы значения. Пропустить любое из них недопустимо! Если значение какого-либо из свойств не будет указано, Xcode сообщит об ошибке.

Инициализатор автоматически вызывается всегда при создании нового экземпляра, поэтому можно опустить его имя и передавать значения свойств сразу после имени структуры (листинг 23.4а).

Листинг 23.4а

```
var harry = PlayerInChess(name: "Гарри", wins: 32)
```

Как вы можете видеть, при объявлении параметра не используется имя инициализатора.

Значения свойств по умолчанию

Для свойств можно задавать значение по умолчанию. При этом Swift автоматически создает новый инициализатор, который позволяет создавать экземпляр без указания значений свойств. Вы сможете увидеть все доступные инициализаторы структуры в окне автодополнения во время создания экземпляра.

На рис. 23.1 изображены два разных инициализатора, доступных при создании экземпляра: один не требует указывать значения свойств, поскольку использует их значения по умолчанию, другой, наоборот, требует указать эти значения. Инициализатор, который не требует указывать какие-либо значения, называется *пустым инициализатором*.

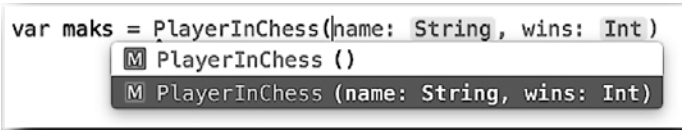

Рис. 23.1. Два инициализатора в окне автодополнения

Значения по умолчанию указываются вместе с объявлением свойств точно так же, как вы указываете значение любой переменной или константы. При этом если вы решили дать значение по умолчанию хотя бы одному свойству, то должны указывать его и для всех остальных свойств.

Объявим значения по умолчанию для структуры `PlayerInChess` (листинг 23.5).

Листинг 23.5

```
struct PlayerInChess {
    var name: String = "Игрок"
    var wins: UInt = 0
}
var john = PlayerInChess(name: "Джон", wins: 32)
var player = PlayerInChess()
```

В обоих случаях создается экземпляр структуры `PlayerInChess`. Если для создания экземпляра выбирается пустой инициализатор, параметрам `name` и `wins` присваиваются их значения по умолчанию.

23.3. Структура как пространство имен

Структура образует отдельное пространство имен, поэтому для доступа к элементам этого пространства имен необходимо в первую очередь получить доступ к самому пространству.

В предыдущем примере была создана структура `PlayerInChess` с двумя свойствами. Каждое из свойств имеет некоторое значение, но от них не будет никакого толку, если не описать механизмы доступа к данным свойствам.

Доступ к элементам структур происходит с помощью экземпляров данной структуры (листинг 23.6).

Листинг 23.6

```
john.name // "Джон"
player.name // "Игрок"
```

Данный способ доступа обеспечивает не только чтение, но и изменение значения свойства экземпляра структуры (листинг 23.7).

Листинг 23.7

```
john.wins // 32
john.wins += 1
john.wins // 33
```

ПРИМЕЧАНИЕ Если свойство структуры или ее экземпляр указаны как константа (`let`), при попытке изменения значения свойства Xcode сообщит об ошибке.

23.4. Собственные инициализаторы

Как говорилось ранее, инициализатор — это специальный метод, который носит имя `init`. Если вас не устраивают инициализаторы, которые создаются для структур автоматически, вы имеете возможность определить собственные.

ПРИМЕЧАНИЕ Автоматически созданные встроенные инициализаторы удаляются при объявлении первого собственного инициализатора.

Вам необходимо постоянно придерживаться правила: «все свойства структуры должны иметь значения при создании экземпляра данной структуры». Вы можете создать инициализатор, который принимает в качестве входного параметра значения не для всех свойств, тогда остальным свойствам должны быть назначены значения либо внутри данного инициализатора, либо через значения по умолчанию. Несмотря на то что инициализатор — метод, он объявляется без использования ключевого слова `func`. При этом одна структура может содержать произвольное количество инициализаторов, каждый из которых должен иметь уникальный набор входных параметров. Доступ к свойствам экземпляра внутри инициализатора осуществляется с помощью оператора `self`.

Создадим инициализатор для структуры `PlayerInChess`, который принимает значение только для свойства `name` (листинг 23.8).

Листинг 23.8

```
struct PlayerInChess {
    var name: String = "Игрок"
    var wins: UInt = 0
    //инициализатор
    init(name: String){
        self.name = name
    }
}
var helga = PlayerInChess(name: "Ольга")
helga.wins // 0
// следующий код должен был бы вызвать ошибку
// var newPlayer = PlayerInChess()
```

Инициализатор принимает значение только для свойства `name`, при этом свойству `wins` будет проинициализировано значение по умолчанию. При создании экземпляра вам будет доступен исключительно разработанный вами инициализатор.

Помните, что создавать собственные инициализаторы для структур не обязательно, так как они уже имеют встроенные инициализаторы.

ВНИМАНИЕ Если экземпляр структуры хранится в константе, модификация его свойств невозможна. Если же он хранится в переменной, то возможна модификация тех свойств, которые объявлены с помощью оператора `var`.

ВНИМАНИЕ Структуры — это типы-значения. При передаче экземпляра структуры от одного параметра в другой происходит его копирование. В следующем примере создаются два независимых экземпляра одной и той же структуры:

```
var olegMuhin = PlayerInChess(name: "Олег")
var olegLapin = olegMuhin
```

23.5. Методы в структурах

Объявление методов

Помимо свойств, структуры, как и перечисления, могут содержать методы. Синтаксис объявления методов в структурах аналогичен объявлению методов в перечислениях. Они, как и обычные функции, могут принимать входные параметры. Для доступа к собственным свойствам структуры используется оператор `self`.

Реализуем метод `description()`, который выводит справочную информацию об игроке в шахматы на консоль (листинг 23.9).

Листинг 23.9

```
struct PlayerInChess {
    var name: String = "Игрок"
    var wins: UInt = 0
    init(name: String){
        self.name = name
    }
    func description(){
        print("Игрок \(self.name) имеет \(self.wins) побед")
    }
}
var andrey = PlayerInChess(name: "Андрей")
andrey.description()
```

Консоль

```
Игрок Андрей имеет 0 побед
```

Изменяющие методы

По умолчанию методы структур, кроме инициализаторов, не могут изменять значения свойств, объявленные в тех же самых структурах. Для того чтобы обойти данное ограничение, перед именем объявляемого метода необходимо указать модификатор `mutating`.

Создадим метод `wins`, который будет изменять значение свойства `wins` (листинг 23.10).

Листинг 23.10

```
struct PlayerInChess {
    var name: String = "Игрок"
    var wins: UInt = 0
    init(name: String){
        self.name = name
    }
    func description(){
        print("Игрок \(self.name) имеет \(self.wins) побед")
    }
    mutating func win( count: UInt = 1 ){
        self.wins += count
    }
}
var harold = PlayerInChess(name: "Гарольд")
harold.wins // 0
harold.win()
harold.wins // 1
harold.win( count: 3 )
harold.wins // 4
```

ПРИМЕЧАНИЕ Структура может изменять значения свойств только в том случае, если экземпляр структуры хранится в переменной.

24 Классы

Классы являются наиболее функциональными конструкциями в разработке приложений. Данная глава призвана познакомить вас с их замечательными возможностями. Если вы ранее разрабатывали приложения на других языках программирования, то, возможно, уже осведомлены о классах. В этом случае имеющийся опыт пригодится вам при их изучении в Swift.

Классы очень похожи на структуры, но их отличают несколько ключевых моментов.

Тип. Класс — это тип-ссылка. Экземпляры класса передаются по ссылке, а не копированием.

Изменяемость. Экземпляр класса может изменять значения своих свойств, объявленных как переменная (`var`), даже если данный экземпляр хранится в константе (`let`). При этом использовать ключевое слово `mutating` для методов не требуется.

Наследование. Два класса могут быть в отношении «суперкласс–субкласс» друг к другу. Но обратите внимание: субкласс наследует и включает в себя все характеристики (свойства и методы) суперкласса и может быть дополнительно расширен. Об ограничениях, связанных с наследованием, рассказано в главе 27.

Инициализаторы. Класс имеет только пустой встроенный инициализатор `init(){}`, который не требует передачи значения входных параметров для их установки в свойства.

Деинициализаторы. Swift позволяет создать деинициализатор — специальный метод, который автоматически вызывается при удалении экземпляра класса.

Приведение типов. В процессе выполнения программы вы можете проверить экземпляр класса на соответствие определенному типу данных.

Каждая из особенностей детально разбирается в книге.

Со временем вы научитесь определять наиболее подходящие конструкции для реализации используемых в программах сущностей: последовательность, коллекция, структура, класс, кортеж или что-то другое.

24.1. Синтаксис классов

Объявление классов очень похоже на объявление структур.

СИНТАКСИС

```
class ИмяКласса {
        // свойства и методы класса
}
```

- Классы объявляются с помощью ключевого слова `class`, за которым следует имя создаваемого класса. Имя класса должно быть написано в верхнем верблюжьем регистре.
- Тело класса заключается в фигурные скобки и может содержать методы и свойства, а также другие элементы, с которыми мы еще не знакомы.

ПРИМЕЧАНИЕ При объявлении нового класса, как и при объявлении перечисления или структуры, создается новый тип данных, который может быть использован.

23.2. Свойства классов

Перейдем к практической стороне изучения классов. Класс, как и структура, может предоставлять механизмы для описания некоторой сущности. Эта сущность обычно обладает рядом характеристик, выраженных в классе в виде *свойств класса*. Для свойств могут быть указаны значения по умолчанию.

Как отмечалось ранее, класс имеет один встроенный инициализатор, который является пустым. Если у структуры инициализатор генерируется автоматически вместе с изменением состава ее свойств, то у класса для установки значений свойств требуется разрабатывать инициализаторы самостоятельно.

При создании экземпляра класса каждому свойству должно быть проинициализировано значение: либо через значения по умолчанию, либо в теле инициализатора.

При выполнении заданий в предыдущих главах мы описывали сущность «шахматная фигура» вначале с помощью перечисления `Chessmen`, а потом с помощью структуры `Chessmen`. Использование класса для мо-

делирования шахматной фигуры предпочтительнее в первую очередь в связи с тем, что каждая отдельная фигура — это уникальный объект со своими характеристиками. Его модификация в программе с использованием ссылок (а класс — это тип-ссылка) значительно упростит работу.

Рассмотрим пример использования классов.

Необходимо смоделировать сущность «шахматная фигура». При этом она должна обладать следующим набором свойств:

- тип фигуры;
- цвет фигуры;
- координаты на игровом поле.

В еще одном дополнительном свойстве мы будем хранить символ, соответствующий шахматной фигуре (в Unicode входят необходимые символы).

Координаты будут служить не только для того, чтобы определить местоположение фигуры на шахматной доске, но и для определения факта ее присутствия. Если фигура убита или еще не выставлена, то значение координат будет `nil`.

Так как у класса будут определены свойства, необходимо разработать инициализатор, который будет устанавливать их значения.

В листинге 24.1 приведен код класса, описывающий сущность «шахматная фигура».

Листинг 24.1

```
class Chessman {
    // тип фигуры
    let type: String
    // цвет фигуры
    let color: String
    //координаты фигуры
    var coordinates: (String, Int)? = nil
    // символ, соответствующий фигуре
    let figureSymbol: Character
    // инициализатор
    init(type: String, color: String, figure: Character){
        self.type = type
        self.color = color
        self.figureSymbol = figure
    }
}
// создаем экземпляр фигуры
var kingWhite = Chessman(type: "king", color: "white", figure:
                "\u{2654}")
```

ПРИМЕЧАНИЕ Коды Unicode-символов, соответствующих шахматным фигурам, вы можете самостоятельно посмотреть в интернете.

Каждая из характеристик фигуры выражена в отдельном свойстве класса. Тип данных свойства `coordinate` является опциональным кортежем. Это связано с тем, что фигура может быть убрана с игрового поля (тогда свойство будет `nil`). Координаты фигуры на шахматном поле задаются с помощью строки и числа.

В разработанном инициализаторе указаны входные аргументы, значения которых используются в качестве значений свойств экземпляра.

В результате выполнения кода в переменной `kingWhite` находится экземпляр класса `Chessman`, описывающий фигуру «Белый король». Фигура еще не имеет координат, а значит, не выставлена на игровое поле (ее координаты равны `nil`).

Свойства `type` и `color` могут принять значения из четко определенного перечня, поэтому имеет смысл реализовать два перечисления: одно должно содержать типы шахматных фигур, второе цвета (листинг 24.2).

Листинг 24.2

```
// тип шахматной фигуры
enum ChessmanType {
    case king, castle, bishop, pawn, knight, queen
}
// цвета фигур
enum ChessmanColor {
    case black, white
}
```

Созданные перечисления должны найти место в качестве типов соответствующих свойств класса `Chessman`. Не забывайте, что и входные аргументы инициализатора должны измениться соответствующим образом (листинг 24.2а).

Листинг 24.2а

```
class Chessman {
    let type: ChessmanType
    let color: ChessmanColor
    var coordinates: (String, Int)? = nil
    let figureSymbol: Character
    init(type: ChessmanType, color: ChessmanColor, figure:Character){
        self.type = type
        self.color = color
        self.figureSymbol = figure
```

```
    }
}
var kingWhite = Chessman(type: .king, color: .white, figure:
                "\u{2654}")
```

Теперь при создании модели шахматной фигуры необходимо передавать значения типов `ChessmanType` и `ChessmanColor` вместо `String`.

Созданные дополнительные связи обеспечивают корректность ввода данных при создании экземпляра класса.

24.3. Методы классов

Сущность «Шахматная фигура» уже является вполне рабочей. На ее основе можно описать любую фигуру. Тем не менее описанная фигура все еще является «мертвой» в том смысле, что она не может быть использована непосредственно для игры. Это связано с тем, что еще не разработаны механизмы, позволяющие установить фигуру на игровое поле и динамически ее перемещать.

Классы, как и структуры с перечислениями, могут содержать произвольные методы, обеспечивающие функциональную нагрузку класса. Не забывайте, что в классах нет необходимости использовать ключевое слово `mutating` для методов, меняющих значения свойств.

Немного оживим созданную модель шахматной фигуры, создав несколько методов (листинг 24.3):

- установка координат фигуры (при выставлении на поле или при движении);
- снятие фигуры с игрового поля (окончание партии или гибель фигуры).

Листинг 24.3

```
class Chessman {
    let type: ChessmanType
    let color: ChessmanColor
    var coordinates: (String, Int)? = nil
    let figureSymbol: Character
    init(type: ChessmanType, color: ChessmanColor, figure:Character){
        self.type = type
        self.color = color
        self.figureSymbol = figure
    }
    // метод установки координат
    func setCoordinates(char c:String, num n: Int){
```

```
        self.coordinates = (c, n)
    }
    // метод, убивающий фигуру
    func kill(){
        self.coordinates = nil
    }
}
var kingWhite = Chessman(type: .king, color: .white, figure:
                "\u{2654}")
kingWhite.setCoordinates(char: "E", num: 1)
```

В результате фигура «Белый король» выставляется в позицию с координатами `E1`.

На самом деле для действительного размещения фигуры на игровом поле необходимо смоделировать саму шахматную доску. И этим вопросом мы займемся уже в скором времени.

24.4. Инициализаторы классов

Класс может содержать произвольное количество разработанных инициализаторов, различающихся лишь набором входных аргументов. Это никоим образом не влияет на работу самого класса, а лишь дает нам более широкие возможности при создании его экземпляров.

Рассмотрим процесс создания дополнительного инициализатора. Существующий класс `Chessman` не позволяет одним выражением создать фигуру и выставить ее на поле. Сейчас для этого используются два независимых выражения. Давайте разработаем второй инициализатор, который будет дополнительно принимать координаты фигуры (листинг 24.4).

Листинг 24.4

```
class Chessman {
    let type: ChessmanType
    let color: ChessmanColor
    var coordinates: (String, Int)? = nil
    let figureSymbol: Character
    init(type: ChessmanType, color: ChessmanColor, figure:Character){
        self.type = type
        self.color = color
        self.figureSymbol = figure
    }
    init(type: ChessmanType, color: ChessmanColor, figure: Character,
                coordinates: (String, Int)){
        self.type = type
```

```
        self.color = color
        self.figureSymbol = figure
        self.setCoordinates(char: coordinates.0, num: coordinates.1)
    }
    func setCoordinates(char c:String, num n: Int){
        self.coordinates = (c, n)
    }
    func kill(){
        self.coordinates = nil
    }
}
var queenBlack = Chessman(type: .queen, color: .black, figure:
                 "\u{2655}", coordinates: ("A", 6))
```

Так как код установки координат уже написан в методе `setCoordinates(char:num:)`, то, во избежание дублирования, в инициализаторе этот метод просто вызывается.

При объявлении нового экземпляра в окне автодополнения будут предлагаться на выбор два инициализатора, объявленные в классе `Chessman`.

24.5. Вложенные в класс типы

Очень часто перечисления, структуры и классы создаются для того, чтобы расширить функциональность и удобство использования определенного типа данных. Такой подход мы встречали, когда разрабатывали перечисления `ChessmanColor` и `ChessmanType`, использующиеся в классе `Chessman`. В данном случае перечисления нужны исключительно в контексте класса, описывающего шахматную фигуру, и нигде больше.

В такой ситуации вы можете вложить перечисления в класс, то есть описать их не глобально, а в пределах тела класса (листинг 24.5).

Листинг 24.5

```
class Chessman {
    enum ChessmanType {
        case king, castle, bishop, pawn, knight, queen
    }
    enum ChessmanColor {
        case black, white
    }
    let type: ChessmanType
    let color: ChessmanColor
    var coordinates: (String, Int)? = nil
```

```
    let figureSymbol: Character
    init(type: ChessmanType, color: ChessmanColor, figure:
        Character){
        self.type = type
        self.color = color
        self.figureSymbol = figure
    }
    init(type: ChessmanType, color: ChessmanColor, figure:
        Character, coordinates: (String, Int)){
        self.type = type
        self.color = color
        self.figureSymbol = figure
        self.setCoordinates(char: coordinates.0, num:coordinates.1)
    }
    func setCoordinates(char c:String, num n: Int){
        self.coordinates = (c, n)
    }
    func kill(){
        self.coordinates = nil
    }
}
```

Структуры `ChessmanColor` и `ChessmanType` теперь являются вложенными в класс `Chessman` и существуют только в пределах области видимости данного класса. Мы уже неоднократно видели с вами подобный подход.

Ссылки на вложенные типы

В некоторых ситуациях может возникнуть необходимость использовать вложенные типы вне определяющего их контекста. Для этого необходимо указать имя родительского типа, после которого должно следовать имя требуемого типа данных (листинг 24.6). В этом примере мы получаем доступ к одному из членов перечисления `ChessmanType`, объявленного в контексте класса `Chessman`.

Листинг 24.6

```
var linkToEnumType = Chessman.ChessmanType.bishop
```

25 Свойства

Вы уже умеете создавать и использовать свойства при разработке типов данных. В этой главе вы получите более глубокие знания по этому вопросу, так как свойства не ограничиваются рассмотренными ранее возможностями.

25.1. Типы свойств

Свойства — это параметры, объявленные в пределах объектного типа данных. Они позволяют хранить и вычислять значения, а также получать доступ к этим значениям.

По типу хранимого значения можно выделить два основных вида свойств:

- хранимые свойства могут использоваться в структурах и классах;
- вычисляемые свойства могут использоваться в перечислениях, структурах и классах.

Хранимые свойства

Хранимое свойство — это константа или переменная, объявленная в объектном типе и хранящая определенное значение. Хранимое свойство может:

- получить значение по умолчанию в случае, если при создании экземпляра ему не передается никакого значения;
- получить значение в инициализаторе (метод с именем `init`);
- изменить значение в процессе использования экземпляра.

Мы уже создавали хранимые свойства, к примеру, при реализации класса `Chessman` в предыдущей главе.

Ленивые хранимые свойства

Хранимые свойства могут быть «ленивыми». Значение, которое должно храниться в ленивом свойстве, не создается до момента первого обращения к нему.

СИНТАКСИС

```
lazy var имяСвойства1: ТипДанных
lazy let имяСвойства2: ТипДанных
```

Перед операторами `var` и `let` добавляется ключевое слово `lazy`, указывающее на «ленивость» свойства.

Рассмотрим пример. Создадим класс, описывающий человека. Он будет содержать свойства, содержащие информацию об имени и фамилии. Также будет определен метод, возвращающий полное имя (имя и фамилию вместе) и ленивое свойство, содержащее значение данного метода (листинг 25.1).

Листинг 25.1

```
class AboutMan{
    var firstName = "Имя"
    var secondName = "Фамилия"
    lazy var wholeName: String = self.generateWholeName()
    init(name: String, secondName: String){
        ( self.firstName, self.secondName ) = ( name, secondName )
    }
    func generateWholeName() -> String{
        return self.firstName + " " + self.secondName
    }
}
var Me = AboutMan(name:"Егор", secondName:"Петров")
Me.wholeName
```

Экземпляр класса `AboutMan` очень упрощенно описывает сущность «человек». В свойстве `wholeName` должно храниться его полное имя, но при создании экземпляра его значение не задается. При этом оно не равно `nil`, оно просто не сгенерировано и не записано. Это связано с тем, что свойство является ленивым. Как только происходит обращение к данному свойству, его значение формируется.

Ленивые свойства позволяют экономить оперативную память и не расходовать ее до тех пор, пока значение какого-либо свойства не потребуется.

ПРИМЕЧАНИЕ Стоит отметить, что в качестве значений для хранимых свойств нельзя указывать элементы (свойства и методы) того же объектного типа. Ленивые свойства не имеют этого ограничения, так как их значения формируются уже после создания экземпляров.

Ленивые свойства являются `lazy-by-need`, то есть вычисляются однажды и больше не меняют свое значение. Это продемонстрировано в листинге 25.2.

Листинг 25.2

```
Me.wholeName // "Егор Петров"
Me.secondName = "Иванов"
Me.wholeName // "Егор Петров"
```

Методы типов данных в некоторой степени тоже являются ленивыми: они вычисляют значение при обращении к ним и делают это каждый раз. Если внимательно посмотреть на структуру класса AboutMan, то для получения полного имени можно было обращаться к методу `generateWholeName()` вместо ленивого свойства `wholeName`. Но также можно было пойти и другим путем: создать ленивое хранимое свойство функционального типа, содержащее в себе замыкание (листинг 25.3).

Листинг 25.3

```
class AboutMan{
    var firstName = "Имя"
    var secondName = "Фамилия"
    lazy var wholeName: ()->String = { "\(self.firstName) \(self.
secondName)" }
    init(name: String, secondName: String){
        ( self.firstName, self.secondName ) = ( name, secondName )
    }
}

var otherMan = AboutMan(name: "Алексей", secondName:"Олейник")
otherMan.wholeName() // "Алексей Олейник"
otherMan.secondName = "Дуров"
otherMan.wholeName() // "Алексей Дуров"
```

Обратите внимание, что так как свойство хранит в себе замыкание, доступ к нему необходимо организовывать с использованием скобок.

Почему необходимо использовать `lazy` для свойства `wholeName`? Как было сказано выше, только ленивые свойства могут обращаться к элементам (свойствам и методам) того же объектного типа. Если убрать `lazy`, то Xcode сообщит об ошибке:

```
error: use of unresolved identifier 'self'
```

При этом свойство будет возвращать актуальное значение каждый раз при обращении к нему.

Таким образом, мы создали ленивое хранимое свойство, которое высчитывает и возвращает значение каждый раз при обращении к нему. Напомню, что такой тип ленивых параметров называется `lazy-by-name`.

Вычисляемые свойства

Также существует иной способ создать параметр, значение которого будет вычисляться при каждом доступе к нему. Для этого можно использовать уже знакомые по перечислениям нам вычисляемые свойства. По сути, это те же ленивые хранимые свойства, имеющие функциональный тип, но определяемые в упрощенном синтаксисе.

Вычисляемые свойства фактически не хранят значение, а вычисляют его с помощью замыкающего выражения.

СИНТАКСИС

```
var имяСвойства:  ТипДанных { тело_замыкающего_выражения }
```

Вычисляемые свойства могут храниться исключительно в переменных (`var`). После указания имени объявляемого свойства и типа возвращаемого замыкающим выражением значения без оператора присваивания указывается замыкание, в результате которого должно быть сгенерировано возвращаемое свойством значение.

Для того чтобы свойство возвращало некоторое значение, в теле замыкания должен присутствовать оператор `return`.

Сделаем свойство `wholeName` класса `AboutName` вычисляемым (листинг 25.4).

Листинг 25.4

```
class AboutMan{
    var firstName = "Имя"
    var secondName = "Фамилия"
    var wholeName: String { return "\(self.firstName)
                                    \(self.secondName)" }
    init(name: String, secondName: String){
        ( self.firstName, self.secondName ) = ( name, secondName )
    }
}
var otherMan = AboutMan(name: "Алексей", secondName:"Олейник")
otherMan.wholeName // "Алексей Олейник"
otherMan.secondName = "Дуров"
otherMan.wholeName // "Алексей Дуров"
```

Теперь доступ к значению свойства `wholeName` производится так же, как и к обыкновенным свойствам (без использования скобок), но при этом всегда возвращается актуальное значение.

25.2. Контроль значений свойств

Геттер и сеттер вычисляемого свойства

Для любого *вычисляемого* свойства существует возможность реализовать две специальные функции:

- *Геттер* (`get`) выполняет некоторый код при попытке получить значение вычисляемого свойства.
- *Сеттер* (`set`) выполняет некоторый код при попытке установить значение вычисляемому свойству.

Во всех объявленных ранее вычисляемых свойствах был реализован только геттер, поэтому они являлись свойствами «только для чтения», то есть попытка изменения вызвала бы ошибку. При этом не требовалось писать какой-либо код, который бы указывал на то, что существует некий геттер.

В случае, если вычисляемое свойство должно иметь и геттер, и сеттер, то необходимо использовать специальный синтаксис.

СИНТАКСИС

```
var имяСвойства: ТипДанных {
    get {
        // тело геттера
        return возвращаемоеЗначение
    }
    set (ассоциированныйПараметр) {
        // телосеттера
    }
}
```

- `ТипДанных: Any` — тип данных возвращаемого свойством значения.
- `возвращаемоеЗначение: ТипДанных` — значение, возвращаемое вычисляемым свойством.

Геттер и сеттер определяются внутри тела вычисляемого свойства. При этом используются ключевые слова `get` и `set` соответственно, за которыми в фигурных скобках следует тело каждой из функций.

Геттер срабатывает при запросе значения свойства. Для корректной работы он должен возвращать значение с помощью оператора `return`.

Сеттер срабатывает при попытке установить новое значение свойству. Поэтому необходимо указывать имя параметра, в который будет записано устанавливаемое значение. Данный ассоциированный параметр является локальным для тела функции `set()`.

Если в вычисляемом свойстве отсутствует сеттер, то есть реализуется только геттер, то можно использовать упрощенный синтаксис записи. В этом случае опускается ключевое слово `set` и указывается только тело замыкающего выражения. Данный формат мы встречали в предыдущих примерах.

Рассмотрим пример.

Необходимо разработать структуру, описывающую сущность «окружность». При этом окружность на плоскости имеет две основные характеристики: координаты центра и радиус. При этом нам также требуется третья характеристика: длина окружности, которая напрямую зависит от радиуса. Необходимо учесть, что в процессе работы с экземпляром как радиус, так и длина окружности могут быть изменены. Но при изменении одной величины также должна измениться и другая (листинг 25.5).

Листинг 25.5

```
struct Circle{
    var coordinates: (x: Int, y: Int)
    var radius: Float
    var perimeter: Float {
        get{
            return 2.0*3.14*self.radius
        }
        set(newPerimeter){
            self.radius = newPerimeter / (2*3.14)
        }
    }
}
var myNewCircle = Circle(coordinates: (0,0), radius: 10)
myNewCircle.perimeter // выводит 62.8
myNewCircle.perimeter = 31.4
myNewCircle.radius // выводит 5
```

При запросе значения свойства `perimeter` происходит выполнение кода в геттере, который генерирует возвращаемое значение с учетом значения свойства `radius`. При инициализации значения свойству `perimeter` срабатывает код из сеттера, который вычисляет и устанавливает значение свойства `radius`.

Сеттер также позволяет использовать сокращенный синтаксис записи, в котором не указывается имя входного параметра. При этом внутри

сеттера для доступа к устанавливаемому значению необходимо задействовать автоматически объявляемый параметр с именем `newValue`. Таким образом, класс `Circle` может выглядеть как в листинге 25.6.

Листинг 25.6

```
struct Circle{
    var coordinates: (x: Int, y: Int)
    var radius: Float
    var perimeter: Float {
        get{
            return 2.0*3.14*self.radius
        }
        set{
            self.radius = newValue / (2*3.14)
        }
    }
}
```

Наблюдатели хранимых свойств

Геттер и сеттер позволяют выполнять код при установке и чтении значения вычисляемого свойства. Другими словами, у вас имеются механизмы контроля попыток изменения и получения значений. Наделив такими полезными механизмами вычисляемые свойства, разработчики `Swift` не могли обойти стороной и хранимые свойства. Специально для них были реализованы наблюдатели (`observers`), также называемые *обсерверами*.

Наблюдатели — это специальные функции, которые вызываются либо непосредственно перед, либо сразу после установки нового значения хранимого свойства.

Выделяют два вида наблюдателей:

- Наблюдатель `willSet` вызывается перед установкой нового значения.
- Наблюдатель `didSet` вызывается после установки нового значения.

СИНТАКСИС

```
var имяСвойства: ТипЗначения {
    willSet (ассоциированныйПараметр){
        // тело обсервера
    }
    didSet (ассоциированныйПараметр){
       // тело обсервера
    }
}
```

Наблюдатели объявляются с помощью ключевых слов `willSet` и `didSet`, после которых в скобках указывается имя входного аргумента. В аргумент наблюдателя `willSet` записывается устанавливаемое значение, в наблюдатель `didSet` — старое, уже стертое.

При объявлении наблюдателей можно использовать сокращенный синтаксис, в котором не требуется указывать входные аргументы (точно так же, как сокращенный синтаксис сеттера). При этом новое значение в `willSet` присваивается параметру `newValue`, а старое в `didSet` — параметру `oldValue`.

Рассмотрим применение наблюдателей на примере. В структуру, описывающую окружность, добавим функционал, который при изменении радиуса окружности выводит соответствующую информацию на консоль (листинг 25.7).

Листинг 25.7

```
    struct Circle{
        var coordinates: (x: Int, y: Int)
        //var radius: Float
        // свойство для листинга 7
        var radius: Float {
            willSet (newValueOfRadius) {
                print("Вместо значения \(self.radius) устанавливается
                       значение \(newValueOfRadius)")
            }
            didSet (oldValueOfRadius) {
                print("Вместо значения \(oldValueOfRadius) установлено
                       значение \(self.radius)")
            }
        }
        var perimeter: Float {
            get{
                return 2.0*3.14*self.radius
            }
            set{
                self.radius = newValue / (2*3.14)
            }
        }
    }

    var myNewCircle = Circle(coordinates: (0,0), radius: 10)
    myNewCircle.perimeter // выводит 62.8
    myNewCircle.perimeter = 31.4
myNewCircle.radius // выводит 5
```

Консоль

```
Вместо значения 10.0 устанавливается значение 5.0
Вместо значения 10.0 установлено значение 5.0
```

Наблюдатели вызываются не только при непосредственном изменении значения свойства вне экземпляра. Так как сеттер свойства `perimeter` также изменяет значение свойства `radius`, то наблюдатели выводят на консоль соответствующий результат.

25.3. Свойства типа

Ранее мы рассматривали свойства, которые позволяют каждому отдельному экземпляру хранить свой, независимый от других экземпляров набор значений. Другими словами, можно сказать, что свойства экземпляра описывают характеристики определенного экземпляра и принадлежат определенному экземпляру.

Дополнительно к свойствам экземпляров вы можете объявлять свойства, относящиеся непосредственно к типу данных. Значения этих свойств едины для всех экземпляров данного типа.

Свойства типа данных очень полезны в том случае, когда существуют значения, которые являются универсальными для всего типа целиком. Они могут быть как хранимыми, так и вычисляемыми. При этом если значение хранимого свойства типа является переменной и изменяется в одном экземпляре, то измененное значение становится доступно во всех других экземплярах типа.

ПРИМЕЧАНИЕ Для хранимых свойств типа в обязательном порядке должны быть указаны значения по умолчанию. Это связано с тем, что сам по себе тип не имеет инициализатора, который бы мог сработать еще во время определения типа и установить требуемые значения для свойств.

Хранимые свойства типа всегда являются ленивыми, при этом они не нуждаются в использовании ключевого слова `lazy`.

Свойства типа могут быть созданы для перечислений, структур и классов.

СИНТАКСИС

```
struct SomeStructure {
     static var storedTypeProperty = "Some value"
     static var computedTypeProperty: Int {
```

```
        return 1
    }
}
enum SomeEnumiration{
    static var storedTypeProperty = "Some value"
    static var computedTypeProperty: Int {
        return 2
    }
}
class SomeClass{
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 3
    }
    class var overrideableComputedTypeProperty: Int {
        return 4
    }
}
```

Свойства типа объявляются с использованием ключевого слова `static` для перечислений, классов и структур. Единственным исключением являются маркируемые словом `class` вычисляемые свойства класса, которые могут быть переопределены в подклассе. О том, что такое подкласс, мы поговорим позже.

Создадим структуру для демонстрации работы свойств типа (листинг 25.8). Класс `AudioChannel` моделирует аудиоканал, у которого есть два параметра:

- максимально возможная громкость ограничена для всех каналов в целом;
- текущая громкость ограничена максимальной громкостью.

Листинг 25.8

```
struct AudioChannel {
    static var maxVolume = 5
    var volume: Int {
        didSet {
            if volume > AudioChannel.maxVolume {
                volume = AudioChannel.maxVolume
            }
        }
    }
}
var LeftChannel = AudioChannel(volume: 2)
var RightChannel = AudioChannel(volume: 3)
RightChannel.volume = 6
RightChannel.volume // 5
AudioChannel.maxVolume // 5
```

```
AudioChannel.maxVolume = 10
AudioChannel.maxVolume // 10
RightChannel.volume = 8
RightChannel.volume // 8
```

Мы использовали тип `AudioChannel` для создания двух каналов: левого и правого. Свойству `volume` не удается установить значение `6`, так как оно превышает значения свойства типа `maxVolume`.

Обратите внимание, что при обращении к свойству типа используется не имя экземпляра данного типа, а имя самого типа.

26 Сабскрипты

С сабскриптами вы уже знакомы. Мы использовали их при работе с массивами. Там сабскриптом был индекс, указываемый для доступа к значению элементов. Однако сабскрипты позволяют также упростить работу со структурами и классами.

26.1. Назначение сабскриптов

С помощью сабскриптов структуры и классы можно превратить в некое подобие коллекций. Таким образом можно организовать доступ к свойствам экземпляра с использованием специальных ключей (индексов).

Предположим, что нами разработан класс `Chessboard`, моделирующий сущность «шахматная доска». Экземпляр данного класса хранится в переменной `desk`:

```
var desk = Chessboard()
```

В одном из свойств данного экземпляра содержится информация о том, какая клетка поля какой шахматной фигурой занята. Для доступа к информации относительно определенной клетки мы можем разработать специальный метод, которому в качестве входных параметров будут передаваться координаты:

```
desk.getCellInfo("A", 5)
```

С помощью сабскриптов можно организовать доступ к ячейкам клетки, передавая координаты, подобно ключам массива, непосредственно экземпляру класса:

```
desk["A", 5]
```

ПРИМЕЧАНИЕ Сабскрипты доступны для структур и классов.

26.2. Синтаксис сабскриптов

В своей реализации сабскрипты являются чем-то средним между методами и вычисляемыми свойствами. От первых им достался синтаксис определения выходных параметров и типа возвращаемого значения, от вторых — возможность создания геттера и сеттера.

СИНТАКСИС

```
subscript(входные_параметры) -> тип_возвращаемого_значения {
    get{
        // тело геттера
    }
    set(ассоциированныйПараметр){
        // тело сеттера
    }
}
```

Сабскрипты объявляются в теле класса или структуры с помощью ключевого слова `subscript`. Далее указываются входные параметры (в точности так же, как у методов) и тип значения. Входные параметры — это значения, которые передаются в виде ключей. Тип значения указывает на тип данных устанавливаемого (в случае сеттера) или возвращаемого (в случае геттера) значения.

Тело сабскрипта заключается в фигурные скобки и состоит из геттера и сеттера по аналогии с вычисляемыми значениями. Геттер выполняет код при запросе значения с использованием сабскрипта, сеттер — при попытке установить значение.

Сеттер также дает возможность дополнительно указать имя ассоциированного параметра, которому будет присвоено устанавливаемое значение. Если данный параметр не будет указан, то новое значение автоматически инициализируется локальной переменной `newValue`. При этом тип данных параметра будет соответствовать типу возвращаемого значения.

Сеттер является необязательным, и в случае его отсутствия может быть использован сокращенный синтаксис:

```
subscript(входные_параметры) -> возвращаемое_значение {
// тело геттера
}
```

Сабскрипты поддерживают перегрузку, то есть в пределах одного объектного типа может быть определено произвольное количество сабскриптов, различающихся лишь набором входных аргументов.

ПРИМЕЧАНИЕ С перегрузками мы встречались, когда объявляли несколько функций с одним именем или несколько инициализаторов в пределах одного объектного типа. Каждый набор одинаковых по имени объектов отличался лишь набором входных параметров.

Для изучения сабскриптов вернемся к теме шахмат и создадим класс, описывающий сущность «шахматная доска». При разработке модели шахматной доски у нее можно выделить одну наиболее важную характеристику: коллекцию игровых клеток с указанием информации о находящихся на них шахматных фигурах. Не забывайте, что игровое поле — это матрица, состоящая из отдельных ячеек.

В данном примере будет использоваться созданный ранее тип `Chessman`, описывающий шахматную фигуру, включая вложенные в него перечисления.

При разработке класса шахматной доски реализуем метод, устанавливающий переданную ему фигуру на игровое поле. При этом стоит помнить о двух моментах:

- фигура, возможно, уже находилась на поле, а значит, ее требуется удалить со старой позиции;
- фигура имеет свойство `coordinates`, которое также необходимо изменять.

В листинге 26.1 показан код класса `gameDesk`, описывающего шахматную доску.

Листинг 26.1

```
class gameDesk {
    var desk: [Int:[String:Chessman]] = [:]
    init(){
        for i in 1...8 {
            desk[i] = [:]
        }
    }
    func setChessman(chess: Chessman , coordinates: (String, Int)){
        let rowRange = 1...8
        let colRange = "A"..."Z"
        if( rowRange.contains( coordinates.1 ) && colRange.contains
                              ( coordinates.0 )) {
            self.desk[coordinates.1]![coordinates.0] = chess
            chess.setCoordinates(char: coordinates.0,
                                 num: coordinates.1)
        } else {
            print("Coordinates out of range")
        }
    }
}
var game = gameDesk()
var queenBlack = Chessman(type: .queen, color: .black, figure:
                 "\u{265B}", coordinates: ("A", 6))
game.setChessman(chess: queenBlack, coordinates: ("B",2))
```

```
queenBlack.coordinates //(.0 "B", .1 2)
game.setChessman(chess: queenBlack, coordinates: ("A",3))
queenBlack.coordinates //(.0 "A", .1 3)
```

Класс `gameDesk` описывает игровое поле. Его единственным свойством является коллекция клеток, на которых могут располагаться шахматные фигуры (экземпляры класса `Chessman`).

При создании экземпляра свойству `desk` устанавливается значение по умолчанию «пустой словарь». Во время работы инициализатора в данный словарь записываются значения, соответствующие номерам строк на шахматной доске. Это делается для того, чтобы обеспечить безошибочную работу при установке фигуры на шахматную клетку. В противном случае при установке фигуры нам пришлось бы сначала узнать состояние линии (существует ли она в словаре), а уже потом записывать фигуру на определенные координаты.

Метод `setChessman(chess:coordinates:)` не просто устанавливает ссылку на фигуру в свойство `desk`, но также проверяет переданные координаты на корректность и устанавливает их в экземпляре фигуры.

Пока что в классе `GameDesk` отсутствует возможность запроса информации о произвольной ячейке. Реализуем ее с использованием сабскрипта (листинг 26.2). В сабскрипт будут передаваться координаты необходимой ячейки в виде двух отдельных входных аргументов. Если по указанным координатам существует фигура, то она возвращается, в противном случае возвращается `nil`.

Листинг 26.2

```
class GameDesk {
    var desk: [Int:[String:Chessman]] = [:]
    init(){
        for i in 1...8 {
            desk[i] = [:]
        }
    }
    //сабскрипт из листинга 2
    subscript(alpha: String, number: Int) -> Chessman? {
        get{
            return self.desk[number]![alpha]
        }
    }
    func setChessman(chess: Chessman , coordinates: (String, Int)){
        let rowRange = 1...8
        let colRange = "A"..."Z"
        if( rowRange.contains( coordinates.1 ) && colRange.contains
                              ( coordinates.0 )) {
```

```
            self.desk[coordinates.1]![coordinates.0] = chess
            chess.setCoordinates(char: coordinates.0, num:
                                        coordinates.1)
        } else {
            print("Coordinates out of range")
        }
    }
}
var game = GameDesk()
var queenBlack = Chessman(type: .queen, color: .black, figure:
                  "\u{265B}", coordinates: ("A", 6))
game.setChessman(chess: queenBlack, coordinates: ("A",3))
game["A",3]?.coordinates //(.0 "A", .1 3)
```

Реализованный сабскрипт имеет только геттер, причем в данном случае можно было использовать краткий синтаксис записи (без ключевого слова `get`).

Так как сабскрипт возвращает опционал, перед доступом к свойству `coordinates` возвращенной шахматной фигуры необходимо выполнить извлечение опционального значения.

Теперь мы имеем возможность установки фигур на шахматную доску с помощью метода `setChessman(chess:coordinates:)` и получения информации об отдельной ячейке с помощью сабскрипта.

Мы можем расширить функционал сабскрипта, добавив в него сеттер, позволяющий устанавливать фигуру на новые координаты (листинг 26.3).

Листинг 26.3

```
class GameDesk {
    var desk: [Int:[String:Chessman]] = [:]
    init(){
        for i in 1...8 {
            desk[i] = [:]
        }
    }
    subscript(alpha: String, number: Int) -> Chessman? {
        get{
            return self.desk[number]![alpha]
        }
        set{
            if let chessman = newValue {
                self.setChessman(chess: chessman, coordinates:
                                 (alpha, number))
            }else{
                self.desk[number]![alpha] = nil
            }
```

```
        }
    }
    func setChessman(chess: Chessman , coordinates: (String, Int)){
        let rowRange = 1...8
        let colRange = "A"..."Z"
        if( rowRange.contains( coordinates.1 ) && colRange.contains
                              ( coordinates.0 )) {
            self.desk[coordinates.1]![coordinates.0] = chess
            chess.setCoordinates(char: coordinates.0, num:
                                       coordinates.1)
        } else {
            print("Coordinates out of range")
        }
    }
}
var game = GameDesk()
var queenBlack = Chessman(type: .queen, color: .black, figure:
                 "\u{265B}", coordinates: ("A", 6))
game["C",5] = queenBlack
game["C",5] // Chessman
game["C",5] = nil
game["C",5] // nil
```

Тип данных переменной `newValue` в сеттере сабскрипта соответствует типу данных возвращаемого сабскриптом значения (`Chessman?`). По этой причине необходимо извлечь значение перед тем, как установить фигуру на шахматную клетку.

ПРИМЕЧАНИЕ Запомните, что сабскрипты не могут использоваться как хранилища, то есть через них мы организуем только доступ к хранилищам значений.

Сабсрипты действительно привносят в `Swift` много интересного. Согласитесь, что к сущности «шахматная доска» обращаться намного удобнее через индексы, чем без них.

Реализовать шахматную доску и шахматные фигуры с помощью Swift можно большим количеством способов. Все зависит от того, как вы будете использовать созданные экземпляры в вашей программе. К примеру, можно отказаться от указания координат в типе `Chessman` и работать с ними исключительно в рамках класса `GameDesk`. Но это, в свою очередь, может создать проблемы: вы не сможете напрямую обратиться к фигуре, например, для проверки ее наличия на поле. Каждый раз вам потребуется обходить все клетки поля. С другой стороны, такой подход поможет избежать дублирования, когда необходимо следить за тем, чтобы информация была актуальна и в экземпляре типа `GameDesk`, и в экземпляре типа `Chessaman`.

27 Наследование

Одной из главных целей объектно-ориентированного подхода является многократное использование кода. Объединять код для его многократного использования позволяют замыкания (включая функции) и объектные типы данных. В методологии ООП, помимо создания экземпляров различных перечислений, структур и классов, существует возможность создания нового класса на основе существующего с автоматическим включением в него всех свойств, методов и сабскриптов класса-родителя. Данный подход называется *наследованием*.

В рамках наследования «старый» класс называется суперклассом (или базовым классом), а «новый» — подклассом (или субклассом, или производным классом).

27.1. Синтаксис наследования

Для наследования одного класса другим необходимо указать имя суперкласса через двоеточие после имени объявляемого класса.

СИНТАКСИС

```
class SuperClass {
    // тело суперкласса
}
class SubClass: SuperClass {
    // тело подкласса
}
```

Для создания производного класса `SubClass`, для которого базовым является `SuperClass`, необходимо указать имя суперкласса через двоеточие после имени подкласса.

В результате все свойства и методы, определенные в классе `SuperClass`, становятся доступными в классе `SubClass` без их непосредственного объявления в производном типе.

ПРИМЕЧАНИЕ Значения наследуемых свойств могут изменяться независимо от значений соответствующих свойств родительского класса.

Рассмотрим пример, в котором создается базовый класс `Quadruped` с набором свойств и методов (листинг 27.1). Данный класс описывает сущность «четвероногое животное». Дополнительно объявляется субкласс `Dog`, описывающий сущность «собака». Все характеристики класса `Quadruped` применимы и к классу `Dog`, поэтому их можно наследовать.

Листинг 27.1

```
// суперкласс
class Quadruped {
    var type = ""
    var name = ""
    func walk(){
        print("walk")
    }
}
// подкласс
class Dog: Quadruped {
    func bark(){
        print("woof")
    }
}
var dog = Dog()
dog.type = "dog"
dog.walk() // выводит walk
dog.bark() // выводит woof
```

Экземпляр `myDog` позволяет получить доступ к свойствам и методам родительского класса `Quadruped`. Кроме того, класс `Dog` расширяет собственные возможности, реализуя в своем теле дополнительный метод `bark()`.

ПРИМЕЧАНИЕ Класс может быть суперклассом для произвольного количества субклассов.

Доступ к наследуемым характеристикам

Доступ к наследуемым элементам родительского класса в производном классе реализуется так же, как к собственным элементам данного производного класса, то есть с использованием ключевого слова `self`. В качестве примера в класс `Dog` добавим метод, выводящий на консоль кличку собаки. Кличка хранится в свойстве `name`, которое наследуется от класса `Quadruped` (листинг 27.2).

Листинг 27.2

```
class Dog: Quadruped {
    func bark(){
        print("woof")
    }
    // метод из листинга 2
    func printName(){
        print(self.name)
    }
}
var dog = Dog()
dog.name = "Dragon Wan Helsing"
dog.printName()
```

Консоль

```
Dragon Wan Helsing
```

Для класса безразлично, с какими характеристиками он взаимодействует, собственными или наследуемыми. Данное утверждение справедливо до тех пор, пока не меняется реализация наследуемых элементов, о которой мы поговорим в следующем разделе.

27.2. Переопределение наследуемых элементов

Субкласс может создавать собственные реализации свойств, методов и сабскриптов, наследуемых от суперкласса. Такие реализации называются *переопределенными*. Для переопределения параметров суперкласса в подклассе необходимо указать ключевое слово `override` перед определением элемента.

Переопределение методов

Довольно часто реализация метода, который «достался в наследство» от суперкласса, не соответствует требованиям разработчика. В таком случае в субклассе нужно переписать данный метод, обеспечив к нему доступ по прежнему имени. Объявим новый класс `NoisyDog`, который описывает сущность «беспокойная собака». Класс `Dog` является суперклассом для `NoisyDog`. В описываемый класс необходимо внедрить собственную реализацию метода `bark()`, но так как одноименный метод уже существует в родительском классе `Dog`, мы воспользуемся механизмом переопределения (листинг 27.3).

Листинг 27.3

```
class NoisyDog: Dog{
    override func bark(){
        print ("woof")
        print ("woof")
        print ("woof")
    }
}
var badDog = NoisyDog()
badDog.bark()
```

Консоль

```
woof
woof
woof
```

С помощью ключевого слова `override` мы сообщаем Swift, что метод `bark()` в классе `NoisyDog` имеет собственную реализацию.

ПРИМЕЧАНИЕ Класс может быть суперклассом вне зависимости от того, является он субклассом или нет.

Переопределенный метод не знает деталей реализации метода родительского класса. Он знает лишь имя и перечень входных параметров родительского метода.

Доступ к переопределенным элементам суперкласса

Несмотря на то что переопределение изменяет реализацию свойств, методов и сабскриптов, `Swift` позволяет осуществлять доступ внутри производного класса к переопределенным элементам суперкласса. Для этого в качестве префикса имени элемента вместо `self` используется ключевое слово `super`.

В предыдущем примере в методе `bark()` класса `NoisyDog` происходит дублирование кода. В нем используется функция вывода на консоль литерала `"woof"`, хотя данный функционал уже реализован в одноименном родительском методе. Перепишем реализацию метода `bark()` таким образом, чтобы избежать дублирования кода (листинг 27.4).

Листинг 27.4

```
class NoisyDog: Dog{
    override func bark(){
        for _ in 1...3 {
            super.bark()
        }
```

```
    }
}
var badDog = NoisyDog()
badDog.bark()
```

Консоль

```
woof
woof
woof
```

Вывод на консоль соответствует выводу реализации класса из предыдущего примера.

Доступ к переопределенным элементам осуществляется по следующим правилам:

- Переопределенный метод с именем `someMethod()` может вызвать одноименный метод суперкласса, используя конструкцию `super.someMethod()` внутри своей реализации (в коде переопределенного метода).
- Переопределенное свойство `someProperty` может получить доступ к свойству суперкласса с таким же именем, используя конструкцию `super.someProperty` внутри реализации своего геттера или сеттера.
- Переопределенный сабскрипт `someIndex` может обратиться к сабскрипту суперкласса с таким же форматом индекса, используя конструкцию `super[someIndex]` внутри реализации сабскрипта.

Переопределение инициализаторов

Инициализаторы являются такими же наследуемыми элементами, как и методы. Если в подклассе набор свойств, требующих установки значений, не отличается, то наследуемый инициализатор может быть использован для создания экземпляра подкласса.

Тем не менее вы можете создать собственную реализацию наследуемого инициализатора. Запомните, что если вы определяете инициализатор с уникальным для суперкласса и подкласса набором входных аргументов, то вы не переопределяете инициализатор, а объявляете новый.

Если подкласс имеет хотя бы один собственный инициализатор, то инициализаторы родительского класса не наследуются.

ВНИМАНИЕ Для вызова инициализатора суперкласса внутри инициализатора субкласса необходимо использовать конструкцию `super.init()`.

В качестве примера переопределим наследуемый от суперкласса `Quadruped` пустой инициализатор. В классе `Dog` значение наследуемого свойства `type` всегда должно быть равно `"dog"`. В связи с этим перепишем реализацию инициализатора таким образом, чтобы в нем устанавливалось значение данного свойства (листинг 27.5).

Листинг 27.5

```
class Dog: Quadruped {
    override init(){
        super.init()
        self.type = "dog"
    }
    func bark(){
        print("woof")
    }
    func printName(){
        print(self.name)
    }
}
```

Прежде чем получать доступ к наследуемым свойствам в переопределенном инициализаторе, необходимо вызвать инициализатор родителя. Он выполняет инициализацию всех наследуемых свойств.

Если в подклассе есть собственные свойства, которых нет в суперклассе, то их значения в инициализаторе необходимо указать до вызова инициализатора родительского класса.

Переопределение наследуемых свойств

Как отмечалось ранее, вы можете переопределять любые наследуемые элементы. Наследуемые свойства иногда ограничивают функциональные возможности субкласса. В таком случае можно переписать геттер или сеттер данного свойства или при необходимости добавить наблюдатель.

С помощью механизма переопределения вы можете расширить наследуемое «только для чтения» свойство до «чтение-запись», реализовав в нем и геттер и сеттер. Но обратное невозможно: если у наследуемого свойства реализованы и геттер и сеттер, вы не сможете сделать из него свойство «только для чтения».

ПРИМЕЧАНИЕ Хранимые свойства переопределять нельзя, так как вызываемый или наследуемый инициализатор родительского класса попытается установить их значения, но не найдет их.

Субкласс не знает деталей реализации наследуемого свойства в суперклассе, он знает лишь имя и тип наследуемого свойства. Поэтому необходимо всегда указывать и имя, и тип переопределяемого свойства.

27.3. Превентивный модификатор final

Swift позволяет защитить реализацию класса целиком или его отдельных элементов. Для этого необходимо использовать превентивный модификатор `final`, который указывается перед объявлением класса или его отдельных элементов:

- `final class` для классов;
- `final var` для свойств;
- `final func` для методов;
- `final subscript` для сабскриптов.

При защите реализации класса его наследование в другие классы становится невозможным. Для элементов класса их наследование происходит, но переопределение становится недоступным.

27.4. Подмена экземпляров классов

Наследование, помимо всех перечисленных возможностей, позволяет заменять требуемые экземпляры определенного класса экземплярами одного из подклассов.

Рассмотрим пример из листинга 27.6. В нем объявим массив элементов типа `Quadruped` и добавим в него несколько элементов.

Листинг 27.6

```
var animalsArray: [Quadruped] = []
var someAnimal = Quadruped()
var myDog = Dog()
var sadDog = NoisyDog()
animalsArray.append(someAnimal)
animalsArray.append(myDog)
animalsArray.append(sadDog)
```

В результате в массив `animalsArray` добавляются элементы типов `Dog` и `NoisyDog`. Это происходит несмотря на то, что в качестве типа массива указан класс `Quadruped`.

27.5. Приведение типов

Ранее нами были созданы три класса: `Quadruped`, `Dog` и `NoisyDog`, а также определен массив `animalsArray`, содержащий элементы всех трех типов данных. Данный набор типов представляет собой иерархию классов, поскольку между всеми классами можно указать четкие зависимости (кто кого наследует). Для анализа классов в единой иерархии существует специальный механизм, называемый *приведением типов*.

Путем приведения типов вы можете выполнить следующие операции:

- проверить тип конкретного экземпляра класса на соответствие некоторому типу или протоколу;
- преобразовать тип конкретного экземпляра в другой тип той же иерархии классов.

Проверка типа

Проверка типа экземпляра класса производится с помощью оператора `is`. Данный оператор возвращает `true` в случае, когда тип проверяемого экземпляра является указанным после оператора классом или наследует его. Для анализа возьмем определенный и заполненный ранее массив `animalsArray` (листинг 27.7).

Листинг 27.7

```
for item in animalsArray {
    if item is Dog {
        print("Yap")
    }
}
// Yap выводится 2 раза
```

Данный код перебирает все элементы массива `animalsArray` и проверяет их на соответствие классу `Dog`. В результате выясняется, что ему соответствуют только два элемента массива: экземпляр класса `Dog` и экземпляр класса `NoisyDog`.

Преобразование типа

Как отмечалось ранее, массив `animalsArray` имеет элементы разных типов данных из одной иерархической структуры. Несмотря на это, при получении очередного элемента вы будете работать исключительно с использованием методов класса, указанного в типе массива

(в данном случае Quadruped). То есть, получив элемент типа Dog, вы не увидите определенный в нем метод bark(), поскольку Swift подразумевает, что вы работаете именно с экземпляром типа Quadruped.

Для того чтобы преобразовать тип и сообщить Swift, что данный элемент является экземпляром определенного типа, используется оператор as, точнее, две его вариации: as? и as!. Данный оператор ставится после имени экземпляра, а после него указывается имя класса, в который преобразуется экземпляр.

Между обеими формами оператора существует разница:

- as? ИмяКласса возвращает либо экземпляр типа ИмяКласса? (опционал), либо nil в случае неудачного преобразования;
- as! ИмяКласса производит принудительное извлечение значения и возвращает экземпляр типа ИмяКласса или вызывает ошибку в случае неудачи.

ВНИМАНИЕ Тип данных может быть преобразован только в пределах собственной иерархии классов.

Снова приступим к перебору массива animalsArray. На этот раз будем вызывать метод bark(), который не существует в суперклассе Quadruped, но присутствует в подклассах Dog и NoisyDog (листинг 27.8).

Листинг 27.8

```
for item in animalsArray {
    if var animal = item as? NoisyDog {
        animal.bark()
    }else if var animal = item as? Dog {
        animal.bark()
    }else{
        item.walk()
    }
}
```

Каждый элемент массива animalArray связывается с параметром item. Далее в теле цикла данный параметр с использованием оператора as? пытается преобразоваться в каждый из типов данных нашей структуры классов. Если item преобразуется в тип NoisyDog или Dog, то у него становится доступным метод bark().

28 Псевдонимы Any и AnyObject

Swift предлагает два специальных псевдонима, позволяющих работать с неопределенными типами:

- `AnyObject` соответствует произвольному экземпляру любого класса;
- `Any` соответствует произвольному типу данных.

Данные псевдонимы позволяют корректно обрабатывать ситуации, когда конкретное наименование типа или класса неизвестно либо набор возможных типов может быть разнородным.

28.1. Псевдоним Any

Благодаря псевдониму `Any` можно создавать хранилища неопределенного типа данных. Объявим массив, который может содержать элементы произвольных типов (листинг 28.1).

Листинг 28.1

```
var things = [Any]()
things.append(0)
things.append(0.0)
things.append(42)
things.append("hello")
things.append((3.0, 5.0))
things.append({ (name: String) -> String in "Hello, \(name)" })
```

Массив `things` содержит значения типов: `Int`, `Double`, `String`, `(Double, Double)` и `(String)->String`. То есть перед вами целый набор различных типов данных.

При запросе любого из элементов массива вы получите значение не того типа данных, который предполагался при установке конкретного значения, а типа `Any`.

ПРИМЕЧАНИЕ Псевдоним `Any` несовместим с протоколом `Hashable`, поэтому использование типа `Any` там, где необходимо сопоставление, невозможно. Это относится, например, к ключам словарей.

Приведение типа Any

Для анализа каждого элемента массива необходимо выполнить приведение типа. Так вы сможете получить каждый элемент, преобразованный в его действительный тип данных.

Рассмотрим пример, в котором разберем объявленный ранее массив поэлементно и определим тип данных каждого элемента (листинг 28.2).

Листинг 28.2

```
for thing in things {
    switch thing {
    case let someInt as Int:
        print("an integer value of \(someInt)")
    case let someDouble as Double where someDouble > 0:
        print("a positive double value of \(someDouble)")
    case let someString as String:
        print("a string value of \"\(someString)\"")
    case let (x, y) as (Double, Double):
        print("an (x, y) point at \(x), \(y)")
    case let stringConverter as (String) -> String:
        print(stringConverter("Troll"))
    default:
        print("something else")
    }
}
```

Консоль

```
an integer value of 0
something else
an integer value of 42
a string value of "hello"
an (x, y) point at 3.0, 5.0
Hello, Troll
```

Каждый из элементов массива преобразуется в определенный тип при помощи оператора `as`. При этом в конструкции `switch-case` данный оператор не требует указывать какой-либо постфикс (знак восклицания или вопроса).

28.2. Псевдоним AnyObject

Псевдоним `AnyObject` позволяет указать на то, что в данном месте должен или может находиться экземпляр любого класса. При этом вы будете довольно часто встречать массивы данного типа при разработке программ с использованием фреймворка `Cocoa Foundation`. Данный фреймворк написан на Objective-C, а этот язык не имеет массивов с явно указанными типами данных.

Объявим массив экземпляров с помощью псевдонима `AnyObject` (листинг 28.3).

Листинг 28.3

```
let someObjects: [AnyObject] = [Dog(), NoisyDog(), Dog()]
```

При использовании псевдонима `AnyObject` нет ограничений на использование классов только из одной иерархической структуры. В данном примере если вы извлекаете произвольный элемент массива, то получаете экземпляр класса `AnyObject`, не имеющий свойств и методов для взаимодействия с ним.

Приведение типа AnyObject

Порой вы точно знаете, что все элементы типа `AnyObject` на самом деле имеют некоторый определенный тип. В таком случае для анализа элементов типа `AnyObject` необходимо выполнить приведение типа (листинг 28.4).

Листинг 28.4

```
for object in someObjects {
    let animal = object as! Dog
    print(animal.type)
}
```

Консоль

```
dog
dog
dog
```

Чтобы сократить запись, вы можете выполнить приведение типа для преобразования всего массива целиком (листинг 28.5).

Листинг 28.5

```
for object in someObjects as! [Dog]{
    print(animal.type)
}
```

29 Инициализаторы и деинициализаторы

Инициализатор — это специальный метод, выполняющий подготовительные действия при создании экземпляра объектного типа данных. Инициализатор срабатывает при создании экземпляра, а при его удалении вызывается деинициализатор.

29.1. Инициализаторы

Инициализатор выполняет установку начальных значений хранимых свойств и различные настройки, которые нужны для использования экземпляра.

Назначенные инициализаторы

При реализации собственных типов данных во многих случаях не требуется создавать собственный инициализатор, так как классы и структуры имеют встроенные инициализаторы:

- классы имеют пустой встроенный инициализатор `init(){}`;
- структуры имеют встроенный инициализатор, принимающий в качестве входных аргументов значения всех свойств.

ПРИМЕЧАНИЕ Пустой инициализатор срабатывает без ошибок только в том случае, если у класса отсутствуют свойства или у каждого свойства указано значение по умолчанию.

Для опциональных типов данных значение по умолчанию указывать не требуется, оно соответствует `nil`.

Инициализаторы класса и структуры, производящие установку значений свойств, называются *назначенными* (designated). Вы можете разработать произвольное количество назначенных инициализаторов с отличающимся набором параметров в пределах одного объектного

типа. При этом должен существовать хотя бы один назначенный инициализатор, производящий установку значений всех свойств (если они существуют), и один из назначенных инициализаторов должен обязательно вызываться при создании экземпляра. Назначенный инициализатор не может вызывать другой назначенный инициализатор, то есть использование конструкции `self.init()` запрещено.

ПРИМЕЧАНИЕ Инициализаторы наследуются от суперкласса к подклассу.

Единственный инициализатор, который может вызывать назначенный инициализатор, — это инициализатор производного класса, вызывающий инициализатор родительского класса для установки значений наследуемых свойств. Об этом мы говорили довольно подробно, когда изучали наследование.

ПРИМЕЧАНИЕ Инициализатор может устанавливать значения констант.

Внутри инициализатора необходимо установить значения свойств класса или структуры, чтобы к концу его работы все свойства имели значения (опционалы могут соответствовать `nil`).

Вспомогательные инициализаторы

Помимо назначенных, в Swift существуют *вспомогательные* (convenience) инициализаторы. Они являются вторичными и поддерживающими. Вы можете определить вспомогательный инициализатор для проведения настроек и обязательного вызова одного из назначенных инициализаторов. Вспомогательные инициализаторы не являются обязательными для их реализации в типе. Создавайте их, если это обеспечивает наиболее рациональный путь решения поставленной задачи.

Синтаксис объявления вспомогательных инициализаторов не слишком отличается от синтаксиса назначенных.

СИНТАКСИС

```
convenience init(параметры) {
    // тело инициализатора
}
```

Вспомогательный инициализатор объявляется с помощью модификатора `convenience`, за которым следует ключевое слово `init`. Данный тип инициализатора также может принимать входные аргументы и устанавливать значения для свойств.

В теле инициализатора обязательно должен находиться вызов одного из назначенных инициализаторов.

Вернемся к иерархии определенных ранее классов `Quadruped`, `Dog` и `NoisyDog`. Давайте перепишем класс `Dog` таким образом, чтобы при установке он давал возможность выводить на консоль произвольный текст. Для этого создадим вспомогательный инициализатор, принимающий на входе значение для наследуемого свойства `type` (листинг 29.1).

Листинг 29.1

```
class Dog: Quadruped {
    //метод из листинга 5
    override init(){
        super.init()
        self.type = "dog"
    }
    //инициализатор листинга 1 со страницы Init
    convenience init(text: String){
         self.init()
         print(text)
    }
    func bark(){
        print("woof")
    }
    // метод из листинга 2
    func printName(){
        print(self.name)
    }
}
var someDog = Dog(text: "Экземпляр класса Dog создан")
```

В результате при создании нового экземпляра класса `Dog` вам будет предложено выбрать один из двух инициализаторов: `init()` или `init(text:)`. Вспомогательный инициализатор вызывает назначенный инициализатор и выводит текст на консоль.

ПРИМЕЧАНИЕ Вспомогательный инициализатор может вызывать назначенный инициализатор через другой вспомогательный инициализатор.

Наследование инициализаторов

Наследование инициализаторов отличается от наследования обычных методов суперкласса. Есть два важнейших правила, которые необходимо помнить:

- Если подкласс имеет собственный назначенный инициализатор, то инициализаторы родительского класса не наследуются.

- Если подкласс переопределяет все назначенные инициализаторы суперкласса, то он наследует и все его вспомогательные инициализаторы.

Отношения между инициализаторами

В вопросах отношений между инициализаторами Swift соблюдает следующие правила:

- Назначенный инициализатор подкласса должен вызвать назначенный инициализатор суперкласса.
- Вспомогательный инициализатор должен вызвать назначенный инициализатор того же объектного типа.
- Вспомогательный инициализатор в конечном счете должен вызвать назначенный инициализатор.

Иллюстрация всех трех правил представлена на рис. 29.1.

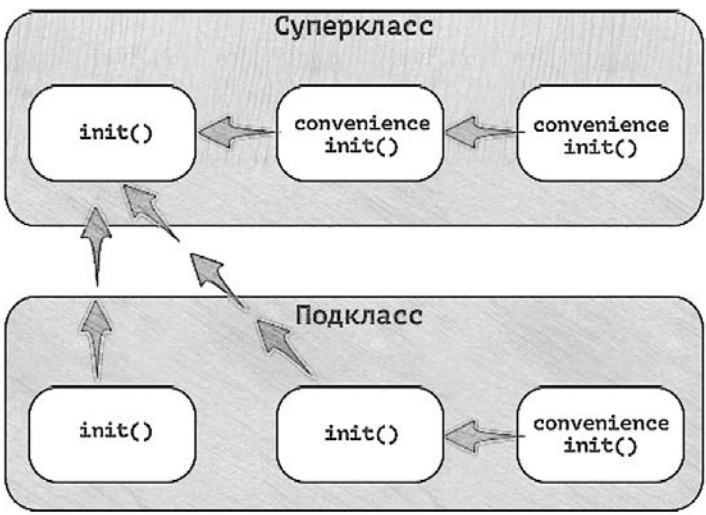

Рис. 29.1. Отношения между инициализаторами

Здесь изображен суперкласс с одним назначенным и двумя вспомогательными инициализаторами. Один из вспомогательных инициализаторов вызывает другой, который, в свою очередь, вызывает назначенный. Также изображен подкласс, имеющий два собственных назначенных инициализатора и один вспомогательный.

Вызов любого инициализатора из изображенных должен в итоге вызывать назначенный инициализатор суперкласса (левый верхний блок).

Проваливающиеся инициализаторы

В некоторых ситуациях бывает необходимо определить объектный тип, создание экземпляра которого может закончиться неудачей, вызванной некорректным набором внешних параметров, отсутствием какого-либо внешнего ресурса или иным обстоятельством. Для этой цели служат *проваливающиеся* (failable) инициализаторы. Они способны возвращать `nil` при попытке создания экземпляра. И это их основное предназначение.

СИНТАКСИС

```
init?(параметры) {
    // тело инициализатора
}
```

Для создания проваливающегося инициализатора служит ключевое слово `init?` (со знаком вопроса), которое говорит о том, что возвращаемый экземпляр будет опционалом или его не будет вовсе.

В теле инициализатора должно присутствовать выражение `return nil`.

Рассмотрим пример реализации проваливающегося инициализатора. Создадим класс, описывающий сущность «прямоугольник». При создании экземпляра данного класса необходимо контролировать значения передаваемых параметров (высота и ширина), чтобы они обязательно были больше нуля. При этом в случае некорректных значений параметров программа не должна завершаться с ошибкой.

Для решения данной задачи используем проваливающийся инициализатор (листинг 29.2).

Листинг 29.2

```
class Rectangle{
    var height: Int
    var weight: Int
    init?(height h: Int, weight w: Int){
        self.height = h
        self.weight = w
        if !(h > 0 && w > 0) {
            return nil
        }
```

```
    }
}
var rectangle = Rectangle(height: 56, weight: -32) // возвращает nil
```

Инициализатор принимает и проверяет значения двух параметров. Если хотя бы одно из них не больше нуля, то возвращается `nil`. Обратите внимание на то, что, прежде чем вернуть `nil`, инициализатор устанавливает значения всех хранимых свойств.

ВНИМАНИЕ В классах проваливающийся инициализатор может вернуть `nil` только после установки значений всех хранимых свойств. В случае структур данное ограничение отсутствует.

Назначенный инициализатор в подклассе может переопределить проваливающийся инициализатор суперкласса, а проваливающийся инициализатор может вызывать назначенный инициализатор того же класса.

Не забывайте, что в случае использования проваливающегося инициализатора возвращается опционал. Поэтому, прежде чем работать с экземпляром, необходимо выполнить извлечение опционального значения.

Вы можете использовать проваливающийся инициализатор для выбора подходящего члена перечисления, основываясь на значениях входных аргументов. Рассмотрим пример из листинга 29.3. В данном примере объявляется перечисление `TemperatureUnit`, содержащее три члена. Проваливающийся инициализатор используется для того, чтобы вернуть член перечисления, соответствующий переданному параметру, или `nil`, если значение параметра некорректно.

Листинг 29.3

```
enum TemperatureUnit {
    case Kelvin, Celsius, Fahrenheit
    init?(symbol: Character) {
        switch symbol {
        case "K":
            self = .Kelvin
        case "C":
            self = .Celsius
        case "F":
            self = .Fahrenheit
        default:
            return nil
        }
```

```
    }
}
let fahrenheitUnit = TemperatureUnit(symbol: "F")
```

При создании экземпляра перечисления в качестве входного параметра `symbol` передается значение. На основе переданного значения возвращается соответствующий член перечисления.

У перечислений, члены которых имеют значения, есть встроенный проваливающийся инициализатор `init?(rawValue:)`. Его можно использовать без определения в коде (листинг 29.4).

Листинг 29.4

```
enum TemperatureUnit: Character {
    case Kelvin = "K", Celsius = "C", Fahrenheit = "F"
}
let fahrenheitUnit = TemperatureUnit(rawValue: "F")
fahrenheitUnit!.hashValue
```

Члены перечисления `TemperatureUnit` имеют значения типа `Character`. В этом случае вы можете вызвать встроенный проваливающийся инициализатор, который вернет член перечисления, соответствующий переданному значению.

Альтернативой инициализатору `init?` служит оператор `init!`. Разница в них заключается лишь в том, что второй возвращает неявно извлеченный экземпляр объектного типа, поскольку для работы с ним не требуется дополнительно извлекать опциональное значение. При этом все еще может возвращаться `nil`.

Обязательные инициализаторы

Обязательный (required) инициализатор — это инициализатор, который обязательно должен быть определен во всех подклассах данного класса.

СИНТАКСИС

```
required init(параметры) {
    // тело инициализатора
}
```

Для объявления обязательного инициализатора перед ключевым словом `init` указывается модификатор `required`.

Кроме того, модификатор `required` необходимо указывать перед каждой реализацией данного инициализатора в подклассах, чтобы последующие подклассы также реализовывали этот инициализатор.

При реализации инициализатора в подклассе ключевое слово `override` не используется.

29.2. Деинициализаторы

Деинициализаторы являются отличительной особенностью классов.

Деинициализатор автоматически вызывается во время уничтожения экземпляра класса. Вы не можете вызвать деинициализатор самостоятельно. Один класс может иметь максимум один деинициализатор.

С помощью деинициализатора вы можете, например, освободить используемые экземпляром ресурсы, вывести на консоль журнал или выполнить любые другие действия.

СИНТАКСИС

```
deinit {
     // тело деинициализатора
}
```

Для объявления деинициализатора предназначено ключевое слово `deinit`, после которого в фигурных скобках указывается тело деинициализатора.

Деинициализатор суперкласса наследуется подклассом и вызывается автоматически в конце работы деинициализаторов подклассов. Деинициализатор суперкласса вызывается всегда, даже если деинициализатор подкласса отсутствует. Кроме того, экземпляр класса не удаляется, пока не закончит работу деинициализатор, поэтому все значения свойств экземпляра остаются доступными в теле деинициализатора.

Рассмотрим пример использования деинициализатора (листинг 29.5).

Листинг 29.5

```
class SuperClass {
    init?(isNil: Bool){
        if isNil == true {
            return nil
        }else{
            print("Экземпляр создан")
        }
    }
    deinit {
        print("Деинициализатор суперкласса")
    }
}
class SubClass:SuperClass{
```

```
    deinit {
        print("Деинициализатор подкласса")
    }
}
var obj = SubClass(isNil: false)
obj = nil
```

Консоль

```
Экземпляр создан
Деинициализатор подкласса
Деинициализатор суперкласса
```

При создании экземпляра класса `SubClass` на консоль выводится соответствующее сообщение, так как данный функционал находится в наследуемом от суперкласса проваливающемся инициализаторе. В конце программы мы удаляем созданный экземпляр, передав ему в качестве значения `nil`. При этом вывод на консоль показывает, что первым выполняется деинициализатор подкласса, потом — суперкласса.

30 Удаление экземпляров и ARC

Любой созданный экземпляр объектного типа данных, как и вообще любое хранилище вашей программы, занимает некоторый объем оперативной памяти. Если не производить своевременное уничтожение экземпляров и освобождение занимаемых ими ресурсов, то в программе может произойти утечка памяти.

ПРИМЕЧАНИЕ Утечка памяти — это программная ошибка, приводящая к излишнему расходованию оперативной памяти.

Одним из средств решения проблемы исключения утечек памяти в Swift является использование деинициализаторов, но возможности Swift не ограничиваются только этим.

30.1. Уничтожение экземпляров

Как мы отмечали в предыдущей главе, непосредственно перед уничтожением экземпляра класса вызывается деинициализатор, при этом остался нерассмотренным вопрос о том, как удаляется экземпляр. Любой экземпляр может быть удален одним из двух способов:

- его самостоятельно уничтожает разработчик;
- его уничтожает Swift.

Область видимости

Ранее мы самостоятельно уничтожали созданный экземпляр опционального типа `SubClass?`, передавая ему в качестве значения `nil`. Теперь обратимся к логике работы Swift. Для этого разработаем класс `myClass`, который содержит единственное свойство `description`. Данное свойство служит для того, чтобы отличать один экземпляр класса от другого.

Необходимо создать два экземпляра, один из которых должен иметь ограниченную область видимости (листинг 30.1).

Листинг 30.1

```
class myClass{
    var description: String
    init(description: String){
        print("Экземпляр \(description) создан")
        self.description = description
    }
    deinit{
        print("Экземпляр \(self.description) уничтожен")
    }
}
var myVar1 = myClass(description: "ОДИН")
if true {
    var myVar2 = myClass(description: "ДВА")
}
```

Консоль

```
Экземпляр ОДИН создан
Экземпляр ДВА создан
Экземпляр ДВА уничтожен
```

Экземпляр `myVar2` имеет область видимости, ограниченную оператором `if`. Несмотря на то что мы не выполняли принудительное удаление экземпляра, для него был вызван деинициализатор, в результате он был автоматически удален.

Причина удаления второго экземпляра лежит в области видимости хранящей его переменной. Первый экземпляр инициализируется вне оператора `if`, а значит, является глобальным для всей программы. Второй экземпляр является локальным для условного оператора. Как только выполнение тела оператора завершается, область видимости объявленной в нем переменной заканчивается, и она вместе с хранящимся в ней экземпляром автоматически уничтожается.

Количество ссылок на экземпляр

Рассмотрим пример, в котором на один экземпляр указывает несколько разных ссылок (листинг 30.2).

Листинг 30.2

```
class myClass{
    var description: String
    init(description: String){
```

```
        print("Экземпляр \(description) создан")
        self.description = description
    }
    deinit{
        print("Экземпляр \(self.description) уничтожен")
    }
}
var myVar1 = myClass(description: "ОДИН")
var myVar2 = myVar1
myVar1 = myClass(description: "ДВА")
myVar2 = myVar1
```

Консоль

```
Экземпляр ОДИН создан
Экземпляр ДВА создан
Экземпляр ОДИН уничтожен
```

В переменной `myVar1` изначально хранится ссылка на экземпляр класса `myClass`. После записи данной ссылки в переменную `myVar2` на созданный экземпляр уже указывают две ссылки. В результате этот экземпляр удаляется лишь тогда, когда удаляется последняя ссылка на него.

Не забывайте, что экземпляры классов в Swift передаются не копированием, а по ссылке.

ПРИМЕЧАНИЕ Экземпляр существует до тех пор, пока на него указывает хотя бы одна ссылка.

30.2. Утечки памяти

Утечка памяти может привести к самым печальным результатам, одним из которых может быть отказ пользователей от вашей программы.

Пример утечки памяти

Рассмотрим ситуацию, при которой может возникнуть утечка памяти. Разработаем класс, который может описать человека и его родственные отношения с другими людьми. Для этого в качестве типа свойств класса будет указан сам тип (листинг 30.3).

Листинг 30.3

```
class Human {
let name: String
var child = [Human?]()
var father: Human?
var mother: Human?
```

```
        init(name: String){
            self.name = name
        }
        deinit {
            print("\(self.name) — удален")
        }
    }
    var Kirill: Human? = Human(name: "Кирилл")
    var Olga: Human? = Human(name: "Ольга")
    var Aleks: Human? = Human(name: "Алексей")
    Kirill?.father = Aleks
    Kirill?.mother = Olga
    Aleks?.child.append(Kirill)
    Olga?.child.append(Kirill)
    Kirill = nil
    Aleks = nil
Olga = nil
```

Несмотря на то что деинициализатор определил вывод на консоль при удалении экземпляра, на консоль ничего не выводится. Это говорит о том, что созданные в коде экземпляры не удаляются (их деинициализатор не вызывается) несмотря на то, что содержащим их параметрам инициализируется `nil`.

Экземпляры остаются неудаленными, поскольку к моменту, когда они должны быть удалены, ссылки на объекты все еще существуют, и Swift не может понять, какие из ссылок можно удалить, а какие нельзя.

Это типичный пример утечки памяти в приложениях.

Сильные и слабые ссылки

Swift пытается не позволить программе создавать ситуации, приводящие к утечкам памяти. Представьте, что произойдет, если объекты, занимающие большую область памяти, не будут удаляться, занимая драгоценное свободное пространство. В конце концов приложение «упадет». Такие ситуации приведут к отказу пользователей от приложения.

Для решения описанной ситуации, когда Swift не знает, какие из ссылок можно удалять, а какие нет, существует специальный механизм, называемый *сильными* и *слабыми ссылками*. Все создаваемые ссылки на экземпляр по умолчанию являются сильными. И когда два объекта указывают друг на друга сильными ссылками, то Swift не может принять решение о том, какую из ссылок можно удалить

первой. Для решения проблемы некоторые ссылки можно преобразовать в слабые.

Слабые ссылки определяются с помощью ключевых слов `weak` и `unowned`. Модификатор `weak` указывает на то, что хранящаяся в параметре ссылка может быть в автоматическом режиме на усмотрение программы заменена на `nil`. Поэтому модификатор weak доступен только для опционалов. Но помимо опционалов, бывают типы данных, которые обязывают переменную хранить значение (все неопциональные типы данных). Для создания слабых ссылок на неопционалы служит модификатор `unowned`.

Перепишем пример из предыдущего листинга, преобразовав свойства `father` и `mother` как слабые ссылки (листинг 30.4).

Листинг 30.4

```
class Human {
    let name: String
    var child = [Human?]()
    weak var father: Human?
    weak var mother: Human?
    init(name: String){
        self.name = name
    }
    deinit {
        print("\(self.name) — удален")
    }
}
var Kirill: Human? = Human(name: "Кирилл")
var Olga: Human? = Human(name: "Ольга")
var Aleks: Human? = Human(name: "Алексей")
Kirill?.father = Aleks
Kirill?.mother = Olga
Aleks?.child.append(Kirill)
Olga?.child.append(Kirill)
Kirill = nil
Aleks = nil
Olga = nil
```

Консоль

```
Алексей — удален
Ольга — удален
Кирилл — удален
```

В результате все три объекта будут удалены, так как после удаления слабых ссылок никаких перекрестных ссылок не остается.

Указанные ключевые слова можно использовать только для хранилища определенного экземпляра класса. Например, вы не можете указать слабую ссылку на массив экземпляров или на кортеж, состоящий из экземпляров класса.

30.3. Автоматический подсчет ссылок

Хотя в названии данной главы фигурирует аббревиатура ARC (Automatic Reference Counting — автоматический подсчет ссылок), в ходе изучения мы еще ни разу к ней не обращались. На самом деле во всех наших действиях с экземплярами классов всегда участвовал механизм автоматического подсчета ссылок.

Понятие ARC

ARC в Swift автоматически управляет занимаемой памятью, удаляя неиспользуемые объекты. С помощью этого механизма вы можете «просто заниматься программированием», не переключаясь на задачи, которые система решает за вас в автоматическом режиме.

Как уже упоминалось, при создании нового хранилища, в качестве значения которому передается экземпляр класса, данный экземпляр помещается в оперативную память, а в хранилище записывается ссылка на него. На один и тот же экземпляр может указывать произвольное количество ссылок, и ARC в любой момент времени знает, сколько таких ссылок хранится в переменных, константах и свойствах. Как только последняя ссылка на экземпляр будет удалена или ее область видимости завершится, ARC незамедлительно вызовет деинициализатор и уничтожит объект.

Таким образом, ARC делает работу со Swift еще более удобной.

Сильные ссылки в замыканиях

Ранее мы выяснили, что использование сильных ссылок может привести к утечкам памяти. Также мы узнали, что для решения возникших проблем могут помочь слабые ссылки.

Сильные ссылки могут также стать источником проблем при их передаче в качестве входных параметров в замыкания. Захватываемые замыканиями экземпляры классов передаются по сильной ссылке и не освобождаются, когда замыкание уже не используется. Рассмотрим

пример. В листинге 30.5 создается пустое опциональное замыкание, которому в зоне ограниченной области видимости передается значение (тело замыкания).

Листинг 30.5

```
class Human{
    var name = "Человек"
    deinit{
        print("Объект удален")
    }
}
var closure : (() -> ())?
if true{
    var human = Human()
    closure = {
        print(human.name)
    }
    closure!()
}
print("Программа завершена")
```

Консоль

```
Человек
Программа завершена
```

Так как условный оператор ограничивает область видимости переменной `human`, содержащей экземпляр класса `Human`, то, казалось бы, данный объект должен быть удален вместе с окончанием условного оператора. Однако по выводу на консоль видно, что экземпляр создается, но перед завершением программы его деинициализатор не вызывается.

Созданное опциональное замыкание использует сильную ссылку на созданный внутри условного оператора экземпляр класса. Так как замыкание является внешним по отношению к условному оператору, а ссылка сильной, Swift самостоятельно не может принять решение о возможности удаления ссылки и последующего уничтожения экземпляра.

Для решения проблемы в замыкании необходимо выполнить захват переменной, указав при этом, что в переменной содержится слабая ссылка (листинг 30.6).

Листинг 30.6

```
class Human{
    var name = "Человек"
    deinit{
```

```
        print("Объект удален")
    }
}
var closure : (() -> ())?
if true{
    var human = Human()
    // измененное замыкание
    closure = {
        [unowned human] in
        print(human.name)
    }
    closure!()
}
print("Программа завершена")
```

Консоль

```
Человек
Объект удален
Программа завершена
```

Захватываемый параметр `human` является локальным для замыкания и условного оператора, поэтому Swift без проблем может самостоятельно принять решение об уничтожении хранящейся в нем ссылки.

В данном примере используется модификатор `unowned`, поскольку объектный тип не является опционалом.

31 Опциональные цепочки

Как вы знаете, опционалы позволяют хранить в параметре некоторое значение или не хранить его вовсе. Опционалы также широко используются и при работе с объектными типами данных, об особенностях их использования мы и поговорим в данной главе.

31.1. Доступ к свойствам через опциональные цепочки

Рассмотрим следующий пример.

Разработаны два класса: первый является простейшим описанием сущности «место жительства», а второй — сущности «персона». При этом у персоны имеется свойство опционального типа, которое может содержать экземпляр «места жительства» (листинг 31.1).

Листинг 31.1

```
class Residence {
    var rooms = 1
}
class Person {
    var residence: Residence?
}
```

Экземпляры класса `Person` имеют единственное свойство со ссылкой на экземпляр класса `Residence`, который также имеет всего одно свойство, характеризующее количество комнат.

Если создать экземпляр класса `Person`, то свойство `residence` получит свойство `nil`, поскольку оно является опционалом. В определенный момент времени мы не сможем сказать, а есть ли значение в данном

опционале. Для того чтобы избежать ошибки, необходимо сперва проверить наличие значения в свойстве `residence`, и только потом обращаться к нему. Мы знаем уже несколько способов сделать это, одним из которых является опциональное связывание (листинг 31.2).

Листинг 31.2

```
var man = Person()
if let manResidence = man.residence {
    print("Есть дом с \(manResidence.rooms) комнатами")
}else{
    print("Нет дома")
}
```

Консоль

```
Нет дома
```

Представленный подход позволяет решить проблему, но потребует писать лишний код, если, к примеру, вложенность классов в качестве опциональных свойств окажется многоуровневой. Для демонстрации этого создадим новый объектный тип, описывающий сущность «комната», и сделаем на него ссылку в классе `Residence` (листинг 31.3).

Листинг 31.3

```
struct Room {
    let square: Int
}
class Residence {
    var rooms:[Room]?
}
class Person {
    var residence: Residence?
}
//создаем объект Персона
var man = Person()
if let residence = man.residence {
    if let rooms = residence.rooms {
        for oneRoom in rooms {
            print("Есть комната площадью \(oneRoom.square)")
        }
    }else{
        // нет комнат
    }
}else{
    // нет резиденции
}
```

ПРИМЕЧАНИЕ Обратите внимание, что свойство `rooms` типа `Residence` теперь является опциональной коллекцией.

Как видите, для доступа к свойству `square` требуется строить вложенные конструкции опционального связывания. И чем сложнее будут ваши программы, тем сложнее будут ваши конструкции.

Для решения данной проблемы можно использовать *опциональные цепочки*. Они позволяют в одном выражении написать полный путь до требуемого элемента, при этом после каждого опционала необходимо ставить знак вопроса (вместо знака восклицания, как мы делали это при принудительном извлечении).

Продемонстрируем использование опциональных цепочек (листинг 31.4)

Листинг 31.4

```
//создаем объект комната
let room = Room(square: 10)
//создаем объект место проживания
var residence = Residence()
//добавляем в него комнату
residence.rooms = [room]
//создаем объект Персона
var man = Person()
//добавляем в него резиденцию
man.residence = residence
if let rooms = man.residence?.rooms {
    for oneRoom in rooms {
        print("Есть комната площадью \(oneRoom.square)")
    }
}else{
    // нет резиденции или комнат
}
```

Консоль

```
Есть комната площадью 10
```

Если в свойстве `residence` не будет значения, то операция опционального связывания выполнена не будет и произойдет переход к альтернативной ветке условного оператора. При этом никаких ошибок не возникнет. В случае, когда опциональная цепочка не может получить доступ к элементу, результатом выражения является `nil`.

Вы можете использовать опциональные цепочки для вызова свойств, методов и сабскриптов для любого уровня вложенности типов друг

в друга. Это позволяет «пробираться» через подсвойства внутри сложных моделей вложенных типов и проверять возможность доступа к свойствам, методам и сабскриптам этих подсвойств. К примеру, если бы у типа `Residence` было свойство `kitchen: Room?`, то вы могли бы осуществить доступ к свойству `square` следующим образом:

```
man.residence?.kitchen?.square
```

31.2. Установка значений через опциональные цепочки

Использование опциональных цепочек не ограничивается получением свойств и вызовом сабскриптов и методов. Они также могут быть использованы и для установки значений вложенных свойств.

Вернемся к примеру с жилищем человека. Пусть нам необходимо изменить значение свойства `rooms`. Мы можем сделать это с помощью опциональных цепочек (листинг 31.5)

Листинг 31.5

```
let room1 = Room(square: 15)
let room2 = Room(square: 25)
man.residence?.rooms = [room1, room2]
```

Для решения поставленной задачи используется опциональная цепочка `man.residence?.rooms`, указывающая на требуемый элемент. Если на каком-то из этапов экземпляр будет отсутствовать, программа не вызовет ошибку, а лишь не выполнит данное выражение.

31.3. Доступ к методам через опциональные цепочки

Как отмечалось ранее, опциональные цепочки могут быть использованы не только для доступа к свойствам, но и для вызова методов. В класс `Residence` добавим новый метод, который должен обеспечивать вывод информации о количестве комнат (листинг 31.6).

Листинг 31.6

```
class Residence {
    var rooms:[Room]?
    func roomsCount()->Int {
```

```
        if let rooms = self.rooms {
            return rooms.count
        }else{
            return 0
        }
    }
}
```

Для вызова данного метода также можно воспользоваться опциональной последовательностью (листинг 31.7).

Листинг 31.7

```
man.residence?.roomsCount()
```

32 Расширения

Расширения позволяют добавить новую функциональность к существующему объектному типу (классу, структуре или перечислению), а также к протоколу. Более того, вы можете расширять типы данных, доступа к исходным кодам которых у вас нет (например, типы, предоставляемые фреймворками, или фундаментальные для Swift типы данных).

Перечислим возможности расширений:

- добавление вычисляемых свойств экземпляра и вычисляемых свойств типа (`static`);
- определение методов экземпляра и методов типа;
- определение новых инициализаторов, сабскриптов и вложенных типов;
- обеспечение соответствия существующего типа протоколу.

Расширения могут добавлять новый функционал к типу, но не могут изменять существующий. Суть расширения состоит исключительно в наращивании возможностей, но не в их изменении.

СИНТАКСИС

```
extension ИмяРасширяемогоТипа {
    // описание новой функциональности для расширяемого типа
}
```

Для объявления расширения используется ключевое слово `extension`, после которого указывается имя расширяемого типа данных. Именно к указанному типу применяются все описанные в теле расширения возможности.

Новая функциональность, добавляемая расширением, становится доступной всем экземплярам расширяемого объектного типа вне зависимости от того, где эти экземпляры объявлены.

32.1. Вычисляемые свойства в расширениях

Расширения могут добавлять вычисляемые свойства экземпляра и вычисляемые свойства типа в определенный тип данных. Рассмотрим пример расширения функционала типа `Double`, создав в нем ряд новых вычисляемых свойств и обеспечив тип `Double` возможностью работы с единицами длины (листинг 32.1).

Листинг 32.1

```
extension Double {
    var km: Double { return self * 1000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1000.0 }
    var ft: Double { return self / 3.28084 }
}
let oneInch = 25.4.mm
print("Один фут — это \(oneInch) метра")
// выводит "Один фут — это 0.0254 метра"
let threeFeet = 3.ft
print("Три фута — это \(threeFeet) метра")
// выводит "Три фута — это 0.914399970739201 метра"
```

Созданные вычисляемые свойства позволяют использовать дробные числа как конкретные единицы измерения длины. Добавленные новые свойства могут применяться для параметров и литералов типа `Double`.

В данном примере подразумевается, что значение `1.0` типа `Double` отражает величину один метр. Именно поэтому свойство `m` возвращает значение `self`.

Другие свойства требуют некоторых преобразований перед возвращением значений. Один километр — это то же самое, что `1000` метров, поэтому при запросе свойства `km` возвращается результат выражения `self * 1000`.

Чтобы после определения новых вычисляемых свойств использовать всю их мощь, требуется создавать и геттеры, и сеттеры.

ПРИМЕЧАНИЕ Расширения могут добавлять только новые вычисляемые свойства. При попытке добавить хранимые свойства или наблюдателей свойств происходит ошибка.

32.2. Инициализаторы в расширениях

Расширения могут добавлять инициализаторы к существующему типу. Таким образом, вы можете расширить существующие типы, напри-

мер, для обработки экземпляров ваших собственных типов в качестве входных аргументов.

ПРИМЕЧАНИЕ Для классов расширения могут добавлять только новые вспомогательные инициализаторы. Попытка добавить назначенный инициализатор или деинициализатор ведет к ошибке.

В качестве примера напишем инициализатор для типа `Double` (листинг 32.2). В этом примере создается структура, описывающая линию на плоскости. Необходимо реализовать инициализатор, принимающий в качестве входного аргумента экземпляр линии и устанавливающий значение, соответствующее длине линии.

Листинг 32.2

```
import Foundation
// сущность "линия"
struct Line{
    var pointOne: (Double, Double)
    var pointTwo: (Double, Double)
}
// расширения для Double
extension Double {
    init(line: Line){
        self = sqrt(pow((line.pointTwo.0 - line.pointOne.0),2) +
                    pow((line.pointTwo.1 - line.pointOne.1),2))
    }
}
var myLine = Line(pointOne: (10,10), pointTwo: (14,10))
var lineLength = Double(line: myLine) // 4
```

Библиотека Foundation обеспечивает доступ к математическим функциям `sqrt(_:)` и `pow(_:_:)` (соответственно квадратный корень и возведение в степень), которые требуются для вычисления длины линии на плоскости.

Структура `Line` описывает сущность «линия», в свойствах которой указываются координаты точек ее начала и конца. Созданный в расширении инициализатор принимает на входе экземпляр класса `Line` и на основе значений его свойств вычисляет требуемое значение.

При разработке нового инициализатора в расширении будьте крайне внимательны к тому, чтобы к завершению инициализации каждое из свойств имело определенное значение.

32.3. Методы в расширениях

Следующей рассматриваемой функцией расширений является создание новых методов в расширяемом типе данных. Рассмотрим пример (листинг 32.3). В этом примере путем расширения типа `Int` мы добавляем метод `repetitions`, принимающий на входе замыкание типа `() -> ()`. Данный метод предназначен для того, чтобы выполнять переданное замыкание столько раз, сколько указывает собственное значение целого числа.

Листинг 32.3

```
extension Int {
    func repetitions(task: () -> ()) {
        for _ in 0..<self {
            task()
        }
    }
}
3.repetitions{
    print("Swift")
}
```

Консоль

```
Swift
Swift
Swift
```

Для изменения свойств перечислений и структур реализуемыми расширением методами необходимо не забывать использовать модификатор `mutating`. В следующем примере реализуется метод `square()`, который возводит в квадрат собственное значение экземпляра. Так как тип `Int` является структурой, то для изменения собственного значения экземпляра необходимо использовать ключевое слово `mutating` (листинг 32.4).

Листинг 32.4

```
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
someInt.square() // 9
```

32.4. Сабскрипты в расширениях

Помимо свойств, методов и инициализаторов, расширения позволяют создавать новые сабскрипты.

Создаваемое в листинге 32.5 расширение типа `Int` реализует новый сабскрипт, который позволяет получить определенную цифру собственного значения экземпляра. В сабскрипте указывается номер позиции числа, которое необходимо вернуть.

Листинг 32.5

```
extension Int {
    subscript( digitIndex: Int ) -> Int {
        var base = 1
        var index = digitIndex
        while index > 0 {
            base *= 10
            index -= 1
        }
        return (self / base) % 10
    }
}
746381295[0] // 5
746381295[1] // 9
```

Если у числа отсутствует цифра с запрошенным индексом, возвращается `0`, что не нарушает логику работы.

33 Протоколы

В процессе изучения Swift мы уже неоднократно встречались с протоколами, но каждый раз касались их поверхностно, без подробного изучения механизмов взаимодействия с ними.

Протоколы содержат перечень свойств, методов и сабскриптов, которые должны быть реализованы в объектном типе. Другими словами, они содержат требования к наличию определенных элементов внутри типа данных. Протокол сам непосредственно не реализует какой-либо функционал, он лишь является своеобразным набором правил и требований к типу. Любой объектный тип данных может принимать протокол. Наиболее важной функцией протокола является обеспечение целостности объектных типов путем указания требований к их реализации.

Протоколы объявляются независимо от каких-либо элементов программы, так же как и объектные типы данных.

СИНТАКСИС

```
protocol ИмяПротокола {
    // тело протокола
}
```

Для объявления протокола используется ключевое слово `protocol`, после которого указывается имя создаваемого протокола.

Для того чтобы принять протокол к исполнению каким-либо объектным типом, необходимо написать его имя через двоеточие сразу после имени реализуемого типа:

```
struct ИмяПринимающейСтруктуры: ИмяПротокола{
    // тело структуры
}
```

После указания имени протокола при объявлении объектного типа данный тип обязан выполнить все требования протокола. Вы можете указать произвольное количество принимаемых протоколов.

Если класс не только принимает протоколы, но и наследует некоторый класс, то имя суперкласса необходимо указать первым, а за ним через запятую — список протоколов:

```
class ИмяПринимающегоКласса: ИмяСуперКласса, Протокол1, Протокол2{
    // тело класса
}
```

33.1. Требуемые свойства

Протокол может потребовать соответствующий ему тип реализовать свойство экземпляра или свойство типа (`static`), имеющее конкретные имя и тип данных. При этом протокол не указывает на вид свойства (хранимое или вычисляемое). Также могут быть указаны требования к доступности и изменяемости параметра.

Если у свойства присутствует требование доступности и изменяемости, то в качестве данного свойства не могут выступать константа или вычисляемое свойство «только для чтения». Требование доступности обозначается с помощью конструкции `{get}`, а требование доступности и изменяемости — с помощью конструкции `{get set}`.

Создадим протокол, содержащий ряд требований к принимающему его типу (листинг 33.1).

Листинг 33.1

```
1  protocol SomeProtocol {
2      var mustBeSettable: Int { get set }
3      var doesNotNeedToBeSettable: Int { get }
4  }
```

Протокол `SomeProtocol` требует, чтобы в принимающем типе были реализованы два изменяемых (`var`) свойства типа `Int`. При этом свойство `mustBeSettable` должно быть и доступным для чтения, и изменяемым, а свойство `doesNotNeedToBeSettable` — как минимум доступным для чтения.

Протокол определяет минимальные требования к типу, то есть тип данных обязан реализовать все, что описано в протоколе, но он может не ограничиваться этим набором элементов. Так, для свойства `doesNotNeedToBeSettable` из предыдущего листинга может быть реализован не только геттер, но и сеттер.

Для указания в протоколе на свойство типа необходимо использовать модификатор `static` перед ключевым словом `var`. Данное требование

выполняется даже в том случае, если протокол принимается структурой или перечислением (листинг 33.2).

Листинг 33.2

```
protocol AnotherProtocol {
    static var someTypeProperty: Int { get set }
}
```

В данном примере свойство типа `someTypeProperty` должно быть обязательно реализовано в принимающем типе данных.

В следующем примере мы создадим протокол и принимающий его требования класс (листинг 33.3).

Листинг 33.3

```
protocol FullyNamed {
    var fullName: String { get }
}
struct Person: FullyNamed {
    var fullName: String
}
let john = Person(fullName: "John Wick")
```

В данном примере определяется протокол `FullyNamed`, который обязывает структуру `Person` иметь доступное свойство `fullName` типа `String`.

33.2. Требуемые методы

Протокол может требовать реализации определенного метода экземпляра или метода типа. Форма записи для этого подобна указанию требований к реализации свойств.

Для требования реализации метода типа необходимо использовать модификатор `static`. Также протокол может описывать изменяющий метод. Для этого служит модификатор `mutating`.

ПРИМЕЧАНИЕ Если вы указали ключевое слово `mutating` перед требованием метода, то вам уже не нужно указывать его при реализации метода в классе. Данное ключевое слово требуется только в реализации структур.

При реализации метода в типе данных необходимо в точности соблюдать все требования протокола, в частности имя метода, наличие или отсутствие входных аргументов, тип возвращаемого значения, модификаторы (листинг 33.4).

Листинг 33.4

```
protocol RandomNumberGenerator {
    func random() -> Double
    static func getRandomString()
    mutating func changeValue(newValue: Double)
}
```

В данном примере реализован протокол `RandomNumberGenerator`, который содержит требования реализации трех методов. Метод экземпляра `random()` должен возвращать значение типа `Double`. Метод `getRandomString` должен быть методом типа, при этом требования к возвращаемому им значению не указаны. Метод `changeValue(_:)` должен быть изменяющим и должен принимать в качестве входного параметра значение типа `Double`.

Данный протокол не делает никаких предположений относительно того, как будет вычисляться случайное число, ему важен лишь факт выполнения требований.

33.3. Требуемые инициализаторы

Дополнительно протокол может предъявлять требования к реализации инициализаторов. Необходимо писать инициализаторы точно так же, как вы пишете их в объектном типе, опуская фигурные скобки и тело инициализатора.

Требования к инициализаторам могут быть выполнены в соответствующем классе в форме назначенного или вспомогательного инициализатора. В любом случае перед объявлением инициализатора в протоколе необходимо указывать модификатор `required`. Это гарантирует, что вы реализуете указанный инициализатор во всех подклассах данного класса.

ПРИМЕЧАНИЕ Нет нужды обозначать реализацию инициализаторов протокола модификатором `required` в классах, которые имеют модификатор `final`.

Реализуем протокол, содержащий требования к реализации инициализатора, и класс, выполняющий требования протокола (листинг 33.5).

Листинг 33.5

```
protocol Named{
    init(name: String)
}
class Person : Named {
```

```
    var name: String
    required init(name: String){
        self.name = name
    }
}
```

33.4. Протокол в качестве типа данных

Протокол сам по себе не несет какой-либо функциональной нагрузки, он лишь содержит требования к реализации объектных типов. Тем не менее протокол является полноправным типом данных.

Используя протокол в качестве типа данных, вы указываете на то, что записываемое в данное хранилище значение должно иметь тип данных, который соответствует указанному протоколу.

Так как протокол является типом данных, вы можете организовать проверку соответствия протоколу с помощью оператора `is`, который мы обсуждали при изучении темы приведения типов. При проверке соответствия возвращается значение `true`, если проверяемый экземпляр соответствует протоколу, и `false` в противном случае.

33.5. Расширение и протоколы

Расширения могут взаимодействовать не только с объектными типами, но и с протоколами.

Добавление соответствия типа протоколу

Вы можете использовать расширения для добавления требований по соответствию некоторого объектного типа протоколу. Для этого в расширении после имени типа данных через двоеточие необходимо указать список новых протоколов (листинг 33.6).

Листинг 33.6

```
protocol TextRepresentable {
    func asText() -> String
}
extension Int: TextRepresentable {
    func asText() -> String {
        return String(self)
    }
}
5.asText() // "5"
```

В данном примере протокол `TextRepresentable` требует, чтобы в принимающем объектном типе был реализован метод `asText()`. С помощью расширения мы добавляем требование о соответствии типа `Int` к данному протоколу, при этом, поскольку где-то ранее был реализован сам тип данных, в обязательном порядке указывается реализация данного метода.

Расширение протоколов

С помощью расширений мы можем не только указывать на необходимость соответствия новым протоколам, но и расширять сами протоколы, поскольку протоколы являются полноценными типами данных.

При объявлении расширения необходимо использовать имя протокола, а в его теле указывать набор требований с их реализациями. После расширения протокола описанные в нем реализации становятся доступны в экземплярах всех классов, которые приняли данный протокол к исполнению.

Напишем расширение для реализованного ранее протокола `TextRepresentable` (листинг 33.7).

Листинг 33.7

```
extension TextRepresentable {
    func description() -> String {
        return "Данный тип поддерживает протокол TextRepresentable"
    }
}
5.description()
```

Расширение добавляет новый метод в протокол `TextRepresentable`. При этом ранее мы указали, что тип `Int` соответствует данному протоколу. В связи с этим появляется возможность обратиться к указанному методу для любого значения типа `Int`.

33.6. Наследование протоколов

Протокол может наследовать один или более других протоколов. При этом он может добавлять новые требования поверх наследуемых, — тогда тип, принявший протокол к реализации, будет вынужден выполнить требования всех протоколов в структуре. При наследовании протоколов используется тот же синтаксис, что и при наследовании классов.

Работа с наследуемыми протоколами продемонстрирована в листинге 33.8.

Листинг 33.8

```
protocol SuperProtocol {
    var someValue: Int { get }
}
protocol SubProtocol: SuperProtocol {
    func someMethod()
}
struct SomeStruct: SubProtocol{
    let someValue: Int = 10
    func someMethod() {
        // тело метода
    }
}
```

Протокол `SuperProtocol` имеет требования к реализации свойства, при этом он наследуется протоколом `SubProtocol`, который имеет требования к реализации метода. Структура принимает к исполнению требования протокола `SubProtocol`, а значит, в ней должны быть реализованы и свойство, и метод.

33.7. Классовые протоколы

Вы можете ограничить протокол таким образом, чтобы его могли принимать к исполнению исключительно классы (а не структуры и перечисления). Для этого у протокола в списке наследуемых протоколов необходимо указать ключевое слово `class`. Данное слово всегда должно указываться на первом месте в списке наследования.

Пример создания протокола приведен в листинге 33.9. В нем мы изменим протокол `SubProtocol` таким образом, чтобы его мог принять исключительно класс.

Листинг 33.9

```
protocol SubProtocol: class, SuperProtocol {
    func someMethod()
}
```

33.8. Композиция протоколов

Иногда бывает удобно требовать, чтобы тип соответствовал не одному, а нескольким протоколам. В этом случае, конечно же, можно создать

новый протокол, наследовать в него несколько необходимых протоколов и задействовать имя только что созданного протокола. Однако для решения данной задачи лучше воспользоваться *композицией протоколов*, то есть скомбинировать несколько протоколов.

СИНТАКСИС

```
Протокол1 & Протокол2 ...
```

Для композиции протоколов необходимо указать имена входящих в данную композицию протоколов, разделив их оператором & (амперсанд).

В листинге 33.10 приведен пример, в котором два протокола комбинируются в единственное требование.

Листинг 33.10

```
protocol Named {
    var name: String { get }
}
protocol Aged {
    var age: Int { get }
}
struct Person: Named, Aged {
    var name: String
    var age: Int
}
func wishHappyBirthday(celebrator: Named & Aged) {
    print("С Днем рождения, \(celebrator.name)! Тебе уже
                            \(celebrator.age)!")
}
let birthdayPerson = Person(name: "Джон Уик", age: 46)
wishHappyBirthday(celebrator: birthdayPerson)
// выводит "С Днем рождения, Джон Уик! Тебе уже 46!"
```

В данном примере объявляются два протокола: `Named` и `Aged`. Созданная структура принимает оба протокола и в полной мере выполняет их требования.

Входным аргументом функции `wishHappyBirthday(celebrator:)` должно быть значение, которое удовлетворяет обоим протоколам. Таким значением является экземпляр структуры Person, который мы и передаем.

34 Разработка приложения в Xcode Playground

Ранее в книге мы уже создавали несколько простейших приложений на Swift, используя для этого среду разработки Xcode. Как вы уже знаете, playground не предназначен для создания полноценных приложений, но набросать небольшой проект и протестировать его, не выходя из среды playground-проекта, вполне возможно. В этой главе мы рассмотрим пример создания приложения уже в среде Xcode Playground. Также вы познакомитесь с понятием API, разграничением доступа к различным объектам, а также я покажу вам важность работы с документацией.

34.1. Модули

Каждая программа или отдельный проект в Xcode — это модуль.

Модуль — это единый функциональный блок, выполняющий определенные задачи. Модули могут взаимодействовать между собой. Каждый внешний модуль, который вы используете в своей программе, для вас «черный ящик». Вам недоступна его внутренняя реализация — вы знаете лишь то, какие функции данный модуль позволяет выполнить (то есть что дать ему на вход и что получите на выходе). Модули состоят из исходных файлов и ресурсов.

Исходный файл — отдельный файл, содержащий программный код и разрабатываемый в пределах одного модуля.

Для того чтобы из набора файлов получить модуль, необходимо провести компиляцию, то есть из кода, понятного разработчику и среде программирования, получается файл (модуль), понятный компьютеру. Модуль может быть собран как из одного, так и из множества исходных файлов и ресурсов. В качестве модуля может выступать, например, целая программа или фреймворк.

С фреймворками (или библиотеками) мы встречались ранее: вспомните про Foundation и UIKit, которые мы подгружали в программу с помощью директивы `import`. Данная директива служит для обеспечения доступа к функционалу внешней библиотеки. Для доступа к ее функциям чаще всего существует специальный набор правил, то есть интерфейс, называемый API.

API (application programming interface) — это набор механизмов (обычно функций и типов данных), включенных в состав некоторой библиотеки (модуля). API доступны при условии, что содержащая их библиотека подключена к проекту (импортирована в проект).

Если вернуться к фреймворку `Foundation`, то функция `arc4random_uniform()`, которую мы использовали ранее, является одной из большого перечня доступных API-функций. Пример подключения фреймворка к библиотеке приведен в листинге 34.1.

Листинг 34.1

```
import Foundation
```

При разработке приложений вы будете использовать большое количество различных библиотек, которые, в том числе, поставляются вместе с Xcode. Самое интересное, что Xcode содержит просто гигантское количество возможностей: работа с 2D и 3D, различные визуальные элементы, физические законы и многое-многое другое. И все это реализуется благодаря дополнительным библиотекам. В этой главе вы познакомитесь с некоторыми их возможностями.

ПРИМЕЧАНИЕ Запомните, что одни библиотеки могут подключать другие. Так, например, UIKit подгружает в проект Foundation, и дополнительное подключение Foundation не требуется.

34.2. Разграничение доступа

Уровни доступа

База системы разграничения доступа при разработке приложений строится на основе понятия «модуль».

Всего Swift предлагает пять различных уровней доступа для объектов вашего кода:

open и public

Открытый и публичный. Данные уровни доступа очень похожи, они открывают полную свободу использования объекта. Вы можете

импортировать модуль и свободно использовать его public-объекты в своем коде. Разница между `open` и `public` описывается далее.

internal

Внутренний. Данный уровень используется в случаях, когда необходимо ограничить использование объекта самим модулем. Таким образом, объект будет доступен во всех исходных файлах модуля, исключая его использование за пределами модуля.

fileprivate

Частный в пределах файла. Данный уровень позволяет использовать объект только в пределах данного исходного файла.

private

Частный. Данный уровень позволяет использовать объект только в пределах конструкции, в которой он объявлен. Например, объявленный в классе параметр не будет доступен в его расширениях.

Открытый уровень доступа является самым высоким и наименее ограничивающим, а частный — самым низким и максимально ограничивающим.

Открытый уровень применяется только к классам и членам класса (свойствам, методам и т. д.) и отличается от публичного следующими характеристиками:

- Класс, имеющий уровень доступа `public` (или более строгий), может иметь подклассы только в том модуле, где он был объявлен.
- Члены класса, имеющие уровень доступа `public` (или более строгий), могут быть переопределены (с помощью оператора `override`) в подклассе только в том модуле, где он объявлен.
- Класс, имеющий уровень доступа `open`, может иметь подклассы внутри модуля, где он определен, и в модулях, импортированных в данном модуле.
- Члены класса, имеющие уровень доступа `open`, могут быть переопределены (с помощью оператора `override`) в подклассе в том модуле, где он объявлен, а также в модулях, импортируемых в данном модуле.

Главное правило определения уровня доступа в Swift звучит следующим образом: объект с более низким уровнем доступа не может определить объект с более высоким уровнем доступа (в контексте модуля).

По умолчанию все объекты вашего кода имеют уровень доступа `internal`. Для того чтобы изменить его, необходимо явно указать уровень. При этом если вы разрабатываете фреймворк, то для того, чтобы сделать некоторый объект частью доступного API, вам необходимо изменить его уровень доступа на `public`.

Синтаксис определения

Чтобы определить уровень доступа к некоторому объекту, необходимо указать соответствующее ключевое слово (`open`, `public`, `internal`, `private`) перед определением объекта (`func`, `property`, `class`, `struct` и т. д.). Пример приведен в листинге 34.2.

Листинг 34.2

```
open class SomePublicClass {}
internal class SomeInternalClass {}
fileprivate class SomePrivateClass {}
public var somePublicVar = 0
private var somePrivatelet = 0
internal func someInternalFunc() {}
```

ПРИМЕЧАНИЕ Если уровень доступа к вашему объекту предполагается `internal`, то можно его не указывать, так как по умолчанию для любого объекта назначен именно этот уровень.

Уровень доступа к типам данных

Следует подробнее остановиться на определении уровня доступа различных типов объектов.

Как мы уже неоднократно говорили, Swift позволяет определять собственные типы данных. Если вам требуется указать уровень доступа к типу данных или его членам, то это необходимо сделать в момент определения типа. Новый тип данных может быть использован там, где это позволяет его уровень доступа.

Если ваш объект имеет вложенные объекты (например, класс со свойствами и методами), то уровень доступа родителя определяет уровни доступа к его членам. Таким образом, если вы укажете уровень доступа `private`, то все его члены по умолчанию будут иметь уровень доступа `private`. Для уровней доступа `public` и `internal` уровень доступа членов — `internal`.

Рассмотрим пример из листинга 34.3.

Листинг 34.3

```
public class SomePublicClass { // public класс
    public var somePublicProperty = 0 // public свойство
    var someInternalProperty = 0 // internal свойство
    fileprivate func somePrivateMethod() {} // fileprivate метод
}
class SomeInternalClass { // internal класс
    var someInternalProperty = 0 // internal свойство
    private func somePrivatemethod() {} // private метод
}

private class  SomePrivateClass { // private класс
    var  somePrivateProperty = 0 // private свойство
    func somePrivateMethod() {} // private метод
}
```

Всего было определено три класса с различными уровнями доступа. Из примера видно, как влияет определение уровня доступа к классу на доступ к его членам.

Обратите внимание, что при наследовании уровень доступа подкласса не может быть выше уровня родительского класса.

Уровень доступа к кортежу типа данных определяется наиболее строгим типом данных, включенным в кортеж. Так, например, если вы скомпонуете кортеж типа из двух разных типов, один из которых будет иметь уровень доступа `internal`, а другой — `private`, то результирующим уровнем доступа будет `private`, то есть самый строгий.

ПРИМЕЧАНИЕ Запомните, что Swift не позволяет явно указать тип данных кортежа. Он вычисляется автоматически.

Уровень доступа к функции определяется самым строгим уровнем типов аргументов функции и типа возвращаемого значения. Рассмотрим пример из листинга 34.4.

Листинг 34.4

```
func someFunction() -> (SomeInternalClass, SomePrivateClass) {
   // тело функции
}
```

Можно было ожидать, что уровень доступа функции будет равен `internal`, так как не указан явно. На самом деле эта функция вообще не будет скомпилирована. Это связано с тем, что тип возвращаемого значения — это кортеж с уровнем доступа `private`. При этом тип этого кортежа определяется автоматически на основе типов данных, входящих в него.

В связи с тем, что уровень доступа функции — `private`, его необходимо указать явно (листинг 34.5).

Листинг 34.5

```
private func someFunction() -> (SomeInternalClass, SomePrivateClass) {
   // тело функции
}
```

Что касается перечислений, стоит обратить внимание на то, что каждый член перечисления получает тот же уровень доступа, что установлен для самого перечисления.

34.3. Разработка интерактивного приложения

Что может быть интереснее, чем интерактивное приложение, в котором можно двигать, щелкать, толкать и т. д.? Разработкой именно такого приложения мы и займемся.

Для реализации возьмем следующую задачу: в квадратном ящике расположены несколько цветных шариков. Пользователь может перемещать шарики и сталкивать их друг с другом.

ЗАДАНИЕ

Сейчас вам необходимо создать новый playground-проект типа `Blank`. Назовите его `"Balls"` и удалите весь программный код, присутствующий в нем.

Interactive Playgrounds, UIKit и PlaygroundSupport

Xcode обладает потрясающим механизмом, который называется Interactive playgrounds. С его помощью вы можете нажимать, перемещать и совершать другие действия с объектами прямо в окне playground-проекта. Interactive playgrounds помогут вам быстро создать прототип приложения, тем самым уменьшив вероятность неприятных ошибок в релизной версии программы.

Interactive playgrounds наделяют вас практически теми же возможностями по построению интерфейсов, что и при полноценной разработке приложений. Именно этот механизм мы будем использовать при разработке вашего первого приложения.

В Xcode присутствует модуль, предназначенный для работы с Interactive Playground. Он называется `PlaygroundSupport`.

ПРИМЕЧАНИЕ `PlaygroundSupport` — модуль, обеспечивающий взаимодействие пользователя с Xcode при работе в playground.

Наиболее важным механизмом данной библиотеки является класс PlaygroundPage, предназначенный для создания интерактивного содержимого страниц в playground-проекте.

`PlaygroundSupport` позволяет взаимодействовать с некоторыми объектами. Поэтому нам потребуется функционал, обеспечивающий графическое построение этих объектов. И таким функционалом обладает модуль `UIKit`.

`UIKit` — это библиотека, обеспечивающая ключевую инфраструктуру, необходимую для построения iOS-приложений. `UIKit` содержит огромное количество классов, позволяющих строить визуальные элементы, анимировать их, обрабатывать физические законы, управлять механизмами печати и обработки текста и многое-многое другое! Это важнейшая и совершенно незаменимая библиотека функций, которую вы будете использовать при разработке каждого приложения.

Импортируем в проект `Balls` библиотеки `PlaygroundSupport` и `UIKit` (листинг 34.6).

Листинг 34.6

```
import PlaygroundSupport
import UIKit
```

Разделим функциональную нагрузку разрабатываемой программы между исходными файлами. В панели Project Navigator в папке Source создайте новый файл с именем Balls.swift. Для этого щелкните по названию папки правой кнопкой мыши и выберите пункт New File. В результате новый файл отобразится в составе папки Sources. Вам останется лишь указать его имя (рис. 34.1). Всю функциональную начинку мы разместим в этом файле, в то время как главный файл Balls будет использовать реализованный функционал.

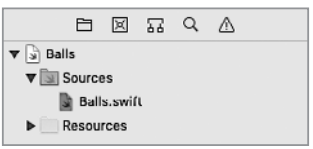

Рис. 34.1. Новый файл в Project Navigator

В только что созданном файле также импортируйте библиотеку `UIKit`.

Класс Balls

Весь функционал будет заключен в одном-единственном классе, который мы назовем `Balls` и расположим в файле Balls.swift. Определите новый класс `Balls`, как это сделано в листинге 34.7.

Листинг 34.7

```
import UIKit
public class Balls: UIView {}
```

В качестве уровня доступа класса `Balls` указан `public`. Это связано с тем, что некоторые свойства и методы класса `UIView`, которые будут переопределены в `Balls`, также соответствуют `public`, а как вы знаете, уровень доступа свойств и методов не может быть выше, чем уровень самого типа.

В состав подключенной библиотеки `UIKit` входит тип данных `UIView`, предназначенный для визуального отображения графических элементов и взаимодействия с ними. Например, мы можем отобразить прямоугольную рабочую область типа `UIView`, наполнив ее другими графическими элементами, каждый из которых также будет представлять собой отдельный экземпляр типа `UIView`. В этом и будет заключаться решение поставленной нами задачи с шариками (рис. 34.2).

В связи с этим реализуемый класс `Balls` является подклассом для `UIView`.

У разрабатываемой системы можно выделить следующие свойства.

1. Существует прямоугольная область.
2. Внутри данной области расположены несколько шариков.
3. Каждый шарик имеет свой цвет.
4. Шарики могут перемещаться.
5. Шарики могут взаимодействовать друг с другом (ударяться).
6. Шарики могут взаимодействовать с границами прямоугольной области.

Благодаря пункту 3 вы получите очень важный опыт взаимодействия с типом данных `UIColor`, который позволяет работать с цветовой палитрой.

Мы будем определять количество шариков, передавая массив цветов. Каждый из переданных цветов будет указывать на один шарик. Объявим два свойства, содержащих информацию о цветах и шариках (листинг 34.8).

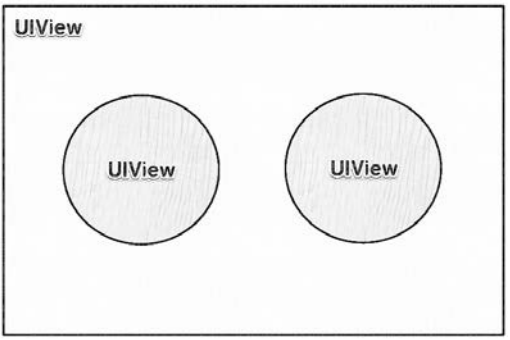

Рис. 34.2. Структура отображений

Листинг 34.8

```
public class Balls: UIView {
    // список цветов для шариков
    private var colors: [UIColor]
    // шарики
    private var balls: [UIView] = []
}
```

Свойство `colors` — это массив значений типа `UIColor`. Сами шарики, как мы говорили выше, представляют собой экземпляры типа `UIView`, наложенные на отображение экземпляра типа `Balls`, который является наследником типа `UIView`. Мы не реализуем дополнительный тип данных для шариков, потому что функционала `UIView` вполне достаточно для решения поставленной задачи.

Для использования разрабатываемого класса нам потребуется реализовать инициализатор (листинг 34.9).

Листинг 34.9

```
//инициализатор класса
public init(colors: [UIColor]){
    self.colors = colors
    super.init(frame: CGRect(x: 0, y: 0, width: 400, height: 400))
    backgroundColor = UIColor.gray
}
```

Инициализатор `init(colors:)` в качестве входного параметра получает массив значений типа `UIColor`. В результате количество и порядок шариков будут зависеть именно от массива `colors`.

Класс `UIView` имеет встроенный инициализатор `init(frame:)`, который позволяет определить характеристики создаваемого графического элемента. При его вызове задаются значения для построения прямоугольной основы, на которую будут накладываться шарики. Для создания прямоугольников служит специальный класс `CGRect`, которому в качестве параметров передаются координаты левого верхнего угла и значения длин сторон.

Свойство `backgroundColor` определяет цвет создаваемого объекта, оно наследуется от класса `UIView`. Класс `UIColor` позволяет нам создать практически любой требуемый цвет. Для этого существует большое количество методов. В данном случае мы используем свойство `gray`, которое определяет серый цвет. При необходимости создания произвольного цвета вы можете использовать соответствующие методы (в этом вам поможет окно автодополнения).

Обратите внимание, что при попытке запустить проект Xcode отобразит ошибку, сообщающую об отсутствии инициализатора `init(coder:)` (рис. 34.3).

```
1  import UIKit
2  public class Balls: UIView {
3      // список цветов для шариков
4      private var colors: [UIColor]
5      // шарики
6      private var balls: [UIView] = []
7      //инициализатор класса
8      public init(colors: [UIColor]){
9          self.colors = colors
10         super.init(frame: CGRect(x: 0, y: 0, width: 400, height: 400))
11         backgroundColor = UIColor.gray
12     }
13 }                    'required' initializer 'init(coder:)' must be provided by subclass of 'UIView'
14
```

Рис. 34.3. Ошибка, сообщающая об отсутствии инициализатора init(coder:)

Вы можете написать код инициализатора самостоятельно (листинг 34.10) или позволить Xcode исправить ошибку.

Листинг 34.10

```
required public init?(coder aDecoder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
}
```

Теперь перейдем к основному файлу проекта Balls в Project Navigator и создадим экземпляр класса `Balls` (листинг 34.11).

Листинг 34.11

```
let balls = Balls(colors: [UIColor.white])
```

В качестве входного параметра инициализатора мы передаем массив объектов типа `UIColor`, содержащий всего один элемент, который определяет белый цвет. В результате в константу `balls` будет записан экземпляр класса `Balls`, описывающий всю разрабатываемую систему целиком.

ПРИМЕЧАНИЕ Xcode позволяет преобразовать некоторые ресурсы из текстового вида в графический. Так, например, описание цвета может быть представлено в виде визуальной палитры цветов прямо в редакторе кода!

Для этого требуется переписать объявление в виде специального литерала:

```
#colorLiteral(red: 0, green: 0, blue: 0, alpha: 1)
```

После его написания данный код будет преобразован к графическому квадрату черного цвета. Вы можете вставить как элемент массива `colors` вместо `UIColor.white` и скопировать необходимое количество раз (рис. 34.4).

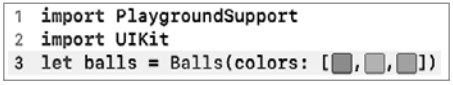

Рис. 34.4. Визуальное отображение ресурсов

Для выбора нового цвета вам необходимо щелкнуть по квадрату и выбрать подходящий из появившейся палитры. Это еще одна из поразительных возможностей среды разработки Xcode!

Обратите внимание на то, что такой подход возможен только в файлах, описывающих страницы playground. В Balls.swift вся информация отображается исключительно в текстовом виде.

Добавьте еще три произвольных цвета в массив `colors`, при этом не забудьте разделить их запятыми.

Теперь наш проект готов для предварительного отображения написанного кода. Для этого добавьте следующую строку в файл Balls (листинг 34.12).

Листинг 34.12

```
PlaygroundPage.current.liveView = balls
```

В данном листинге используется функционал подключенной ранее библиотеки `PlaygroundSupport` — класс `PlaygroundPage`, который позволяет вывести графические элементы на странице `playground`.

Теперь нажмите кнопку Assistant Editor, и в правой части Xcode отобразится вывод экземпляра класса `Balls` (рис. 34.5). В данный момент отображается только прямоугольник — он будет подложкой для отображения набора шариков.

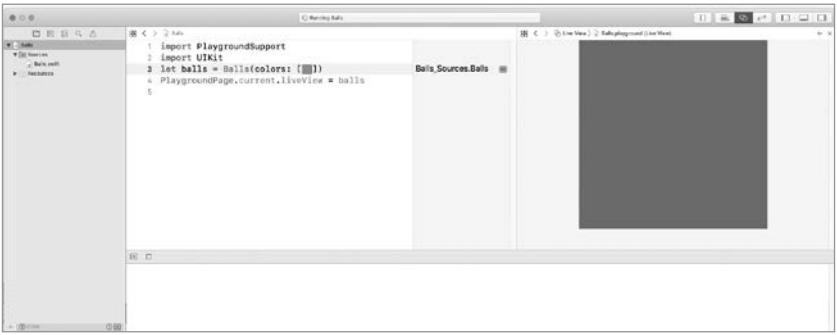

Рис. 34.5. Работа в режиме Assistant Editor

Вернемся к файлу Balls.swift.

Xcode имеет довольно внушительное количество типов данных, — как говорится, на все случаи жизни. Для указания размера шариков мы будем использовать тип данных `CGSize`. Создадим свойство класса `Balls`, описывающее высоту и ширину шарика (листинг 34.13).

Листинг 34.13

```
// размер шариков
private var ballSize: CGSize = CGSize(width: 40, height: 40)
```

В качестве аргументов инициализатор класса `CGSize` получает параметры `width` и `height`, определяющие ширину и высоту.

Теперь напишем метод, отвечающий за отображение шариков (листинг 34.14).

Листинг 34.14

```
func ballsView () {
    /* производим перебор переданных цветов
       именно они определяют количество шариков */
    for (index, color) in colors.enumerated() {
        /* шарик представляет собой
           экземпляр класса UIView */
        let ball = UIView(frame: CGRect.zero)
        /* указываем цвет шарика
```

```
            он соответствует переданному цвету */
          ball.backgroundColor = color
          // накладываем отображение шарика на отображение подложки
          addSubview(ball)
          // добавляем экземпляр шарика в массив шариков
          balls.append(ball)
          /* вычисляем отступ шарика по осям X и Y, каждый
             последующий шарик должен быть правее и ниже
           предыдущего */
          let origin = 40*index + 100
          // отображение шарика в виде прямоугольника
          ball.frame = CGRect(x: origin, y: origin,
         width: Int(ballSize.width), height: Int(ballSize.height))
          // с закругленными углами
          ball.layer.cornerRadius = ball.bounds.width / 2.0
    }
}
```

Для обработки шариков мы используем свойство `colors`, которое хранит в себе массив переданных цветов. Метод `enumerated()` уже знаком нам — он позволяет перебрать все элементы массива, получая индекс и значение каждого элемента.

Как уже неоднократно говорилось, шарики, как и вся система целиком, это отображения. Но если подложка — это экземпляр потомка класса `UIView`, то каждый шарик — экземпляр самого `UIView`. Изначально в качестве отображения мы используем конструкцию `CGRect.zero`, которая соответствует `CGRect(x: 0, y: 0, width: 0, height:0)`, то есть прямоугольнику с нулевыми размерами.

Для того чтобы подложка и шарики выводились совместно, необходимо использовать функцию `addSubview(_:)`, которая накладывает одно отображение на другое.

Для того чтобы шарики обрисовывались так же, как и их подложка, необходимо добавить вызов метода `ballsVew()` в инициализатор `init(colors:)` (листинг 34.15).

Листинг 34.15

```
//инициализатор класса
public init(colors: [UIColor]){
    self.colors = colors
    super.init(frame: CGRect(x: 0, y: 0, width: 400, height: 400))
    backgroundColor = UIColor.gray
    // вызов функции отрисовки шариков
    ballsView()
}
```

Вернитесь к файлу Balls. В режиме Assistant Editor можно увидеть, что кроме серой подложки отрисовываются четыре разноцветных шарика. В данный момент шарики статичны: вы не сможете с ними взаимодействовать.

Теперь займемся интерактивностью. Нам необходимо реализовать возможность перемещения шариков указателем мыши, их столкновения между собой и с бортами подложки.

Для анимации движения используется класс-аниматор `UIDynamicAnimator`. Он позволяет отображать обрабатываемые другими классами события, например перемещения.

Создадим новое свойство класса `Balls` (листинг 34.16).

Листинг 34.16

```
// аниматор графических объектов
private var animator: UIDynamicAnimator?
```

Обратите внимание, что аниматор — это опционал. Это связано с тем, что данное свойство не будет иметь какого-либо значения к моменту вызова родительского инициализатора `super.init(frame:)`.

Теперь необходимо подключить аниматор к отображению. Для этого добавим в инициализатор соответствующий код (листинг 34.17).

Листинг 34.17

```
public init(colors: [UIColor]){
    self.colors = colors
    super.init(frame: CGRect(x: 0, y: 0, width: 400, height: 400))
    backgroundColor = UIColor.blueColor()
    // подключаем аниматор с указанием на сам класс
    animator = UIDynamicAnimator(referenceView: self)
    ballsView()
}
```

Сам по себе аниматор не производит каких-либо действий. Для того чтобы он отображал некоторое изменение состояния объектов, необходимо использовать ряд дополнительных классов и связать каждый из этих классов с аниматором.

Для взаимодействия пользователя с графическими элементами `UIKit` предоставляет нам специальный класс `UISnapBehavior`.

ПРИМЕЧАНИЕ Каждый тип данных, в названии которого присутствует `Behavior`, предназначен для обработки некоторого поведения. Так, `UISnapBehavior` обрабатывает поведение при перемещении объектов от точки к точке.

Определим новое свойство типа `UISnapBehavior` (листинг 34.18).

Листинг 34.18

```
// обработчик перемещений объектов
private var snapBehavior: UISnapBehavior?
```

Класс `UISnapBehavior` позволяет обрабатывать касания экрана устройства (или щелчки мышки). Для этого в состав `UISnapBehavior` включены три метода:

`touchesBegan(_:with:)`

Метод вызывается в момент касания экрана.

`touchesMoved(_:with:)`

Метод срабатывает при каждом перемещении пальца, уже коснувшегося экрана.

`touchesEnded(_:with:)`

Метод вызывается по окончании взаимодействия с экраном (когда палец убран).

Все методы уже определены в `UISnapBehavior`, поэтому при создании собственной реализации этих методов необходимо их переопределять, то есть использовать ключевое слово `override`.

Реализуем метод `touchesBegan(_:with:)` (листинг 34.19).

Листинг 34.19

```
override public func touchesBegan(_ touches: Set<UITouch>, with event:
UIEvent?) {
    if let touch = touches.first {
        let touchLocation = touch.location( in: self )
        for ball in balls {
            if (ball.frame.contains( touchLocation )) {
                snapBehavior = UISnapBehavior(item: ball,snapTo:
touchLocation)
                snapBehavior?.damping = 0.5
                animator?.addBehavior(snapBehavior!)
            }
        }
    }
}
```

Коллекция `touches` содержит данные обо всех касаниях. Это связано с тем, что тач-панель современных устройств поддерживает мульти-

тач, то есть одновременное касание несколькими пальцами. В начале метода извлекаются данные о первом элементе набора `touches` и помещаются в константу `touch`.

Константа `touchLocation` содержит координаты касания относительно всего отображения.

С помощью метода `ball.frame.contains()` мы определяем, входят ли координаты касания в какой-либо из шариков. Если находится соответствие, то в свойство `snapBehavior` записываются данные об объекте, с которым происходит взаимодействие, и о координатах, куда данный объект должен быть перемещен.

Свойство `damping` определяет плавность и затухание при движении шарика.

Далее, используя метод `addBehavior(_:)` аниматора, указываем, что обрабатываемое классом `UISnapBehavior` поведение объекта должно быть анимировано. Таким образом, все изменения состояния объекта, производимые в свойстве `snapBehavior`, будут анимированы.

После обработки касания необходимо обработать перемещение пальца. Для этого реализуем метод `touchesMoved(_:with:)` (листинг 34.20).

Листинг 34.20

```
override public func touchesMoved(_ touches: Set<UITouch>, with event:
UIEvent?) {
    if let touch = touches.first {
        let touchLocation = touch.location( in: self )
        if let snapBehavior = snapBehavior {
            snapBehavior.snapPoint = touchLocation
        }
    }
}
```

Так как в свойстве `snapBehavior` уже содержится указание на определенный шарик, с которым происходит взаимодействие, нет необходимости проходить по всему массиву шариков снова. Единственной задачей данного метода является изменение свойства `snapPoint`, которое указывает на координаты объекта.

Для завершения обработки перемещения объектов касанием необходимо переопределить метод `touchesEnded(_:with:)` (листинг 34.21).

Листинг 34.21

```
public override func touchesEnded(_ touches: Set<UITouch>,
                                  with event: UIEvent?) {
    if let snapBehavior = snapBehavior {
```

```
            animator?.removeBehavior(snapBehavior)
        }
        snapBehavior = nil
    }
```

Данный метод служит для решения одной очень важной задачи — очистки используемых ресурсов. После того как взаимодействие с шариком окончено, нет необходимости хранить информацию об обработчике поведения в `snapBehavior`.

ПРИМЕЧАНИЕ Возьмите в привычку удалять ресурсы, пользоваться которыми вы уже не будете. Это сэкономит изрядное количество памяти и уменьшит вероятность возникновения ошибок.

Перейдите в файл Balls, и в окне Assistant Editor вы увидите изображение четырех шариков, но, в отличие от предыдущего раза, сейчас вы можете указателем мыши перемещать любой из шариков (рис. 34.6)!

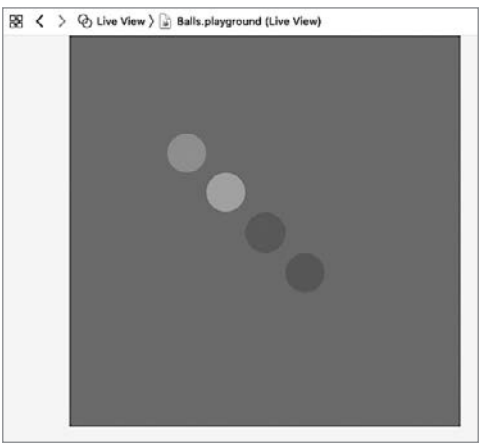

Рис. 34.6. Шарики, расположенные внутри подложки

Хотя шарики и могут перемещаться, они не взаимодействуют между собой и с границами подложки. Для обработки поведения при столкновениях служит класс `UICollisionBehavior`. Создадим новое свойство (листинг 34.22).

Листинг 34.22

```
// обработчик столкновений
private var collisionBehavior: UICollisionBehavior
```

Следующим шагом будет редактирование инициализатора (листинг 34.23).

Листинг 34.23

```
public init(colors: [UIColor]){
    self.colors = colors
    // создание значения свойства
    collisionBehavior = UICollisionBehavior(items: [])
    /* указание на то, что границы отображения
      также являются объектами взаимодействия */
    collisionBehavior.setTranslatesReferenceBoundsIntoBoundary( with:
                    UIEdgeInsets(top: 1, left: 1, bottom: 1, right: 1))
    super.init(frame: CGRect(x: 0, y: 0, width: 400, height: 400))
    backgroundColor = UIColor.gray
    // подключаем аниматор с указанием на сам класс
    animator = UIDynamicAnimator(referenceView: self)
    // вызов функции отрисовки шариков
    ballsView()
}
```

Одним из свойств класса `UICollisionBehavior` является `items`. Оно указывает на набор объектов, которые могут взаимодействовать между собой. В момент работы инициализатора шарики еще не созданы, поэтому в качестве входного параметра при создании объекта типа `UICollisionBehavior` мы указываем пустой массив.

Для того чтобы обеспечить взаимодействие шариков не только друг с другом, но и с границами подложки, мы используем метод `setTrans latesReferenceBoundsIntoBoundary(with:)`. Он устанавливает границы коллизий с внутренним отступом в 1 пиксел с каждой стороны подложки.

В самом конце необходимо добавить обработчик поведения при коллизиях аниматору `animator`. Делается это с помощью уже известного нам метода `addBehavior(_:)`.

Теперь вам потребуется добавить каждый шарик в обработчик коллизий. Для этого существует метод `addItem(_:)`. Добавим соответствующую строку в метод `ballsView()` (листинг 34.24).

Листинг 34.24

```
func ballsView () {
    /* производим перебор переданных цветов
     именно они определяют количество шариков */
    for (index, color) in colors.enumerated() {
        /* шарик представляет собой
         экземпляр класса UIView */
```

```
        let ball = UIView(frame: CGRect.zero)
        /* указываем цвет шарика
         он соответствует переданному цвету */
        ball.backgroundColor = color
        // накладываем отображение шарика на отображение подложки
        addSubview(ball)
        // добавляем экземпляр шарика в массив шариков
        balls.append(ball)
        /* вычисляем отступ шарика по осям X и Y, каждый
         последующий шарик должен быть правее и ниже
         предыдущего */
        let origin = 40*index + 100
        // отображение шарика в виде прямоугольника
        ball.frame = CGRect(x: origin, y: origin,
                            width: Int(ballSize.width), height:
                            Int(ballSize.height))
        // с закругленными углами
        ball.layer.cornerRadius = ball.bounds.width / 2.0
        // добавим шарик в обработчик столкновений
        collisionBehavior.addItem(ball)
    }
}
```

Поздравляю вас! Это была последняя строчка кода, которую необходимо было написать для решения поставленной задачи. Сейчас вы можете перейти к файлу Balls и в Assistant Editor протестировать созданный прототип.

Как вы можете видеть, шарики невозможно переместить за границы, а при попытке столкнуть их они разлетаются.

`UIKit` и `Foundation` предоставляет еще огромное количество возможностей помимо рассмотренных. Так, например, можно создать гравитационное, магнитное и любое другое взаимодействие элементов, смоделировать силу тяжести или турбулентность, установить параметры скорости и ускорения. Все это и многое-многое другое позволяет вам создавать поистине функциональные приложения, которые обязательно найдут отклик у пользователей.

35 Универсальные шаблоны

Универсальные шаблоны (generic) являются одним из мощнейших инструментов Swift. На их основе написано большинство библиотек. Даже если вы никогда специально не использовали универсальные шаблоны, на самом деле вы взаимодействовали с ними практически в каждой написанной программе.

Универсальные шаблоны позволяют создавать гибкие конструкции (функции, объектные типы) без привязки к конкретному типу данных. Вы лишь описываете требования и функциональные возможности, а Swift самостоятельно определяет, каким типам данных доступен разработанный функционал. Примером может служить тип данных `Array` (массив). Элементами массива могут выступать значения произвольных типов данных, и для этого разработчикам не требуется создавать отдельные типы массивов: `Array<Int>`, `Array<String>` и т. д. Для реализации коллекции использован универсальный шаблон, позволяющий при необходимости указать требования к типу данных. Так, например, в реализации типа `Dictonary` существует требование, чтобы тип данных ключа соответствовал протоколу `Hashable` (его предназначение мы обсуждали ранее).

35.1. Универсальные функции

Разработаем функцию, с помощью которой можно поменять значения двух целочисленных переменных (листинг 35.1).

Листинг 35.1

```
func swapTwoInts( a: inout Int,  b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

```
var firstInt = 4010
var secondInt = 13
swapTwoInts(a: &firstInt, b: &secondInt)
firstInt // 13
secondInt // 4010
```

Функция `swapTwoInts(a:b:)` использует сквозные параметры, чтобы обеспечить доступ непосредственно к параметрам, передаваемым в функцию, а не к их копиям. В результате выполнения значения в переменных `firstInt` и `secondInt` меняются местами.

Данная функция является крайне полезной, но очень ограниченной в своих возможностях. Для того чтобы поменять значения двух переменных других типов, придется писать отдельную функцию: `swapTwoStrings()`, `swapTwoDoubles()` и т. д. Если обратить внимание на то, что тела всех функций должны быть практически одинаковыми, то мы просто-напросто займемся дублированием кода, хотя ранее в книге неоднократно рекомендовалось всеми способами этого избегать.

Для решения задачи было бы намного удобнее использовать универсальную функцию, позволяющую передать в качестве аргумента значения любого типа с одним лишь требованием: типы данных обоих аргументов должны быть одинаковыми.

Универсальные функции объявляются точно так же, как и стандартные, за одним исключением: после имени функции в угловых скобках указывается заполнитель имени типа, то есть литерал, который далее в функции будет указывать на тип данных переданного аргумента.

Преобразуем функцию `swapTwoInts(a:b:)` к универсальному виду (листинг 35.2).

Листинг 35.2

```
func swapTwoValues<T>( a: inout T,  b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
var firstString = "one"
var secondString = "two"
swapTwoValues(a: &firstString, b: &secondString)
firstString // "two"
secondString // "one"
```

В универсальной функции заполнителем типа является литерал `T`, который и позволяет задать тип данных в списке входных аргументов

вместо конкретного типа (`Int`, `String` и т. д.). При этом определяется, что `a` и `b` должны быть одного и того же типа данных.

Функция `swapTwoValues(a:b:)` может вызываться точно так же, как и определенная ранее функция `swapTwoInts(a:b:)`.

Используемый заполнитель называется параметром типа. Как только вы его определили, можете применять его для указания типа любого параметра или значения, включая возвращаемое функцией значение. При необходимости можно задать несколько параметров типа, вписав их в угловых скобках через запятую.

35.2. Универсальные типы

В дополнение к универсальным функциям универсальные шаблоны позволяют создать универсальные типы данных. К универсальным типам относятся, например, упомянутые ранее массивы и словари.

Создадим универсальный тип данных `Stack` (стек) — упорядоченную коллекцию элементов, подобную массиву, но со строгим набором доступных операций:

- метод `push(_:)` служит для добавления элемента в конец коллекции;
- метод `pop()` служит для возвращения элемента из конца коллекции с удалением его оттуда.

Никаких иных доступных операций для взаимодействия со своими элементами стек не поддерживает.

В первую очередь создадим неуниверсальную версию данного типа (листинг 35.3).

Листинг 35.3

```
struct IntStack {
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

Данный тип обеспечивает работу исключительно со значениями типа `Int`. В качестве хранилища элементов используется массив `[Int]`.

Сам тип для взаимодействия с элементами коллекции предоставляет нам оба описанных ранее метода.

Недостатком созданного типа является его ограниченность в отношении типа используемого значения. Реализуем универсальную версию типа, позволяющую работать с любыми однотипными элементами (листинг 35.4).

Листинг 35.4

```
struct Stack<T> {
    var items = [T]()
    mutating func push(_ item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
}
```

Универсальная версия отличается от неуниверсальной только тем, что вместо указания конкретного типа данных задается заполнитель имени типа.

Создавая новую коллекцию типа `Stack`, в угловых скобках необходимо указать тип данных, после чего можно использовать описанные методы для модификации хранилища (листинг 35.5).

Листинг 35.5

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
let fromTheTop = stackOfStrings.pop() // "dos"
```

В коллекцию типа `Stack<String>` были добавлены два элемента и удален один.

Мы можем доработать описанный тип данных таким образом, чтобы при создании хранилища не было необходимости указывать тип элементов стека (листинг 35.6). Для реализации этой задачи опишем инициализатор, принимающий в качестве входного аргумента массив значений.

Листинг 35.6

```
struct Stack<T> {
    var items = [T]()
    init(){}
    init(_ elements: T...){
```

```
        self.items = elements
    }
    mutating func push(_ item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
}
var stackOfInt = Stack(1, 2, 4, 5)
type(of:stackOfInt) // Stack<Int>.Type
var stackOfStrings = Stack<String>()
type(of:stackOfStrings) // Stack<String>.Type
```

Так как мы объявили собственный инициализатор, принимающий входной параметр, для сохранения функциональности пришлось описать также пустой инициализатор.

Теперь мы можем не создавать стек без указания типа элементов, а просто передать значения в качестве входного аргумента в инициализатор типа.

35.3. Ограничения типа

Иногда бывает полезно указать определенные ограничения, накладываемые на типы данных универсального шаблона. В качестве примера мы уже рассматривали тип данных `Dictionary`, где для ключа существует требование: тип данных должен соответствовать протоколу `Hashable`.

Универсальные шаблоны позволяют накладывать определенные требования и ограничения на тип данных значения. Вы можете указать список типов, которым должен соответствовать тип значения. Если элементом этого списка является протокол (который также является типом данных), то проверяется соответствие типа значения данному протоколу; если типом является класс, структура или перечисления, то проверяется, соответствует ли тип значения данному типу.

Для определения ограничений необходимо передать перечень имен типов через двоеточие после заполнителя имени типа.

Реализуем функцию, производящую поиск элемента в массиве и возвращающую его индекс (листинг 35.7).

ПРИМЕЧАНИЕ Для обеспечения функционала сравнения двух значений в Swift существует специальный протокол `Equatable`. Он обязывает поддерживающий его

тип данных реализовать функционал сравнения двух значений с помощью операторов равенства (==) и неравенства (!=). Другими словами, если тип данных поддерживает этот протокол, то его значения можно сравнивать между собой.

Листинг 35.7

```
func findIndex<T: Equatable>(array: [T], valueToFind: T) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
var myArray = [3.14159, 0.1, 0.25]
let firstIndex = findIndex(array: myArray, valueToFind: 0.1) // 1
let secondIndex = findIndex(array: myArray, valueToFind: 31) // nil
```

Параметр типа записывается как `<T: Equatable>`. Это означает «любой тип, поддерживающий протокол `Equatable`». В результате поиск в переданном массиве выполняется без ошибок, поскольку тип данных `Int` поддерживает протокол `Equatable`, следовательно, значения данного типа могут быть приняты к обработке.

35.4. Расширения универсального типа

Swift позволяет расширять описанные универсальные типы. При этом имена заполнителей, использованные в описании типа, могут указываться и в расширении.

Расширим описанный ранее универсальный тип `Stack`, добавив в него вычисляемое свойство, возвращающее верхний элемент стека без его удаления (листинг 35.8).

Листинг 35.8

```
extension Stack {
    var topItem: T? {
        return items.isEmpty ? nil : items[items.count - 1]
    }
}
stackOfInt.topItem // 5
stackOfInt.push(7)
stackOfInt.topItem // 7
```

Свойство `topItem` задействует заполнитель имени типа `T` для указания типа свойства. Данное свойство является опционалом, так как значение в стеке может отсутствовать. В этом случае возвращается `nil`.

35.5. Связанные типы

При определении протокола бывает удобно использовать связанные типы, указывающие на некоторый, пока неизвестный, тип данных. Связанный тип позволяет задать заполнитель типа данных, который будет использоваться при заполнении протокола. Фактически тип данных не указывается до тех пор, пока протокол не будет принят каким-либо объектным типом. Связанные типы указываются с помощью ключевого слова `associatedtype`, за которым следует имя связанного типа.

Определим протокол `Container`, использующий связанный тип `ItemType` (листинг 35.9).

Листинг 35.9

```
protocol Container {
    associatedtype  ItemType
    mutating func append(item: ItemType)
    var count: Int { get }
    subscript(i: Int) -> ItemType { get }
}
```

Протокол `Container` (контейнер) может быть задействован в различных коллекциях, например в описанном ранее типе коллекции `Stack`. В этом случае тип данных, используемый в свойствах и методах протокола, заранее неизвестен.

Для решения проблемы используется связанный тип `ItemType`, который определяется лишь при принятии протокола типом данных.

Пример принятия протокола к исполнению типом данных `Stack` представлен в листинге 35.10.

Листинг 35.10

```
struct Stack<T>: Container {
    typealias ItemType = T
    var items = [T]()
    var count: Int {
        return items.count
    }
    init(){}
    init(_ elements: T...){
        self.items = elements
    }
    subscript(i: Int) -> T {
        return items[i]
    }
```

```
    mutating func push(item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
    mutating func append(item: T) {
        items.append(item)
    }
}
```

Так как тип `Stack` теперь поддерживает протокол `Container`, в нем появилось три новых элемента: свойство, метод и сабскрипт. Ключевое слово `typealias` указывает на то, какой тип данных является связанным в данном объектном типе.

ПРИМЕЧАНИЕ Обратите внимание на то, что при описании протокола используется ключевое слово `associatedtype`, а при описании структуры — `typealias`.

Так как заполнитель имени использован в качестве типа аргумента `item` свойства `append` и возвращаемого значения сабскрипта, Swift может самостоятельно определить, что заполнитель `T` указывает на тип `ItemType`, соответствующий типу данных в протоколе `Container`. При этом указывать ключевое слово `associatedtype` не обязательно: если вы его удалите, то тип продолжит работать без ошибок.

36 Обработка ошибок

Обработка ошибок подразумевает реагирование на возникающие в процессе выполнения программы ошибки. Некоторые операции не могут гарантировать корректное выполнение вследствие возникающих обстоятельств. В этом случае очень важно определить причину возникновения ошибки и правильно обработать ее, чтобы не вызвать внезапного завершения всей программы.

В качестве примера можно привести запись информации в файл. При попытке доступа файл может не существовать или у пользователя могут отсутствовать права доступа для записи в него.

Отличительные особенности ситуаций могут помочь программе самостоятельно решать возникающие проблемы.

36.1. Выбрасывание ошибок

В Swift для создания перечня возможных ошибок служат перечисления, где каждый член перечисления соответствует отдельной ошибке. Само перечисление при этом должно поддерживать протокол `Error`, который хотя и является пустым, сообщает о том, что данный объектный тип содержит варианты ошибок.

Не стоит создавать одно перечисление на все случаи жизни. Группируйте возможные ошибки по их смыслу в различных перечислениях.

В следующем примере объявляется тип данных, который описывает ошибки в работе торгового автомата по продаже еды (листинг 36.1).

Листинг 36.1

```
enum VendingMachineError: Error {
    case InvalidSelection
    case InsufficientFunds(coinsNeeded: Int)
    case OutOfStock
}
```

Каждый из членов перечисления указывает на отдельный тип ошибки:

- неправильный выбор;
- нехватка средств;
- отсутствие выбранного товара.

Ошибка позволяет показать, что произошла какая-то нестандартная ситуация и обычное выполнение программы не может продолжаться. Процесс появления ошибки называется *выбрасыванием ошибки*. Для того чтобы выбросить ошибку, необходимо воспользоваться оператором `throw`. Так, следующий код при попытке совершить покупку выбрасывает ошибку о недостатке пяти монет (листинг 36.2).

Листинг 36.2

```
throw VendingMachineError.InsufficientFunds(coinsNeeded: 5)
```

36.2. Обработка ошибок

Сам по себе выброс ошибки не приносит каких-либо результатов. Выброшенную ошибку необходимо перехватить и корректно обработать. В Swift существует четыре способа обработки ошибок:

- передача ошибки;
- обработка ошибки оператором `do-catch`;
- преобразование ошибки в опционал;
- запрет на выброс ошибки.

Если при вызове какой-либо функции или метода вы знаете, что они могут выбросить ошибку, то необходимо перед вызовом указывать ключевое слово `try`.

Теперь разберем каждый из способов обработки ошибок.

Передача ошибки

При передаче ошибки блок кода (функция, метод или инициализатор), ставший источником ошибки, самостоятельно не обрабатывает ее, а передает выше в код, который вызвал данный блок кода.

Для того чтобы указать блоку кода, что он должен передавать возникающие в нем ошибки, в реализации данного блока после списка параметров указывается ключевое слово `throws`.

В листинге 36.3 приведен пример объявления двух функций, которые передают возникающие в них ошибки выше.

Листинг 36.3

```
func anotherFunc() throws {
    // тело функции
    var value = try someFunc()
    // ...
}
func someFunc() throws -> String{
    // тело функции
    try anotherFunc()
    // ...
}
try someFunc()
```

Функция `someFunc()` возвращает значение типа `String`, поэтому ключевое слово `throws` указывается перед типом возвращаемого значения.

Функция `anotherFunc()` в своем теле самостоятельно не выбрасывает ошибки, она может лишь перехватить ошибку, выброшенную функцией `anotherFunc()`. Для того чтобы перехватить ошибку, выброшенную внутри блока кода, необходимо осуществлять вызов с помощью упомянутого ранее оператора `try`. Благодаря ему функция `anotherFunc()` сможет отреагировать на возникшую ошибку так, будто она сама является ее источником. А так как эта функция также помечена ключевым словом `throws`, она просто передаст ошибку в вызвавший ее код.

Если функция не помечена ключевым словом `throw`, то все возникающие внутри нее ошибки она должна обрабатывать самостоятельно.

Рассмотрим пример из листинга 36.4.

Листинг 36.4

```
struct Item {
    var price: Int
    var count: Int
}
class VendingMachine {
    var inventory = [
        "Candy Bar": Item(price: 12, count: 7),
        "Chips": Item(price: 10, count: 4),
        "Pretzels": Item(price: 7, count: 11)
    ]
    var coinsDeposited = 0
    func dispenseSnack(snack: String) {
        print("Dispensing \(snack)")
    }
    func vend(itemNamed name: String) throws {
        guard var item = inventory[name] else {
            throw VendingMachineError.InvalidSelection
```

```
        }
        guard item.count > 0 else {
            throw VendingMachineError.OutOfStock
        }
        guard item.price <= coinsDeposited else {
            throw VendingMachineError.InsufficientFunds(coinsNeeded:
                  item.price - coinsDeposited)
        }
        coinsDeposited -= item.price
        item.count -= 1
        inventory[name] = item
        dispenseSnack(snack: name)
    }
}
```

Структура `Item` описывает одно наименование продукта из автомата по продаже еды. Класс `VendingMachine` описывает непосредственно сам аппарат. Его свойство `inventory` является словарем, содержащим информацию о наличии определенных товаров. Свойство `coinsDeposited` указывает на количество внесенных в аппарат монет. Метод `dispenseSnack(snack:)` сообщает о том, что аппарат выдает некий товар. Метод `vend(itemNamed:)` непосредственно обслуживает покупку товара через аппарат.

При определенных условиях (запрошенный товар недоступен, его нет в наличии или количества внесенных монет недостаточно для покупки) метод `vend(itemNamed:)` может выбросить ошибку, соответствующую перечислению `VendingMachineError`. Сама реализация метода использует оператор `guard` для реализации преждевременного выхода с помощью оператора `throw`. Оператор `throw` мгновенно изменяет ход работы программы, в результе выбранный продукт может быть куплен только в том случае, если все условия покупки выполняются.

Так как метод `vend(itemNamed:)` передает все возникающие в нем ошибки вызывающему его коду, то необходимо выполнить дальнейшую обработку ошибок с помощью оператора `try` или `do-catch`.

Реализуем функцию, которая в автоматическом режиме пытается приобрести какой-либо товар (листинг 36.5). В данном примере словарь `favoriteSnacks` содержит указатель на любимое блюдо каждого из трех человек.

Листинг 36.5

```
let favoriteSnacks = [
    "Alice": "Chips",
    "Bob": "Licorice",
    "Eve": "Pretzels",
```

```
]
func buyFavoriteSnack(person: String, vendingMachine: VendingMachine)
                      throws {
    let snackName = favoriteSnacks[person] ?? "Candy Bar"
    try vendingMachine.vend(itemNamed: snackName)
}
```

Сама функция `buyFavoriteSnack(person:vendingMachine:)` не может выбросить ошибку, но так как она вызывает метод `vend(itemNamed:)`, для передачи ошибки выше необходимо использовать операторы `throws` и `try`.

Оператор do-catch

Выброс и передача ошибок вверх в конце концов должна вести к их обработке таким образом, чтобы это принесло определенную пользу пользователю и разработчику. Для этого вы можете задействовать оператор `do-catch`.

СИНТАКСИС

```
do {
    try имяВызываемого Блока
} catch шаблон1 {
    // код...
} catch шаблон2 {
    // код...
}
```

Оператор содержит блок `do` и произвольное количество блоков `catch`. В блоке `do` должен содержаться вызов функции или метода, которые могут выбросить ошибку. Вызов осуществляется с помощью оператора `try`.

Если в результате вызова была выброшена ошибка, то данная ошибка сравнивается с шаблонами в блоках `catch`. Если в одном из них найдено совпадение, то выполняется код из данного блока.

Вы можете использовать ключевое слово `where` в шаблонах условий.

Блок `catch` можно задействовать без указания шаблона. В этом случае данный блок соответствует любой ошибке, а сама ошибка будет находиться в локальной переменной `error`.

Используем оператор `do-catch`, чтобы перехватить и обработать возможные ошибки (листинг 36.6).

Листинг 36.6

```
var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8
```

```
do {
    try buyFavoriteSnack(person: "Alice", vendingMachine:
vendingMachine)
} catch VendingMachineError.InvalidSelection {
    print("Invalid Selection.")
} catch VendingMachineError.OutOfStock {
    print("Out of Stock.")
} catch VendingMachineError.InsufficientFunds(let coinsNeeded) {
    print("Недостаточно средств. Пожалуйста, внесите еще \(coinsNeeded)
          монет(ы).")
}
// выводит "Недостаточно средств. Пожалуйста, внесите еще 2 монет(ы)."
```

В приведенном примере функция `buyFavoriteSnack(person:vendingMachine:)` вызывается в блоке `do`. Поскольку внесенной суммы монет не хватает для покупки любимой сладости покупателя `Alice`, возвращается ошибка и выводится соответствующее этой ошибке сообщение.

Преобразование ошибки в опционал

Для преобразования выброшенной ошибки в опциональное значение используется оператор `try`, а точнее, его форма `try?`. Если в этом случае выбрасывается ошибка, то значение выражения вычисляется как `nil`.

Рассмотрим пример из листинга 36.7.

Листинг 36.7

```
func someThrowingFunction() throws -> Int {
    // ...
}
let x = try? someThrowingFunction()
```

Если функция `someThrowingFunction()` выбросит ошибку, то в константе `x` окажется значение `nil`.

Запрет на выброс ошибки

В некоторых ситуациях можно быть уверенными, что блок кода во время исполнения не выбросит ошибку. В этом случае необходимо использовать оператор `try!`, который сообщает о том, что данный блок гарантированно не выбросит ошибку, — это запрещает передачу ошибки в целом.

Рассмотрим пример из листинга 36.8.

Листинг 36.8

```
let photo = try! loadImage("./Resources/John Appleseed.jpg")
```

Функция `loadImage(_:)` производит загрузку локального изображения, а в случае его отсутствия выбрасывает ошибку. Так как указанное в ней изображение является частью разрабатываемой вами программы и гарантированно находится по указанному адресу, с помощью оператора `try!` целесообразно отключить режим передачи ошибки.

Будьте внимательны: если при запрете передачи ошибки блок кода все же выбросит ее, то ваша программа экстренно завершится.

36.3. Отложенные действия по очистке

Swift позволяет определить блок кода, который будет выполнен лишь по завершении выполнения текущей части программы. Для этого служит оператор `defer`, который содержит набор отложенных выражений. С его помощью вы можете выполнить необходимую очистку независимо от того, как произойдет выход из данной части программы.

Отложенные действия выполняются в обратном порядке, то есть вначале выполняется блок последнего оператора `defer`, затем предпоследнего и т. д.

Рассмотрим пример использования блока отложенных действий (листинг 36.9).

Листинг 36.9

```
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }
        while let line = try file.readline() {
            // работа с файлом.
        }
    }
}
```

В данном примере оператор `defer` просто обеспечивает закрытие открытого ранее файла.

37 Нетривиальное использование операторов

Вы уже познакомились с большим количеством операторов, которые предоставляет Swift. Однако возможна ситуация, в которой для ваших собственных объектных типов данных эти операторы окажутся бесполезными. В таком случае вам потребуется самостоятельно создать свои реализации стандартных операторов или полностью новые операторы.

37.1. Операторные функции

С помощью операторных функций вы можете обеспечить взаимодействие собственных объектных типов посредством стандартных операторов Swift.

Предположим, что вы разработали структуру, описывающую вектор на плоскости (листинг 37.1).

Листинг 37.1

```
1  struct Vector2D {
2       var x = 0.0, y = 0.0
3  }
```

Свойства `x` и `y` показывают координаты конечной точки вектора. Начальная точка находится либо в точке с координатами `(0, 0)`, либо в конечной точке предыдущего вектора.

Если перед вами возникнет задача сложить два вектора, то проще всего воспользоваться операторной функцией и создать собственную реализацию оператора сложения (`+`), как показано в листинге 37.2.

Листинг 37.2

```
func + (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x + right.x, y: left.y + right.y)
}
```

```
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
```

Здесь операторная функция определена с именем, соответствующим оператору сложения. Так как оператор сложения является бинарным, он должен принимать два заданных значения и возвращать результат сложения.

Ситуация, когда несколько объектов имеют одно и то же имя, в Swift носит название *перегрузки*. С данным понятием мы уже неоднократно встречались по ходу чтения книги.

Префиксные и постфиксные операторы

Оператор сложения является бинарным инфиксным, то есть он ставится между двумя операндами. Помимо инфиксных операторов, в Swift существуют префиксные (предшествуют операнду) и постфиксные (следуют за операндом) операторы.

Для перегрузки префиксного или постфиксного оператора перед объявлением операторной функции необходимо указать модификатор `prefix` или `postfix` соответственно.

Реализуем префиксный оператор унарного минуса для структуры `Vector2D` (листинг 37.3).

Листинг 37.3

```
prefix func - (vector: Vector2D) -> Vector2D {
    return Vector2D(x: -vector.x, y: -vector.y)
}
let positive = Vector2D(x: 3.0, y: 4.0)
let negative = -positive  // negative — экземпляр Vector2D
                          // со значениями (-3.0, -4.0)
```

Благодаря созданию операторной функции мы можем использовать унарный минус для того, чтобы развернуть вектор относительно начала координат.

Составной оператор присваивания

В составных операторах присваивания оператор присваивания (+) комбинируется с другим оператором. Для перегрузки составных операторов в операторной функции первый передаваемый аргумент

необходимо сделать сквозным (`inout`), так как именно его значение будет меняться в ходе выполнения функции.

В листинге 37.4 приведен пример реализации составного оператора присваивания-сложения для экземпляров типа `Vector2D`.

Листинг 37.4

```
func += ( left: inout Vector2D, right: Vector2D) {
    left = left + right
}
var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd
// original теперь имеет значения (4.0, 6.0)
```

Так как оператор сложения был объявлен ранее, вам нет нужды реализовывать его в теле данной функции. Вы можете просто сложить два значения типа `Vector2D`.

Обратите внимание, что первый входной аргумент функции является сквозным.

Оператор эквивалентности

Пользовательские объектные типы не содержат встроенной реализации оператора эквивалентности, поэтому чтобы сравнить два экземпляра, необходимо перегрузить данный оператор с помощью операторной функции.

В следующем примере приведена реализация оператора эквивалентности и оператора неэквивалентности (листинг 37.5).

Листинг 37.5

```
func == (left: Vector2D, right: Vector2D) -> Bool {
    return (left.x == right.x) && (left.y == right.y)
}
func != (left: Vector2D, right: Vector2D) -> Bool {
    return !(left == right)
}
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
    print("Эти два вектора эквивалентны.")
}
// выводит "Эти два вектора эквивалентны."
```

В операторной функции == мы реализуем всю логику сравнения двух экземпляров типа Vector2D. Так как данная функция возвращает false в случае неэквивалентности операторов, мы можем использовать ее внутри собственной реализации оператора неэквивалентности.

37.2. Пользовательские операторы

В дополнение к стандартным операторам языка Swift вы можете определять собственные. Собственные операторы объявляются с помощью ключевого слова operator и модификаторов prefix, infix и postfix, причем вначале необходимо объявить новый оператор, а уже потом задавать его новую реализацию в виде операторной функции.

В следующем примере реализуется новый оператор +++, который складывает экземпляр типа Vector2D сам с собой (листинг 37.6).

Листинг 37.6

```
prefix operator +++
prefix func +++ (vector: inout Vector2D) -> Vector2D
{
    vector += vector
    return vector
}
var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
// toBeDoubled теперь имеет значения (2.0, 8.0)
// afterDoubling также имеет значения (2.0, 8.0)
```

Часть VII. Введение в мобильную разработку

Вы прошли весь предлагаемый материал, и если выполнили все задания, то можете с гордостью сказать, что имеете навыки программирования на Swift. Но при этом до нашей основной цели «создание приложений под iOS» предстоит преодолеть большой путь.

Каждая из последующих глав в книге будет вашим проводником на нелегком пути Apple-разработчика. Несмотря на то что до этого мы занимались не мобильной разработкой, а созданием консольных приложений для настольного компьютера, нашей основной задачей было изучение важнейших элементов интерфейса Xcode и прохождение начальных этапов в вопросах взаимодействия с ним.

Пришло время перейти к наиболее интересной теме: разработке под мобильную операционную систему. Скоро вы практически самостоятельно создадите несколько простейших приложений, каждое из которых будет направлено на изучение возможностей Xcode. В процессе разработки мы разберем множество новых функциональных элементов и таких понятий, как storyboard, view controller, Interface Builder, user interface, модальные окна, кнопки, надписи и многое-многое другое.

Держитесь крепче, будет очень интересно!

38 Разработка приложения под iOS

Очень важно, чтобы вы погружались в разработку плавно, без рывков в изучении материала. Мы будем максимально подробно разбирать каждый шаг, останавливаясь на новых и пока еще неизвестных понятиях и механизмах Xcode.

В этой главе вы впервые перейдете к созданию приложения, имеющего пусть минимальный, но действующий функционал.

38.1. Создание проекта MyName

Сейчас мы приступим к созданию простейшего iOS-приложения, в процессе разработки которого вы познакомитесь с некоторыми базовыми понятиями и функциональными возможностями.

ЗАДАНИЕ

Создайте новый проект для платформы iOS типа Single View Application с именем MyName.

Прежде чем приступить к разработке приложения, обратимся к редактору проекта. Сейчас в нем отображаются настройки вашего приложения (рис. 38.1).

Напомню, что изменение настроек проекта доступно в том случае, когда в Project Navigator выбран корневой объект дерева ресурсов (в данном случае это MyName).

В верхней части рабочей области расположена панель вкладок, с помощью которой можно перемещаться между различными группами настроек. В настоящий момент активна вкладка General, и именно ее состав отображается в редакторе проекта. С ее помощью вы можете

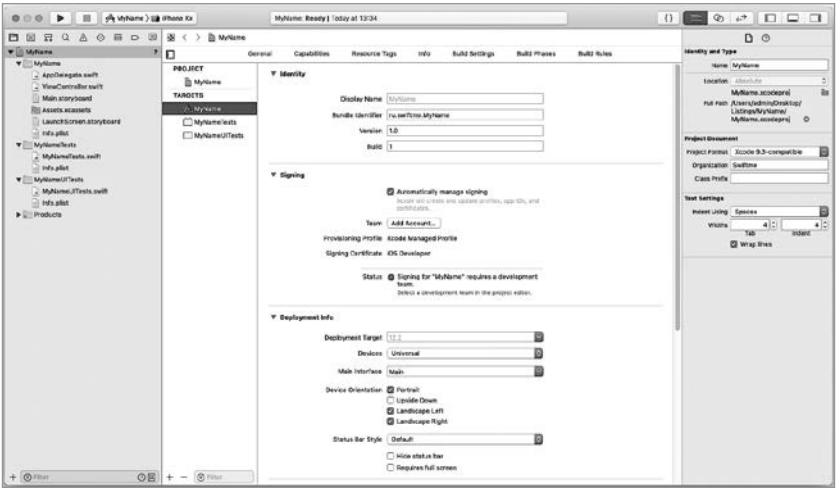

Рис. 38.1. Настройки проекта MyName

изменить основные настройки вашего приложения, в том числе ряд тех, которые были указаны при создании проекта.

Все параметры вкладки General разделены на разделы. Раздел Identify содержит следующие параметры:

- Display Name — имя проекта, которое вы задавали при его создании. Значение данного поля не подлежит редактированию.
- Bundle Identify — идентификатор продукта.
- Version — версия приложения.
- Build — номер сборки приложения.

Прежде чем вы разместите свое новое приложение в магазине AppStore, его необходимо подписать с помощью цифровой подписи. Возможность (или невозможность) совершения этой операции указана в разделе Signing. Обратите внимание на раздел Deployment Info. В нем содержатся параметры, непосредственно влияющие на работу приложения. Так, например, параметр Devices определяет устройства, под которыми будет работать ваше приложение. Значение Universal говорит о том, что ваше приложение будет работать как под iPhone, так и под iPad. Щелкните по выпадающему списку и выберите iPhone.

38.2. Interface Builder, Storyboard и View Controller

Пришло время перейти к разработке пользовательского интерфейса (User Interface, сокращенно UI) вашего приложения.

Взгляните на Project Navigator: там появилось множество новых, по сравнению с проектом консольного приложения, ресурсов (рис. 38.2).

Рис. 38.2. Структура проекта MyName

ПРИМЕЧАНИЕ Обратите внимание, что в случае, если при создании проекта вы не сняли галочки с пунктов Include Unit Tests и Include UI Tests, в структуре проекта будут расположены папки MyNameTests и MyNameUITests.

Рассмотрение их содержимого и возможностей тестирования приложений выходит за рамки темы данной книги.

Файлы с расширением swift (AppDelegate.swift и ViewController.swift) содержат исходный код приложения. Файлы с расширением storyboard (Main.storyboard и LaunchScreen.storyboard) ранее нам не встречались. Они предназначены для создания UI приложения, и этот процесс в Xcode очень прост и понятен, с ним мы вскоре познакомимся. Щелкните по файлу Main.storyboard в области навигации и обратите внимание на то, как изменился Project Editor (рис. 38.3). Перед вами открылся редактор интерфейса (Interface Builder, сокращенно IB).

Interface Builder обеспечивает удобный визуальный способ создания и редактирования UI приложений. С помощью IB создается раскадровка (storyboard), состоящая из одного или нескольких рабочих

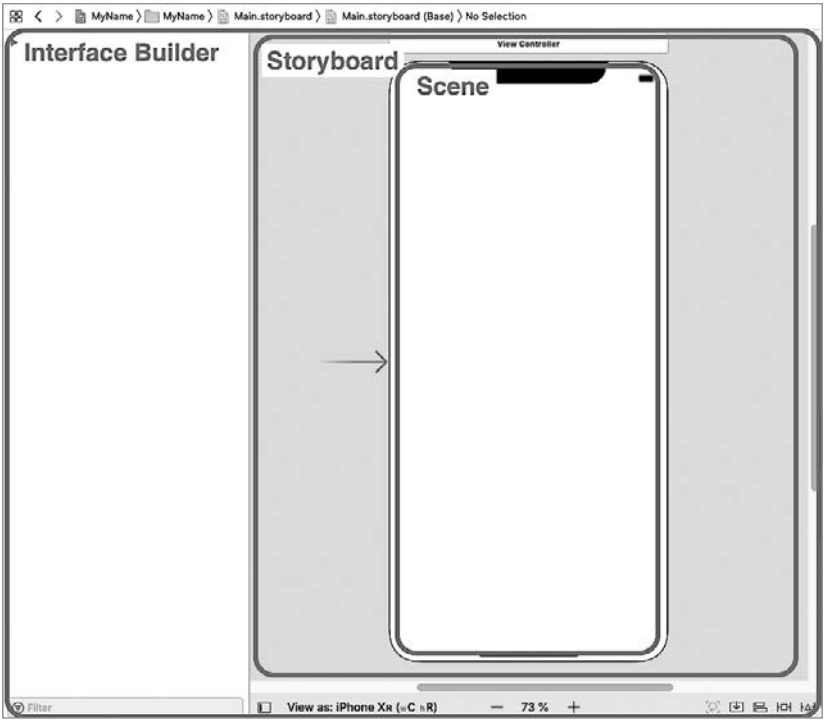

Рис. 38.3. Редактор интерфейса

экранов и набора связей между этими экранами. Каждый отдельный рабочий экран вашего приложения называется сценой (scene). С помощью редактора интерфейса вы можете создать необходимое количество сцен и обеспечить их взаимодействие между собой. И все это может быть сделано без единой строчки кода!

Сейчас storyboard вашего приложения состоит всего из одной сцены.

ПРИМЕЧАНИЕ Во время своей работы Interface Builder взаимодействует с файлами xib (они не отображаются в Project Navigator, вы можете увидеть их лишь в файловой системе, в папке с проектом). Это обычные текстовые файлы, имеющие XML-формат и описывающие графические элементы интерфейса. Storyboard-файлы, в свою очередь, компонуют множество xib-файлов. Читая документацию или статьи о разработке в Xcode, вы можете встретить упоминания nib-файлов. В настоящее время nib-формат является устаревшим, ему на смену пришел xib, но в силу исторических причин такие файлы порой все еще называются nib.

Обратите внимание на Jump Bar (панель перехода), расположенную прямо над Interface Builder (рис. 38.4). Это очень полезный функциональный элемент Xcode, с его помощью, так же как и в Project Navigator, вы можете выбирать ресурс, который требуется отобразить в редакторе проекта. В дальнейшем вы будете очень часто его использовать.

Рис. 38.4. Jump Bar, предназначенный для перехода между ресурсами проекта

Ранее на этапе создания проекта в качестве шаблона был выбран Single View Application, что дословно переводится как «приложение с одним отображением». В соответствии с этим шаблоном файл Main.storyboard содержит одну сцену, то есть один рабочий экран приложения (вернитесь к рис. 38.3). Со временем storyboard вашего приложения может обзавестись большим количеством сцен. Пример storyboard с большим количеством сцен приведен на рис. 38.5.

Рис. 38.5. Storyboard с большим количеством сцен

Как говорилось ранее, в состав storyboard любого приложения входят:

- одна или несколько сцен;
- связи между сценами.

При этом в любой сцене можно выделить следующие элементы:

- отображения (`views`, среди программистов также называемые «вьюшками») графических элементов сцены (кнопок, надписей, таблиц и т. д.);
- дополнительные элементы, обеспечивающие функциональность сцены.

Отображения могут входить в состав друг друга. Они являются базовыми строительными блоками, из которых и создается UI приложение.

ПРИМЕЧАНИЕ Вы наверняка помните, как использовали вложенные друг в друга отображения при разработке приложения, позволяющего двигать шарики в Xcode Playground.

В левой части Interface Builder находится Document outline (схема документа) (рис. 38.6). Если отобразить все вложенные в View Controller Scene элементы, то вы увидите полную структуру вашего storyboard. Если Project Navigator позволяет вам получить доступ к различным ресурсам проекта, в том числе к storyboard-файлам, то благодаря Document outline вы получаете очень удобный способ доступа к элементам, входящим в состав storyboard.

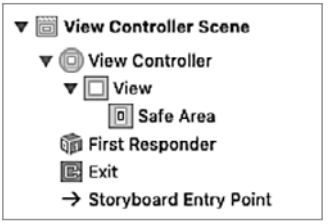

Рис. 38.6. Структура storyboard

Пока что storyboard состоит всего из одной сцены, которые называются View Controller Scene в Document outline. Он, в свою очередь, состоит из следующих элементов:

- View Controller — это очень важный объект, от которого во многом зависит функционирование сцены. Он загружает сцену на экран вашего устройства, а также следит за происходящим на сцене. Дополнительно он обеспечивает эффективный и однозначный доступ к любому из объектов, созданных в пределах сцены (например, к кнопкам).

- View определяет непосредственно саму сцену, а точнее, экземпляр класса `UIView`, который является визуальным макетом, загружаемым и отображаемым средствами View Controller на экран вашего устройства. View — это иерархическая структура, а значит, отображения различных графических элементов могут создавать дерево и входить в состав друг друга. Ранее вы встречались с классом `UIView` при создании интерактивного Xcode Playground. Если вы щелкнете по иконке View в Document Outline, то Xcode автоматически подсветит квадрат, соответствующий экрану на вашей сцене в storyboard.
- Safe Area описывает сцену приложения, исключая верхний бар с данными о сотовой сети, заряде батареи и другой системной информацией.
- First Responder всегда указывает на объект, с которым пользователь взаимодействует в данный момент. При работе с iOS-приложениями пользователь потенциально может начать взаимодействие со многими элементами, расположенными на сцене. Например, если он начал вводить данные в текстовое поле (tetxfield), то именно оно и будет являться first responder в данный момент.
- Exit служит единственной специфической цели и используется только в приложениях с множеством сцен. Когда вы создаете программу, которая перемещает пользователя между набором экранов, то exit обеспечивает перемещение к одному из предыдущих экранов.
- Storyboard Entry Point является указателем на то, какая именно сцена будет отображена при запуске приложения.

ПРИМЕЧАНИЕ Не переживайте, если предназначение каких-либо (или всех) элементов вам не в полной мере понятно. Со временем вы поработаете с каждым из них.

ПРИМЕЧАНИЕ Стоит отметить, что указанная иерархия сцены является условной: например, элементы Storyboard Entry Point и Exit описывают связи сцен, а ранее мы говорили о том, что связи являются составной частью storyboard, но не сцены. Похожая ситуация и с View Controller: по сути, он не входит в состав сцены — он обеспечивает ее работу.

В Xcode 9 был добавлен новый элемент Safe Area. Его появление связано с выходом iPhone X (и его экраном необычной формы) и появившимся в связи с этим новым подходом к работе со сценами. В правой панели откройте File inspector и взгляните в раздел Interface Builder Document (рис. 38.7). В нем расположен переключатель Use Safe Area Layout Guides, с помощью которого можно выбрать вариант работы со сценой (использовать Safe zone или старый вариант).

Рис. 38.7. Переключатель Use Safe Area Layout Guides

Уберите галочку переключателя и обратите внимание на изменения в Document outline. Из View Controller исчез Safe zone, но появились два новых элемента:

- Top Layout Guide описывает «шапку» вашего приложения, обычно расположенную сразу под статусбаром с часами (рис. 38.8).

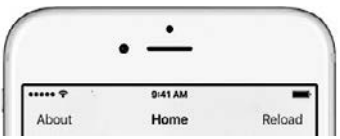

Рис. 38.8. Пример «шапки» приложения

- Bottom Layout Guide описывает «подвал» вашего приложения, обычно расположенный в самом низу сцены (рис. 38.9).

Рис. 38.9. Пример «подвала» приложения

В следующей версии Xcode данный функционал, вероятно, будет полностью удален. В связи с этим поставим галочку напротив Use Safe Area Layout Guides и в дальнейшем будем работать именно с Safe zone.

Одним из пунктов в схеме документа является View Controller (контроллер отображения). Запомните, данный контроллер в действительности не входит в состав сцены, он лишь обеспечивает ее отображение. View Controller — это внешний объект, предназначенный для управления ассоциированной с ним сценой и набором отображений, входящих в эту сцену.

ПРИМЕЧАНИЕ Чуть позже мы подробнее разберем, что такое View Controller и как организуется его взаимосвязь с отображениями.

Некоторые элементы на панели структуры сцены имеют соответствующие им графические элементы на самой сцене (рис. 38.10). Пощелкайте по всем элементам в Document outline и посмотрите, что будет подсвечиваться на изображении сцены справа.

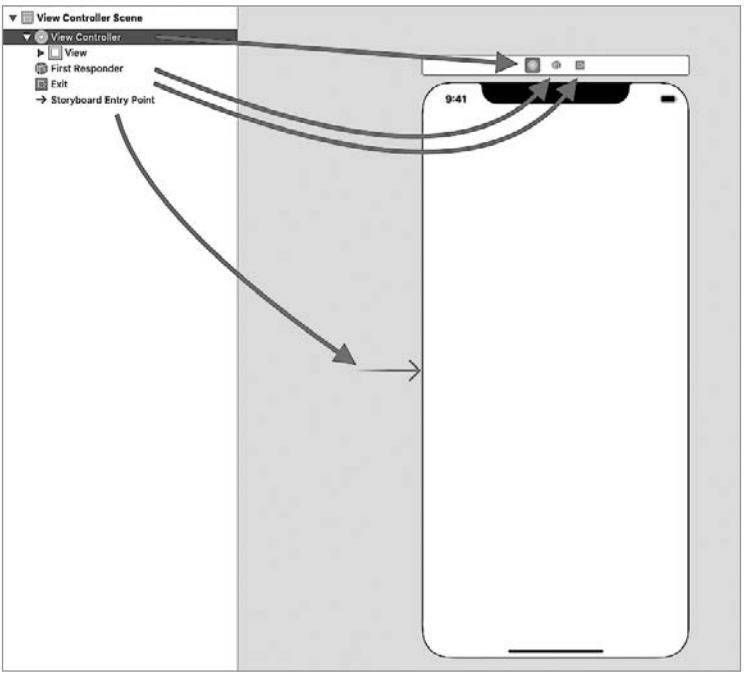

Рис. 38.10. Соответствие элементов схемы элементам сцены

ПРИМЕЧАНИЕ Если в верхней части вашей сцены вместо трех значков отображается надпись View Controller, то для их отображения достаточно щелкнуть по любому элементу на панели структуры сцены.

38.3. Разработка простейшего UI

Убедитесь, что в Project Navigator файл Main.storyboard является активным, а в редакторе проекта отображается Interface Builder.

При разработке графических приложений Xcode позволяет использовать шаблоны визуальных элементов, таких как текстовое поле, надпись, ползунок, кнопка и многие другие, которые могут быть полезны во время разработки графического интерфейса приложений. Для доступа к библиотеке объектов нажмите кнопку Library ( ), расположенную в Toolbar, после чего перед вами появятся все доступные элементы (рис. 38.11).

Рис. 38.11. Библиотека объектов

С помощью кнопки Switch to в правом верхнем углу библиотеки объектов вы можете выбрать вид отображения объектов в виде списка или сетки.

ПРИМЕЧАНИЕ Все объекты доступны благодаря фреймворку `UIKit`, в состав которого они входят.

Перейдем непосредственно к использованию IB и разработке UI приложения. Следующим шагом станет размещение на сцене кнопки. Для этого в библиотеке объектов найдите элемент Button (можете использовать поле поиска, введя в него имя искомого элемента) и перетащите его в IB, прямо в центр вашей сцены. После размещения объекта он появится на панели структуры сцены в разделе Views (рис. 38.12). Теперь View сцены (экземпляр класса `UIView`) содержит вложенный View кнопки, который является экземпляром класса `UIButton`.

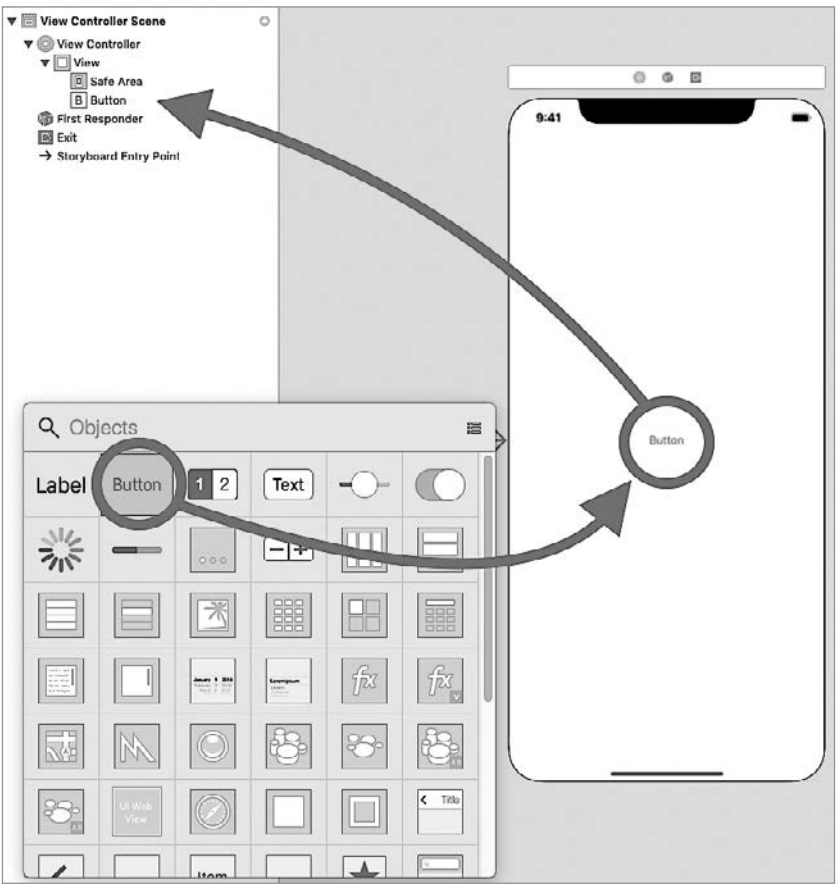

Рис. 38.12. Размещение элемента Button на сцене

ПРИМЕЧАНИЕ Обратите внимание, что при размещении объекта в центре сцены начинают отображаться вспомогательные прерывистые линии синего цвета. С их помощью вы можете эффективно размещать графические элементы на сцене.

Теперь изменим текст кнопки. Для его изменения дважды щелкните по ней и введите «`Hello World`». После этого для сохранения изменений нажмите Enter на клавиатуре.

ПРИМЕЧАНИЕ После изменения текста вам может потребоваться повторная отцентровка кнопки на сцене.

38.4. Запуск приложения в эмуляторе

Разработка iOS-проектов была бы чрезвычайно сложной в том случае, если бы в целях тестирования их требовалось запускать только на реальном устройстве. С целью оптимизации этого процесса в состав Xcode включены эмуляторы всех имеющихся на данный момент на рынке устройств «яблочной» корпорации. Таким образом, разрабатываемое вами приложение MyName при запуске, если не указано иное, будет отображаться в окне эмулятора iPhone.

С тем, как производить запуск приложения, вы познакомились при создании консольного приложения. В текущем проекте панель инструментов также содержит набор кнопок для запуска процесса сборки и осуществления принудительной остановки запущенного приложения. Дополнительно имеется возможность выбрать устройство, на эмуляторе которого будет запущен проект (рис. 38.13). Вопрос тестирования приложений на различных эмуляторах становится очень актуальным в связи с тем, что сегодня iOS функционирует на большом числе устройств с большим количеством разрешений.

Рис. 38.13. Кнопки запуска проекта

ПРИМЕЧАНИЕ Xcode позволяет также разрабатывать приложения, которые будут работать одновременно на разных устройствах, в том числе с различным соотношением сторон и разрешением экрана.

Запустите приложение с помощью кнопки Build and run на панели инструментов.

В статус-баре виден текущий статус проекта. После запуска сборки проекта информация, отображаемая в нем, позволяет определить, на каком этапе работы находится ваш проект (рис. 38.14).

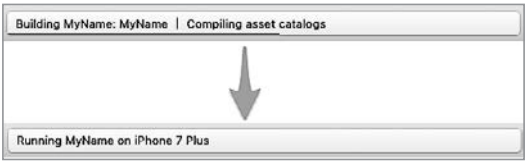

Рис. 38.14. Изменения в статусбаре во время запуска проекта

Рис. 38.15. Окно эмулятора

В результате ваше iOS-приложение отобразится в окне эмулятора (рис. 38.15), в центре которого будет расположена кнопка с текстом «Hello World». Попробуйте щелкнуть на ней. При этом ничего не происходит, так как пока мы не запрограммировали какие-либо дальнейшие действия.

Теперь остановите запущенное приложение с помощью кнопки Stop the running scheme or application и на панели инструментов щелкните на названии модели iPhone, на котором производилась эмуляция (вероятно, был выбран iPhone Xr). В выпадающем списке выберите iPhone 8 Plus и произведите повторный запуск проекта.

На этот раз кнопка «Hello World» будет расположена не в центре, а немного ниже от него! И это на первый взгляд удивительно. Чтобы понять, в чем проблема, вернитесь в Xcode. В нижней части редактора проекта указано, что IB должен отображать экран, соответствующий iPhone Xr, но при этом программа запускалась на эмуляторе iPhone 8 Plus (рис. 38.16). Эти устройства имеют различные разрешения, а в связи с тем, что координаты кнопки фиксируются в пикселах по отступу от левого верхнего угла, на разных устройствах кнопка располагается в разных местах.

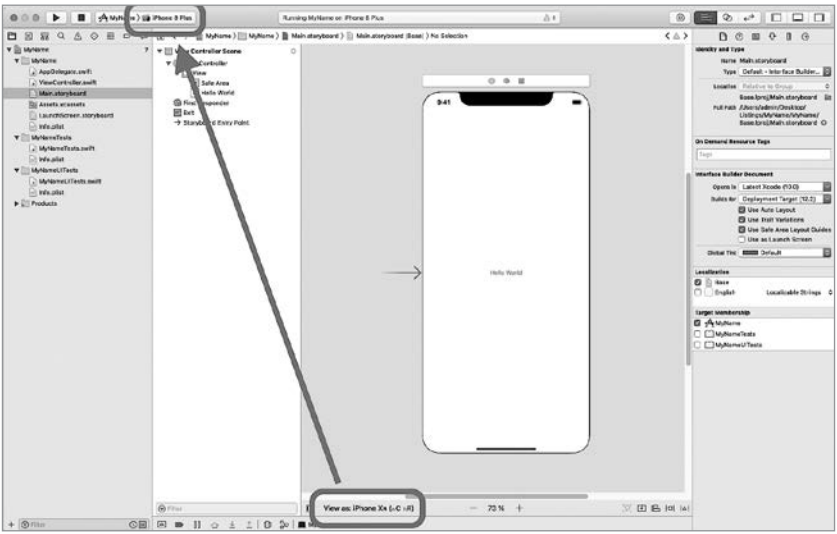

Рис. 38.16. Различные типы устройств

Обратите внимание на то, как просто вы разместили кнопку в приложении. Однако при нажатии на нее ничего не происходит. Сейчас мы реализуем функционал, обеспечивающий появление всплывающего окна с вашим именем. Для этого потребуется написать всего несколько строчек кода!

38.5. View Controller сцены и класс UIViewController

На этом этапе мы добавим нашей кнопке функциональные возможности. В Project Navigator щелкните по файлу ViewController.swift, после чего его содержимое отобразится в Project Editor. Как вы видите, в коде файла уже присутствует класс `ViewController` (листинг 38.1). Это связано с тем, что в качестве шаблона приложения был выбран Single View Application.

Листинг 38.1

```
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.
    }
}
```

Xcode позволяет создать связь и организовать взаимодействие кода и элементов интерфейса. Для этого нужно связать объект View Controller, отображенный в Document outline и входящий в состав сцены, и класс `ViewController`, описанный в файле ViewController.swift. Это позволит влиять на элементы интерфейса с помощью программного кода.

Шаблон Single View Application, который был выбран при создании проекта, изначально содержит одну сцену со своим View Controller, один класс ViewController и уже настроенную связь между ними. Схематично эта связь приведена на рис. 38.17.

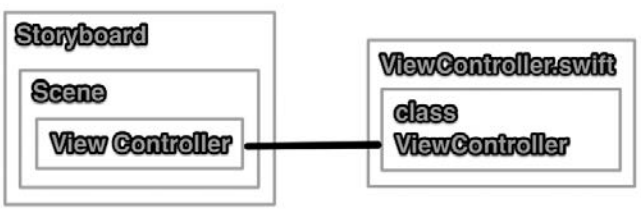

Рис. 38.17. Связь объекта View Controller и класса ViewController

Работая над сложными проектами, вы будете создавать большое количество различных сцен (а значит, и элементов View Controller), классов и связей между ними. Для этой цели в Xcode есть возможность настройки и изменения уже созданных связей между View Controller

и классами. Для этого в Project Navigator выберите файл Main.storyboard, после чего в Document Outline выделите элемент View Controller. Далее, в Utilities Area откройте вкладку Identify Inspector, нажав кнопку Show the Identify Inspector ( ).

Параметр Class в разделе Custom Class содержит ссылку на связанный со сценой программный класс, в данном случае это ViewController (рис. 38.18).

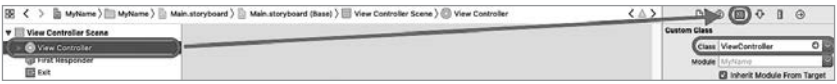

Рис. 38.18. Отображение связи элемента сцены View Controller и класса ViewController

Вернемся к классу ViewController в файле ViewController.swift. Обратите внимание на два важных момента.

В самом начале подключается фреймворк `UIKit` (листинг 38.2).

Листинг 38.2

```
import UIKit
```

Класс `ViewController` является подклассом `UIViewController` (листинг 38.3).

Листинг 38.3

```
class ViewController: UIViewController {…}
```

ПРИМЕЧАНИЕ Напомню, что два класса могут находиться в отношении «суперкласс — подкласс (субкласс)». Подкласс наследует все характеристики (свойства и методы) родительского класса. Имя суперкласса (если он имеется) указывается при определении подкласса через двоеточие.

С библиотекой `UIKit` мы встречались уже не один раз. Напомню, что `UIKit` — это фреймворк, входящий в стандартную поставку Xcode. Он обеспечивает функциональные возможности для построения и управления пользовательским интерфейсом, анимацией, текстом, изображениями для ваших приложений, а также для обработки событий, происходящих во время работы пользователя. Если говорить проще, то `UIKit` — это набор основных возможностей, которые разработчики используют при построении интерфейса практически любого приложения.

В библиотеке UIKit описан класс UIViewController, предназначенный для описания инфраструктуры и управления отображениями приложения. Основной функцией UIViewController (а также его потомков) является обновление содержимого отображений, в том числе в ответ на действия пользователей.

ПРИМЕЧАНИЕ В дальнейшем вы практически не будете создавать экземпляры класса UIViewController, наиболее часто используемым решением будет, как и в нашем случае, создание потомков UIViewController с целью расширения его возможностей собственным функционалом.

Обратите внимание на метод viewDidLoad(), описанный в классе ViewController. Так как вы создали приложение, используя шаблон Single View Application, Xcode автоматически переопределил его в коде класса ViewController. Родительский класс UIViewController имеет большое количество методов, которые могут быть переопределены в потомках. Каждый из этих методов служит для решения конкретной задачи (например, выполнение блока кода сразу после вывода отображения на экран устройства). Со многими из таких методов вы познакомитесь в процессе изучения вопроса разработки приложений в Xcode.

Несмотря на то что viewDidLoad() не содержит какого-либо кода, а лишь вызывает одноименный родительский метод, при необходимости он может быть расширен для реализации требуемого функционала. Но в некоторых случаях в его использовании нет необходимости, а значит, его можно удалить. О том, для чего предназначен viewDidLoad(), мы поговорим позже. Сейчас вам требуется убрать его из класса ViewController (листинг 38.4).

Листинг 38.4

```
class ViewController: UIViewController {
}
```

38.6. Доступ UI к коду. Определитель типа @IBAction

Добавим в наше приложение простейшие функциональные возможности, а именно вывод на консоль текстового сообщения при нажатии на кнопку. Ранее мы говорили, что класс ViewController ассоциирован с элементом View Controller сцены. Соответственно, для создания допол-

нительного функционала необходимо работать именно с этим классом. Расширим его новым методом `showMessage()`, использующим внутри своей реализации функцию `print(_:)` с текстом `"you press Hello World button"` (листинг 38.5).

ПРИМЕЧАНИЕ Не забывайте, что вы можете использовать кодовые сниппеты для упрощения создания кодовых конструкций.

Листинг 38.5

```
class ViewController: UIViewController {
     func showMessage(){
          print("you press Hello World button")
     }
}
```

Для того чтобы появилась возможность создать связь между методом класса и элементом сцены, используется определитель типа `@IBAction` (среди разработчиков называемый «экшеном»), указываемый перед определением метода. Добавим `@IBAction` к методу `showMessage()` (листинг 38.6).

Листинг 38.6

```
class ViewController: UIViewController {
    @IBAction func showMessage(){
         print("you press Hello World button")
    }
}
```

ПРИМЕЧАНИЕ Обратите внимание, что слева от метода `showMessage()` вместо номера строки теперь отображается серый квадрат.

Ключевое слово `@IBAction` разрешает графическим элементам в Interface Builder осуществлять доступ к методам класса.

ПРИМЕЧАНИЕ Для создания обратной связи, когда метод обращается к элементам интерфейса, необходимо использовать определитель `@IBOutlet`. С ним мы познакомимся позже.

Так как метод `showMessage()` помечен с помощью экшена, мы можем организовать его вызов по событию «нажатие на кнопку». Для этого в IB откройте Main.storyboard, зажмите Control на клавиатуре и с помощью мышки потяните кнопку «Hello World» на желтый круглый символ View Controller (рис. 38.19), после чего отпустите. В процессе перетаскивания от кнопки к указателю мышки будет идти линия синего цвета.

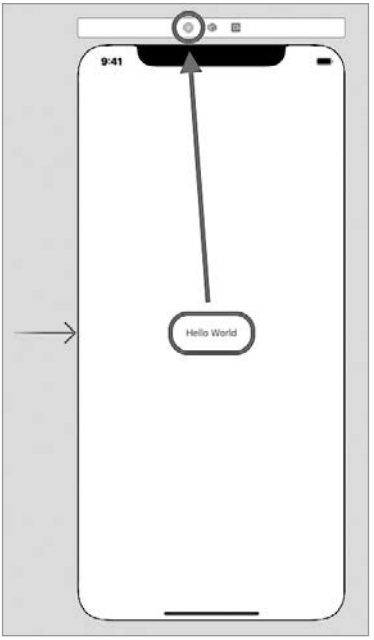

Рис. 38.19. Создание связи между View Controller и showMessage()

В появившемся окне выберите элемент showMessage, находящийся в разделе Sent Events (рис. 38.20).

ПРИМЕЧАНИЕ Метод `showMessage()` доступен для связывания и находится в выпадающем списке, так как помечен ключевым словом `@IBAction`.

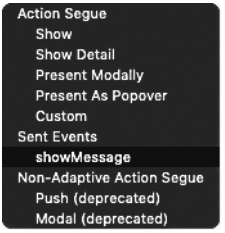

Рис. 38.20. Выбор showMessage() для связи с кнопкой на сцене

Теперь, если запустить приложение и щелкнуть на кнопке, то будет вызван метод `showMessage()` класса `ViewController`, а значит, произойдет вывод фразы `"you press Hello World button"` на консоль (рис. 38.21).

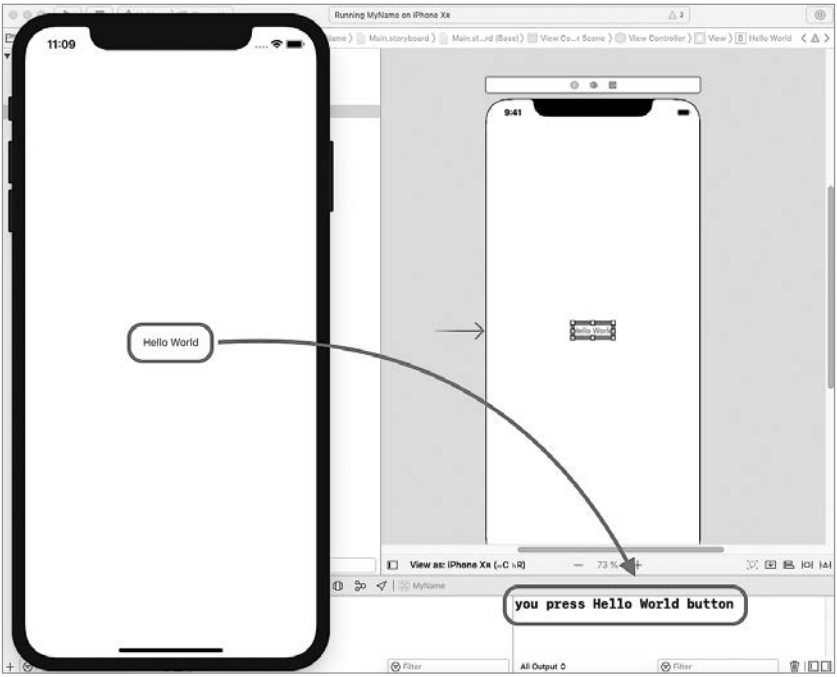

Рис. 38.21. Вывод на консоль по нажатии кнопки на сцене

ПРИМЕЧАНИЕ Обратите внимание, что на рис. 38.21 приложение снова запущено на эмуляторе iPhone Xr.

38.7. Отображение всплывающего окна. Класс UIAlertController

Следующим шагом станет реализация появления всплывающего окна при нажатии на кнопку на сцене. Для этого также будем использовать метод `showMessage()`.

В Xcode для работы со всплывающими окнами (оповещениями) используется класс `UIAlertController`. Оповещение может быть отображено в классическом виде в центре экрана (тип `Alert`) (рис. 38.22) или в нижней части сцены (тип `Action Sheet`) (рис. 38.23).

Рис. 38.22. UIAlertController в виде Alert

Рис. 38.23. UIAlertController в виде Action Sheet

Рассмотрим использование `UIAlertController` подробнее.

СИНТАКСИС

Класс `UIAlertController`

Предназначен для создания всплывающего окна. Является потомком класса `UIViewController`.

Инициализаторы:

```
init( title: String?, message: String?, preferedStyle:
UIAlertControllerStyle).
```

Доступные свойства и методы:

`var title: String?` — заголовок окна.

`var message: String?` — текст, выводимый ниже заголовка.

`var preferedStyle: UIAlertControllerStyle` — стиль всплывающего окна.

`func addAction( UIAlertAction )` — создание кнопки на всплывающем окне.

Пример

```
let alertController = UIAlertController(title: "Alert", message: "Text
of alert", preferredStyle: UIAlertControllerStyle.alert).
```

Аргумент `preferedStyle` инициализатора класса, имеющий тип `UIAlert-Controller.Style`, определяет стиль всплывающего окна. Именно значение данного параметра указывает на то, оповещение какого типа будет создано: `Alert` или `Action Sheet`.

СИНТАКСИС

Перечисление `UIAlertController.Style`

Определяет тип всплывающего окна. Входит в состав типа `UIAlertController`.

Доступные свойства:

`alert` — всплывающее окно типа Alert.

`actionSheet` — всплывающее окно типа Action Sheet.

Пример

```
UIAlertController.Style.actionSheet
```

Реализуем всплывающее окно в коде нашего приложения. В методе `showMessage()` создадим экземпляр класса `UIAlertController` и запишем его в константу `alertController` (листинг 38.7).

Листинг 38.7

```
@IBAction func showMessage(){
     let alertController = UIAlertController(
        title: "Welcome to MyName App",
        message: "Vasiliy Usov",
        preferredStyle: UIAlertController.Style.alert)
}
```

Данные, хранящиеся в `alertController`, являются не чем иным, как экземпляром класса `UIAlertController`. При выполнении метода `showMessage()` всплывающее окно не отобразится. Для того чтобы графический элемент был показан поверх сцены, используется метод `present(_:animated:competition)`, входящий в состав класса `UIViewController`. Он должен быть применен не к всплывающему окну, а к самому отображению сцены, а экземпляр класса `UIAlertController` передается ему в качестве аргумента.

СИНТАКСИС

Метод `UIViewController.present(_:animated:competition)`

Предназначен для показа отображения графического элемента поверх экземпляра класса `UIViewController` в виде модального окна. Данный метод должен быть вызван после внесения всех изменений в выводимое отображение.

Аргументы:

`_: UIViewController` — отображение, которое будет показано на экране устройства.

`animated: Bool` — флаг, указывающий на необходимость использования анимации при отображении элемента.

`competition: (() -> Void)? = nil` — замыкание, исполняемое после завершения показа элемента.

Пример

```
self.present(alertController, animated: true, completion: nil)
```

Добавим вывод всплывающего окна с помощью метода `present(_:animated:competition)` (листинг 38.8).

Листинг 38.8

```
@IBAction func showMessage(){
    let alertController = UIAlertController(
        title: "Welcome to MyName App",
        message: "Vasiliy Usov",
        preferredStyle: UIAlertController.Style.alert)
    // вывод всплывающего окна
    self.present(alertController, animated: true, completion: nil)
}
```

Запустите программу и щелкните на кнопке «Hello World». При этом будет произведен вывод оповещения (рис. 38.24).

Добавим к всплывающему окну пару кнопок. Для этого используется метод `addAction(_:)` класса `UIAlertController`.

СИНТАКСИС

Метод UIAlertController.addAction(_:)

Предназначен для добавления во всплывающее окно функциональных элементов (кнопок).

Аргументы:

`_: UIAlertAction` — экземпляр, описывающий функциональный элемент.

Пример:

```
alertController.addAction(
  UIAlertAction(
    title: "OK",
    style: UIAlertAction.Style.default,
    handler: nil))
```

Рис. 38.24. UIAlertController типа Alert без функциональных кнопок

Аргумент типа `UIAlertAction` в методе `addAction(_:)` определяет и описывает функциональную кнопку.

СИНТАКСИС

Класс UIAlertAction

Создает функциональную кнопку и определяет ее текст, стиль и реакцию на нажатие.

Инициализаторы:

```
init(title: String?, style: UIAlertActionStyle, handler:
((UIAlertAction) -> Void)? = nil)
```

Доступные свойства и методы:

`var title: String?` — текст, расположенный на кнопке.

`var style: UIAlertActionStyle` — стиль кнопки.

`var handler: ((UIAlertAction) -> Void)? = nil` — обработчик нажатия кнопки.

Пример

```
UIAlertAction(
  title: «Text of title»,
  style: UIAlertAction.Style.default,
  handler: nil)
```

Аргумент `style` инициализатора класса имеет тип данных `UIAlertAction.Style` и определяет внешний вид (стиль) кнопки.

СИНТАКСИС

Перечисление `UIAlertAction.Style`

Определяет внешний вид функционального элемента (кнопки) во всплывающем окне. Входит в состав типа `UIAlertAction`.

Доступные свойства:

`Default` — текст кнопки написан без выделения.

`Cancel` — текст кнопки написан жирным.

`Destructive` — текст кнопки написан красным.

Пример

```
UIAlertAction.Style.cancel
```

Мы рассмотрели все типы данных, методы и свойства, участвующие в процессе создания и отображения всплывающего окна. Добавим с помощью метода `addAction(_:)` (листинг 38.9) две кнопки в оповещение.

ПРИМЕЧАНИЕ Не забывайте, что все изменения с элементом, в том числе и создание кнопок, должны быть произведены до его вывода с помощью метода `presen t(_:animated:competition)`.

Листинг 38.9

```
@IBAction func showMessage(){
    let alertController = UIAlertController(
    title: "Welcome to MyName App",
    message: "Vasiliy Usov",
    preferredStyle: UIAlertController.Style.alert)
```

```
    // добавляем кнопки к всплывающему сообщению
    alertController.addAction(UIAlertAction(title: "First", style:
UIAlertAction.Style.default, handler: nil))
    alertController.addAction(UIAlertAction(title: "Second", style:
UIAlertAction.Style.default, handler: nil))
    // вывод всплывающего окна
    self.present(alertController, animated: true, completion: nil)
}
```

Теперь вы можете запустить приложение в эмуляторе и наслаждаться его работой. Модальное окно будет отображаться при нажатии кнопки на сцене и закрываться после нажатия любой из его кнопок (рис. 38.25).

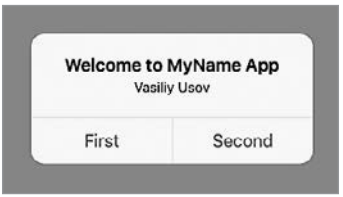

Рис. 38.25. Вывод модального окна с двумя кнопками

Все методы, имеющие связь с каким-либо графическим элементом в редакторе кода, отмечаются с помощью серого кружка вместо номера строки (рис. 38.26). Нажав на него, вы сможете осуществить переход в IB к элементу сцены.

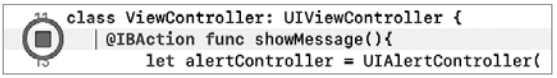

Рис. 38.26. Указатель на связь кодовой конструкции и графического элемента

38.8. Изменение атрибутов кнопки

Каждый графический элемент, который может быть расположен на сцене, имеет набор атрибутов, доступных для редактирования. С их помощью можно изменять размер, шрифт, цвет текста, цвет фона кнопки и многие другие параметры.

Для доступа к атрибутам, а также их редактирования служит Attribute Inspector (Инспектор атрибутов), который можно отобразить с помощью кнопки Show the Attribute Inspector , расположенной в верхней части Utilities Area (рис. 38.27).

С помощью Attribute Inspector можно изменить некоторые свойства кнопки:

- Цвет текста кнопки — на красный.
- Стиль текста — на жирный.
- Размер текста — на значение «20».

Для этого в IB выделите кнопку на сцене и включите Панель атрибутов. За цвет текста отвечает атрибут Text Color, находящийся в разделе Button. Вы можете изменить его значение на любой другой цвет, щелкнув по синему прямоугольнику и выбрав Custom в появившемся списке, после чего отобразится всплывающее окно с палитрой доступных цветов (рис. 38.28). Выберите в палитре красный цвет, после чего закройте окно палитры.

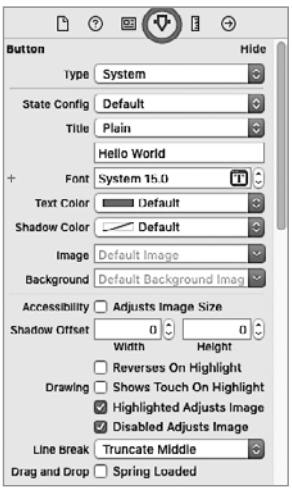

Рис. 38.27. Инспектор атрибутов элемента «Кнопка»

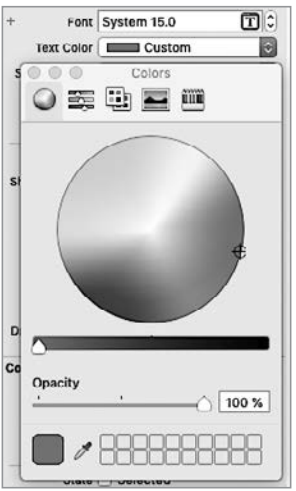

Рис. 38.28. Смена цвета текста кнопки

Настройки шрифта определены в атрибуте Font. Если нажать на иконку с буквой T, то в появившемся окне вы сможете изменить шрифт, стиль его начертания и размер (рис. 38.29). Для установки жирного шрифта выберите Bold в качестве значения параметра Style. Размер текста можно изменить в поле Size: установите его равным 20.

Теперь кнопка в вашей программе выглядит иначе (рис. 38.30), при этом ее функциональность совершенно не изменилась — она по-прежнему вызывает всплывающее окно.

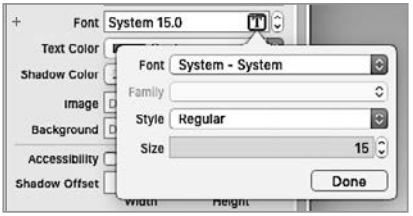

Рис. 38.29. Смена настроек шрифта текста кнопки

ПРИМЕЧАНИЕ Возможно, что после проведения настроек текст на вашей кнопке перестал в нее помещаться, а вместо некоторых букв появились точки. Чтобы это исправить, просто выделите кнопку в IB и растяните нужным вам образом, потянув за уголок или грань.

Рис. 38.30. Измененная кнопка в приложении

С помощью атрибутов вы можете изменять внешний вид ваших программ. Их возможности значительно шире, чем смена шрифта и цвета. Я советую вам самостоятельно протестировать смену значений различных атрибутов.

ЗАДАНИЕ

Сейчас вам потребуется создать на сцене несколько новых элементов. Разместите две кнопки (элемент `Button`) и надпись/метку (элемент `Label` в библиотеке объектов) так, как показано на рис. 38.31. Кнопки должны иметь текст «Left» и «Right», а метка — «Press some button». Элементы расположите в одну строку.

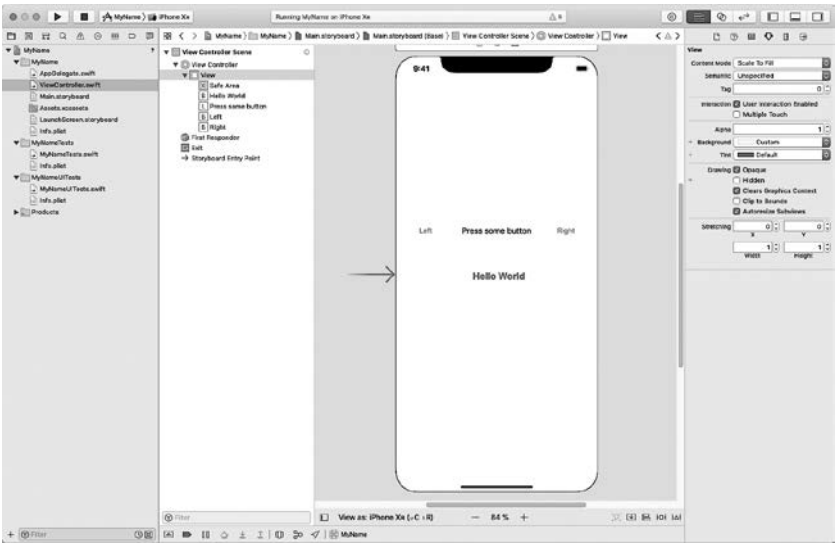

Рис. 38.31. Новые элементы на форме

ПОДСКАЗКА 1

Для поиска элементов в Object Library вы можете использовать поле Filter, вводя в него Button и Label. После этого достаточно просто перетащить нужные элементы на сцену.

ПОДСКАЗКА 2

Изменить текст элементов Button и Label можно двойным щелчком мышки по каждому из них либо с помощью инспектора атрибутов: параметр Text для элемента Label и параметр Title для Button.

38.9. Доступ кода к UI. Определитель типа @IBOutlet

Вы уже знаете, как обеспечить доступ элементов сцены к методам класса (если кто забыл — с помощью определителя `@IBAction`). Для осуществления обратной операции (доступ к элементу из кода) служит ключевое слово `@IBOutlet`, называемое аутлетом.

Сейчас мы рассмотрим пример его использования: запрограммируем изменение текста метки, размещенной на сцене, по нажатии на кнопки Left и Right.

Измените режим отображения Project Editor на Assistant Editor, нажав кнопку Show the Assistant Editor на панели инструментов (Toolbar). После этого Project Editor будет разделен на две рабочие зоны. Вы можете изменять их размеры и подстраивать удобным для вас способом, а также скрывать другие панели.

ЗАДАНИЕ

С помощью панели Jump Bar, расположенной выше Project Editor, организуйте отображение Main.storyboard в левой зоне, а ViewController.swift — в правой (рис. 38.32).

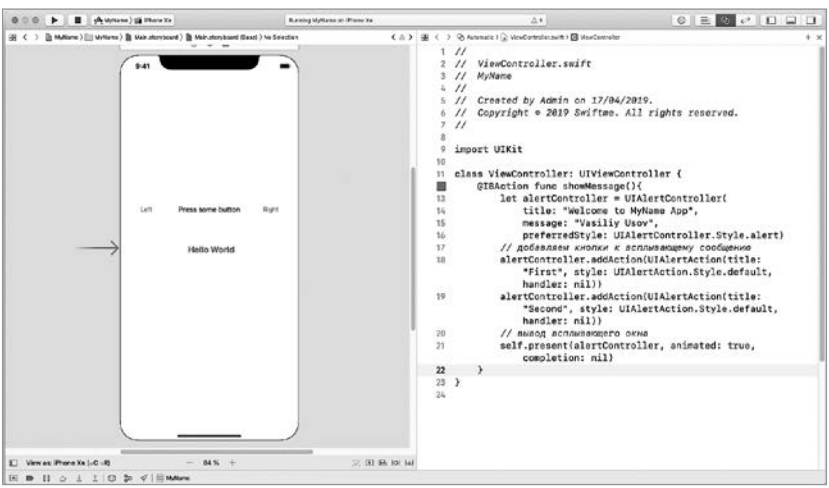

Рис. 38.32. Assistant Editor в действии

Режим Assistant Editor предоставляет очень удобный способ создания экшенов и аутлетов. С зажатой клавишей Ctrl в IB выделите элемент Label, после чего перетащите его в соседнюю зону, в которой открыт файл

ViewController.swift. Расположите появившуюся синюю линию выше метода `showMessage()`. При этом должна быть надпись «Insert Outlet or Outlet Collection» (рис. 38.33). После того как вы отпустите кнопку мышки, отобразится окно создания Outlet (рис. 38.34). Поле Connection определяет тип создаваемой связи. Выберите Outlet в качестве его значения. В поле Name укажите «`label`». Не изменяя значения остальных полей, нажмите кнопку Connect. После этого у класса `ViewController` будет создано новое свойство `Label` (листинг 38.10).

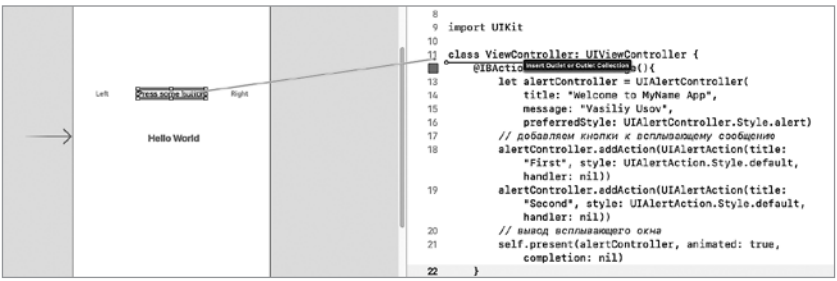

Рис. 38.33. Создание связи между кодом и элементом

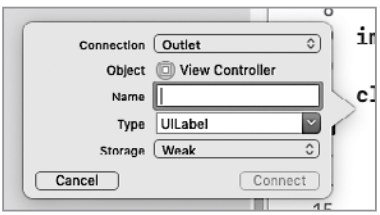

Рис. 38.34. Окно создания Outlet

ПРИМЕЧАНИЕ В данном случае поле Connection не содержит значения Action. Это связано с тем, что Action в принципе не может быть создан для элемента Label. Если вы проделаете ту же операцию с кнопкой, то увидите Action в качестве доступного для выбора значения.

Листинг 38.10

```
class ViewController: UIViewController {
    @IBOutlet weak var label: UILabel!
    @IBAction func showMessage(){
    //...
    }
}
```

Свойство `label` содержит определитель типа `@IBOutlet`. Слева от него отображена пиктограмма в виде серого круга, указывающая на то, что свойство имеет настроенную связь с элементом. Если навести на свойство указатель мышки, то связанный с ним элемент будет подсвечен прямо на сцене в соседней рабочей зоне (речь, конечно же, идет о работе в режиме Assistant Editor) (рис. 38.35).

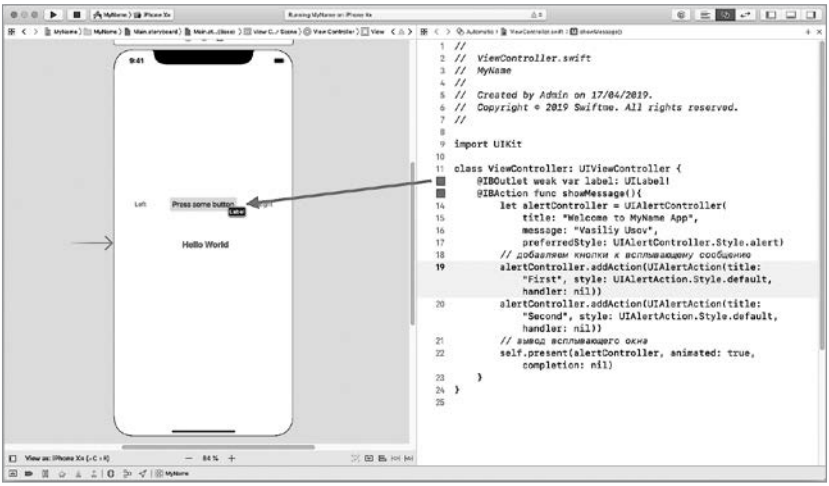

Рис. 38.35. Отображение связи между элементом и аутлетом

Следующим шагом в реализации функции смены текста метки станет создание action-метода `changeLabelText()` (метода с определителем `@IBAction`). Внутри своей реализации он будет использовать аутлет `label` для доступа к элементу Label и изменения свойства, отвечающего за его текст. На этот раз не станем создавать отдельный метод для каждой из кнопок — Xcode позволяет создать связь одного action-метода сразу с несколькими элементами на сцене.

Вполне возможно, у вас возник вопрос: как внутри метода будет определяться, какая из кнопок нажата? Дело в том, что метод, помеченный определителем `@IBAction`, позволяет использовать входной аргумент `sender`, содержащий ссылку на тот элемент, который вызвал данный метод. В качестве типа этого аргумента можно указать как какой-то конкретный (как, например, `UIButton`), так и `AnyObject`, позволяющий связать данный метод с любым типом элементов на сцене.

Вы уже умеете создавать action-методы вручную. Сейчас рассмотрим более удобный способ с использованием возможностей Assistant Editor.

Ранее вы создавали аутлет с помощью перетаскивания элемента Label с зажатой клавишей Ctrl в область редактора кода. Удерживайте нажатой клавишу Ctrl, и перетащите кнопку Right прямо под созданное ранее свойство label. В появившемся окне в поле Connection вы сможете выбрать значение Action (рис. 38.36). В качестве имени укажите «changeLabelText», а в поле Type — `UIButton` (так как использовать данный метод будут исключительно кнопки). Поле Event позволяет указать тип события, по которому будет вызван данный Action, оставьте его без изменений. Обратите внимание на поле Arguments: если выбрать None, то создаваемый метод не будет иметь каких-либо входных аргументов, и мы не сможем определить, какая кнопка нажата. Для того чтобы action-метод мог обратиться к вызвавшему его элементу, в данном поле необходимо указать Sender. После нажатия кнопки Connect в редакторе кода появится новый метод `changeLabelText(_:)` с определителем типа `@IBAction` (листинг 38.11).

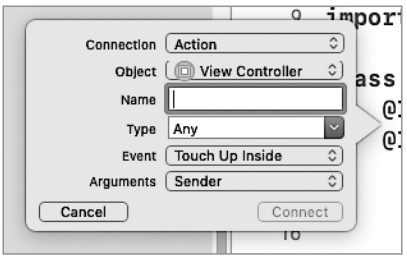

Рис. 38.36. Окно создания Action

Листинг 38.11

```
class ViewController: UIViewController {
   //...
   @IBAction func changeLabelText(_ sender: UIButton) {
   }
   //...
}
```

Внутри созданного метода вы можете обратиться к свойству `label` для доступа к элементу Label на сцене с целью изменения его параметров. Реализуем в методе функционал, изменяющий текст метки, используя при этом текст самой кнопки (листинг 38.12).

Листинг 38.12

```
@IBAction func changeLabelText(_ sender: UIButton) {
      label.text = ("The \(sender.titleLabel!.text!.lowercased())
button was pressed")
}
```

Аутлет-свойство `label` хранит в себе экземпляр класса `UILabel`, соответствующий элементу Label, расположенному на сцене, поэтому мы используем его для изменения текста метки (свойство text).

Рассмотрим подробнее выражение `sender.titleLabel!.text!.lowercased()`. Параметр `sender` позволяет обратиться к экземпляру класса `UIButton`, который соответствует той кнопке на сцене, которая была использована для вызова метода `changeLabelText(_:)` и передана в качестве входного аргумента. Свойство `titleLabel` возвращает опциональный экземпляр класса `UILabel`, который как раз и содержит описание текста кнопки. Для доступа к самому тексту используется свойство `text`, после чего происходит его преобразование к нижнему регистру с помощью метода `lowercased()`. Вот такая запутанная и одновременно простая цепочка.

Самое интересное в том, что созданный метод может быть использован в том числе и для кнопки Left, при этом свойство `sender` будет корректно отрабатывать и ее. Для создания связи между action-методом `changeLabelText(_:)` и кнопкой Left в режиме Assistant Editor достаточно нажать кнопку мыши на сером квадрате левее метода `changeLabelText(_:)` и перетащить его на элемент сцены (рис. 38.37). Теперь, если запустить программу и нажать на любую из двух кнопок, то текст метки будет изменяться.

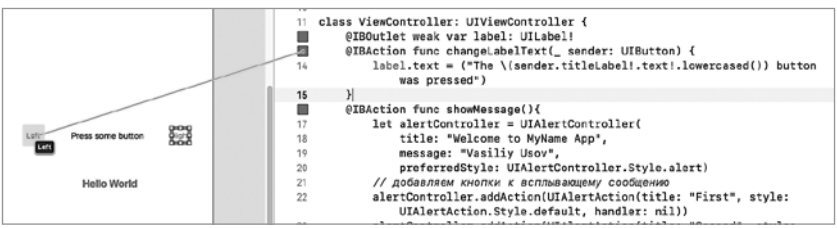

Рис. 38.37. Создание дополнительной связи action-метода и Button

Возможно, что при изменении текста элемент не будет соответствовать размерам метки и не отобразится полностью. В этом случае необходимо растянуть элемент Label во всю ширину сцены и изменить свойство `Alignment` в инспекторе атрибутов на «выравнивание по центру».

Таким образом, мы разобрали с вами основы создания простейшего пользовательского интерфейса и связывания между собой кода и графических элементов. Я советую вам дальше самостоятельно тренироваться использовать другие объекты библиотеки `UIKit`, активно подключая официальную справку от Apple.

39 Паттерны проектирования при разработке в Xcode

Мы близки к завершению нашего курса. Вам осталось познакомиться всего с одной темой, прежде чем вы самостоятельно продолжите свой путь в качестве iOS-разработчика.

В своей практике разработчикам часто приходится решать типовые задачи, для реализации которых уже найдены оптимальные пути и шаблоны. Они называются паттернами проектирования. Благодаря паттернам программисты могут, что называется, не изобретать велосипед, а использовать опыт сотен тысяч или даже миллионов людей. Говоря другими словами, шаблоны проектирования формализуют и описывают решение типовых задач программирования.

В настоящее время существует большое количество книг, описывающих огромное число всевозможных паттернов, часть из которых важно знать и понимать при разработке приложений под iOS и macOS. Если вы занимались программированием ранее, то наверняка слово «паттерны» неоднократно вам встречалось, а может, вы даже использовали некоторые из них в своей работе.

В этой главе приведены несколько основных паттернов, с которыми вы встретитесь в ходе создания приложений в Xcode.

39.1. Паттерн MVC. Фреймворк Cocoa Touch

MVC расшифровывается как Model-View-Controller (Модель-Отображение-Контроллер) и является основой построения программных продуктов в среде Xcode. Этот паттерн предполагает полное разделение кода, данных и внешнего вида приложения друг от друга. Каждый из этих элементов создается и управляется отдельно. Сегодня большинство объектно-ориентированных библиотек построены с учетом MVC.

Как говорилось ранее, данный шаблон подразумевает разделение всех составляющих разработки на три категории:

- **Модель** — классы, которые обеспечивают хранение данных ваших приложений.
- **Отображение (Представление)** — позволяют создать различные графические элементы, которые видит пользователь при работе с приложением.
- **Контроллер** — обеспечивает совместную работу «отображения» и «модели». Данный блок содержит логику работы приложения.

Каждый объект, который вы создаете в своей программе, может быть легко отнесен к одной из категорий, но при этом не должен реализовать какие-либо функции, присущие двум другим. Например, экземпляр класса `UIButton`, обеспечивающий отображение кнопки, не должен содержать код, выполняемый при нажатии на нее, а код, производящий работу с базой аккаунтов, не должен рисовать таблицу на экране смартфона.

MVC позволяет достичь максимального разделения трех основных категорий, что впоследствии позволяет удобно обновлять и перерабатывать программу, а также повторно использовать отдельные компоненты. Так, например, класс, который обеспечивает отображение кнопки, без труда может быть дополнен, расширен и многократно использован в любых приложениях.

Все возможности по разработке iOS-приложений предоставляет iOS SDK (software development kit, комплект средств разработки), который входит в состав Xcode. Данный SDK предоставляет огромное число ресурсов, благодаря которым вы можете строить UI, организовывать мультитач-управление, хранение данных в БД, работу с мультимедиа, передачу данных по сети, использование функций устройств (акселерометр и т. д.) и многое-многое другое. В состав iOS SDK входит фреймворк `Cocoa Touch`, который как раз построен на принципе MVC.

Во время разработки приложений в Xcode вы работали с категорией «Отображения» с помощью Interface Builder, но при этом не обеспечивали решения каких-либо бизнес-процессов с помощью графических элементов.

Категория «Контроллер» включает в себя специфические классы, обеспечивающие функциональность ваших приложений, например UIViewController, а точнее, его потомок ViewController, с которым вы работали ранее.

Элементы категории «Модель» не были рассмотрены в книге, тем не менее в будущем вы станете создавать механизмы, обеспечивающие хранение данных ваших приложений.

Чем глубже вы будете погружаться в разработку приложений для iOS, тем лучше и яснее вы будете видеть реализацию всех принципов паттерна MVC.

39.2. Паттерн Singleton. Класс UIApplication

Глобальные переменные могут стать значительной проблемой при разработке программ с использованием ООП. Они привязывают классы к их контексту, и повторное использование этих классов в других проектах становится просто невозможным. Если в классе используется глобальная переменная, то его невозможно извлечь из одного приложения и применить в другом, не убедившись, что в новом проекте используется в точности такой же набор глобальных переменных.

Несмотря на то что глобальные переменные — очень удобный способ хранения информации, доступной всем классам, их использование приносит больше проблем, чем пользы.

В хорошо спроектированных системах внешние относительно классов параметры обычно передаются в виде входных аргументов для методов этого класса. При этом каждый класс сохраняет свою независимость от других. Тем не менее время от времени возникает необходимость использовать общие для некоторых классов ресурсы.

Предположим, в вашей программе требуется хранить набор параметров в константе `prefences`, который должен быть доступен любому классу в рамках приложения. Одним из выходов является объявление этого параметра в виде глобального и его использование внутри классов, но, как мы говорили ранее, такой способ не является правильным. Другим способом решения этой задачи может служить использование паттерна Singleton.

Шаблон проектирования Singleton подразумевает существование только одного экземпляра класса, который может быть использован в любом другом контексте. Для того чтобы обеспечить это требование, в классе создается единая точка доступа к экземпляру этого класса. Так, например, мог бы существовать класс `Prefences`, для доступа к экземпляру которого мы могли бы использовать метод `Prefences.`

`singleton()`. При попытке использования данного метода из любого другого класса будет возвращен один и тот же экземпляр.

Примером использования паттерна Singleton при разработке под iOS может служить класс `UIApplication`, экземпляр которого является стартовой точкой каждого приложения. Любое приложение, которое вы создаете, содержит в себе и использует только один экземпляр класса `UIApplication`, доступ к которому обеспечивается с помощью шаблона `Singleton`. `UIApplication` выполняет большое количество задач, в том числе обеспечивает вывод на экран устройства окна вашего приложения (экземпляр класса `UIWindow`) и отображение в нем стартовой сцены. Вам, как разработчику, никогда не придется самостоятельно создавать экземпляр класса `UIApplication`, система делает это автоматически, независимо от кода приложения, и постоянно работает фоном.

ПРИМЕЧАНИЕ Мы еще никогда не сталкивались с классом `UIWindow`. Он создается автоматически и обеспечивает отображение UI ваших приложений. Мобильные программы обычно имеют только один экземпляр класса `UIWindow`, так как одновременно отображают только одно окно (исключением является подключение внешнего дисплея), в отличие от программ настольных компьютеров, которые могут отображать несколько окон одной программы в один момент времени.

39.3. Паттерн Delegation. Класс UIApplicationDelegate

Паттерн Delegation (делегирование) является еще одним очень важным для iOS-разработчика паттерном. И знание о нем понадобится вам при создании приложений в Xcode. Его суть состоит в том, чтобы один класс делегировал (передавал) ответственность за выполнение некоторых функций другому классу. Со стороны это выглядит так, словно главный класс самостоятельно выполняет все возложенные на него функции (даже делегированные другому классу). Фреймворк `Cocoa Touch` очень активно использует делегаты в своей работе, чтобы одни объекты выполняли часть своих функций от лица других.

Ярким примером использования паттерна делегирования является уже знакомый нам класс `UIApplication`. Напомню, что каждое приложение имеет один-единственный экземпляр этого класса-синглтона. Во время работы приложения `UIApplication` вызывает некоторые специфические методы своих делегатов, если, конечно, делегаты существуют и реализуют в себе эти методы.

Откройте в Xcode ранее разработанный нами проект MyName. Мы рассмотрели еще не все ресурсы, доступные разработчику в Project Navigator. Одним из них является AppDelegate.swift. Откройте его в редакторе проекта и взгляните на строку инициализации класса AppDelegate (листинг 39.1).

Листинг 39.1

```
class AppDelegate: UIResponder, UIApplicationDelegate {
  //...
}
```

AppDelegate является делегатом для `UIApplication`, об этом говорит протокол `UIApplicationDelegate`, который принимает данный класс. Во время выполнения программы экземпляр класса `UIApplication` в определенные моменты вызывает методы, описанные в AppDelegate.

Заключение

Итак, цели, которые мы ставили перед собой во время чтения книги, достигнуты — вы прошли полноценный курс подготовки к разработке приложений на Swift. Конечно, пока вы не обладаете всеми знаниями в предметной области, но теперь у вас есть хороший задел для изучения материала в будущем.

Я уверен, что ваши знания стали намного глубже и вы прониклись этим необыкновенным духом разработки, который подарила нам Apple. Сейчас пришло время реализации собственных идей и разработки приложений.

Многие из вас задаются вопросом: «А что дальше?»

Вы можете двигаться по одному из двух путей:

- Продолжить изучение по другим курсам/книгам. Правда, в этом случае вам может потребоваться знание английского языка, так как в Рунете все еще не так много материалов для начинающих iOS-разработчиков.
- Найти для себя интересную задачу и двигаться вперед, используя официальную документацию (там есть ответы на все вопросы), данную книгу, сайт http://swiftme.ru и другие ресурсы в Сети.

Создавайте и творите!

Вы сделали первый и поэтому самый важный шаг — дальше все будет только интереснее.

Василий Усов

Swift. Основы разработки приложений под iOS, iPadOS и macOS. 5-е изд., дополненное и переработанное

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*А. Сизова*
Художественный редактор	*С. Заматевская*
Корректор	*С. Беляева*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 08.2019.

Наименование: книжная продукция.

Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

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

Подписано в печать 21.08.19. Формат 60×90/16. Бумага офсетная.
Усл. п. л. 31,000. Тираж 1000. Заказ 0000.