ВАСИЛИЙ УСОВ

SWIFT

ОСНОВЫ РАЗРАБОТКИ ПРИЛОЖЕНИЙ под iOS, iPadOS и macOS

6 ИЗДАНИЕ

ВАСИЛИЙ УСОВ

SWIFT

ОСНОВЫ РАЗРАБОТКИ ПРИЛОЖЕНИЙ
под iOS, iPadOS и macOS

6-Е ИЗДАНИЕ

ПИТЕР®

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

2021

ББК 32.973.2-018.1
УДК 004.438
У76

Усов В.

У76 Swift. Основы разработки приложений под iOS, iPadOS и macOS. 6-е изд. дополненное и переработанное. — СПб.: Питер, 2021. — 544 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1796-3

Мечтаете стать iOS-разработчиком, написать собственное приложение и работать в крутой компании? Тогда эта книга для вас!

Язык Swift прост, понятен и отлично подойдет как новичкам, так и опытным программистам. Чтобы начать писать код, вам потребуются только эта книга, компьютер и желание учиться. Все базовые концепции программирования и основы синтаксиса объясняются доступным языком, поэтому если вы никогда раньше не занимались разработкой, то эта книга — отличный старт. Теория чередуется с практическими примерами и кодом — так вы сразу сможете связать абстрактные понятия с реальными ситуациями. В каждой главе вас ждут тесты и домашние задания, которые помогут закрепить материал.

А еще Swift — это дружелюбное сообщество в Telegram, где можно обсуждать проекты и получать поддержку.

Учитесь, создавайте и творите свое будущее!

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

ББК 32.973.2-018.1
УДК 004.438

ISBN 978-5-4461-1796-3

Краткое содержание

Часть IV
ВВЕДЕНИЕ В РАЗРАБОТКУ ПРИЛОЖЕНИЙ

Часть V
НЕТРИВИАЛЬНЫЕ ВОЗМОЖНОСТИ SWIFT

Часть VI
ВВЕДЕНИЕ В МОБИЛЬНУЮ РАЗРАБОТКУ

Оглавление

Часть I
Базовые возможности Swift

Посвящается моим родным

Спасибо маме и папе за то, что всегда поддерживали мои интересы и подтолкнули меня к выбору пути, определившего всю мою жизнь.

Спасибо супруге за то, что рядом, за то, что стала моим вдохновением.

Спасибо сестре и брату за то, что были со мной всю мою жизнь и всегда искренне радовались моим успехам.

Спасибо небольшой команде начинающих и состоявшихся разработчиков и IT-специалистов. Без вас книга не была бы такой, какая она есть:

Тимуру — @tima5959

Александру — @mosariot

Виктору — @krylwte

Рустаму — @Zed_Null

Денису — @g01dt00th

Игорю — @iAryslanov

Олегу — @GodOfSwift

Читателю

Привет! Меня зовут Василий Усов. Я автор учебных материалов по разработке на Swift под iOS и macOS. Прошло уже более пяти лет с написания первой книги по Swift, но ежегодно я обновляю и дополняю ее, описывая нововведения языка, и корректирую в соответствии с вашими рекомендациями и просьбами.

В этой книге я ставлю себе две основные задачи: популяризировать разработку на Swift под платформу Apple, а также найти наиболее эффективные способы и подходы к обучению. Благодаря вам я вижу, что мой многолетний труд не проходит впустую и приносит свои плоды. Многие из тех, кто купил прошлые издания и ввязался в эту вечную гонку за знаниями, уже разрабатывают свои приложения или нашли отличную работу, где могут полностью реализоваться.

> **ПРИМЕЧАНИЕ** Писать книги по программированию, как и любую другую техническую литературу, — дело совершенно неприбыльное и отнимающее просто гигантское количество личного времени. Каждое издание книги пишется не менее 5 месяцев, а основная часть прибыли достается магазинам, дистрибьюторам и издателям, оставляя автору не более 5 %.
>
> Если вы скачали книгу в Сети, но нашли ее очень полезной, то я буду благодарен, если вы внесете вклад в дальнейшее развитие проекта, совершив покупку на сайте https://swiftme.ru. С 2015 года и по сегодняшний день мои книги продаются только благодаря вашей поддержке.

Я рад каждому, кто зарегистрировался на сайте или добавился в Telegram-канал. Неважно, сколько вам лет: 15 или 40. Желание разрабатывать приложения уже само по себе бесценно, так как именно оно приведет вас к тому, чтобы создать «то самое приложение», которое совершит очередную революцию.

Если вы еще учитесь в школе, то это прекрасно, что поздно вечером вы не заставляете своих родителей переживать, а мирно сидите перед монитором с этой книгой в руках и изучаете невероятный язык Swift. Я также уверен, что вы не против купить новую модель iPhone, на который можно заработать, создав свое собственное приложение. И вот что я скажу: у вас все получится!

Если вы уже учитесь в институте, то Swift — это то, что будет держать ваш ум в тонусе. Помните, что многие из известных новаторов придумали и реализовали свои удивительные идеи именно в вашем возрасте. Сейчас вы находитесь на пике активности, воспользуйтесь этим!

Если вы старше, то наверняка у вас созрела гениальная идея, для реализации которой вы и пришли в разработку. Swift — это именно тот инструмент, который вам необходим. Главное, погрузившись в разработку, не забывайте о своих близких, хотя уверен, что они с полным пониманием отнесутся к вашему начинанию.

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

Присоединяйтесь к нам

Самые важные ссылки я собрал в одном месте. Обязательно посетите каждую из них, это поможет в изучении материала книги.

Сайт сообщества

https://swiftme.ru

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

Курс с видеоуроками, тестами, домашними заданиями и дополнительными материалами

https://swiftme.ru/course16

Материал книги будет усваиваться значительно лучше, если его закреплять просмотром видеоуроков и решением домашних заданий.

Мы в Telegram

https://swiftme.ru/telegramchat или https://t.me/usovswift

Если по ходу чтения книги у вас появились вопросы, то вы можете задать их в нашем чате в Telegram.

Опечатки книги

https://swiftme.ru/typo16

Здесь вы можете посмотреть перечень всех опечаток, а также оставить информацию о найденных вами и еще не отмеченных. Документ создан в Google Docs, для доступа нужен Google-аккаунт.

Листинги

https://swiftme.ru/listings16

Здесь вы можете найти большую часть листингов и проектов, создаваемых в книге.

Введение

На ежегодной конференции для разработчиков Apple WWDC (WorldWide Developers Conference) 2 июня 2014 года компания удивила iOS-общественность, представив новый язык программирования — Swift. Это стало большой неожиданностью: максимум, на что рассчитывали разработчики, — это обзор возможностей iOS 8 и новые API для работы с ними.

Начиная с 2014 года и по настоящий момент Swift активно развивается. Сегодня он стал основным языком разработки под платформу Apple, опередив все еще популярный язык Objective-C[1]. Swift — это язык программирования с открытым исходным кодом, а значит, он скоро будет поддерживаться и другими операционными системами[2].

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». Имейте в виду, что Swift затягивает и не отпускает!

Для кого написана книга

Ответьте для себя на следующие вопросы:

- Хотите ли вы научиться создавать программы под iOS, iPadOS, macOS, watchOS и tvOS?
- Вам больше нравится обучение в практической форме, чем скучные и монотонные теоретические лекции?

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

Материал в книге подкреплен практическими домашними заданиями. Мы вместе пройдем путь от самых простых понятий до решения наиболее интересных задач, которые заставят ваш мозг усердно работать.

---

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

[2] Приложения на Swift можно разрабатывать не только для операционных систем iOS, macOS, watchOS и tvOS, но и для Linux и Windows. Также есть решения, позволяющие писать на Swift и Server Side Code.

Не стоит бояться, Swift вовсе не отпугнет вас (как это мог бы 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 современных мобильных систем iOS и iPadOS, стационарной системы macOS или Linux, смарт-часов Apple Watch или телевизионной приставки Apple TV.

Что нужно знать, прежде чем начать читать

Единственное и самое важное требование — уметь работать с компьютером: скачивать, устанавливать и открывать программы, пользоваться мышью и клавиатурой, а также иметь общие навыки работы с операционной системой. Как видите, для начала надо 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.

Особенности 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 Swift, вы удивитесь, насколько он превосходит другие языки программирования, на которых вы раньше писали. Его простота, лаконичность и невероятные возможности просто поразительны!

Современность

Swift — это комбинация последних изысканий в области программирования и опыта, полученного в процессе работы по созданию продуктов экосистемы Apple. Более того, Swift имеет очень большое сообщество и открытый исходный код, что позволяет оперативно добавлять в него новые возможности.

Объектно-ориентированность

Swift — объектно-ориентированный язык программирования, придерживающийся парадигмы «всё — это объект». Если сейчас вы не поняли, что это значит, не переживайте: чуть позже мы еще к этому вернемся.

Читабельность, экономичность и лаконичность кода

Swift создан для того, чтобы быть удобным в работе и максимально понятным. У него простой и функциональный синтаксис, позволяющий вместо сложных многострочных выражений писать однострочные (а в некоторых случаях — односимвольные!).

Безопасность

В рамках Swift разработчики попытались создать современный язык, свободный от уязвимостей и не требующий от программиста лишних усилий в процессе создания приложений. Swift имеет строгую типизацию: в любой момент времени вы точно знаете, с объектом какого типа работаете. Более того, в процессе программирования вам практически не надо думать о расходе оперативной памяти, Swift все делает за вас в автоматическом режиме.

Производительность

Swift 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

Благодаря активно растущему сообществу разработчиков, штаб-квартирой которых стал портал swift.org, язык Swift поддерживает свою актуальность и стабильность работы. Для русскоязычного сообщества есть несколько профильных сайтов, одним из которых является swiftme.ru.

Все это делает Swift по-настоящему удивительным языком программирования.

О том, как работать с книгой

Использование смартфонов для решения возникающих задач стало нормой. Поэтому все больше компаний используют приложения для своего бизнеса и выделяют большие бюджеты на их обслуживание. Операционная система iOS является одной из популярнейших мобильных систем в мире, и в такой ситуации спрос на Swift-разработчиков растет необычайными темпами.

---

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

Книга является вводным курсом для всех желающих научиться программировать на замечательном языке Swift, чтобы создавать собственные приложения или устроиться на работу iOS-разработчиком.

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 он будет очень интересным.

Несмотря на то что книга предназначена в первую очередь для изучения синтаксиса и возможностей языка Swift, вы узнаете о принципах разработки полноценных приложений. Можете быть уверены, эта информация будет очень полезной. С этой книгой вы освоите новый язык и напишете свой первый проект.

Проекты студентов

https://swiftme.ru/projects

Здесь вы можете найти проекты некоторых студентов, которые уже прошли обучение по нашим книгам и курсам. При желании вы сможете разместить здесь и свои работы. Просто напишите об этом в чате в Telegram.

Код и проекты в книге написаны с использованием Swift 5.3, iOS 14, Xcode 12 beta и операционной системы macOS 10.15 Catalina. Если у вас более свежие версии программного обеспечения (например, macOS 11 Big Sur и стабильная версия Xcode 12), то вполне возможно, что показанный на скриншотах интерфейс будет немного отличаться от вашего. При этом весь приведенный программный код с большой долей вероятности будет работать без каких-либо правок. Тем не менее если вы встретитесь ситуацией, при которой интерфейс или код необходимо исправить, то прошу сообщить мне об этом одним из способов:

- Отметить в специальном электронном файле (https://swiftme.ru/typo16).
- Написать в Telegram-канале (https://swiftme.ru/telegramchat или https://t.me/usovswift).

О самостоятельной работе

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 поможет 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!

Листинги

https://swiftme.ru/listings16

Здесь вы можете найти большую часть листингов и проектов, создаваемых в книге.

О домашних заданиях

Задания для самостоятельной работы на сайте swiftme.ru. Все, что вам нужно сделать, — зайти на swiftme.ru, зарегистрироваться, после чего получить доступ к базе учебных материалов.

Курс с видеоуроками, тестами, домашними заданиями и дополнительными материалами

https://swiftme.ru/course16

Советую ознакомиться с сайтом прямо сейчас, так как я не буду ограничиваться одними лишь заданиями. Со временем там появятся дополнительные учебные видео, которые, без сомнения, упростят изучение Swift.

Заглядывайте в учебный раздел перед каждой главой книги — так вы будете в курсе всех имеющихся дополнительных материалов и заданий для самостоятельного решения.

Очень часто во время написания кода начинающие разработчики пользуются нехитрым приемом «копировать/вставить». Они копипастят все, начиная от решения домашних заданий и заканчивая найденными в Сети готовыми участками кода, решающими определенную проблему. Недостаток такого подхода в том, 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 достигнута.

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

Если у вас возникают проблемы с решением задания — обратитесь к нашему сообществу в Telegram.

Старайтесь решать задания самостоятельно, используя при этом помощь сообщества, книгу и другие справочные материалы. Но не ориентируйтесь на то, чтобы посмотреть (или правильнее — подсмотреть?) готовое решение.

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

Исправления в шестом издании

В результате долгого и плодотворного общения со многими из вас была выработана масса идей, благодаря которым новое издание стало еще более полезным. Огромное спасибо всем участникам каналов в Telegram — с вашей помощью книга становится лучше и интереснее.

По сравнению с предыдущим изданием эта книга содержит следующие изменения и дополнения:

- Формат книги изменен (теперь она больше и толще).
- Учебный материал актуализирован в соответствии со Swift 5.3 и Xcode 12.
- Переписаны и обновлены большинство глав в книге.
- Добавлен новый материал, который ранее не входил в книгу:
  - Пример использования фреймворка `SwiftUI`.
  - О выборе между классами и структурами.
  - О протокол-ориентированном программировании.
  - О числовом типе данных `Decimal`.
  - О ключевом слове `some`.
  - О принципах работы ARC и хранении value type и reference type в памяти.
  - О новых методах для работы с массивами.
- Добавлены разделы «Для чего это использовать», которые кратко показывают, для чего в реальных проектах могут быть использованы изученные возможности.
- Наборы (`set`) переименованы в множества.

Обновлены графические материалы (схемы, рисунки, графики и скриншоты).

- Исправлены найденные опечатки и учтены пожелания и замечания читателей по оформлению и содержанию.

Структура книги

Вы уже начали путешествие в мир Swift. Совсем скоро вы выполните первые обязательные шаги перед разработкой собственных приложений. Вы узнаете, как завести собственную учетную запись Apple ID, как подключиться к программе

Apple-разработчиков, где взять среду разработки Swift-приложений и как с ней работать.

Весь последующий материал книги разделен на шесть частей:

- **Часть I. Базовые возможности Swift**. После знакомства со средой разработки Xcode 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 II. 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 III. Основные возможности Swift.** Третья часть фокусируется на рассмотрении и изучении наиболее простых, но очень интересных средств Swift, позволяющих управлять ходом выполнения приложений.
- **Часть IV. Введение в разработку приложений**. Эта часть посвящена изучению основ среды разработки Xcode, а также созданию двух первых консольных приложений.
- **Часть V. Нетривиальные возможности Swift.** В пятой 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 здесь вас ждет большая практическая работа — создание первого интерактивного приложения в Xcode Playground.
- **Часть VI. Введение в iOS-разработку.** В конце долгого и увлекательного пути изучения языка и создания 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

```
Это примеры программного кода.
```

СИНТАКСИС

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

➤ Такие блоки содержат указания, которые вам необходимо выполнить.

О важности изучения английского языка

Сложно найти специальность, для которой знание английского языка было бы настолько важным, как для разработчика. Конечно, есть программисты, которые учатся исключительно по русским источникам, но в этом случае они сами себя ограничивают.

Для работы с официальной документацией и чтения самых новых и познавательных статей необходимо знание английского языка! И к сожалению, ситуация вряд ли изменится. Отечественные разработчики не спешат создавать новые и 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 не обязательно тратить тысячи рублей ежемесячно на занятия с преподавателем (особенно если у вас уже есть школьная или институтская база). Лично я изучаю и поддерживаю уровень языка с помощью следующих средств:

- Мобильные приложения. Duolingo, Simpler, «Полиглот», Lingualeo и многие другие. В каждом из них есть свои положительные стороны, и каждое из них позволяет с пользой проводить 20–30 минут 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 прилагаются аудиоматериалы (тренировка восприятия речи на слух).
- Учебные видео на YouTube. Начинающим я бы посоветовал канал «Английский по плейлистам».
- Общение с носителями языка по Skype. 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 мною английского языка!

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

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.

Подготовка к разработке Swift-приложений

ПРИМЕЧАНИЕ Если вы не прочитали раздел «О том, как работать с книгой», то вернитесь к нему. В нем даны важные рекомендации по работе с книгой и дополнительными учебными материалами.

Сегодня Swift может быть использован не только для разработки приложений под платформу от Apple, но и под Linux, а также для написания Server Side-приложений. Моя цель — ввести вас в разработку под iOS, iPad и macOS. Тем не менее вы можете использовать ее и просто для изучения синтаксиса и возможностей языка для создания приложений под другие платформы.

Какие варианты обучения доступны

В этом разделе книги описаны основные доступные сегодня варианты обучения Swift. В качестве источника знаний вы будете использовать эту книгу, курс с домашними заданиями (https://swiftme.ru/course16), а также дополнительные материалы из Сети. Но вся сложность в том, где именно вам писать программный код, какую среду разработки использовать.

На сегодняшний день доступны следующие варианты:

1. Использовать компьютер Mac с операционной системой macOS и средой разработки Xcode. Данный вариант является наиболее предпочтительным и будет подробно описываться в книге.

ПРИМЕЧАНИЕ У среды разработки Xcode есть конкурент — AppCode от JetBrains.

2. Использовать операционную систему Linux (например, Ubuntu), установив на нее пакет необходимых компонентов (https://swift.org/download), а также произвольный редактор кода с подсветкой синтаксиса Swift (один из вариантов — Sublime Text).
3. Использовать онлайн-редактор со встроенным компилятором Swift-кода (например, http://online.swiftplayground.run или один из его аналогов).

Каждый из указанных вариантов позволит вам попробовать Swift в деле, изучить его синтаксис и возможности, но если вы нацелены создавать приложения под платформу Apple, то вам необходим Mac, операционная система macOS и среда разработки Xcode (вариант № 1). Материал книги ориентирован именно на это.

> **ПРИМЕЧАНИЕ** О других способах изучения Swift и разработки собственных приложений, не имея при этом Mac, вы можете узнать в нашем чате в Telegram (https://swiftme.ru/telegramchat).

Подготовка к разработке на Mac

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 Catalina (10.15) или выше. 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. Под другие операционные системы ее не существует.

Если вы ранее никогда не работали с Xcode, то будете поражены широтой возможностей данной среды и необычным подходом к разработке приложений.

Зарегистрируйте учетную запись Apple ID

Следующим шагом станет 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).

> **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, используйте ее данные для входа в Центр разработчиков. Или нажмите кнопку Создайте сейчас и введите требуемые для регистрации данные.

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 обязательно, только когда решите опубликовать приложение в AppStore.

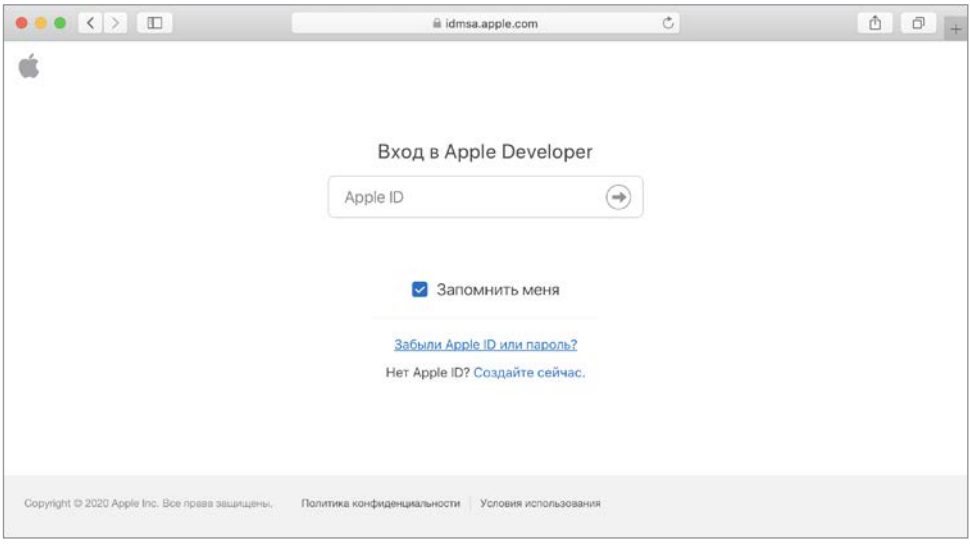

Рис. 1. Страница входа в Центр разработчиков

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

Скачайте и установите среду разработки Xcode

На стартовой странице Центра разработчиков необходимо перейти по ссылке Download Tools, расположенной в центральной части окна, после чего откроется страница, содержащая ссылки на основные ресурсы для разработчиков. В Центре разработчиков вы можете получить доступ к огромному количеству различной документации, видео, примеров кода — ко всему, что поможет создавать отличные приложения.

> **ПРИМЕЧАНИЕ** На момент написания книги Xcode 12 находился на этапе beta-тестирования. Именно beta-версия среды разработки была использована для учебного материала. С большой долей вероятности в данный момент на сайте разработчиков уже доступна стабильная версия Xcode 12, которую вам и следует использовать.

Чтобы скачать актуальную версию среды разработки Xcode, перейдите по ссылке Release, расположенной в верхнем правом углу страницы (рис. 2). После этого вы перейдете на страницу с релизными (стабильными) версиями программных средств. Найдите в списке Xcode, нажмите на кнопку View on the App Store, после чего произойдет переход в App Store. 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.

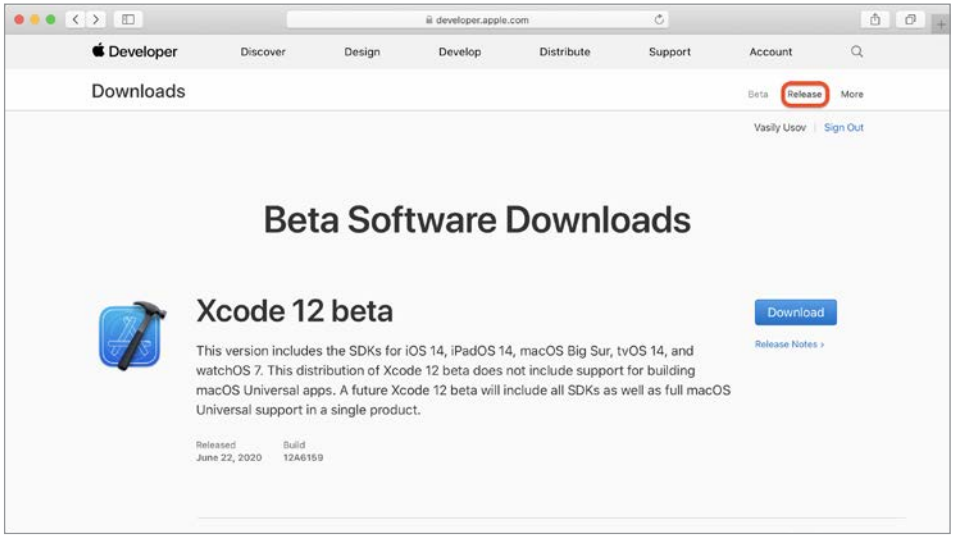

Рис. 2. Перечень ресурсов, доступных в Центре разработчиков

Обзор Xcode

Мы начнем изучение программирования на языке Swift со знакомства со средой разработки Xcode.

> **ПРИМЕЧАНИЕ** Интегрированная среда разработки (Integrated Development Environment, IDE) — набор программных средств, используемых 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 (и других продуктов Apple). Это наиболее важный инструмент, который использует разработчик, и он по-настоящему удивительный! Xcode предоставляет широкие возможности, и изучать их следует постепенно, исходя из поставленных и возникающих задач. Внешний вид рабочей среды приведен на рис. 3.

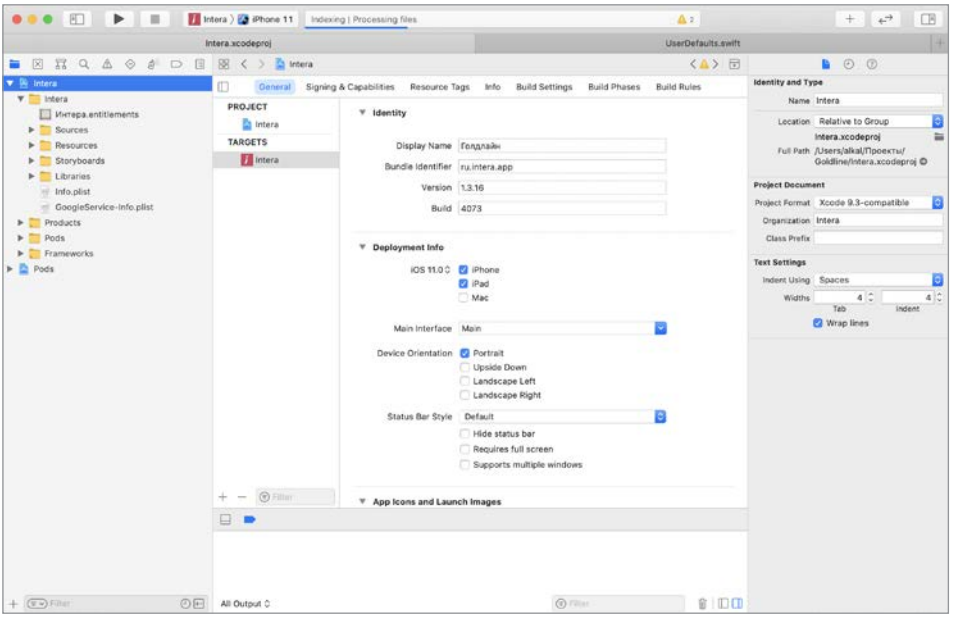

Рис. 3. Интерфейс Xcode

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

Xcode распространяется на бесплатной основе. Это многофункциональное 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). Подробно изучать состав и возможности данной IDe мы начнем непосредственно в процессе разработки приложений.

Я надеюсь, что вы уже имеете на своем компьютере последнюю версию Xcode, а значит, мы можем перейти к первому знакомству с этой 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. 4).

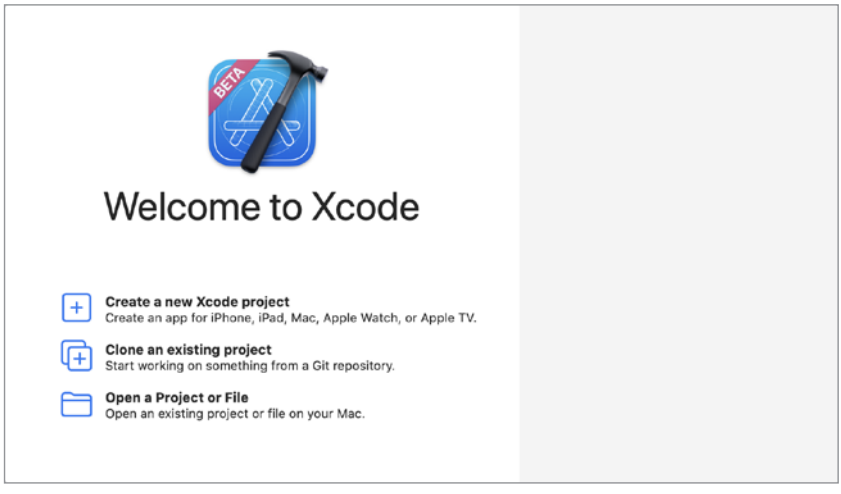

Рис. 4. Стартовое окно Xcode

Стартовое окно служит двум целям: созданию 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:

- Create a new Xcode project — создать новый проект.
- Clone an existing project — подключить репозиторий (при использовании системы контроля версий Git).
- Open a Project or File — открыть уже существующий проект.

Правая часть окна содержит список созданных ранее проектов. В вашем случае, если вы запускаете Xcode впервые, данный список будет пустым. Но 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 — 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 нее 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. 5.

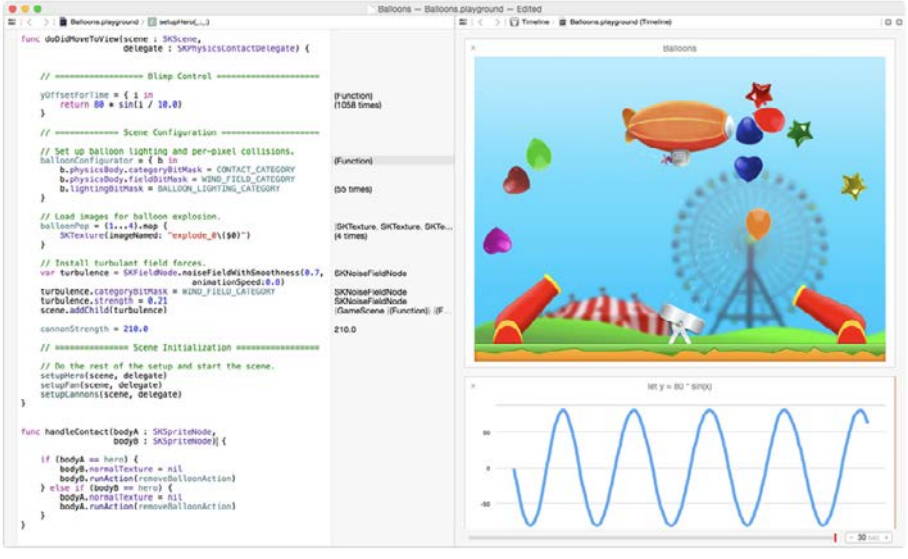

Рис. 5. Пример playground-проекта

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

Интерфейс playground-проекта

Нет способа лучше для изучения языка программирования, чем написание кода. Playground-проект предназначен именно для этого. Для создания нового playground-проекта выберите пункт главного меню File > New > Playground. Далее Xcode предложит вам выбрать типы создаваемого проекта (рис. 6), которые отличаются лишь предустановленным в проекте кодом.

> **ПРИМЕЧАНИЕ** Обратите внимание, что в верхней части окна есть возможность выбора платформы (iOS, tvOS, macOS).

Сейчас вам необходимо выбрать тип Blank, который содержит минимальное количество кода. После нажатия кнопки Next среда разработки попросит вас ввести имя создаваемого проекта. Измените имя на «Part 1 Basics» (или любое другое) и щелкните на кнопке Create. После этого откроется рабочий интерфейс созданного проекта (рис. 7).

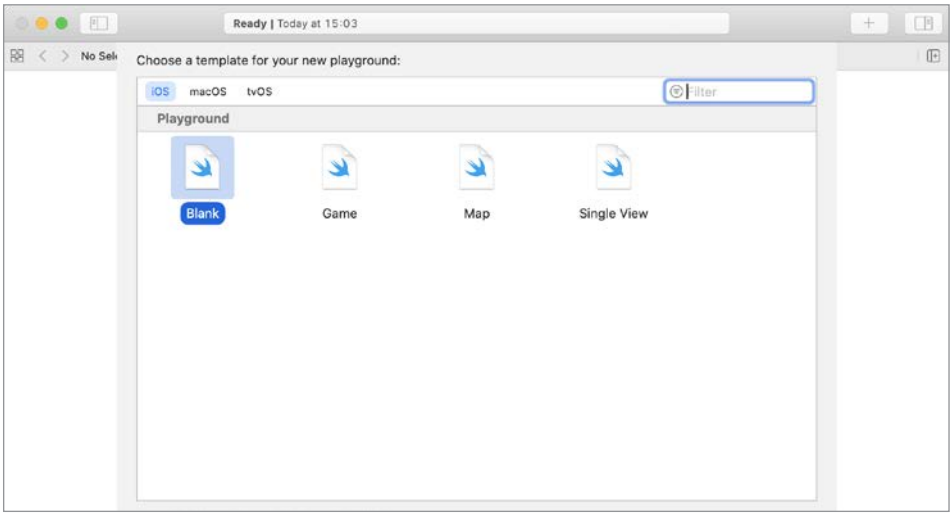

Рис. 6. Окно выбора типа создаваемого проекта

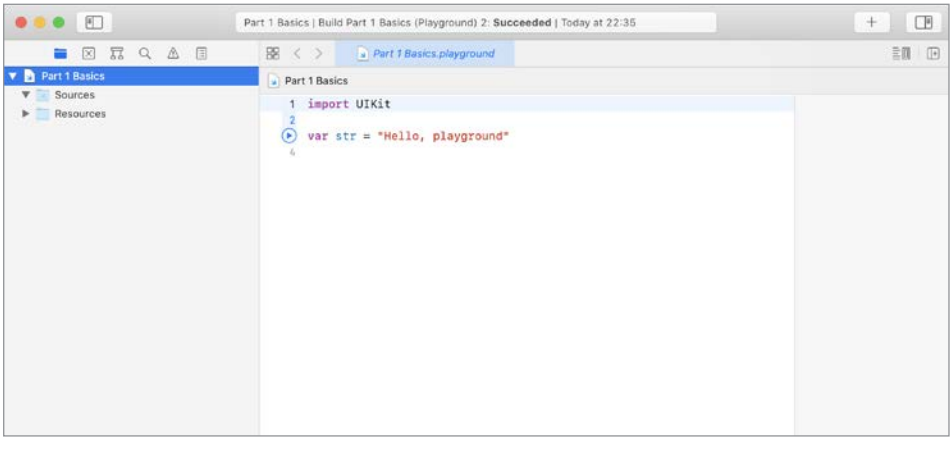

Рис. 7. Рабочий интерфейс playground-проекта

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

Как показано на рис. 7, рабочее окно состоит из четырех основных областей.

- *Toolbar*, или *панель инструментов*, расположенная в верхней части окна. Данная область содержит кнопки управления окном и строку состояния.
- *Navigators*, или *панель навигации*, расположенная в левой части окна. С ее помощью вы можете производить навигацию по различным ресурсам, входящим в состав playground-проекта.

В ближайшее время данная панель нам будет не нужна, а значит, мы можем скрыть ее.

➤ Нажмите на кнопку Hide or show the Navigator, расположенную в левой части Toolbar.

- *Code editor*, или *редактор кода*, расположенная в центральной части окна. В ней вы можете писать и редактировать программный код. В только что созданном проекте уже имеются две строки с кодом.
- *Results*, или *панель результатов*, расположенная в правой части окна. В ней будет отображаться результат выполнения кода, написанного в Code editor.

➤ Нажмите на кнопку Execute Playground, расположенную в нижней левой части Code editor.

Теперь на панели результатов отображается результат выполнения программного кода (рис. 8).

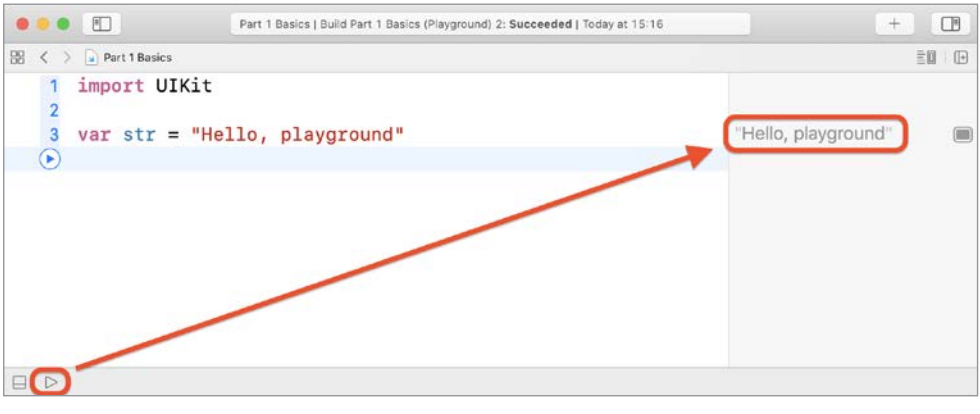

Рис. 8. Отображение значения в области результатов

ПРИМЕЧАНИЕ Для принудительного запуска обработки кода вы можете использовать либо кнопку с изображением стрелки слева от строчки кода, либо кнопку Execute Playground, расположенную в нижней части окна (рис. 9).

Сейчас вы можете видеть результат выполнения строки 3 (значение параметра `str`) в области результатов. В дальнейшем мы вместе будем писать код и обсуждать результаты его выполнения.

Если навести указатель мыши на строку `"Hello, playground"` в области результатов, то рядом появятся две кнопки, как показано на рис. 10.

Левая кнопка (изображение глаза) включает отображение результата в отдельном всплывающем окне, правая — прямо в области кода.

➤ Нажмите на каждую из кнопок и посмотрите на результат.

Рис. 9. Кнопки запуска обработки кода

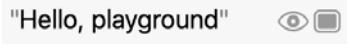

Рис. 10. Дополнительные кнопки в области результатов

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. У playground-проектов есть возможности, благодаря которым процесс разработки можно значительно улучшить.

В скором времени вы увидите, что в качестве результатов могут выводиться не только текстовые, но и графические данные. На рис. 11 показан проект, который будет реализован в книге.

В области результатов могут выводиться строки формата *N* `times`, где *N* — целое число. Они говорят о том, что данная строка кода выводится *N* раз (рис. 12).

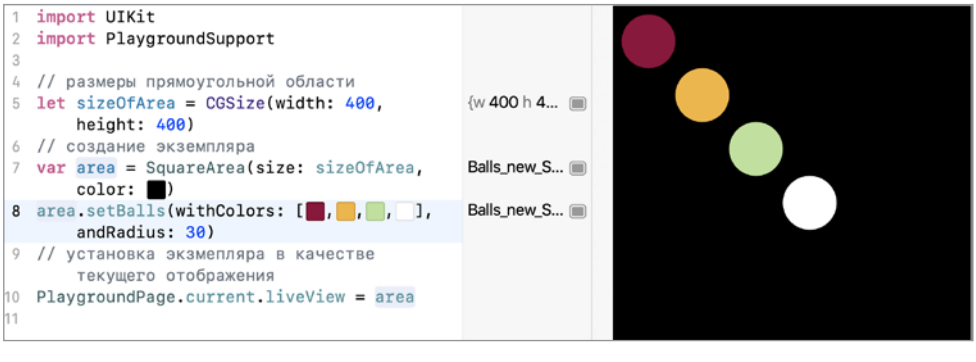

Рис. 11. Пример вывода графических элементов

Рис. 12. Вывод данных 5 раз в одной строке

Также Xcode имеет в своем арсенале такой полезный механизм, как автодополнение (в Xcode известное как автокомплит).

- В редакторе кода на новой строке напишите латинский символ `s`, после чего вы увидите окно автодополнения (рис. 13).

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.

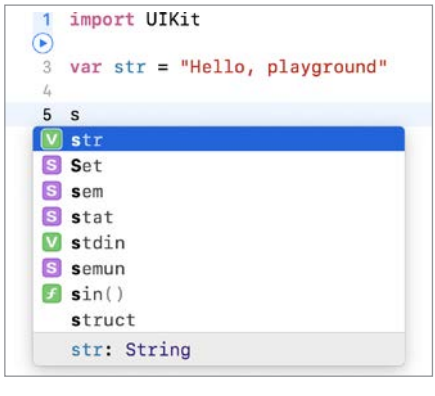

Рис. 13. Окно автодополнения в Xcode

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 (рис. 14).

Также информация об ошибке выводится на консоли в области отладки (Debug Area), показанной на рис. 15. Вывести ее на экран можно одним из следующих способов:

- выбрав в меню пункт View > Debug Area > Show Debug Area;
- щелкнув на кнопке Show the Debug Area, расположенной в левом нижнем углу редактора кода.

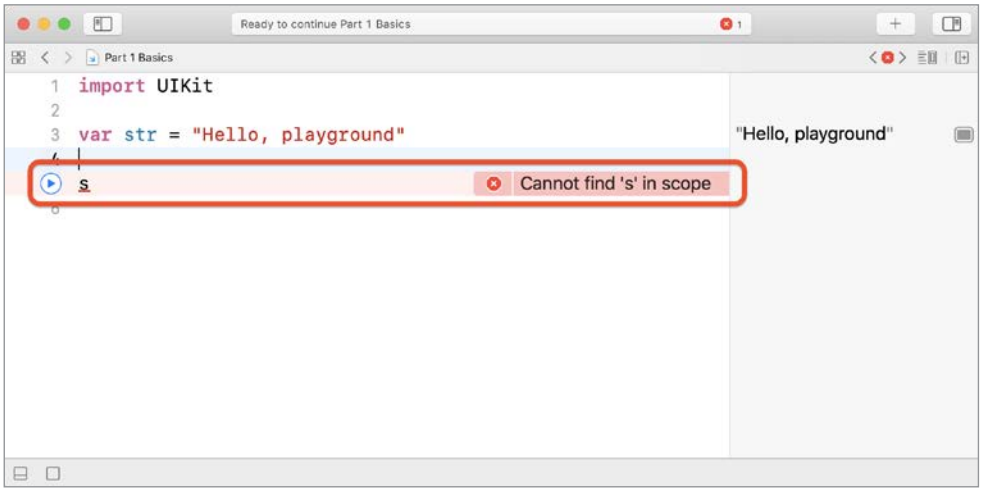

Рис. 14. Отображение ошибки в окне playground-проекта

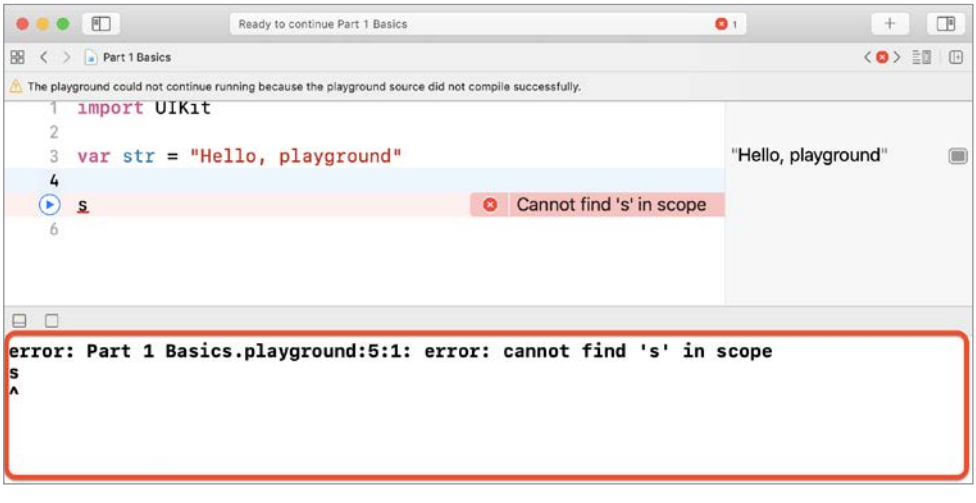

Рис. 15. Информация об ошибке выводится в Debug Area

Вы будете регулярно взаимодействовать с консолью в процессе разработки программ.

ПРИМЕЧАНИЕ Если Debug Area пуста, то просто запустите проект одним из рассмотренных ранее способов.

➤ Удалите из редактора кода строку с ошибкой (с символом `s`).

Swift дает исчерпывающую информацию об используемых в коде объектах. Если нажать клавишу Option (Alt) и щелкнуть на любом объекте в области кода (например, на `str`), то появится вспомогательное окно, в котором можно узнать тип объекта, а также имя файла, где он расположен (рис. 16).

Рис. 16. Всплывающее окно с информацией об объекте

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

Часть I

БАЗОВЫЕ ВОЗМОЖНОСТИ SWIFT

Вы стоите на пороге удивительного мира разработки приложений. С каждым пройденным разделом книги, с каждой изученной страницей и каждым выполненным заданием вы будете все ближе к своей цели — к карьере успешного программиста под iOS или macOS.

Swift — удобный и интересный язык программирования. Если вы работали с языками C, PHP, Python или другими, то уже скоро заметите их сходство с детищем Apple. В этой части книги вы узнаете о базовых понятиях, которые предшествуют успешному программированию. Я обучу вас основам синтаксиса и работы с различными фундаментальными механизмами, которые легли в основу разработки программ на языке Swift.

- ✓ Глава 1. Отправная точка
- ✓ Глава 2. Фундаментальные типы данных

ПРИМЕЧАНИЕ Весь изученный материал должен подкрепляться решением практических задач. Для этого переходите на сайт https://swiftme.ru. Там вы найдете не только задания для самостоятельного решения, но и дополнительные главы, не вошедшие в книгу, и, в дальнейшем, видеоуроки по каждой теме.

Глава 1. Отправная точка

По моей статистике, примерно 50% читателей этой книги ранее не имели каких-либо знаний в области IT и опыта разработки (рис. 1.1). Только подумайте об этом! Каждый второй купивший книгу, каждый второй подключившийся к нашему чату в Telegram на старте обладает знаниями не бóльшими, чем вы. Но при этом уже тысячи людей[1], начавших свое обучение с моих книг, с успехом продолжают прокачивать свои навыки, создавая новые проекты.

Рис. 1.1. Статистика читателей книги

> **ПРИМЕЧАНИЕ** У каждого из вас за плечами свой уровень подготовки и разный опыт, 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.
>
> В первую очередь эта книга ориентирована на начинающих разработчиков. Помните, что вы всегда можете пропустить то, что уже знаете.

1.1. Вычислительное мышление

То, чем вам предстоит заниматься (разработка программ) помимо самого языка Swift и среды разработки Xcode, предоставит вам возможность получить более глубокие знания в областях устройства компьютера и теории программирования:

---

[1] С момента выпуска первого издания данной книги было продано более 10 000 бумажных экземпляров, а также более 5000 электронных.

принципы функционирования приложений, типы и структуры данных, алгоритмы, шаблоны программирования и многое-многое другое — все это откроется вам, если вы проявите достаточное упорство и стремление. И что самое важное, если вы разовьете в себе **вычислительное мышление**, благодаря которому сможете находить оптимальные пути решения поставленных задач. Но помните, эта книга должна стать лишь толчком, первым и самым важным шагом, но основной объем знаний вы получите именно в ходе дальнейшей самостоятельной работы.

Рассмотрим один пример[1], который поможет вам понять, что такое вычислительное мышление. Представьте, что у вас есть 500-страничный телефонный справочник, в котором требуется найти телефонный номер Джона Смита (John Smith). Каким способом лучше всего решить данную задачу?

> **ПРИМЕЧАНИЕ** Не знаю, в каком году вы родились, но вполне вероятно, что уже не застали телефонные справочники. На всякий случай уточню, что в них содержится информация обо всех абонентах стационарной телефонной сети населенного пункта. То есть, зная имя и фамилию человека, вы можете найти там номер его домашнего телефона. Это было очень актуально до повсеместного внедрения сотовой связи.

Самый простой вариант, который приходит на ум, — это открыть начало книги и листать страницу за страницей, пока Джон Смит не будет найден. Этот способ называется *последовательным перебором*. Поиск перебором довольно медленный, так как нам нужно анализировать каждую страницу (рис. 1.2). В самом плохом случае поиск Джона займет `n` действий, где `n` — количество страниц в справочнике. А при увеличении количества страниц количество действий будет расти пропорционально.

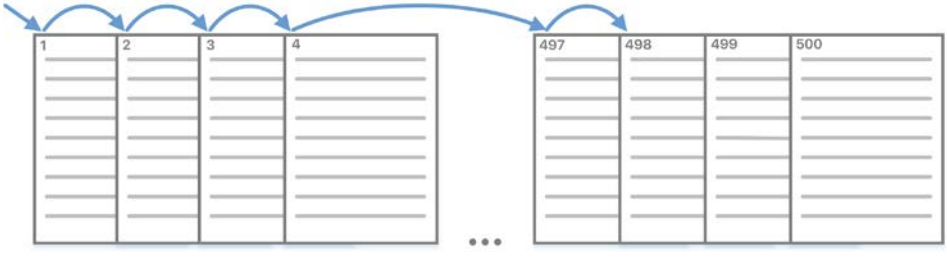

Рис. 1.2. Последовательный перебор каждой страницы справочника

Но мы могли бы пойти иным путем и немного изменить алгоритм поиска. Попробуем перелистывать не по одной, а по две страницы (рис. 1.3). И если вдруг на очередной странице мы поймем, что пропустили Джона, то просто перелистнем одну страницу назад. Данный вариант однозначно менее затратный, так как в самом плохом случае количество действий будет равно `n/2`, где `n` — количество

[1] Этот пример впервые был продемонстрирован в рамках курса CS50 Гарвардского университета.

страниц в справочнике. При этом если увеличить количество страниц в 2 раза (до 1000), то количество действий также пропорционально увеличится (до 500). Этот способ в 2 раза более эффективный, чем первый.

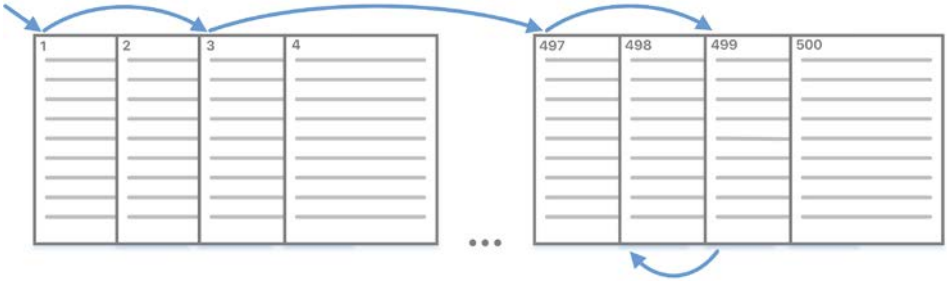

Рис. 1.3. Последовательный перебор каждой второй страницы справочника

Какие варианты решения данной задачи мы могли бы еще использовать? А что, если вместо последовательного просмотра страниц мы будем многократно делить справочник пополам, определяя, в какой из половин находится Джон Смит? Сперва мы делим справочник на два блока по 250 страниц и определяем, в какой из половин расположена искомая запись. Далее эту половину снова делим на два, получая блоки по 125, и повторяем данные операции, пока страница с Джоном не будет найдена.

Для нашего справочника в самом худшем случае поиск Джона будет состоять всего из 9 операций. А при увеличении количества страниц в 2 раза нам потребуется совершить всего одно дополнительное действие, чтобы разделить книгу пополам. Даже если справочник будет содержать миллиард страниц, поиск потребует не больше 30 действий!

Каждый из рассмотренных способов — это **алгоритм**, то есть набор инструкций для решения определенной задачи (в данном случае — для поиска номера абонента по его имени). В дальнейшем алгоритмы станут вашим верным спутником. Каждая задача, которую вы будете ставить перед собой (в том числе при выполнении домашних заданий), будет решаться с помощью того или иного алгоритма.

Вычислительное мышление — это ваш инструмент правильной постановки проблемы, а также поиска оптимального пути ее решения. В этой книге я постараюсь не только научить вас синтаксису Swift, но и заставить ваш мозг думать рационально и системно.

Создавая любую программу на основе входных данных и используя множество алгоритмов, вы будете решать определенную проблему (рис. 1.4). То есть входные данные — это условия для корректного решения проблемы, а выходные данные — это результат, принятое или рассчитанное решение. В примере про справочник входные данные — это сам справочник и имя, которое необходимо найти, а выходные — телефонный номер.

Рис. 1.4. Программа и данные

Но программы не могут существовать без аппаратной составляющей, без компьютера. Опустимся на уровень ниже и посмотрим, как именно компьютер работает с вашими программами (алгоритмами и данными).

ПРИМЕЧАНИЕ Включайтесь в обсуждение учебного материала книги в нашем чате в Telegram по ссылке https://swiftme.ru/telegramchat. Тут вы найдете более тысячи участников, новичков и более опытных разработчиков, способных помочь вам в процессе обучения.

1.2. Как компьютер работает с данными

Сегодня компьютер для решения любой задачи использует нули и единицы, проводя с ними совсем нехитрые операции. Возможно, вы удивитесь, но современные электронные устройства не такие уж и интеллектуальные, как нам рассказывают в рекламе: вся умная функциональность 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.

Рис. 1.5. Функциональные уровни компьютера

Все элементы компьютера в общем случае можно разделить на три основных функциональных уровня (рис. 1.5): аппаратный, операционной системы и программный.

Аппаратный уровень

Аппаратный уровень представлен физическим оборудованием компьютера. Это та основа, на которой базируются остальные уровни. Самые важные элементы аппаратного уровня — это *центральный процессор* (CPU, Central Processing Unit) и *оперативная память*.

Процессор — сердце любого современного электронного устройства (персонального компьютера, планшета, смартфона и т. д.). Компьютеры называются вычислительными машинами неспроста. Они постоянно обрабатывают числа и вычисляют значения, выполняя простые арифметические и логические операции. Все данные в любом компьютере превращаются в числа. Причем неважно, говорим мы о фотографии, любимой песне или книге, — для компьютера все это лишь нули и единицы. CPU как раз и предназначен для работы с числами. Он получает на вход несколько значений и, в зависимости от переданной команды (инструкции), выполняет с ними заданные операции.

Данные, с которыми работает процессор, хранятся во временном хранилище — оперативной памяти.

Оперативная память — это основное средство хранения данных, используемое программами в ходе их функционирования. Память компьютера состоит из миллионов отдельных ячеек, каждая из которых может хранить небольшой объем информацию. Каждая ячейка имеет уникальный адрес, по которому ее можно найти в памяти. Но размер ячейки не позволяет хранить в ней большое количество данных, по этой причине ячейки логически объединяются в *хранилища данных* (рис. 1.6).

Хранилище может объединять произвольное количество ячеек, его размер определяется количеством записанной информации и при необходимости может изменяться. Как и ячейка памяти, хранилище имеет уникальный адрес, по которому может быть получено хранящееся в нем значение (обычно он соответствует адресу первой входящей в него ячейки).

> **ПРИМЕЧАНИЕ** Описание структуры и работы оперативной памяти здесь максимально упрощено. Это сделано для лучшего восприятия вами учебного материала и плавного перехода к простейшим понятиям. Их изучение — первый и обязательный шаг в вашем обучении.

Адреса ячеек (а соответственно и адреса хранилищ) — это наборы цифр, которые очень неудобны для использования программистами. По этой причине Swift, как и любой другой язык высокого уровня, вместо числовых адресов позволяет указывать удобные и понятные имена (идентификаторы), например `username` или `myFriend`, для хранения данных в оперативной памяти. Такие имена использовать намного проще, чем запоминать и передавать в программу настоящие адреса ячеек.

Рис. 1.6. Память с набором ячеек

Предположим, что вы хотите сохранить в вашем приложении имя пользователя, а также имена его друзей. При записи информации в память необходимо передать значение и идентификатор. На рис. 1.7 показан пример трех значений, записанных в память. Каждое из значений записано в свое собственное хранилище данных, каждому из которых соответствует уникальный идентификатор. Независимо от того, кто в настоящий момент пользуется вашей программой, при обращении к ячейке `username` всегда будет возвращено актуальное имя пользователя.

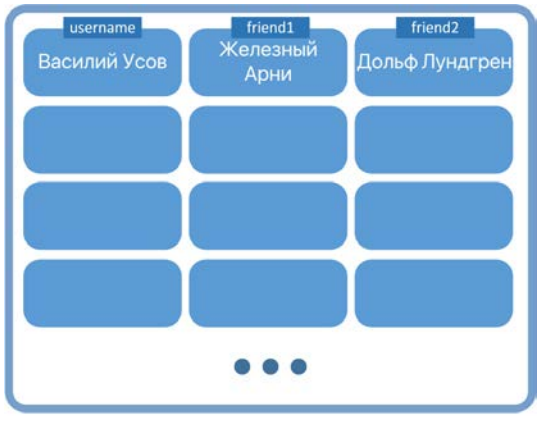

Рис. 1.7. Память с записанными значениями

По идентификатору можно не только получить значение, но и изменить его. Если `Василий Усов` перестал дружить с `Железным Арни`, то новым другом мог бы стать `Сильвестр Сталлоне`. Для изменения потребуется лишь передать новое значение,

используя уже существующий идентификатор `friend1`. И все, новое значение уже в памяти.

> **ПРИМЕЧАНИЕ** Оперативная память хранит нужные данные так долго, как этого требует ваше приложение (ограничиваясь лишь временем, пока оно запущено). Для постоянного (долговременного) хранения информации используются другие элементы аппаратного уровня, например HDD или SSD.

Уровень операционной системы

Операционная система (ОС) — это посредник между вашей программой и аппаратной частью. Благодаря ОС приложения могут использовать все возможности компьютера: воспроизводить музыку, отображать графику на экране, передавать данные по Wi-Fi и Bluetooth, выполнять математические операции и многое другое.

Операционная система предоставляет интерфейсы, которые используются в ходе работы программного обеспечения. Хотите сохранить фото своей кошки? Не вопрос! Напишите соответствующую команду, и ОС сама передаст ваши байты куда следует, после чего вернет результат.

Программный уровень

Программный уровень — это все программы, которые работают на вашем компьютере в среде операционной системы. Каждая программа решает множество задач и для этого взаимодействует с уровнем операционной системы и аппаратным уровнем (рис. 1.8).

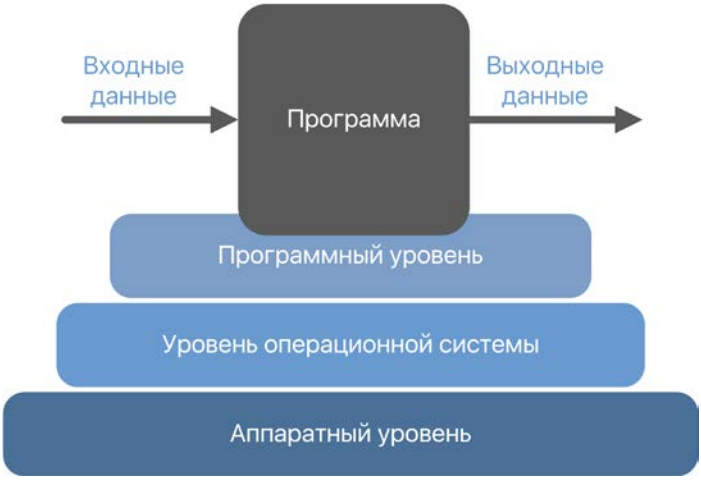

Рис. 1.8. Память с записанными значениями

1. С целью решения задачи приложение может передать операционной системе команды.
2. Система обрабатывает полученные данные и передает их на аппаратный уровень.
3. Аппаратный уровень производит необходимые операции, используя при этом техническую составляющую устройства, после чего передает данные обратно на уровень операционной системы и далее на программный уровень.

Каждая задача, выполняемая приложением, в результате может привести к выполнению сотен и тысяч инструкций в процессоре.

Программный уровень подразумевает использование языка программирования, так как программы создаются именно с его помощью.

Каждая программа взаимодействует с данными, которые могут быть получены разными способами: загружены с жесткого диска, введены пользователем, получены из Сети и т. д. Часто эти данные нужно не просто получить, обработать и отправить в указанное место, но и сохранить, чтобы использовать в будущем. Так, например, получив ваше имя (предположим, что вы ввели его при открытии приложения), программа запишет его в память и при необходимости будет получать его оттуда, чтобы отобразить в соответствующем месте графического интерфейса.

1.3. Базовые понятия

В предыдущем разделе, когда мы говорили об оперативной памяти, то упоминали понятие *хранилища данных*.

Хранилище данных — это виртуальный объект со следующими свойствами:

- записанное значение;
- идентификатор (имя);
- тип информации, для хранения которой предназначено хранилище (числовой, строковый и др.).

Практически весь процесс программирования заключается в том, чтобы создавать (объявлять) объекты, задавать (инициализировать) им значения, получать эти значения из памяти и производить с ними операции.

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

Объявление — это создание нового объекта (хранилища данных).

Инициализация — это присвоение значения объекту.

Рассмотрим простейший пример: числовое значение `23` умножить на числовое значение `145` (листинг 1.1).

Листинг 1.1

```
23 * 145
```

Данный математический пример является математическим **выражением**, то есть командой на языке математики. В нем можно выделить одну операцию умножения и два числовых значения, с которыми будет производиться действие (`23` и `145`).

Во время вычисления значения выражения программисту надо сделать следующее:

- Объявить хранилище с именем `value1` и значением `23`.
- Объявить хранилище с именем `value2` и значением `145`.
- Объявить хранилище с именем `result`, а в качестве значения проинициализировать ему результат умножения значений хранилищ `value1` и `value2`.

ПРИМЕЧАНИЕ Имена `value1`, `value2` и `result` приведены лишь для примера. В процессе работы вы сможете использовать любые имена.

- В листинге 1.1 оператор * указывает на то, что надо выполнить умножение. В результате выполнения выражения программист получит значение `3335`, записанное в новую ячейку памяти (рис. 1.9).

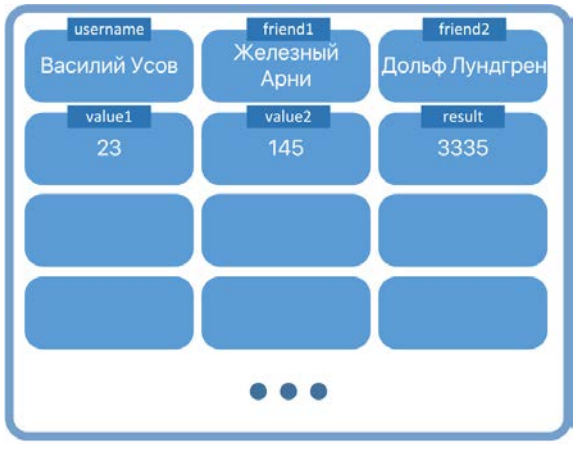

Рис. 1.9. Результат умножения двух чисел помещен в новое хранилище

Если данный пример умножения чисел рассмотреть со стороны языка программирования, то его можно представить так, как показано в листинге 1.2.

Листинг 1.2

```
СоздатьХранилище с именем value1 и значением 23
СоздатьХранилище с именем value2 и значением 145
СоздатьХранилище с именем result и значением value1 * value2
```

ПРИМЕЧАНИЕ Пример из листинга 1.2 написан не на Swift, а с помощью псевдокода. Он приведен для лучшего понимания вами описываемого процесса.

Как видно из рассмотренных примеров работы с памятью (рис 1.9), 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 хранилища ему **всегда** назначается тип данных (это делает программист или компьютер), определяющий, какие именно данные будут храниться в нем. Это нужно затем, чтобы вы, загрузив данные из памяти, знали, для чего именно они предназначены.

Любая программа — это набор выражений, или, другими словами, набор 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 пример: «Отправь этот файл, но перед этим преобразуй его из docx в rtf».

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

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

Операнд — это значение, с которым оператор производит операцию.

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

ПРИМЕЧАНИЕ В процессе изучения материала вам предстоит запомнить некоторые определения, но не переживайте, если они даются вам с трудом. Для удобства советую выписать их в свой конспект и при необходимости возвращаться к ним.

1.4. Введение в операторы

Оператор — это одно из основных понятий, с которым вам предстоит встречаться в ходе обучения разработке на Swift. Благодаря им мы можем работать с данными.

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

Ваш первый программный код

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 (рис. 1.10).

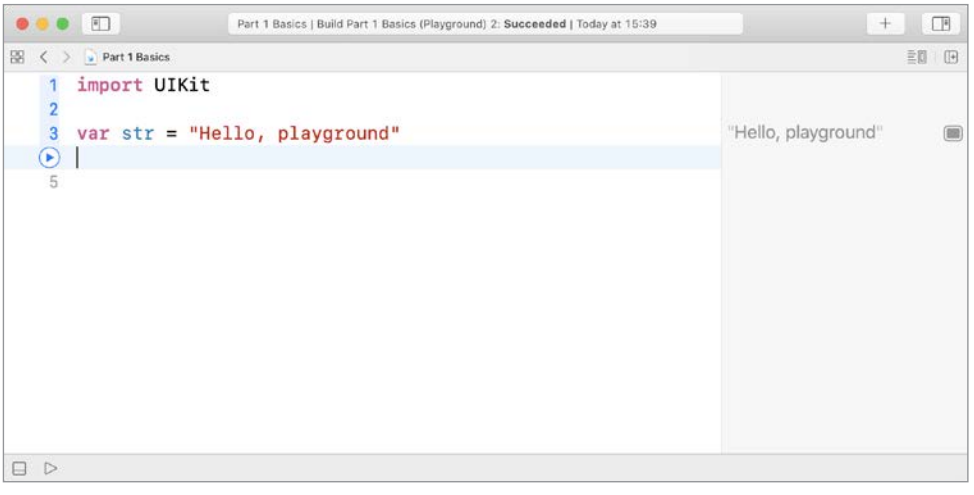

Рис. 1.10. Окно нового playground

Взгляните на код в Xcode Playground (листинг 1.3).

Листинг 1.3

```
import UIKit
var str = "Hello, playground"
```

В первой строке подключается библиотека `UIKit`, расширяющая стандартные возможности языка. О том, что это за библиотека и как работать с оператором `import`, поговорим далее. Сейчас же рассмотрим вторую строчку кода (листинг 1.4).

Листинг 1.4

```
var str = "Hello, playground"
```

В ней объявляется хранилище данных с идентификатором (именем) `str`, после чего инициализируется текстовое значение `Hello, playground`. Данная строка представляет собой полноценное выражение. Его можно разбить на отдельные шаги (рис. 1.11):

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

Шаг 2. Этому хранилищу инициализируется (присваивается) текстовое значение `Hello, playground`. Этот процесс и называется инициализацией значения с помощью оператора `=` (присваивания).

Рис. 1.11. Выражение, состоящее из шагов

Если представить схему оперативной памяти сейчас (с учетом записанных ранее значений), то она выглядит так, как показано на рис. 1.12.

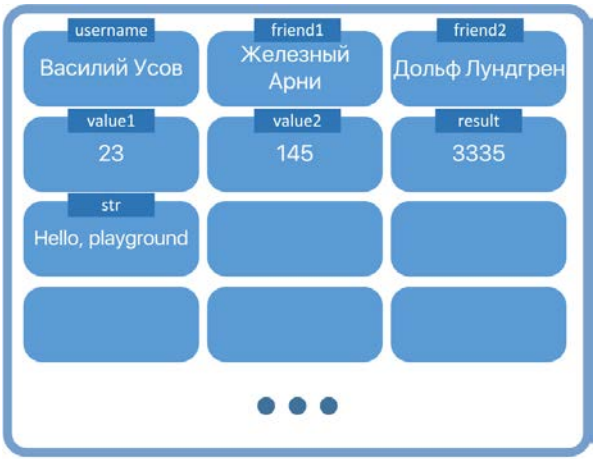

Рис. 1.12. Результат создания нового хранилища данных

В данном выражении используется два оператора: на шаге 1 — оператор `var`, на шаге 2 — оператор `=`.

Классификация операторов

В Swift существуют следующие виды операторов:

- **Простые операторы**, выполняющие операции со значениями (операндами). В их состав входят унарные и бинарные операторы.
- **Унарные операторы** выполняют операцию с одним операндом. Они могут находиться перед операндом (префиксные) или после него (постфиксные).

  Оператор `var`, с помощью которого объявляется хранилище данных, или оператор минус (`-a`), с помощью которого создаются отрицательные числа, являются унарными префиксными.

  Далее в книге будут рассмотрены примеры постфиксных операторов, например, многоточие (`...`) в выражении `1...` указывает на все целые числа после единицы.

- **Бинарные операторы** выполняют операцию с двумя операндами. Оператор, который располагается между операндами, называется инфиксным.

  Оператор инициализации (`=`) и оператор сложения (`+`) являются бинарными инфиксными, так как использует в работе два операнда и находятся между ними (`value1 = 12` или `34+12`).

- **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. Подробно структурные операторы будут рассмотрены в главе «Операторы управления».

1.5. Оператор инициализации

Как вы уже знаете, для хранения данных компьютер использует оперативную память, выделяя под значения ячейки (хранилища данных) с уникальными именами. 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 1.5).

Листинг 1.5

```
var str = "Hello, playground"
```

Для инициализации значения используется оператор присваивания (оператор инициализации), обозначаемый как знак равенства (`=`).

Оператор инициализации (присваивания) (`=`) — это бинарный оператор. Он используется в типовом выражении `a = b`, присваивая хранилищу данных с именем `a` значение хранилища данных с именем `b`. В листинге 1.5 объекту `str` инициализируется текстовое значение `Hello, playground`.

В общем случае левая и правая части оператора присваивания будут однотипными (то есть иметь одинаковый тип, предназначаться для хранения данных одного и того же типа). В примере выше строка `Hello, playground` — 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.

1.6. Переменные и константы

Всего в Swift выделяют два вида хранилищ данных:

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

В листинге 1.5 для объявления хранилища данных `str` используется оператор `var`, это значит, что объявляется переменная.

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

1) **имя**, по которому можно проинициализировать новое или получить записанное ранее значение. Ранее мы назвали его идентификатором;
2) **тип данных**, определяющий множество значений, которые могут храниться в этом хранилище (целые числа, строки и т. д.);
3) **значение**, которое в данный момент находится в хранилище.

Рассмотрим каждый вид хранилища подробнее.

Переменные

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

Переменная — это хранилище данных, значение которого может быть многократно изменено разработчиком в процессе работы программы. В качестве примера можно привести переменную, которая хранит текущую дату. Ее значение должно меняться ежедневно в полночь.

Для объявления переменной в Swift используется оператор `var`.

СИНТАКСИС

```
var имяПеременной = значениеПеременной
```

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

По имени переменной мы получаем доступ к значению переменной для его считывания или изменения.

Рассмотрим пример объявления переменной (листинг 1.6).

> **ПРИМЕЧАНИЕ** Вы можете продолжать писать код в окне Xcode Playground, не убирая предыдущие выражения.

Листинг 1.6

```
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 любое количество переменных. Вы ограничены только своими потребностями (ну и конечно, объемом оперативной памяти устройства).

Изменить значение переменной можно, инициализировав ей новое значение. Повторно использовать оператор `var` в этом случае не надо (листинг 1.7).

Листинг 1.7

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

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

Будьте внимательны: во время инициализации нового значения старое уничтожается. В результате выполнения кода из листинга 1.7 в переменной `age` будет храниться значение `22`. В области результатов выводятся два значения: старое (`21`) и новое (`22`). Старое — напротив первой строки кода, новое — напротив второй (рис. 1.13).

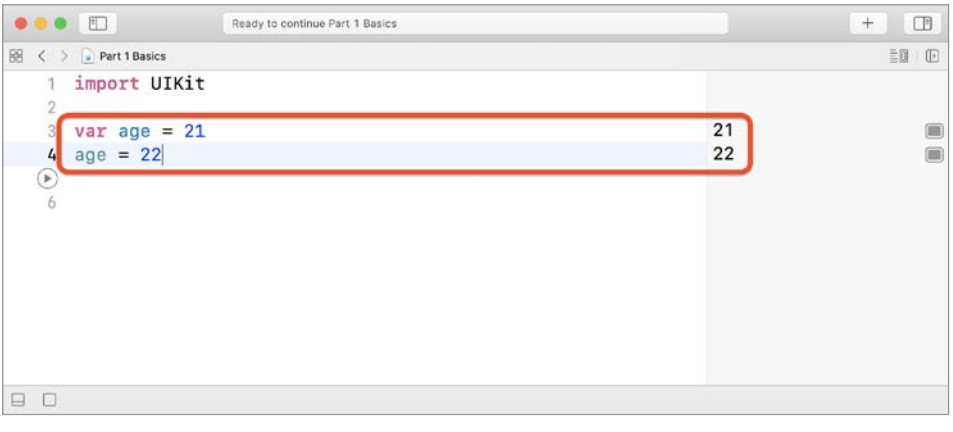

Рис. 1.13. Отображение значения переменной в области результатов

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 в Playground активирован 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, и выберите Manually Run (рис. 1.14).

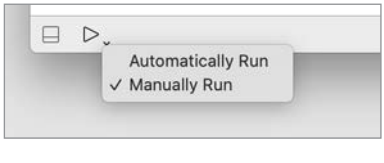

Рис. 1.14. Выбор режима выполнения кода

Теперь для того, чтобы ваш код был выполнен, а его результаты отображены в области результатов, нажмите на треугольник в нижней части окна или слева от строки кода (появляется при наведении указателя мыши) (рис. 1.15).

Но и этот подход не гарантирует того, что Xcode успешно выполнит ваш код. Если у вас все еще возникают с этим проблемы, то используйте описанные ниже методы по порядку, пока один из них не поможет:

- Немного подождите.
- Повторно дайте команду на выполнение кода.
- Перезапустите Xcode.
- Завершите процесс зависшего симулятора. Для этого откройте Программы > Утилиты > Мониторинг системы, после чего найдите и завершите процесс com.apple.CoreSimulator.CoreSimulatorService.
- Измените платформу playground-проекта с iOS на macOS. Для этого нажмите кнопку Hide or Show the Inspector, расположенную в правом верхнем углу, и в поле Platform выберите необходимый пункт (см. рис. 1.16). В этом случае вам надо внести правки в первую строку в редакторе кода: заменить `import UIKit` на `import Foundation`, так как библиотека `UIKit` доступна исключительно в мобильных операционных системах iOS и iPadOS.

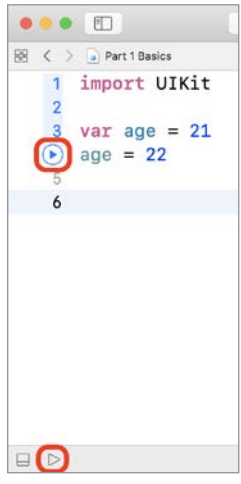

Рис. 1.15. Принудительный запуск кода на выполнение

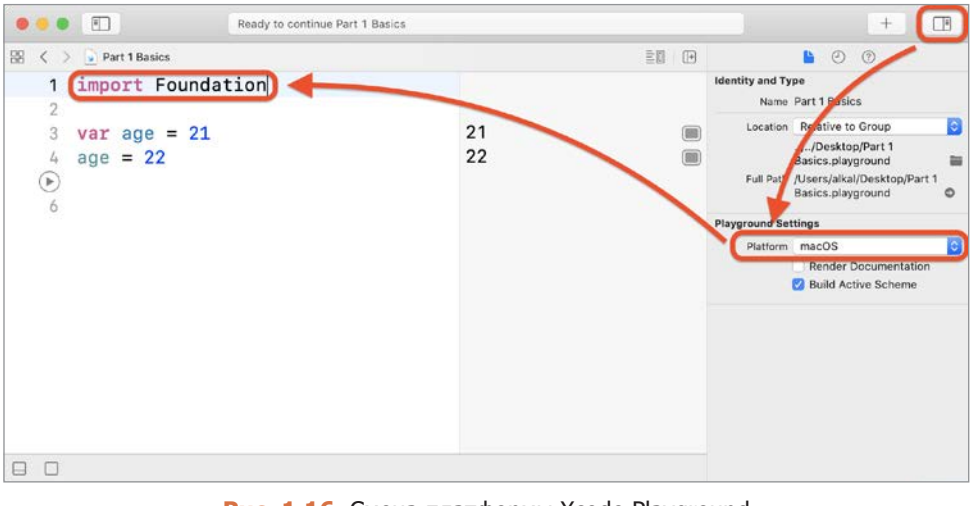

Рис. 1.16. Смена платформы Xcode Playground

В области результатов Xcode Playground можно узнать текущее значение любого параметра. Для этого достаточно лишь написать его имя. Чтобы убедиться, что значение изменилось, напишите в следующей строке имя переменной и посмотрите в область результатов (рис. 1.17).

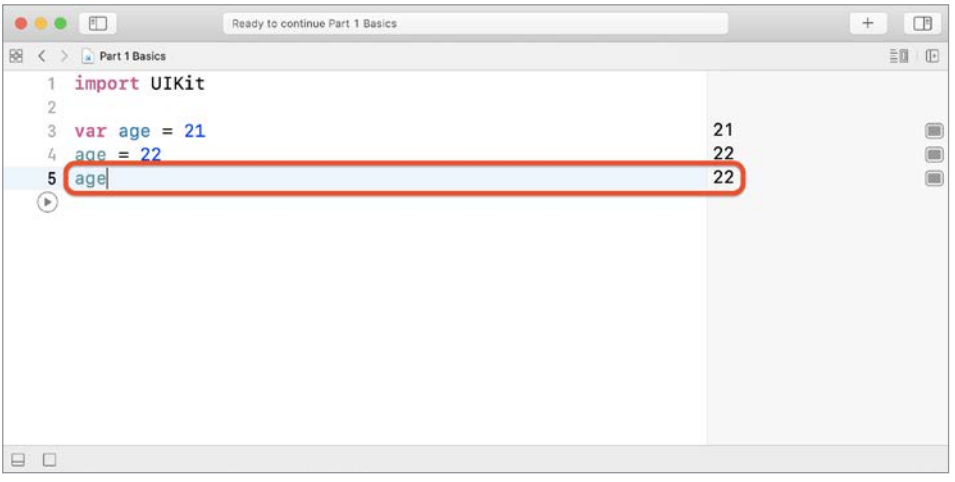

Рис. 1.17. Уточнение значения переменной

Константы

В отличие от переменных, присвоить значение константе можно только один раз. Все последующие попытки изменить его вызовут ошибку. Константы объявляются с помощью оператора `let`.

СИНТАКСИС

```
let имяКонстанты = значениеКонстанты
```

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

Рассмотрим пример объявления константы (листинг 1.8).

Листинг 1.8

```
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. 1.18).

> **ПРИМЕЧАНИЕ** Xcode пытается сделать вашу работу максимально продуктивной. В большинстве случаев он не просто сообщает о возникшей ошибке, а дает рекомендации по ее устранению и даже самостоятельно вносит необходимые правки.
>
> Чтобы ознакомиться с рекомендациями, нажмите на кружок слева от сообщения (рис. 1.19). После нажатия на кнопку Fix ошибка будет исправлена. В данном примере рекомендовано изменить оператор `let` на `var`.

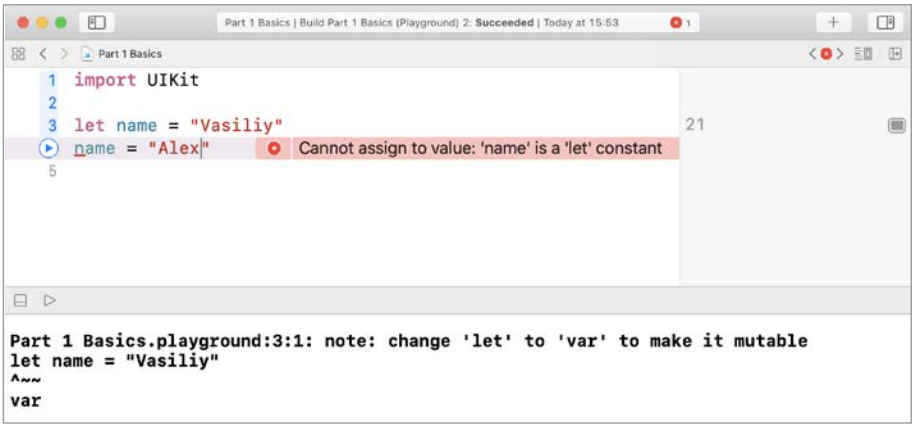

Рис. 1.18. Ошибка при изменении значения константы

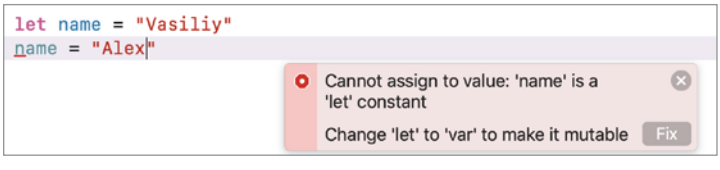

Рис. 1.19. Автоматическое исправление ошибки

Если ранее вы программировали на других языках, то, вполне вероятно, часто избегали использования констант, обходясь переменными. Swift — это язык, который позволяет очень гибко управлять ресурсами. Константы — базовое средство оптимизации используемых мощностей. Поэтому если инициализированное значение не планируется изменять, программисты всегда должны использовать константы.

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

Объявление нескольких параметров в одном выражении

При объявлении нескольких параметров вы можете использовать один оператор — `var` или `let` и через запятую попарно указать их имена и инициализируемые значения (листинг 1.9).

Листинг 1.9

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

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

Где использовать переменные и константы

Подведем краткие итоги изученного материала.

Что используем	Где используется
Переменные	Для хранения значений, которые могут изменяться в ходе выполнения кода. **Пример:** Количество секунд с момента запуска таймера. `var second = 0` Количество очков, заработанных пользователем. `var userPoints = 100`
Константы	Для хранения значений, которые не должны и не будут изменяться в ходе выполнения кода. **Пример:** Имя пользователя. `let username = "Василий Усов"` Устройство, на котором запущено приложение. `let phoneModel = "iPhone 11"`

1.7. 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 1.10).

Листинг 1.10

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

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

Стоит отметить, что таким образом вы создаете копию исходного значения, то есть в результате операции будет объявлена константа и переменная с двумя независимыми значениями. Изменение одного из них не повлияет на другое.

Тип данных параметра, который обеспечивает передачу значения копированием, называется **value type** (значимый тип, или тип-значение). Если рассмотреть схе-

матичное изображение памяти, то скопированные значения будут занимать две независимые ячейки (рис. 1.20).

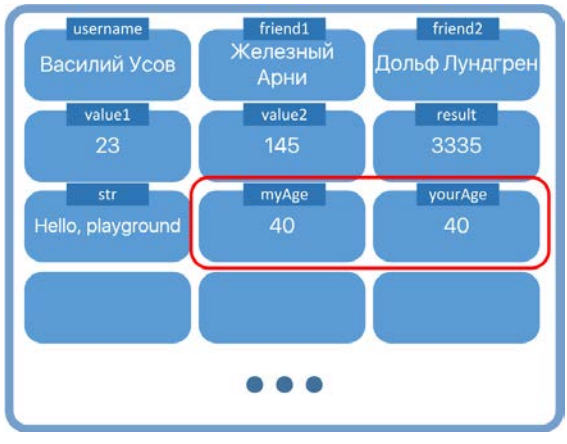

Рис. 1.20. Пример value type

Помимо передачи значения копированием, существует передача ссылки на значение, когда несколько параметров содержат в себе ссылку на одно и то же значение, хранящееся в памяти. Тип данных, который обеспечивает передачу значения ссылкой, называется **reference type** (ссылочный тип, или тип-ссылка). При этом изменение значения через любой из параметров отразится и на всех копиях-ссылках. Другими словами, параметры ссылочного типа устанавливают для ячейки памяти несколько имен (рис. 1.21).

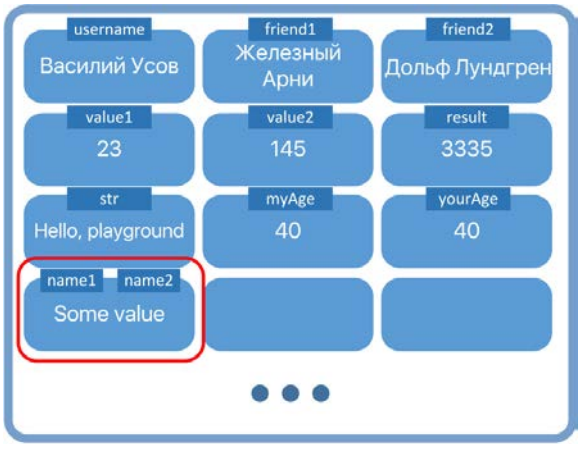

Рис. 1.21. Пример reference type

Пока что вы будете работать только с value type. Далее, при рассмотрении объектных типов и управления памятью, мы познакомимся и с reference type.

1.8. Правила именования переменных и констант

При написании программ на Swift вы можете создавать параметры с любыми именами. При этом существуют правила, которых необходимо придерживаться:

1. Имя переменных и констант пишутся в *верблюжьем регистре* (camelCase).

При именовании используются латинские символы, без подчеркиваний, тире, математических формул и т. д. Каждое значимое слово, кроме первого, начинается с заглавной буквы.

Пример хорошего названия: `userData`, `messageFromServer`, `highScoreOfAllUsers`.

В листинге 1.11 показан пример плохого имени параметра.

Листинг 1.11

```
let dØw = "Значение константы с невоспроизводимым именем"
```

2. Имена параметров должны быть осмысленными и раскрывать их реальное предназначение.

Чем больше кода в вашей программе, тем сложнее в нем ориентироваться. Чтобы облегчить навигацию, старайтесь давать максимально осмысленные имена для всех объектов. При этом ясность должна преобладать над краткостью.

`let hsAllUsers` — пример плохого имени константы. Для чего предназначен данный параметр?

`let highScoreOfAllUsers` — пример хорошего имени константы, из которого однозначно понятно, что данный параметр предназначен для хранения лучшего показателя заработанных очков среди всех пользователей приложения.

1.9. Возможности автодополнения и кодовые сниппеты

Ранее мы уже упоминали об окне автодополнения в Xcode (рис. 1.22). Содержимое окна строится на основе уже введенных вами символов. С его помощью вы можете значительно сократить время написания кода, выбирая наиболее подходящий из предложенных вариантов.

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

В окне автодополнения можно найти имена всех параметров, операторов и ключевых слов, соответствующих введенным символам и данному контексту. Помимо этого, там находятся и кодовые сниппеты, или, другими словами, шаблоны кода.

Так, к примеру, если в новой строчке кода ввести ключевое слово `var`, то в окне автодополнения вы увидите элемент, позволяющий вставить кодовый сниппет для данной конструкции (он обозначен двумя фигурными скобками) (рис. 1.23).

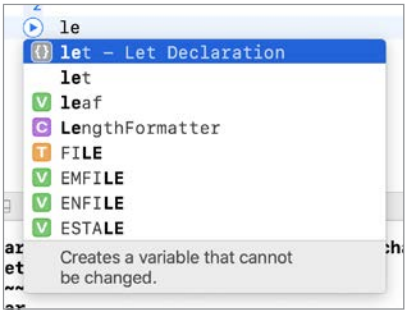

Рис. 1.22. Окно автодополнения в Xcode

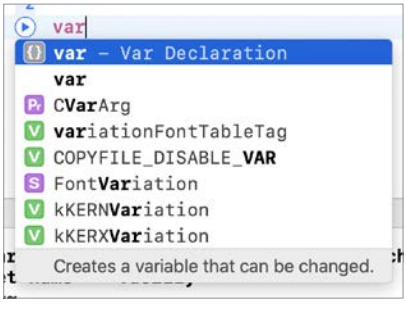

Рис. 1.23. Сниппет оператора var

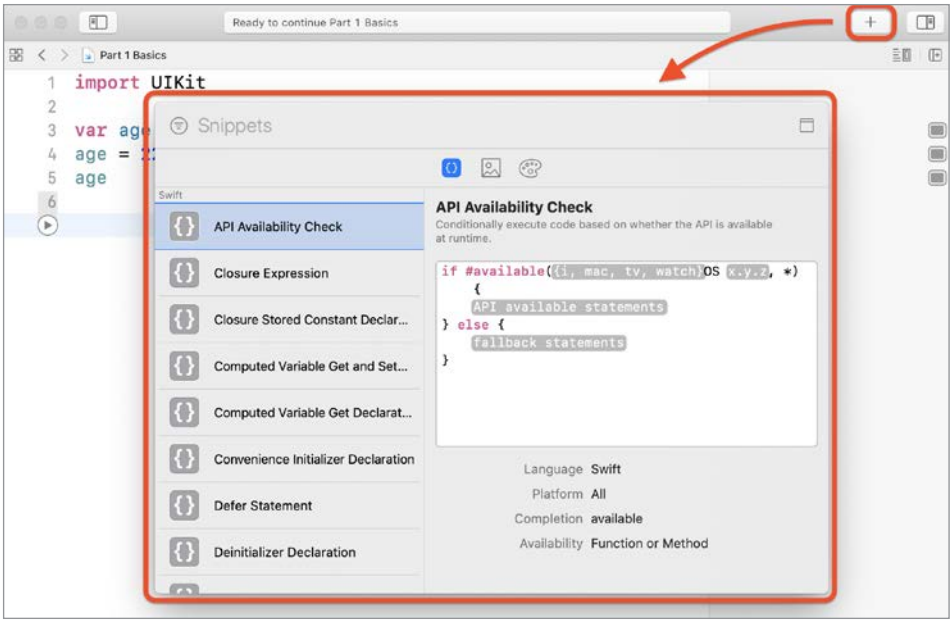

Рис. 1.24. Библиотека кодовых сниппетов

ПРИМЕЧАНИЕ Кодовые сниппеты — это шаблоны, позволяющие быстро создавать типовые конструкции.

Кликнув 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.

Библиотеку кодовых сниппетов можно открыть с помощью кнопки с изображением символа +, расположенной в верхней части Xcode Playground (рис. 1.24). В появившемся окне отображается вся библиотека кодовых шаблонов с возможностью поиска по ней.

1.10. Область видимости (scope)

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 область его применения, или, другими словами, область видимости (scope), которая определяет, где именно данный объект может быть использован. Например, область видимости объекта может быть ограничена файлом или отдельным блоком кода.

По области видимости объекты можно разделить на два вида:

Глобальные — это объекты, доступные в любой точке программы.

Локальные — 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. Помимо этого, одни объекты являются дочерними по отношению к другим (рис. 1.25).

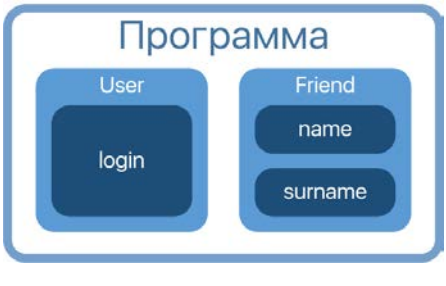

Рис. 1.25. Вариант иерархии объектов внутри программы

Объекты `User` и `Friend` объявлены непосредственно в корне программы. Такие объекты называются глобальными, они доступны в любой точке программного кода.

Каждый из глобальных объектов из примера выше содержит внутренние, локальные по отношению к нему, объекты:

- В составе `User` есть объект `login`;
- В составе `Friend` есть два объекта: name и surname.

Локальные объекты доступны только в пределах своего контекста, то есть родительского объекта. Таким образом, попытка получить доступ из корня программы `User` к объектам `name` и `surname` приведет к ошибке. Для доступа к локальным объектам извне надо обратиться к их родительскому объекту: `Friend -> name`.

Область видимости — это один из базовых механизмов управления доступом. Если бы все объекты были глобальными, то в процессе работы программы вы бы не могли гарантировать того, что случайно не удалите или не измените то, к чему по логике в данный момент не должны быть допущены.

Другой положительной стороной области видимости является возможность создавать объекты с одинаковыми именами. К примеру, если бы вам потребовалось ввести имя (name) в состав `User`, то это не привело бы к ошибке, так как у одноименных объектов различная область видимости (рис. 1.26).

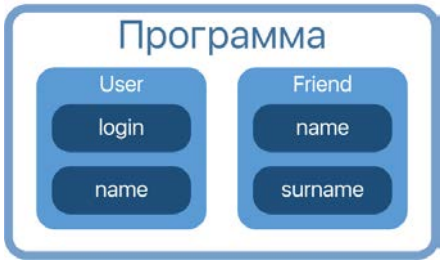

Рис. 1.26. Объекты с одинаковыми именами

Время жизни локальных объектов равно времени жизни их родительских объектов.

1.11. Комментарии

Классические комментарии

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

Комментарии в Swift, как и в любом другом языке программирования, — это 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. 1.27 приведен пример комментариев. По умолчанию они выделяются серым цветом.

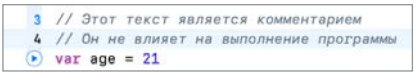

Рис. 1.27. Пример комментариев на странице с кодом

Хорошо, если ваш код содержит комментарии. Они помогают нам не запутаться в написанном.

> **ПРИМЕЧАНИЕ** Помните, что комментарии не должны становиться заменой качественным именам параметров! Не стоит называть переменную ud, а в комментарии к ней описывать ее предназначение. Лучше сразу укажите корректное `userData`.
>
> Есть много рекомендаций 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 комментарии только там, где это действительно необходимо: например, чтобы дать некоторые пояснения к алгоритму работы. Существует множество приемов, которые помогают писать удобный и читабельный код практически без использования комментариев. О некоторых из них мы поговорим в ходе изучения Swift.

В Swift можно использовать однострочные и многострочные комментарии.

Однострочные комментарии пишутся с помощью двойной косой черты (`// комментарий`) перед текстом комментария, в то время как многострочные обрамляются звездочками с косыми чертами с обеих сторон (`/* комментарий */`). Пример комментариев приведен в листинге 1.12.

Листинг 1.12

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

Весь текст, находящийся в комментарии, игнорируется языком Swift и совершенно не влияет на выполнение программы.

Markdown-комментарии

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

Markdown-комментарии должны начинаться с двойной косой черты и двоеточия (`//:`), после которых следует текст комментария. Несколько примеров неформатированных комментариев приведены на рис. 1.28.

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. 1.29.

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

```
1  //: это markdown комментарий
2  //: объявим переменную
3  var str = "Привет, Мир!"
4  //: а это - *курсивный текст*
5  var str2 = "Привет, Мир"
6  //: а это - **жирный текст**
7  var str3 = "Привет, Мир"
8  //: ----------
```

Рис. 1.28. Неформатированные markdown-комментарии

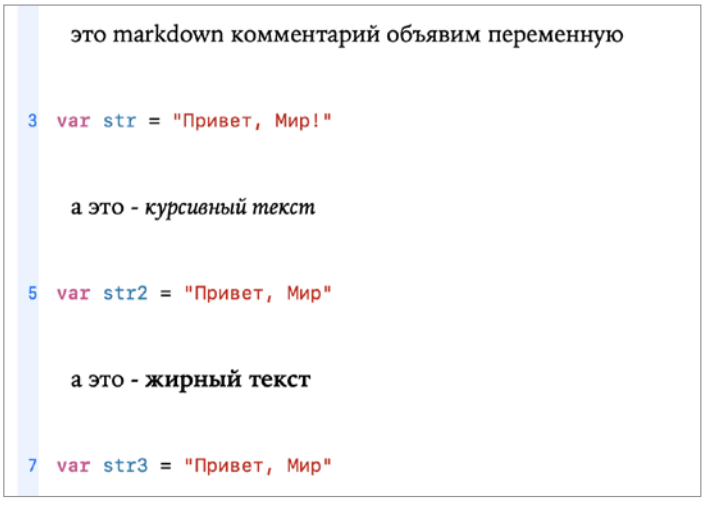

Рис. 1.29. Форматированные markdown-комментарии

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

1.12. 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 этот символ не используется. Это связано с тем, что для этого языка нет такой необходимости. Строчка считается завершенной, когда в конце присутствует (невидимый) символ переноса (он вставляется автоматически при нажатии клавиши Enter).

Единственное исключение — когда в одной строке необходимо написать сразу несколько выражений (листинг 1.13).

Листинг 1.13

```
// одно выражение в строке – точка с запятой не нужна
let number = 18
// несколько выражений в строке – точка с запятой нужна
number = 55; let 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.

1.13. Отладочная консоль и функция print(_:)

Консоль

Консоль — это интерфейс командной строки, в котором отображаются текстовые данные. Если у вас есть опыт работы с компьютером (в любой из существующих ОС), то вы наверняка неоднократно сталкивались с консольными приложениями (рис. 1.30).

Рис. 1.30. Пример консоли в различных операционных системах

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

Одной из составных частей Xcode также является отладочная консоль. Ее основная функция — вывод отладочной текстовой информации (например, сообщений об ошибках), а также текстовых данных, указанных вами в процессе разработки программы. Ранее мы отображали отладочную консоль с помощью кнопки Show the Debug Area, расположенной в нижнем левом углу редактора кода (рис. 1.31).

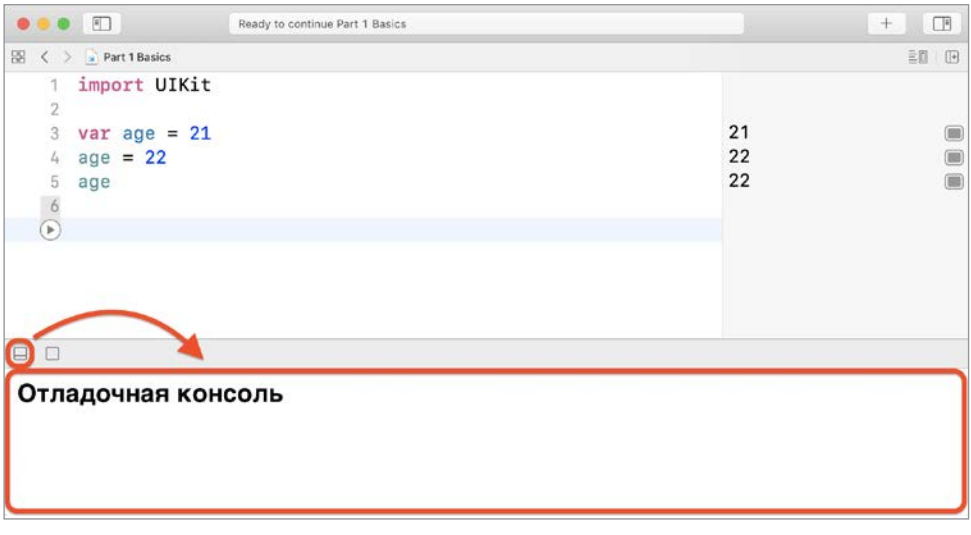

Рис. 1.31. Отладочная консоль Xcode Playground

Отладочная консоль выводит разработчику или пользователю необходимую 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 обнаруживают и устраняют ошибки.

Вывод текстовой информации

При работе в Xcode Playground значения переменных и констант отображаются в области результатов. Напомню, что для этого в редакторе кода достаточно всего лишь написать имя объявленной и проинициализированной ранее переменной или константы (листинг 1.14).

Листинг 1.14

```
let userName = "Dolf" // в области результатов отобразится "Dolf"
var countOfFriends = 5 // в области результатов отобразится 5
userName // в области результатов отобразится "Dolf"
countOfFriends + 3 // в области результатов отобразится 8
```

Такой механизм контроля значений очень полезен, но 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, данные можно отобразить на отладочной консоли.

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

СИНТАКСИС

```
print(сообщение)
```

Вывод на отладочную консоль текстового сообщения.

`сообщение` — текстовое сообщение, выводимое на консоль.

Пример

```
print("Hello, Junior Swift Developer")
```

Консоль

```
Hello, Junior Swift Developer
```

Пример использования функции `print(_:)` приведен на рис. 1.32.

Рис. 1.32. 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 по имени.

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

Некоторые функции построены так, что при их вызове нужно передать входные данные, которые будут использованы функцией внутри нее (как у функции `print(_:)`). Такие данные называются **аргументами функции**. Они указываются в скобках после имени вызываемой функции. Пример вызовов функций с передачей аргументов приведен в листинге 1.15.

Листинг 1.15

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

Здесь вызывается функция с именем `anotherFunction` (такой функции не существует, это лишь пример), которая принимает входной аргумент `name` с текстовым значением `Bazil`.

> **ПРИМЕЧАНИЕ** Далее вы научитесь создавать собственные функции, некоторые из них также будут принимать аргументы в качестве входных данных. Переданное в функцию значение используется внутри нее и в ее пределах называется входным параметром.
>
> Не стоит путать понятия **аргумент функции** и **входной параметр**.
>
> **Аргумент функции** — это то, что передается в функцию в скобках при ее вызове.
>
> **Входной параметр** — это то, что используется внутри функции (разберемся с этим в ходе изучения функций).
>
> По сути, это может быть одно и то же значение, но во время вызова функции — это аргумент, а в теле функции — это параметр.

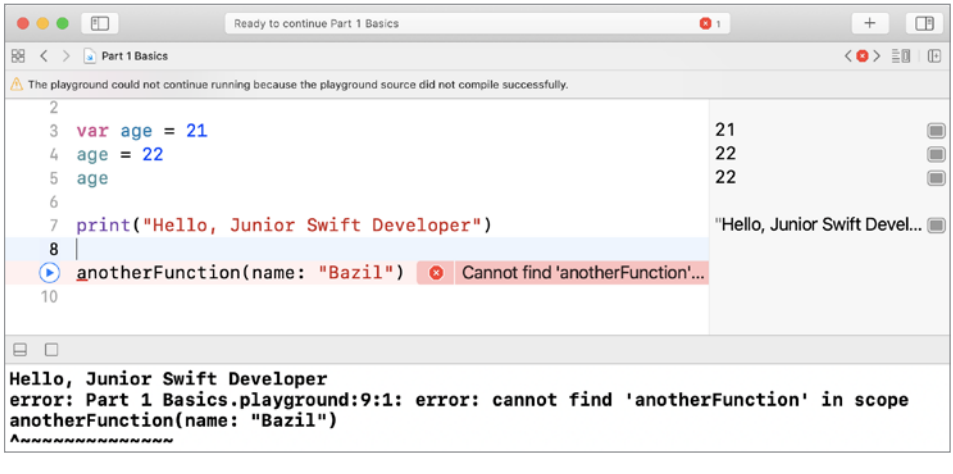

Рис. 1.33. Ошибка при вызове несуществующей функции

Если вы напишете данный код в своем Playground, то получите сообщение об ошибке, так как такой функции еще не существует, она пока не объявлена (рис. 1.33).

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. 1.34).

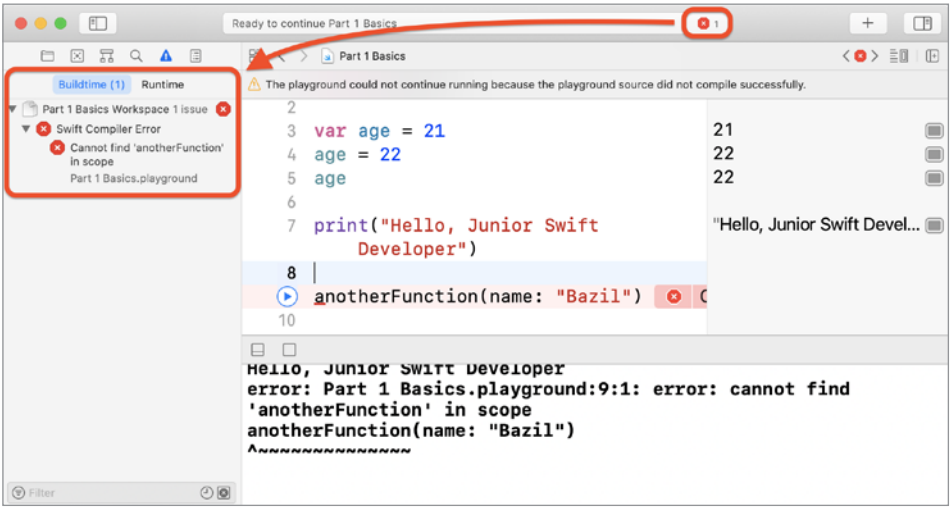

Рис. 1.34. Панель с информацией об ошибках

ПРИМЕЧАНИЕ Каждая функция имеет сигнатуру, то есть краткое описание. Сигнатура содержит информацию об имени функции, ее аргументах и возвращаемом функцией значении. Она может быть в одной из трех форм:

1. Полная, с указанием типов данных:

```
someFunction(a:String, b: String) -> String
```

2. Стандартная, без указания типов данных:

```
someFunction(a: b:)
```

3. Краткая, с указанием только имени функции:

```
someFunction
```

Если функция принимает входные аргументы, то это отражается в сигнатуре (далее приведены примеры с использованием стандартной формы).

- `print(_:)` описывает функцию с именем print, которая принимает один входной безымянный аргумент.
- `anotherFunction(name:)` описывает функцию с именем `anotherFunction`, которая принимает один аргумент с именем name.

- `someFunction(a:b:)` описывает функцию с именем `someFunction`, которая имеет два входных аргумента с именами a и b.
- `myFunction()` описывает функцию с именем myFunction, которая не имеет аргументов.

Таким образом, сигнатура кратко описывает не только название функции, но и ее аргументы.

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

Таким образом, сигнатура `goodFunction(_:text:)` указывает на функцию с именем `goodFunction`, которой нужны два аргумента: первый не имеет имени, а второй должен быть передан с именем `text`.

Пример вызова функции `goodFunction (_:text:)`:

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

Вернемся к рассмотрению функции `print(_:)` (листинг 1.16).

Листинг 1.16

```
// вывод информации на отладочную консоль
print("Hello, Swift Junior Developer")
```

Консоль

```
Hello, Swift Junior Developer
```

Код выводит в отладочную консоль текст, переданный в функцию `print(_:)`. Обратите внимание, что выводимый на консоль текст дублируется и в области вывода результатов, но при этом в конце добавляется символ переноса строки (`\n`) (рис. 1.35).

ПРИМЕЧАНИЕ Если отображаемое значение не влезает в область результатов и вы не видите окончания выведенного значения, расширьте ее, потянув за ее левую границу.

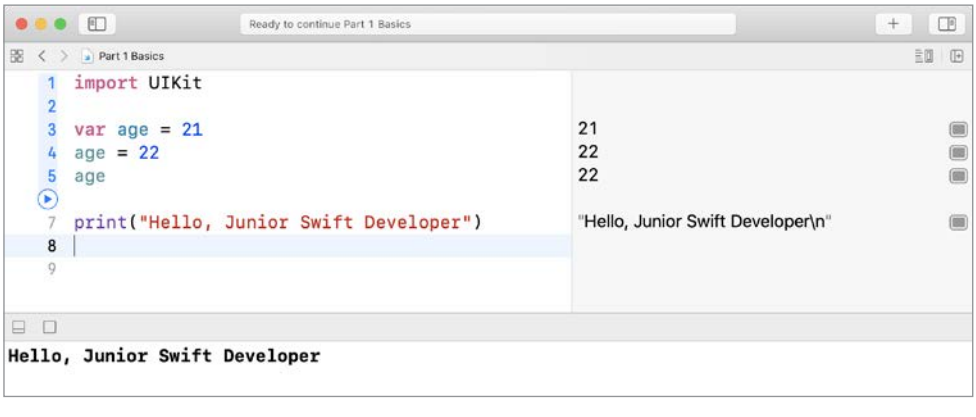

Рис. 1.35. Вывод текста на отладочную консоль

Функция `print(_:)` может принимать на вход не только текст, но и произвольный аргумент (переменную или константу), как показано в листинге 1.17.

Листинг 1.17

```
let foo = "Текст для вывода на консоль"
print(foo)
```

Консоль

```
Текст для вывода на консоль
```

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

Помимо этого, существует возможность объединить вывод текстовой информации со значением некоторого параметра (или параметров). Для этого используется символ обратной косой черты (слеша), после которого в круглых скобках нужно указать имя выводимого параметра (листинг 1.18).

Листинг 1.18

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

Консоль

```
Я изучаю Swift
```

Вы будете использовать функцию `print(_:)` довольно часто, особенно в ходе обучения разработке на Swift. Этой функцией удобно проверять текущее значение параметров, а также искать ошибки в алгоритме работы программы.

Глава 2. Фундаментальные типы данных

С понятием типа данных мы уже встречались в предыдущей главе. Там говорилось, что тип данных определяет вид информации, которая может храниться в параметре. Например, параметр со строковым типом данных сможет хранить только строковые значения.

Если точнее, то **тип данных** — это множество всех возможных значений, а также операций над ними. Например, если у параметра числовой тип данных, то ему может быть присвоено числовое значение, с которым можно проводить математические операции (сложение, вычитание, деление, умножение и т. д.). Воспринимайте типы данных как абстракции, шаблоны, на основе которых создаются конкретные значения. Например, на основе типа данных «целое число» можно создать значения 2, 17, 36. Сам по себе тип «целое число» лишь выделяет диапазон возможных значений, но не несет никакой конкретики, никакой точной величины.

Для лучшего понимания рассмотрим, что такое тип данных, на примере геометрических фигур. На рис. 2.1 изображены несколько кругов, квадратов и прямоугольников. Как вы думаете, что в данном случае можно назвать типом данных, а что конкретным значением? Порассуждаем на эту тему.

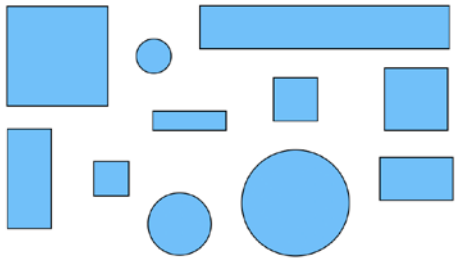

Рис. 2.1. Различные геометрические фигуры

Каждая фигура обладает множеством характеристик, одни из них могут быть уникальными (например, размер), а другие — общими сразу для нескольких фигур (например, форма). Используя какую-либо общую характеристику (например, форму), мы можем объединить фигуры в группы (рис. 2.2).

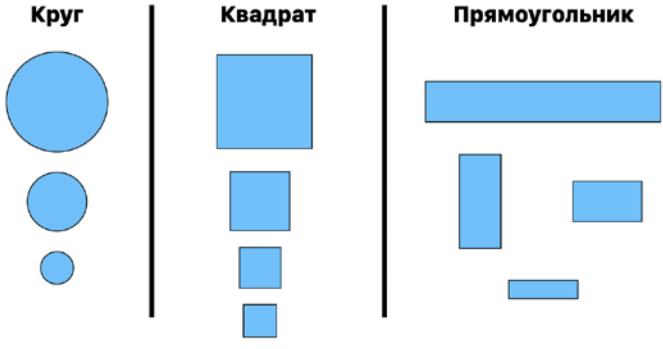

Рис. 2.2. Фигуры, объединенные по группам

Теперь форма фигуры, или, другими словами, тип фигуры, — это и есть аналог типа данных. Это шаблон, на основе которого создается фигура. А каждая конкретная фигура на рисунке — это и есть значение определенного типа (шар с радиусом 5 см, квадрат со стороной 10 см и т. д.). Тип фигуры позволяет указать уникальные характеристики для данной фигуры:

- Для типа «Круг» — это радиус.
- Для типа «Квадрат» — это длина стороны.
- Для типа «Прямоугольник» — это длины сторон.

Если мы говорим о какой-то типовой фигуре (о квадрате или о шаре в широком смысле), то говорим о типе данных; если мы говорим о конкретной фигуре (этот большой зеленый куб), расположенной перед нами, то говорим о конкретном значении этого типа.

Тип данных определяет не только характеристики объекта, но и то, что с ними можно делать. Так, круг можно катить, а квадрат и прямоугольник устойчиво устанавливать друг на друга.

Тип данных — это множество возможных значений и операций с этими значениями. В Swift типы данных определяют:

- *значения*, которые хранятся в параметрах: целые числа, дробные числа, строки и отдельные символы, логические значения или другой вид информации;
- *операции*, которые с этими значениями можно произвести: числа складывать или вычитать, строки объединять между собой и т. д.

В Swift, как и в других языках программирования, есть предустановленные фундаментальные типы данных, доступные вам «из коробки».

> **ПРИМЕЧАНИЕ** Swift Standard Library — это программная библиотека, в которой содержатся все фундаментальные типы данных и функциональные возможности языка. Вместе с компилятором, отладчиком и другими библиотеками и подсистемами они, по сути, и создают Swift таким, каким вы его скоро узнаете.

2.1. Предназначение типов данных

При объявлении параметра у него должен быть определен тип данных. У вас может возникнуть вопрос: как это было сделано в предыдущих листингах книги? Переменные и константы создавались, но каких-либо специальных конструкций для указания типа не использовалось. Swift, если можно так выразиться, — умный язык программирования. Он способен автоматически определить тип объявляемого параметра на основании инициализированного ему значения. Рассмотрим пример из листинга 2.1.

Листинг 2.1

```
let someText = "Я учу Свифт"
```

Данный пример можно прочитать так:

- Объявить константу с именем `someText` и присвоить ей значение `"Я учу Свифт"`.

При этом не надо дополнительно указывать, какой именно вид информации будет храниться в параметре, то есть не надо указывать его тип данных. Swift самостоятельно анализирует инициализируемое значение, после чего принимает решение о типе.

Основываясь на этом, прочитаем код предыдущего листинга:

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

Или по-другому:

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

`String` — это ключевое слово, используемое в Swift для указания на строковый тип данных. Он позволяет хранить в переменных и константах текст.

Операция, в которой Swift самостоятельно определяет тип объявляемого параметра, основываясь на переданном значении, 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 имени ставится двоеточие и с помощью ключевого слова указывается тип данных.

Рассмотрим еще несколько примеров объявления переменных со строковым типом данных (листинг 2.2).

Листинг 2.2

```
// 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 у всех переменных строковый тип данных `String`, то есть они могут хранить в себе строковые значения (текст).

У любой объявленной переменной или константы явно или неявно должен быть определен тип данных. В ином случае будет показано сообщение об ошибке (листинг 2.3).

Листинг 2.3

```
let firstHobbitsName // ОШИБКА: Type annotation missing in pattern (пропущено
указание типа данных)
```

Swift — язык со строгой типизацией. Однажды определив тип данных переменной или константы, вы уже не сможете его изменить. В каждый момент времени вы должны четко представлять, с каким типом значения работает ваш код.

> **ПРИМЕЧАНИЕ** К моменту первого обращения к параметру в нем должно содержаться значение. То есть вы можете создать произвольный параметр, указать его тип данных, но не указывать первоначальное значение. Но к моменту, когда данный параметр будет использован, в нем уже должно находиться значение.
>
> ```
> var str: String
> str = "Hello, playground"
> print(str)
> ```
>
> При этом совершенно неважно, создаете вы переменную или константу.
>
> Иногда при работе Xcode Playgroun среда разработки некорректно обрабатывает и отображает ошибку, сообщающую об отсутствующем значении у объявленного параметра. В этом случае просто укажите значение параметра в момент его объявления.

Все фундаментальные типы данных (строковые, числовые, логические и т. д.) являются значимыми (value type), то есть их значения передаются копированием. При передаче значения переменной или константы значимого типа в другую переменную или константу происходит копирование этого значения, в результате чего мы получаем два независимых параметра. Рассмотрим еще один пример работы со значимым типом данных (листинг 2.4).

Листинг 2.4

```
// неявно определим параметр целочисленного типа данных
var variableOne = 23
// явно определим параметр целочисленного типа данных
```

```
// после чего передадим ему значение другого параметра в качестве
// первоначального
let 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, предназначенный для хранения целых чисел.

В данном примере хранилище `variableOne` — значимого типа. При передаче значения, хранящегося в `variableOne`, в новую переменную `variableOneCopy` создается полная независимая копия. Никакие изменения, вносимые в `variableOne`, не повлияют на значение, хранящееся в `variableOneCopy`.

2.2. Числовые типы данных

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

ПРИМЕЧАНИЕ Для всех рассматриваемых в этой главе фундаментальных типов данных инициализируемые значения называются литералами: строковый литерал, числовой литерал и т. д.

Целочисленные типы данных

Целые числа — это числа, у которых отсутствует дробная часть, например 81 или 18. Целочисленные типы данных бывают знаковыми (позволяют хранить ноль, положительные и отрицательные значения) и беззнаковыми (позволяют хранить только ноль и положительные значения). Swift поддерживает как знаковые, так и беззнаковые целочисленные типы данных. Для записи значений 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 используются ключевые слова `Int` (для хранения только положительных значений) и `UInt` (для хранения как положительных, так и отрицательных значений).

Рассмотрим пример в листинге 2.5.

Листинг 2.5

```
// объявим переменную знакового целочисленного типа и присвоим ей значение
var signedNum: Int = -32
// объявим константу беззнакового целочисленного типа
// и проинициализируем ей значение
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` со значением 128.

Разница между знаковыми и беззнаковыми целочисленными типами в том, что значение знакового типа данных может находиться в интервале от -2^{n-2} до $+2^{n-2}$, а беззнакового — от 0 до $+2^{n-1}$, где n — разрядность типа данных (8, 16, 32 или 64).

В Swift существуют дополнительные целочисленные типы данных: `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64` и `UInt64`. Они определяют диапазон возможных значений, хранимых в параметрах: 8, 16, 32 и 64-битные числа. В табл. 2.1 приведены минимальные и максимальные значения каждого из перечисленных типов.

ПРИМЕЧАНИЕ Вам не надо запоминать приведенные в таблице значения. Достаточно сделать закладку на этой странице.

Таблица 2.1. Типы данных и значения

Тип данных	Минимальное значение	Максимальное значение
`Int`	–9223372036854775808	9223372036854775807
`Int8`	–128	127
`Int16`	–32768	32767
`Int32`	–2147483648	2147483647
`Int64`	–9223372036854775808	9223372036854775807
`UInt`	0	18446744073709551615
`UInt8`	0	255
`UInt16`	0	65535
`UInt32`	0	4294967295
`UInt64`	0	18446744073709551615

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 запомнить!

Код, приведенный ниже, вызовет ошибку, так как это попытка инициализировать значение разного типа.

```
let someNum: Int = 12
let 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 сущность, описанная кодом, объединяющая данные и действия, которые с этими данными можно совершить. Например, объектом является цифра `2` или `продуктовый автомат`, если, конечно, он описан языком программирования. Каждая сущность обладает 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 объекта в Swift осуществляется с помощью их имен, написанных через точку после имени объекта, к примеру:

```
ПродуктовыйАвтомат.максимальнаяВместимость
ПродуктовыйАвтомат.выдатьТовар("шоколадка")
```

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 — тоже объект. Каждый из рассмотренных числовых типов описывает сущность «целое число». Различие 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 2.6).

Листинг 2.6

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

Так как тип данных `UInt8` является беззнаковым и не предназначен для хранения отрицательных чисел, то и максимально возможное значение будет `255` вместо `127` у знакового `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 `Int`, в Swift наделен широкими возможностями. В дальнейшем вы узнаете о других возможностях обработки числовых значений.

Числа с плавающей точкой

Помимо целых чисел, при разработке приложений вы можете использовать дробные числа. Например, `3.14` и `-192.884022`.

ПРИМЕЧАНИЕ Дробные числа в программировании также называются числами с плавающей точкой.

Для хранения дробных чисел в большинстве случаев используются всего два типа данных: `Float` и `Double`, оба являются знаковыми (позволяют хранить положительные и отрицательные значения).

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

Пример объявления параметров приведен в листинге 2.7.

Листинг 2.7

```
// дробное число типа Float с явным указанием типа
let 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
let someNumber = 8.36
```

Обратите внимание, что тип константы `someNumber` задается неявно (с помощью переданного дробного числового значения). При передаче дробного значения без явного указания типа Swift всегда самостоятельно определяет для параметра 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:

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

Арифметические операторы

Ранее мы познакомились с типами данных, позволяющими хранить числовые значения в переменных и константах. С числами, которые мы храним, можно проводить 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.

  ```
  let 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.

  ```
  let res = + 3
  ```

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

  ```
  let res = 4 - 3 // 1
  ```

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

  ```
  let 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 могли 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.

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

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

```
let res = 20 / 4 // 5
```

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

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

ПРИМЕЧАНИЕ В Swift отсутствуют довольно популярные в других языках операторы инкремента (++) и декремента (--), увеличивающие и уменьшающие значение на единицу соответственно.

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

Чтобы продемонстрировать использование данных операторов, создадим две целочисленные и две дробные константы (листинг 2.8).

Листинг 2.8

```
// целочисленные константы
let numOne = 19
let numTwo = 4
// константы, имеющие тип числа с плавающей точкой
let numThree = 3.13
let numFour = 1.1
```

Для первых двух констант неявно задан целочисленный тип данных `Int`, для вторых двух неявно задан тип `Double`. Рассмотренные ранее операторы позволяют выполнить арифметические операции с объявленными переменными (листинг 2.9).

Листинг 2.9

```
// операция сложения
let sum = numOne + numTwo // 23
// операция вычитания
let diff = numOne - numTwo // 15
// операция умножения
let mult = numOne * numTwo // 76
// операция деления
let 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 константы `firstNum`, равной `19`, на константу `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 2.10).

Листинг 2.10

```
// операция сложения
let sumD = numThree + numFour  // 4,23
// операция вычитания
let diffD = numThree - numFour // 2,03
// операция умножения
let multD = numThree * numFour // 3,443
// операция деления
let 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. 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 2.11.

Листинг 2.11

```
// операция получения остатка от деления
let res1 = numOne % numTwo   // 3
let res2 = -numOne % numTwo  // -3
let res3 = numOne %  -numTwo // 3
```

Распишем подробнее вычисление значения константы 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 (рис. 2.3).

Таким образом, остаток от деления всегда меньше делителя.

В листинге 2.11 при вычислении значений констант 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). Пример использования метода приведен в листинге 2.12.

Рис. 2.3. Операция вычисления остатка от деления

Листинг 2.12

```
// дробные константы
let firstFloat: Float = 3.14
let secondFloat: Float = 1.01
// операция получения остатка от деления
let result1 = firstFloat.truncatingRemainder(dividingBy: secondFloat) // 0.1100001
let result2 = -firstFloat.truncatingRemainder(
              dividingBy: secondFloat)  // -0.1100001
let result3 = firstFloat.truncatingRemainder(
              dividingBy: -secondFloat) // 0.1100001
```

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

Операция вычисления остатка от деления очень удобна в тех случаях, когда нужно 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.

ПРИМЕЧАНИЕ Для определения четности 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. Этот 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 — методы. 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 вызывается так:

```
ИмяТипаДанных.init(_:)
```

например:

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

В результате создается константа `numObj` целочисленного знакового типа `Int` со значением 2.

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

```
ИмяТипаДанных(_:)
```

например:

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

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

В дальнейшем 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 нужно перемножить два числа: целое типа `Int` и дробное типа `Double` (листинг 2.13).

Листинг 2.13

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

Существует два подхода к перемножению чисел в константах `numberInt` и `numberDouble`:

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

По выводу в области результатов видно, что переменная `resD` имеет более точное значение, чем константа `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 максимально сокращает объем кода. И чем глубже вы будете постигать этот замечательный язык, тем больше приемов, способных облегчить жизнь, вы узнаете. Один из них — 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 2.14).

Листинг 2.14

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

Чтобы использовать составной оператор присваивания, после оператора арифметической операции без пробелов нужно 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. В листинге 2.15 показаны несколько вариантов увеличения значения на единицу.

Листинг 2.15

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

Каждое выражение увеличивает `itterationValue` на единицу.

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

Способы записи числовых значений

Если в школе у вас была информатика, то, возможно, вы знаете, что существуют различные системы счисления, например десятичная или двоичная. В реальном мире используется десятичная система, в то время как в компьютере все вычисления происходят в двоичной системе.

Swift позволяет записывать числовые литералы в самых популярных системах 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. Ниже (листинг 2.16) приведен пример записи числа `17` в различных системах.

Листинг 2.16

```
// 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 счисления) 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 2.17 приведен пример записи десятичного числа 12,1875 в различных системах счисления и с использованием экспоненты.

Листинг 2.17

```
// 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 любой из систем счисления. В области результатов вы всегда увидите результат выполнения в десятичном виде.

Для визуального отделения порядков числа можно использовать символ нижнего подчеркивания (underscore) (листинг 2.18).

Листинг 2.18

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

В обоих случаях, что с символом нижнего подчеркивания, что без него, получилось одно и то же число. Нижнее подчеркивание можно использовать для любого числового типа данных и для любой системы счисления.

Тип данных Decimal и точность операций

Вы уже знаете, как проводить операции с числовыми типами данных. И в большинстве случаев у вас не возникнет каких-либо проблем с точностью вычислений. Но рано или поздно вы можете встретиться с тем, что прогнозируемый ответ не совпадает с реально полученным. Рассмотрим пример из листинга 2.19.

Листинг 2.19

```
var myWallet: Double = 0
let incomeAfterOperation: Double = 0.1
```

Переменная `myWallet` описывает кошелек, а константа `incomeAfterOperation` — доход, который получает пользователь после совершения операции. Подразумевается, что в результате каждой операции общая сумма денег в кошельке увеличивается на 10 копеек. Предположим, что пользователь провел три операции, значит, значение кошелька нужно увеличить трижды (листинг 2.20).

Листинг 2.20

```
myWallet += incomeAfterOperation
print(myWallet)
myWallet += incomeAfterOperation
print(myWallet)
myWallet += incomeAfterOperation
print(myWallet)
```

Консоль

```
0.1
0.2
0.30000000000000004
```

Что вы видите на консоли? Неожиданно, но по неизвестной пока причине третий вывод вместо `0.3` показал очень приближенное к нему, но все же отличающееся значение `0.30000000000000004`.

Причина этого заключается в особенностях работы компьютера с дробными числами. Как вы знаете, любое число на аппаратном уровне рассматривается компьютером как совокупность 0 и 1. Так, число `2` — это `10`, число `13` — это `1101`, и т. д. Компьютеру очень просто и удобно оперировать с такими числами, перемножать их, вычитать, делить, то есть проводить любые математические операции.

Ошибки в точности вычислений могут возникнуть, когда вы работаете с числами с плавающей точкой. В некоторых случаях компьютер не может точно представить число в двоичной форме и использует максимально близкое по значению. Мы видим это и в случае с `incomeAfterOperation`. При попытке перевести `0.1` в двоичный вид мы получим бесконечное число с повторяющимся набором символов `0011` (рис. 2.4). Из-за ограничений на длину значения при проведении расчетов потребуется отбросить часть «бесконечного хвоста», что в результате и может привести к неожиданным ошибкам.

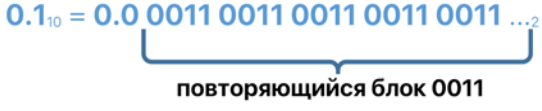

Рис. 2.4. Перевод числа в двоичную систему

Самое интересное, что примеров подобных чисел — бесконечное множество. И для любого из них с увеличением количества проведенных операций будет накапливаться ошибка, которая в конечном счете всплывет в расчетах.

> **ПРИМЕЧАНИЕ** При желании, и я вам очень советую сделать это, вы можете самостоятельно изучить метод перевода чисел из десятичной системы в двоичную. Это поможет вам еще лучше разобраться в проблеме, а не слепо верить тому, что я пишу.

Теперь представьте, что вы владелец биржи криптовалют (или любого другого финансового сервиса). Пользователи ежесекундно проводят операции с валютой, ошибка накапливается, позволяя вам «оставлять» незначительную сумму с каждой операции внутри сервиса. И в один момент об этом узнают правоохранительные органы, — результат предсказуем. Такая маленькая оплошность привела к таким большим проблемам.

Когда ваша программа оперирует числовыми значениями, имеющими базу 10 (к примеру, денежные суммы), а вам критически важна точность вычислений, используйте тип данных `Decimal`.

> **ПРИМЕЧАНИЕ** Если число записано с использованием только 0 и 1 (в двоичном виде) говорят, что оно имеет базу 2, то есть две цифры, с помощью которых представляется. У чисел в разных системах счисления разная база: у шестнадцатеричной — база 16, у десятичной — база 10 и т. д.

Тип `Decimal` отличается от `Float` и `Double` тем, что с его помощью можно с высокой точностью проводить операции с числами с плавающей точкой, имеющими

базу 10. Этот тип обслуживается не просто аппаратной частью вашего компьютера, где из десятичного числа получается двоичное (и вместе с этим преобразованием возникают и ошибки в точности), в интересах `Decimal` работают специальные программные компоненты, которые обеспечивают его точность. Тем не менее на аппаратном уровне (в процессоре) числа все равно представляются как совокупность 0 и 1, но сложная логика работы этих программных компонентов компенсирует все возникающие ошибки.

Изменим тип данных примера выше на `Decimal` и посмотрим на результаты работы (листинг 2.21).

> **ПРИМЕЧАНИЕ** Обратите внимание, что при работе с `Decimal` в самом начале страницы Playground должно присутствовать выражение `import Foundation`. Данный тип не фундаментальный, он входит в состав библиотеки Foundation, по этой причине ее нужно импортировать в проект.

Листинг 2.21

```
import Foundation
var decimalWallet: Decimal = 0
let income: Decimal = 0.1
decimalWallet += income
print(decimalWallet)
decimalWallet += income
print(decimalWallet)
decimalWallet += income
print(decimalWallet)
```

Консоль

```
0.1
0.2
0.3
```

Проблема решена, количество денег в кошельке ровно то, какое и ожидалось.

Использование `Decimal` имеет свои недостатки. К примеру, этот тип данных работает значительно медленнее, чем `Float` или `Double`, а также потребляет больше памяти. Нет необходимости использовать `Decimal` постоянно. В случаях, когда значения могут быть измерены (физические величины), применяйте `Float` и `Double`. В случаях, когда значения могут быть сосчитаны (деньги), используйте `Decimal`.

2.3. Строковые типы данных

Числа очень важны в программировании, но все же это не единственный вид данных, с которым вы будете работать в ходе разработки приложений. Строковые типы данных также очень распространены. Они позволяют работать с текстом. Можете представить программу без текста? Это непростая задача! В качестве примеров текстовых данных могут быть имена людей, адреса их проживания,

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 вы 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 о принципах работы компьютера было сказано, что компьютер представляет 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 должен существовать механизм, используя который возможно представить любые текстовые данные в виде последовательности нулей и единиц. Такое преобразование происходит в два этапа:

1) символы преобразуются в числовую последовательность в десятичной или шестнадцатеричной системе счисления;
2) полученные числовые комбинации переводятся в понятные компьютеру двоичные числа.

Для решения этих задач используются стандарты кодирования, наиболее современным из которых является Юникод (Unicode).

Unicode — это международный стандарт, описывающий, каким образом строки преобразуются в числа. До принятия Юникода было множество других стандартов, но в силу своей ограниченности они не могли стать международными. Наверняка многие из вас помнят те времена, когда, заходя на сайт или открывая документ, вы видели устрашающие кракозябры и совершенно нечитаемый текст. С тех пор как Всемирная паутина перешла на Юникод, такая проблема исчезла.

Главная особенность Юникода в том, 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 `a` имеет кодовую точку `97` (в десятичной системе счисления) или `0x61` (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 наиболее часто вы будете использовать десятичную систему, однако время от времени придется прибегать и к шестнадцатеричной, и к двоичной.

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 Юникоде 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. 2.2).

Таблица 2.2. 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, `0x100A1`).

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

Хоть мы и перевели слова в шестнадцатеричные числа, на аппаратном уровне должны быть лишь 0 и 1. Поэтому получившуюся числовую последовательность необходимо повторно закодировать. Для этого 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 последовательности кодов `53`, `57`, `49`, `46`, `54` может получиться `00110101001110010011000100101110000110110`, а эту последовательность без каких-либо проблем можно передать по каналам связи или обработать на аппаратном уровне.

Таким образом, стандарт 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 очень пригодится 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 основных типа данных:

- тип `Character` предназначен для хранения отдельных символов;
- тип `String` предназначен для хранения произвольной текстовой информации.

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. Пример использования `Character` приведен в листинге 2.22.

Листинг 2.22

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

Консоль

```
a
```

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

Если 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 любой длины.

Чтобы определить параметр строкового типа, нужно использовать ключевое слово `String`. Пример приведен в листинге 2.23.

Листинг 2.23

```
// константа типа String
// тип данных задается явно
let stringOne: String = "Dragon"
```

При объявлении константы `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 нужно указывать тип объявляемого строкового параметра, так как Swift неявно задает его при передаче строки любого размера (листинг 2.24).

Листинг 2.24

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

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

Запомните, что литерал, состоящий даже из одного символа, всегда неявно определяется как `String`. Для `Character` нужно явно указать тип данных.

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 2.25).

Листинг 2.25

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

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

> **ПРИМЕЧАНИЕ** 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 несоответствия типов, так как пустая строка не является *отдельным символом* (рис. 2.5). Она является *пустой строкой*.

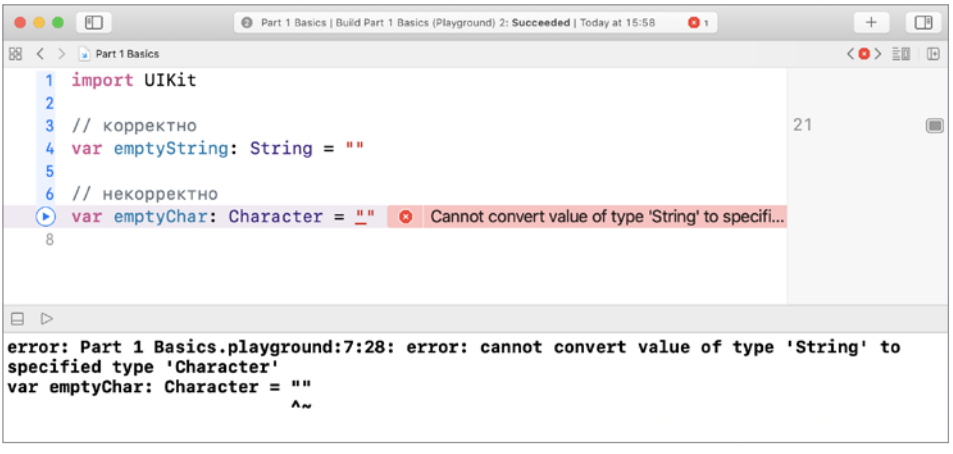

Рис. 2.5. Попытка инициализировать пустой строковый литерал

Многострочные строковые литералы

В Swift можно писать строковые литералы в несколько строк, разделяя их символом переноса (нажатием клавиши Enter). В этом случае текст нужно обрамить с обеих сторон тремя двойными кавычками. При этом:

- открывающие и закрывающие тройки кавычек должны находиться на строке, не содержащей текст литерала:

```
// корректно
"""
строковый_литерал
"""
```

- закрывающая тройка кавычек должна находиться в одном столбце или левее, чем текст:

```
// некорректно
// у нижних кавычек есть отступ
"""
строковый_литерал
 """
```

Пример приведен в листинге 2.26.

Листинг 2.26

```
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 (листинг 2.27).

Листинг 2.27

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

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

С помощью этой 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 2.28).

Листинг 2.28

```
// константа типа Double
let someDoubleNumber = 74.22
// строка, созданная на основе константы типа Double
let someStringNumber = String(someDoubleNumber) // "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 2.29).

Листинг 2.29

```
// переменная типа String
let name = "Дракон"
// константа типа String с использованием интерполяции
let hello = "Привет, меня зовут \(name)!"
// интерполяция с использованием выражения
let 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 константа `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 нескольких строковых значений в одно с помощью оператора сложения (+) (листинг 2.30).

Листинг 2.30

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

Консоль

```
Мой вес 12.4 тонны
```

В данном примере используется оператор сложения для объединения строковых значений. Тип данных константы `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

В Swift мы также можем сравнивать строки. Для этого используется оператор сравнения (==). Рассмотрим пример в листинге 2.31.

Листинг 2.31

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

Значения, отображаемые в области результатов (`false`, `true`), показывают результат сравнения строк: `false` означает отрицательный результат, а `true` — положительный.

ПРИМЕЧАНИЕ Такая операция сравнения называется *логической*. Она может быть проведена и для числовых значений. Подробнее с ней мы познакомимся 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 2.32 приведен пример использования юникод-скаляра для инициализации кириллического символа `K` в качестве значения параметра типа `Character`.

Листинг 2.32

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

Но не только тип `Character` совместим с Юникод, вы также можете использовать скаляры и для значений типа `String` (листинг 2.33).

Листинг 2.33

```
let 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 дальнейшем мы вернемся к вопросу работы с Юникодом, более подробно изучив методы взаимодействия со строковыми данными.

2.4. Логический тип данных

Изучение фундаментальных типов данных не заканчивается 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 две константы логического типа данных (листинг 2.34).

Листинг 2.34

```
// константа с неявно заданным логическим типом
let isDragon = true
// константа с явно заданным логическим 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` (мы познакомимся с ним несколько позже), который, в зависимости от значения переданного ему параметра, пускает выполнение программы по различным ветвям (листинг 2.35).

Листинг 2.35

```
// 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. Язык 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 а» (листинг 2.36).

Листинг 2.36

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

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

Бинарный оператор *логического И* записывается в виде удвоенного символа амперсанда 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 2.37).

Листинг 2.37

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

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

Бинарный оператор *логического ИЛИ* — это 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 2.38).

Листинг 2.38

```
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 определяет, есть ли среди значений переданных ему операндов хотя бы одно истинное.

Логические операторы можно группировать между собой, создавая сложные логические структуры. Пример показан в листинге 2.39.

Листинг 2.39

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

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

Указать порядок вычисления операций можно с помощью круглых скобок (точно как в математических примерах). То, что указано в скобках, будет выполняться в первую очередь (листинг 2.40).

Листинг 2.40

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

Операторы сравнения

Swift позволяет сравнивать однотипные значения друг с другом. Для этого используются операторы сравнения. Результат их работы — 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 операторов возвращает значение, которое указывает на истинность утверждения. Несколько примеров и значений, которые они возвращают, приведены в листинге 2.41.

Листинг 2.41

```
// 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.

2.5. Псевдонимы типов

В Swift есть возможность создать псевдоним для любого типа данных. *Псевдонимом типа* называется дополнительное имя, по которому будет происходить обращение к этому типу. Для создания псевдонима используется оператор typealias. Псевдоним нужен тогда, когда существующее имя типа неудобно использовать в контексте программы (листинг 2.42).

Листинг 2.42

```
// 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 любое количество псевдонимов. И все псевдонимы вместе с оригинальным названием типа можно использовать в программе (листинг 2.43).

Листинг 2.43

```
// определяем псевдоним для типа String
typealias TextType = String
typealias WordType = String
typealias CharType = String
// создаем константы каждого типа
let someText: TextType = "Это текст"
let someWord: WordType = "Слово"
let someChar: CharType = "Б"
let someString: String = "Строка типа String"
```

В данном примере для типа данных String определяется три разных псевдонима. Каждый из них, наравне с основным типом, может использоваться для объявления параметров.

Созданный псевдоним наделяет параметры теми же возможностями, что и родительский тип данных. Однажды объявив его, вы сможете использовать этот псевдоним для доступа к свойствам и методам типа (листинг 2.44).

Листинг 2.44

```
// объявляем псевдоним
typealias AgeType = UInt8
/* используем свойство типа
UInt8 через его псевдоним */
let 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 `AgeType` для доступа к типу данных `UInt8` и свойству `max`.

> **ПРИМЕЧАНИЕ** Запомните, что псевдонимы можно использовать совершенно для 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.

2.6. 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 параметра

Объявим новую целочисленную переменную (листинг 2.45).

Листинг 2.45

```
var intVar = 12
```

Для параметра `intVar` неявно (на основании его значения) определен тип данных `Int`. Чтобы убедиться в этом, сделайте следующее:

- Зажмите клавишу `Option` на клавиатуре.
- Наведите указатель мыши на название параметра `intVar`. При этом вместо стрелочки вы должны увидеть знак вопроса.
- Щелкните левой кнопкой мыши.

В появившемся всплывающем окне вы можете найти справочную информацию о параметре, включая и указание его типа данных (рис. 2.6).

Рис. 2.6. Справочное окно с указанием типа данных

Этот способ работает с любыми объектами в коде. В справочном окне содержится вспомогательная информация и текстовое описание.

Другой способ узнать тип данных — функция `type(of:)`. В качестве аргумента необходимо передать имя параметра, тип которого нужно определить. Функция вернет значение, указывающее на тип данных. Пример использования `type(of:)` показан в листинге 2.46.

Листинг 2.46

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

Консоль

```
Double
```

В дальнейшем мы довольно часто будем применять данную функцию для определения типа параметров.

> **ПРИМЕЧАНИЕ** При использовании `type(of:)` 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 распределены по категориям, каждая из которых определяет наличие у конкретного типа некоторой функциональности. Другими словами, если тип данных обеспечивает определенную функциональность, то он может быть отнесен к определенной категории, к которой также могут быть отнесены и другие типы данных. При этом типы одновременно могут входить в несколько категорий, то есть обеспечивать разную функциональность.

Примером таких категорий могут выступать *сопоставимые типы данных* и *хешируемые типы данных*.

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

Определить, является ли тип данных сопоставимым, очень просто. Достаточно сравнить два значения этого типа с помощью логических операторов. 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 (логично, что `true` не может быть больше или меньше `false`).

> **ПРИМЕЧАНИЕ** Обратите внимание, что для сравнения используются логические операторы <, <=, >= и >. Операторы эквивалентности == и неэквивалентности != отсутствуют в этом списке. Если тип данных позволяет использовать == и != для сравнения значений, то он относится к категории **эквивалентных** (`Equatable`).
>
> Swift имеет большое количество различных категорий, по которым могут быть разделены типы данных. Более того, со временем вы научитесь использовать категории при создании собственных типов.

В листинге 2.47 показан пример проверки типа `String` и `Bool` на предмет возможности сопоставления значений.

Листинг 2.47

```
"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).

С помощью разделения типов по категориям можно задать требования к обрабатываемым данным. К примеру, вы можете создать специальную функцию, рассчитывающую квадрат переданного числа, а в качестве требования к входному аргументу жестко определить возможность передачи значений только числового типа (категория `Numeric`).

Хешируемым (`Hashable`) называется тип данных, для значения которого может быть рассчитан специальный цифровой код — хеш.

> **ПРИМЕЧАНИЕ** Хеш — это понятие из области криптографии. В общем случае хеш 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 значение его типа имеет свойство `hashValue`, к которому вы можете обратиться (листинг 2.48).

Листинг 2.48

```
let stringForHash = "Строка текста"
let intForHash = 23
let 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.

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 https://swiftme.ru.

2.7. Где использовать фундаментальные типы

Описываемый механизм	Где используется
Тип `Int`	Используется для хранения отрицательных и положительных целочисленных значений. **Пример:** • Текущая температура. `var temperature: Int = 32` • Рейтинг пользователя. `var userRating: Int = -182`
Тип `UInt`	Используется для хранения неотрицательных целочисленных значений. **Пример:** • Количество оставшихся противников. `var enemyCount: UInt = 5`
Типы `Int8`, `Int16`, `Int32`, `Int64`, `UInt8`, `UInt16`, `UInt32`, `UInt64`	Указанными типами очень удобно моделировать предметную область, где значения натуральным образом ограничены количеством битов, соответствующим типу данных. Например, интенсивность цветового канала удобно хранить в `UInt8` или `UInt16`, отсчет аудиосигнала — в `UInt16`, и т. п.
Тип `Float`	Используется при необходимости компактного хранения и эффективной работы с дробными числами, имеющими до 6 знаков в дробной части. **Пример:** • Скорость ветра. `var windSpeed: Float = 5.2 // 5.2 метра в секунду`

Описываемый механизм	Где используется
Тип `Double`	Используется для хранения и эффективной работы с дробными числами, имеющими до 15 знаков в дробной части. **Пример:** • Число Pi. `let pi: Double = 3.1415926535`
Тип `Decimal`	Используется для хранения дробных чисел, имеющих до 38 знаков в дробной части в случаях, когда требуется повышенная точность расчетов чисел. **Пример:** • Баланс счета пользователя. `var criptoMoney = Decimal(string: "0.10000000000000000000000000000000000001")`
Тип `String`	Используется для хранения строковых данных. **Пример:** • Логин пользователя. `let userLogin = "dexter"` • Название организации. `let organizationName = "Рога и копыта"`
Тип `Character`	Используется для хранения строковых данных длиной в один символ.
Тип `Bool`	Используется для хранения логических значений (`true` и `false`). **Пример:** • Состояние переключателя. `var isSwitcherOn = true` • Пользователь «в сети». `var userOnline = false`
Псевдонимы типов	Используется, когда это может улучшить читабельность программного кода. В особенности для сложных типов данных. **Пример:** • Для соответствия контексту. `typealias Group = Array<Student>` `// Тип Student – это структура, описывающая студента` • Для упрощения типа при его частом использовании. `typealias MapObjectProperties = [String: Any]` • Для упрощения сложного типа данных. `typealias UserInfo = (firstname: String, lastname: String)`

Описываемый механизм	Где используется
	• При указании соответствия нескольким типам. `typealias TransitionDelegate = UIViewController & UIViewControllerTransitionDelegate` • При передаче замыкания. `typealias Completion = (Int) -> String` `func someMethod(completion: Completion) {` `    // ...` `}` `// Идентификатор Completion несет для читателя кода больше информации о предназначении, чем (Int) -> String`

Часть II
КОНТЕЙНЕРНЫЕ ТИПЫ ДАННЫХ

Вернемся к рассмотренному ранее примеру, в котором мы проводили соответствие типов данных и геометрических фигур. Каждая фигура — это конкретное значение, созданное на основе определенного типа данных. У вас есть возможность взаимодействовать с любой из фигур, взять ее в руки, перекрасить, толкнуть и т. д. Но при необходимости переместить все объекты потребуется сделать это с каждым из них в отдельности. Но что, если мы разместим все фигуры в корзине? Для нас эта корзина — один объект, удобный для транспортировки и использования. При таком подходе мы можем переместить все фигуры за один раз, а при необходимости достать одну из фигур, после чего использовать ее по предназначению.

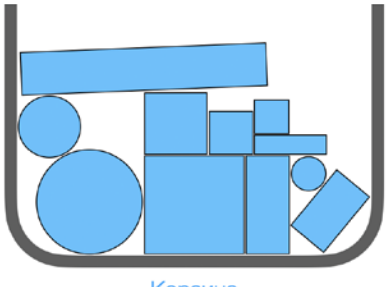

Рис II.1. Корзина для фигур

В Swift доступны специальные функциональные элементы, называемые **контейнерными типами данных**, позволяющие объединять различные значения в единое. В примере выше: для вас «корзина» — это один объект, но в нем содержится множество отдельных фигур. Знание и умение применять их очень важно, так как без них нельзя представить ни одной хорошей программы.

- ✓ Глава 3. Кортежи (Tuple)
- ✓ Глава 4. Последовательности и коллекции
- ✓ Глава 5. Диапазоны (Range)
- ✓ Глава 6. Массивы (Array)
- ✓ Глава 7. Множества (Set)
- ✓ Глава 8. Словари (Dictionary)
- ✓ Глава 9. Строка — коллекция символов (String)

Глава 3. 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, например, могут использоваться для работы с координатами. Куда удобнее использовать конструкцию (*x, y, z*), записанную в одну переменную, чем создавать по отдельной переменной для каждой оси координат.

3.1. Основные сведения о кортежах

Кортеж (tuple) — это объект, который группирует значения различных типов в пределах одного составного значения. При этом у вас есть возможность обратиться к каждому элементу кортежа напрямую, по его идентификатору (индексу). Если вернуться к примеру с корзиной геометрических фигур, то кортеж — это маленькая корзина, которая может содержать объекты любой формы (значения любых типов данных) в строго упорядоченном порядке (рис. 3.1). В любой момент вы можете получить доступ к требуемому элементу в корзине.

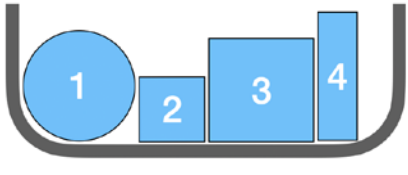

Рис 3.1. Кортеж в виде корзины с объектами

Кортежи представляют собой 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 я не рекомендую использовать больше трех или четырех, в ином случае стоит рассмотреть использование других возможностей языка (структуры, классы и перечисления).

ПРИМЕЧАНИЕ В ходе чтения книги вы будете встречать большое количество блоков «Синтаксис». В них описывается, каким образом могут быть использованы рассматриваемые конструкции. 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 (`:`):

```
описываемыйЭлемент: String
```

— этот элемент должен иметь тип данных `String`.

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 (`->`):

```
описываемыйЭлемент -> Int
```

— в качестве этого элемента может быть использовано выражение, возвращающее целочисленное значение типа `Int`.

Может быть определен как один

```
описываемыйЭлемент: Int
```

так и множество типов данных:

```
описываемыйЭлемент: Int, UInt
```

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)
```

Так происходит 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 3.1).

Листинг 3.1

```
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 некой 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 в константу.

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 3.1 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 3.2 производится сравнение типов данных различных кортежей.

Листинг 3.2

```
let tuple1 = (200, "In Work", true)
let 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 3.3).

Листинг 3.3

```
// 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 делать больше 7 элементов). В листинге 3.4 приведен пример создания кортежа из 4 элементов одного типа. При этом используется псевдоним типа данных `Int`, что не запрещается.

Листинг 3.4

```
// 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)
```

3.2. 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 3.5).

Листинг 3.5

```
// записываем значения кортежа в переменные
let (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 3.6).

Листинг 3.6

```
/* 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 3.7).

Листинг 3.7

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

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

ПРИМЕЧАНИЕ Символ нижнего подчеркивания в Swift интерпретируется как 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 3.8 приведен пример доступа к отдельным элементам кортежа.

Листинг 3.8

```
// 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. Имя элемента не является обязательным параметром и используется только для удобства использования кортежей. Имена могут быть даны как всем, так и части элементов. Имя элемента может быть определено как в литерале кортежа, так и при явном определении его типа.

В листинге 3.9 показан пример определения имен элементов кортежа через литерал.

Листинг 3.9

```
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 3.10 показано, как используются имена элементов совместно с индексами.

Листинг 3.10

```
// 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 3.11 показан соответствующий пример.

Листинг 3.11

```
/* 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 3.12.

Листинг 3.12

```
var myFirstTuple: (Int, String) = (0, "0")
let 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 3.13).

Листинг 3.13

```
// 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 небольшое 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.

3.3. 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 3.14).

Листинг 3.14

```
(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 7. 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.

3.4. Где используются кортежи

Описываемый механизм	Где используется
Кортежи	Используются для группировки нескольких значений «на лету», то есть оперативно, без создания более сложных сущностей. В ином случае задумайтесь об использовании коллекции (`Collection`), структуры (`struct`) или класса (`class`) вместо кортежа. **Пример:** • при возвращении нескольких значений из функции: `func someFunction(id: Int) -> (firstname: String, lastname: String) { // ... } // могут быть удобно разобраны на отдельные параметры // на приемной стороне let (firstname, lastname) = someFunction(id: 12)` • при необходимости передать несколько разнотипных элементов, так как в коллекциях хранятся только однотипные элементы; • при необходимости передать конкретное количество значений, так как в коллекциях количество элементов может меняться.

Глава 4. Последовательности и коллекции

Как и в любом языке программирования, в Swift есть механизмы агрегации значений с целью манипуляции ими в дальнейшем 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.

4.1. Классификация понятий

Для любого реального объекта (объекта физического мира) можно выделить множество характерных для него черт, которые могут быть как уникальными, так и не уникальными. На основе общих свойств различные объекты могут быть сгруппированы и связаны друг с другом.

Предположим, что есть группа объектов, относящихся к категории «Автомобили». Какие свойства характерны для этих объектов? Автомобиль — это техническое средство, обладающее, в самом упрощенном виде, колесами и двигателем (механизмом, позволяющим приводить колеса в действие). Любая конкретная модель авто обладает описанными свойствами (имеет колеса и двигатель), то есть входит в состав данной категории (рис. 4.1).

Рис. 4.1. Классификация автомобилей

Мы можем пойти дальше и выделить еще множество других характеристик, которые позволят разбить категорию «Автомобили» на подкатегории. К примеру, по источнику энергии автомобили можно разделить на те, что используют электродвигатель, и те, что используют двигатель внутреннего сгорания. Теперь объекты (автомобили) можно разбить на подкатегории (рис. 4.2).

Рис. 4.2. Классификация автомобилей

Каждая категория определяет различные свойства, которыми должен обладать объект. При этом объекты могут входить сразу в несколько категорий (рис. 4.3).

Рис. 4.3. Классификация автомобилей

Категории предъявляют требования к объекту, но не описывают, как именно эти требования должны быть реализованы. Даже если у нового BMW будут квадратные колеса, но он будет ехать — это все равно автомобиль.

Но категория — это понятие из реального, физического мира. Мы же с вами занимаемся разработкой на Swift.

Ранее мы рассматривали примеры классификации типов данных по различным категориям: `Equatable`, `Hashable` и `Comparable`. Но в данном языке они называются **протоколами** (а в некоторых других языках программирования вы могли встречать понятие **интерфейса**). На рис. 4.4 приведен пример классификации типов данных по указанным категориям (протоколам).

> **ПРИМЕЧАНИЕ** Данный вариант классификации типов данных является упрощенным, в нем показаны далеко не все доступные протоколы.

Протокол, как и категория, — это набор требований. Например, протокол `Comparable` требует, чтобы тип данных позволял сравнить два значения этого типа между собой. Как именно будет происходить сравнение, не вопрос протокола, а вопрос конкретной реализации этого требования в типе данных. Именно реализация (внутренняя структура) типа данных определяет, как будут выполнены требования

того или иного протокола. При этом, как и физические объекты, типы данных могут выполнять требования множества протоколов, а могут и не выполнять их вовсе.

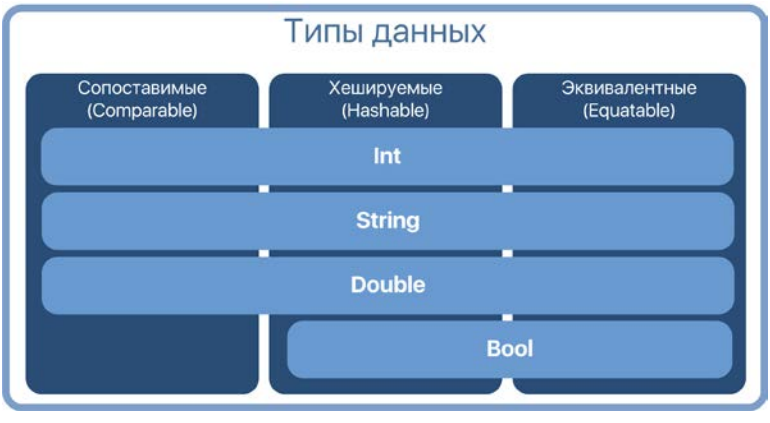

Рис. 4.4. Классификация типов данных

На основании типа данных создается **объект** (или **экземпляр**, это синонимы). К примеру, на основании `Int` создается объект «цифра 5».

Протокол > Тип данных > Объект (экземпляр) — это типовой вариант реализации функционала в Swift, это относится к методологиям объектно-ориентированного и протокол-ориентированного программирования.

> **ПРИМЕЧАНИЕ** Думаю, вы заметили, что материал книги постепенно усложняется. Не все из вас 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.
>
> В любом случае все, о чем я говорю, — это очень важно! Вам необходимо читать, перечитывать и вчитываться, стараясь получить из описания максимум полезной информации.

4.2. Последовательности (Sequence)

Вернемся к рассмотренному ранее примеру с корзиной, но внесем два изменения (рис. 4.5):

- все объекты имеют единую форму (одного типа данных), но различные характеристики (цвет и размер);
- объекты располагаются один над другим.

При таком расположении вы имеете доступ только к верхнему объекту. Вы можете взять его и изучить характеристики (определить цвет и размер). Все, что вы знаете об остальных объектах, — это только тип данных, но их общее количество, цвет или

размер вам неизвестны. Для доступа к нижележащим объектам вам потребуется перебрать все предыдущие.

Перед вами возникла задача определить, есть ли среди содержимого корзины объект зеленого цвета. Вы смотрите на верхний элемент и понимаете, что он синий. Достаете его — и видите, что он фиолетовый. Достаете следующий — и видите нужный зеленый (рис. 4.6).

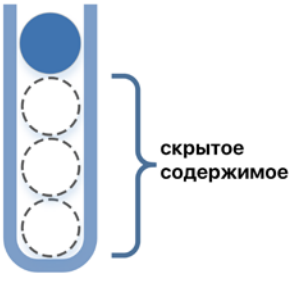

Рис. 4.5. Последовательность в виде корзины с объектами одной формы

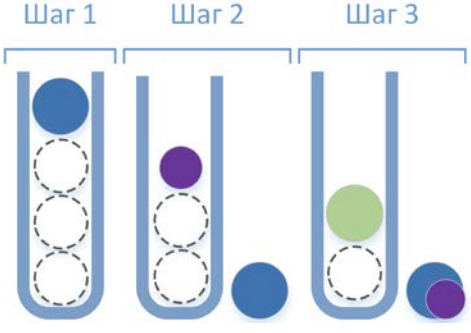

Рис. 4.6. Поиск элемента последовательности

Говоря языком программирования, некий указатель в исходном состоянии направлен на верхний объект. Проведя его анализ (определив цвет), вы достаете его и получаете доступ к следующему, лежащему ниже. И так далее.

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 напрямую. Вы, как было сказано выше, можете лишь последовательно перебирать элементы.

`Sequence` — это протокол, определяющий требования, при которых значение типа данных может называться последовательностью. Протокол `Sequence` требует, чтобы тип обеспечивал хранение множества однотипных значений и организовывал доступ к ним с помощью последовательного перебора.

Рассмотрим пример.

Вы выстроили в очередь множество целочисленных значений: `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`. При этом для вас скрыто все, что находится между первым и последним элементами: `1...8`. Однозначно можно сказать лишь то, что у всех элементов одинаковый тип данных (в данном случае — целое число типа `Int`). Чтобы увидеть, что же скрыто

за многоточием, вам придется последовательно с начала перебирать элементы один за другим.

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.

Следующие главы будут посвящены знакомству с типами данных, реализующими протокол `Sequence`.

4.3. Коллекции (Collection)

Коллекция (`Collection`) — это последовательность (`Sequence`), в которой можно обращаться к отдельному элементу напрямую. Другими словами, `Collection` — это протокол, основанный на протоколе `Sequence`, который при этом имеет дополнительное требование по обеспечению прямого доступа к элементам. Также коллекция не может быть бесконечной (в отличие от `Sequence`). Если вернуться к примеру с шариками и корзиной, то коллекция — это корзина с дырками, в которой можно получить доступ к любому объекту напрямую.

Для доступа к элементам коллекции используются **индексы**. 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). То есть обращаясь к коллекции и сообщая ей индекс интересующего вас элемента, вы получите значение именно того элемента, индекс которого передали. Но при этом элементы коллекции точно так же могут последовательно перебираться.

Рассмотрим пример.

Перед вами стоит строй солдат. У вас есть две возможности:

- Назвать фамилию солдата и в ответ услышать громкое и четкое «Я!».
- Дать команду «Рассчитайсь» и последовательно услышать отклик от каждого из солдат.

Строй в данном примере — это коллекция элементов.

На рис. 4.7 приведена упрощенная схема классификации рассмотренных протоколов и их требований.

Чуть позже 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 дальнейшем.

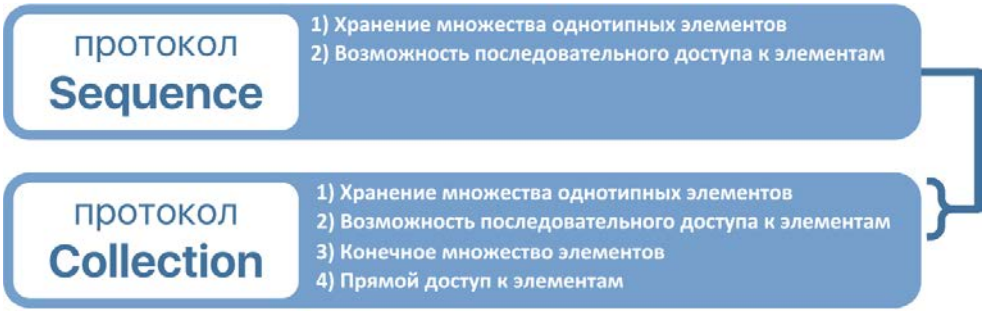

Рис. 4.7. Классификация протоколов и их требования

4.4. 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 Swift само по себе не дает навыков разработки приложений. Для этого требуется изучить большое количество дополнительного материала, а самое главное — необходимо научиться работать с официальной документацией, с помощью которой вы сможете ответить на большинство возникающих вопросов.

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

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

Утверждение: «Все фундаментальные типы являются `Hashable` (хешируемыми)».

Хешируемым называется такой тип данных, для значений которого может быть рассчитан специальный числовой хеш. Доступ к нему может быть получен через свойство `hashValue`. Если тип называется хешируемым, значит, он реализует требования протокола `Hashable`.

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

В Xcode Playground напишите код из листинга 4.1.

Листинг 4.1

```
var intVar = 12
```

В данном примере создается переменная целочисленного типа со значением `12`. Удерживая клавишу `Option`, щелкните по имени переменной, после чего отобразится информационное окно, содержащее сведения об имени, типе переменной и месте ее объявления (рис. 4.8).

Рис. 4.8. Всплывающее информационное окно

Если щелкнуть по имени типа данных (`Int`), то откроется новое окно, содержащее справочную документацию (рис. 4.9).

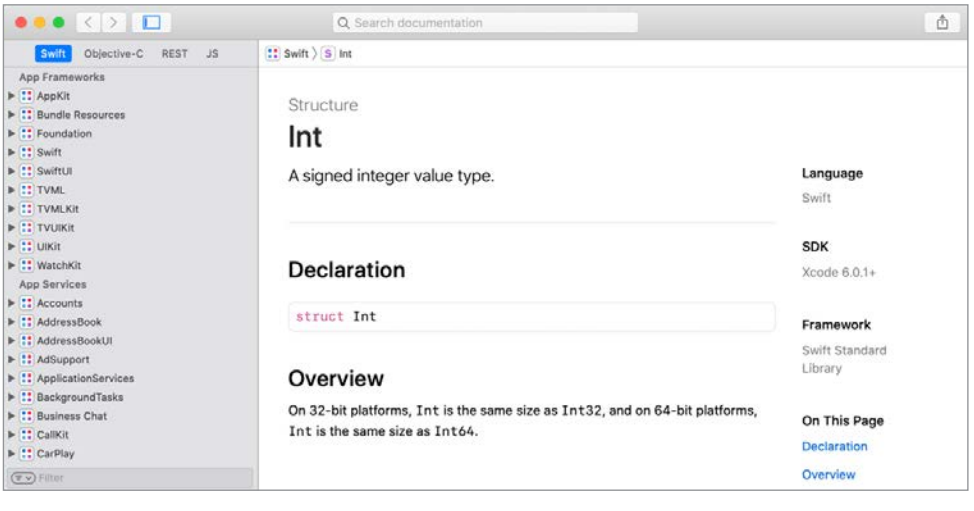

Рис. 4.9. Окно со справочной документацией

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

Если перейти в самый конец страницы описания типа `Int`, то там вы найдете раздел `Conforms To`, где описаны протоколы, которым соответствует данный тип (рис. 4.10). Среди них вы найдете и упомянутый выше `Hashable`.

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.

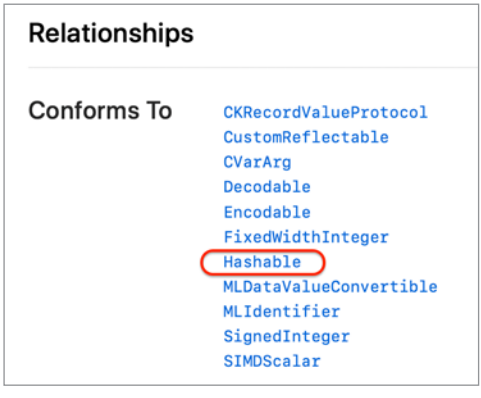

Рис. 4.10. Протоколы типа данных Int

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 и с ним.

В разделе Adopted By перечислены все типы данных, которые соответствуют данному протоколу.

В Inherits From указаны все родительские, по отношению к данному, протоколы. Там вы можете найти, например, `Equatable`. 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` (выше у данного типа в разделе Comforts To отсутствовала ссылка на `Equatable`).

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. 4.11).

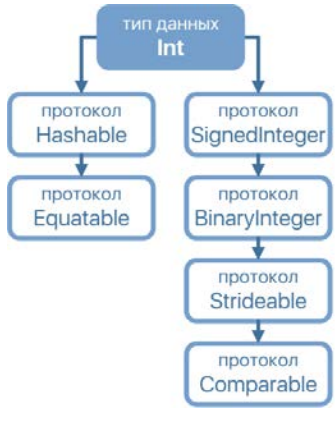

Рис. 4.11. Порядок следования протоколов в типе Int

Таким образом, тип `Int` выполняет требования протокола `Comparable` через цепочку протоколов `SignedInteger`, `BinaryInteger` и `Strideable`. Каждый дочерний протокол выполняет требования родительского, но также накладывает дополнительные требования.

К примеру, протоколы Collection и Sequence также находятся в отношении «предок — наследник» (или «отец — сын»). Один основывается на другом.

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

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

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

Глава 5. Диапазоны (Range)

Знакомиться с типами данных, относящимися к последовательностям и коллекциям, начнем с диапазонов — специальных типов, 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, следующие по порядку (1, 2, 3 ... 99, 100).

Диапазоны 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 в ходе последовательного перебора.

Для создания диапазонов используются два вида операторов:

- полуоткрытый (`..<`);
- закрытый (`...`).

5.1. Оператор полуоткрытого диапазона

Оператор полуоткрытого диапазона обозначается в виде двух точек и знака меньше (`..<`). Swift предлагает две формы данного оператора:

- бинарную (оператор размещен между операндами);
- 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

```
let 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 5.1).

Листинг 5.1

```
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 в качестве значения константы `rangeInt`.

Для каждого вида диапазона, который будет рассмотрен в этой главе, Swift использует свой собственный тип данных. 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, содержащими 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 5.2).

Листинг 5.2

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

Как говорилось ранее, диапазон может содержать не только целочисленные значения, но и элементы других типов. В листинге 5.3 показаны примеры создания диапазонов с элементами типа `String`, `Character` и `Double`.

Листинг 5.3

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

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

// диапазон с элементами типа Double
let rangeDouble = 1.0..<5.0
type(of:rangeDouble) // Range<Double>.Type
```

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 о фундаментальных типах 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 5.4).

Листинг 5.4

```
let firstElement = 10
var lastElement = 18
let 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.

Это 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

```
let myRange = ..<500
```

В листинге 5.5 приведен пример использования данной формы оператора.

Листинг 5.5

```
let 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 префиксной формы — `PartialRangeUpTo<T>`, где `T` — *это сопоставимый* (`Comparable`) тип данных.

5.2. Оператор закрытого диапазона

Оператор закрытого диапазона обозначается в виде трех точек (`...`). Язык Swift предлагает три формы: бинарная, постфиксная и префиксная.

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

```
let myRange = 1...100
```

В листинге 5.6 приведен пример использования данной формы оператора.

Листинг 5.6

```
let 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...
```

`левая_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

```
let myRange = 10...
```

Данный тип диапазона может быть использован, например, при работе с коллекциями, когда вы хотите получить все элементы, начиная с *N*, но размер коллекции при этом неизвестен. В листинге 5.7 приведен пример использования постфиксной формы оператора для создания диапазона и получения элементов.

Листинг 5.7

```
let infRange = 2...
type(of: infRange) // PartialRangeFrom<Int>.Type
let collection = [1, 6, 76, 12, 51]
print( collection[infRange] )
```

Консоль

```
[76, 12, 51]
```

> **ПРИМЕЧАНИЕ** Константа collection — это пример массива, рассмотрению которых посвящена следующая глава.

При этом совершенно неважно, сколько элементов будет содержать коллекция `collection`, так как в любом случае будут получены все элементы, начиная с третьего (почему с третьего, а не со второго, вы узнаете при рассмотрении массивов).

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

```
let myRange = ...0
```

Тип данных диапазона, созданного с помощью постфиксного оператора, — `PartialRangeThrough<T>`, где `T` — это сопоставимый (`Comparable`) тип данных.

В следующих главах мы подробно разберем примеры использования рассмотренных операторов.

5.3. 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 5.8).

Листинг 5.8

```
let intR = 1...10
intR.count // 10
```

Для определения наличия элемента в диапазоне служит метод `contains(_:)` (листинг 5.9).

Листинг 5.9

```
let 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 5.10).

Листинг 5.10

```
// диапазон без элементов
let emptyR = 0..<0
emptyR.count // 0
emptyR.isEmpty // true

// диапазон с единственным элементом - 0
let 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 значение, правда, доступны они только при работе с целочисленными значениями (листинг 5.11).

Листинг 5.11

```
let anotherIntR = 20..<34
anotherIntR.lowerBound // 20
anotherIntR.upperBound // 34
anotherIntR.min() // 20
anotherIntR.max() // 33
```

5.4. Классификация диапазонов

Как вы теперь знаете, диапазоны в Swift представлены следующими типами данных:

- `Range<T>`, например `1..<5`;
- `ClosedRange<T>`, например `1...10`;
- `PartialRangeUpTo<T>`, например `..<10`;
- `PartialRangeFrom<T>`, например `1...`;
- `PartialRangeThrough<T>`, например `...10`,

где `T` — тип данных элементов диапазона.

Рассмотрим некоторые нюансы данных типов, а точнее, выполнение ими требований различных протоколов.

Взгляните на пример из листинга 5.12.

Листинг 5.12

```
let rangeOne = ...10
type(of: rangeOne) // PartialRangeThrough<Int>
let rangeTwo = ..<10
type(of: rangeTwo) // PartialRangeUpTo<Int>
```

Диапазоны `rangeOne` и `rangeTwo` не имеют начального элемента, определена лишь их верхняя граница (`10` и `9` соответственно). По этой причине типы `PartialRangeThrough<T>` и `PartialRangeUpTo<T>` не относятся к последовательностям, а соответственно, и к коллекциям. Нам просто не с чего начать перебор элементов.

Чтобы убедиться в этом, откройте документацию к языку Swift (пункт `Help > Developer Documentation` меню Xcode) и найдите описание типа `PartialRangeThrough` или `PartialRangeUpTo`. Ни у одного из них в разделе Conforms to нет упоминаний ни `Sequence`, ни `Collection` (рис. 5.1).

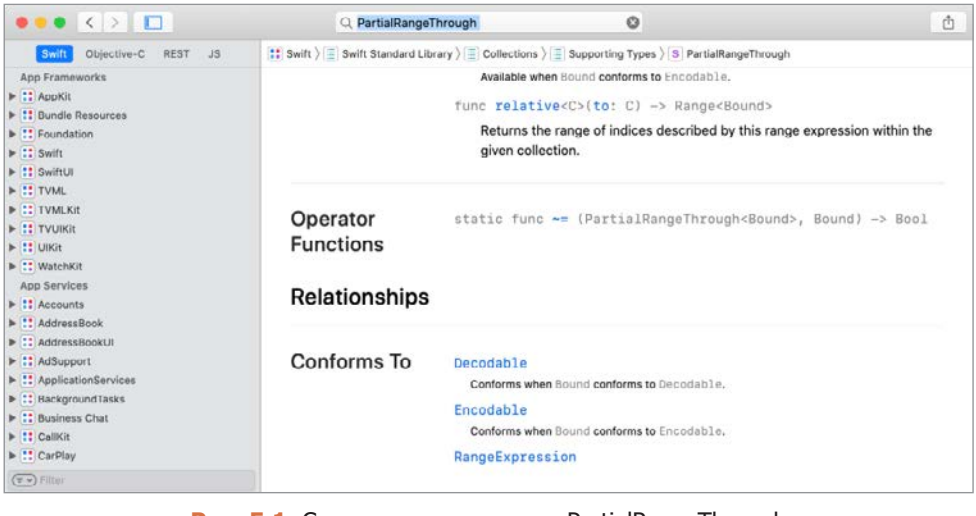

Рис. 5.1. Справка к типу данных PartialRangeThrough

Теперь перейдите к описанию типа данных `Range`. На этот раз в разделе `Conforms to` указаны и `Sequence`, и `Collection` (рис. 5.2). Это говорит о том, что тип `Range` является и последовательностью, и коллекцией. Но есть некоторые оговорки.

Обратите внимание на примечания, указанные под `Sequence` и `Collection`:

```
Conforms when Bound conforms to Strideable and Bound.Stride conforms to
SignedInteger
```

или иначе

```
Тип Range является последовательностью и коллекцией когда
1) тип данных, которым представлены границы диапазона, выполняет требования
   протокола Strideable;
2) шаг между элементами диапазона может быть выражен в виде целого числа.
```

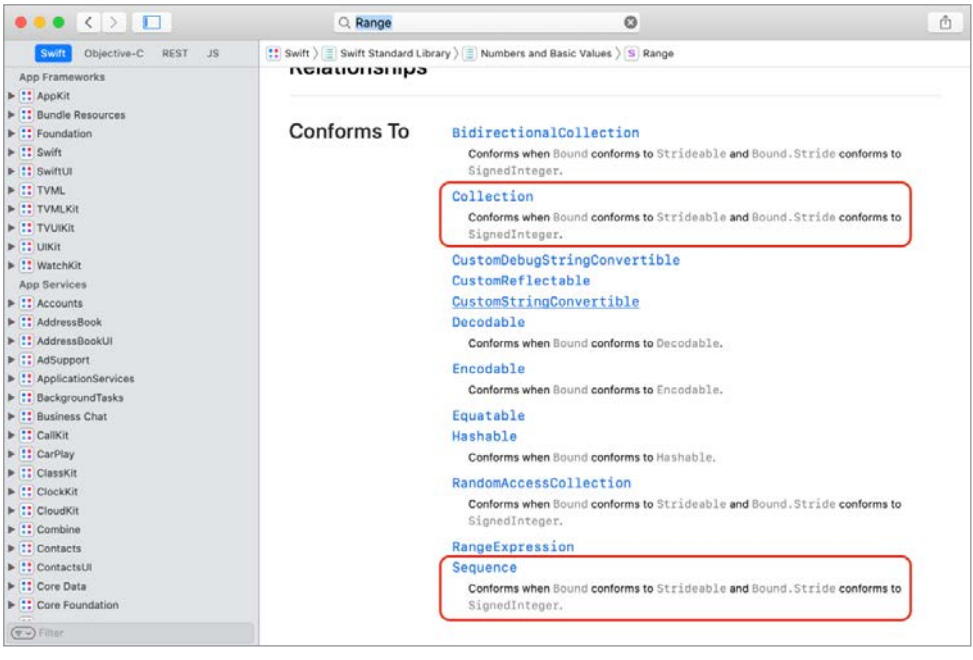

Рис. 5.2. Справка к типу данных Range

Первое условие предъявляет требования к границам диапазона: их значения должны соответствовать протоколу `Strideable`, с которым вы еще не встречались. Его название можно перевести как «Шаговый» или «Шагающий», а в его описании сказано, что реализующий его тип данных должен представлять собой непрерывное множество значений, между которыми можно перемещаться.

К примеру, тип `Int` реализует требования `Strideable`, — это непрерывное множество целых чисел, между которыми можно перемещаться с шагом 1. А вот `String` не выполняет требования `Strideable`, так как нет какого-то конкретного шага, чтобы перейти от символа `a` к символу `b`. Прибавить один? Думаю, нет. Прибавить букву? Звучит довольно странно.

Таким образом, значение типа `Range<Int>` является и последовательностью, и коллекцией, а вот `Range<String>` — нет.

Теперь откройте справку к типу данных `PartialRangeFrom`. В разделе `Conforms To` можно найти запись о `Sequence` (с уже знакомым примечанием), но `Collection`, в свою очередь, отсутствует (рис. 5.3). И это вполне объяснимо. Значение типа `PartialRangeFrom` имеет только начальную границу, а значит, оно бесконечно. Коллекции же не могут быть бесконечными.

Вам необходимо хорошо понимать, что такое диапазоны, так как в дальнейшем вы будете регулярно их использовать. Разберем с вами одну частую ошибку, с которой может столкнуться любой разработчик.

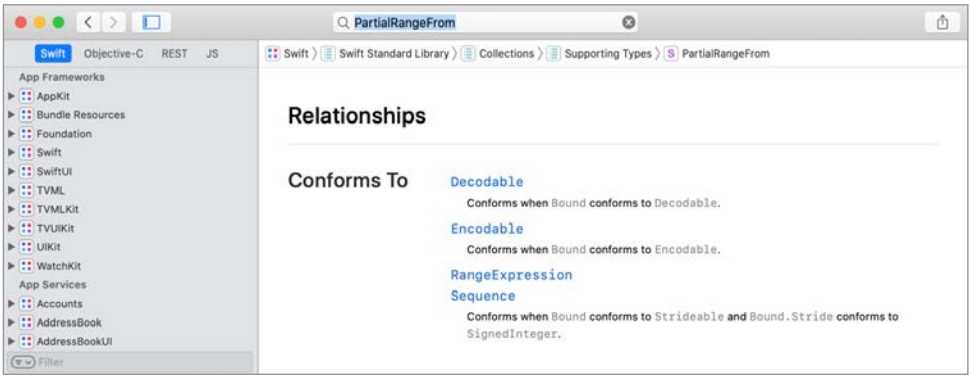

Рис. 5.3. Справка к типу данных PartialRangeFrom

Объявим диапазон типа `PartialRangeFrom<UInt8>` (листинг 5.13).

Листинг 5.13

```
let range = UInt8(1)...
```

Так как данный диапазон — последовательность, то вы можете последовательно перебирать его элементы с помощью конструкции `for-in` (будет подробно рассмотрена в следующей части книги) (листинг 5.14).

Листинг 5.14

```
// данный код выводит значения всех элементов переданной
// последовательности на консоль
for i in range {
    print(i)
}
```

Максимальное значение, которое позволяет принять тип `UInt8`, — 255. По этой причине, как только цикл (конструкция `for-in`) дойдет до 255 и попытается перейти к 256, произойдет критическая ошибка, и приложение аварийно завершит работу. Таким образом, вы столкнулись не с ограничением самой последовательности, а с ограничением типа данных элементов этой коллекции.

Хешируемые диапазоны (Hashable)

Диапазоны, ограниченные с обеих сторон, являются хешируемыми (`Hashable`), то есть для них возможно получить значение свойства `hashValue` (листинг 5.15).

Листинг 5.15

```
let range = 1...10
range.hashValue // 1819967165199576418
let range2 = 1..<10
range2.hashValue // 1819967165199576418
```

Интересным является тот факт, что хеш высчитывается на основании значений границ диапазона. Как видно из листинга 5.15, у обоих диапазонов, созданных с помощью различных операторов, но имеющих одни и те же пределы, идентичное значение хеша.

Эквивалентные диапазоны (Equatable)

Диапазоны, ограниченные с двух сторон, соответствуют протоколу `Equatable`, а значит, могут быть проверены на эквивалентность (листинг 5.16).

Листинг 5.16

```
let range = 1...10
let range2 = 1...10
range == range2 // true
```

Сопоставимые диапазоны (Comparable)

Все типы диапазонов не являются сопоставимыми, то есть не соответствуют требованиям протокола `Comparable`. Их нельзя сравнивать с помощью операторов `<=`, `<`, `>` и `>=`, это приведет к ошибке (листинг 5.17).

Листинг 5.17

```
let range = 1...10
let range2 = 1...10
range < range2 // Ошибка
```

5.5. Где использовать диапазоны

Описываемый механизм	Где используется
Диапазоны	Используются при необходимости указать на множество последовательных значений или элементов. **Пример:** • При переборе в циклах. `for iteratorItem in 1…100 { ... }` • При указании на числовой диапазон. `let successValuesRange = 0.0..<5.0` `// проверим, входит ли некоторое значение в данный диапазон` `successValuesRange.contains(7.2) // false` • При создании массива. `Array(1...4) // [1, 2, 3, 4]` • При получении требуемых элементов массива. `let array: [Int] = [0, 1, 2, 3, 4, 5, 6, 7, 8]` `array[..<5] // [0, 1, 2, 3, 4]`

Глава 6. Массивы (Array)

Массив — это один из наиболее важных представителей коллекции, который вы, с большой долей вероятности, будете использовать при реализации функциональности любой программы. Уделите особое внимание приведенному в этой главе материалу.

6.1. 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. Удаляя некоторый элемент, 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.

Хранение массива в памяти компьютера

Предположим, что в вашей программе уже создан массив, содержащий несколько строковых значений. Он, как и любой параметр, хранится в оперативной памяти (рис. 6.1).

Условно говоря, массив объединяет несколько хранилищ данных под одним идентификатором (именем). При этом идентификатор есть как у всего массива целиком (название параметра, в который он записан), так и у каждого хранилища в составе массива (индекс элемента). Идентификаторы (индексы)

хранилищ внутри массива имеют последовательные числовые значения и начинаются с нуля.

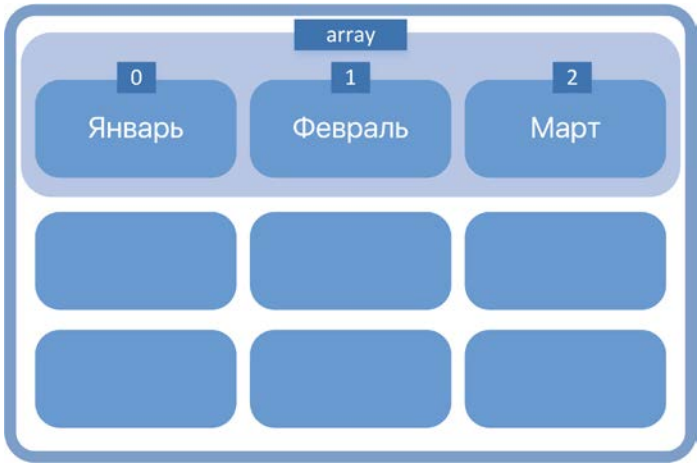

Рис. 6.1. Хранение массива в оперативной памяти

Но почему так происходит? Чтобы ответить на этот вопрос, нужно понять, а что же вообще означает индекс элемента массива. «Порядковый номер элемента» — скажете вы, и будете правы, но лишь частично. Индекс массива — это смещение элемента в массиве относительно его начала (рис. 6.2).

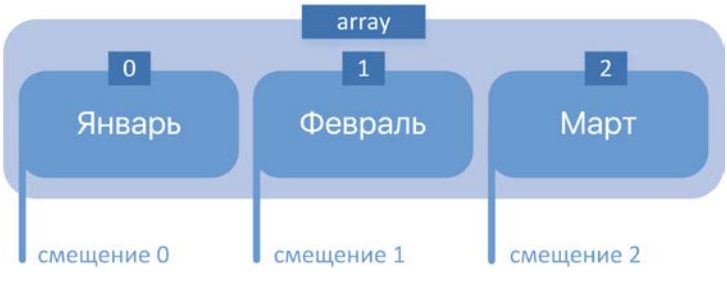

Рис. 6.2. Индексы и смещение элементов массива

Изначально, при осуществлении доступа к массиву, указатель показывает на ячейку первого элемента массива. При необходимости получить значение элемента с индексом 0 компьютер просто вернет значение данной ячейки. При необходимости получить второй элемент потребуется совершить одно смещение, то есть сместить указатель на одну позицию вправо, к следующей ячейке (индекс второго элемента — 1). Для доступа к третьему элементу необходимо совершить два смещения (индекс третьего элемента — 2). И так далее.

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 в нее массив является неизменяемым. Пример создания изменяемого и неизменяемого массива приведен в листинге 6.1.

Листинг 6.1

```
// неизменяемый массив
// с элементами типа String
let alphabetArray = ["a", "b", "c"]
// изменяемый массив
// с элементами типа Int
var mutableArray = [2, 4, 8]
```

В данном листинге массивы с помощью литералов проинициализированы в переменные. `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, множествам (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 6.2.

Листинг 6.2

```
// 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 входной аргумент arrayLiteral со значением 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 6.3).

Листинг 6.3

```
// 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 аргумента 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, который состоит 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 повторения 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 6.4).

Листинг 6.4

```
// 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. Индекс указывается в квадратных скобках после массива (листинг 6.5).

Листинг 6.5

```
// 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 6.6).

Листинг 6.6

```
// 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 6.7 приведен пример замены двух элементов массива на один новый с помощью использования оператора диапазона.

Листинг 6.7

```
// 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.

6.2. 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 6.8.

Листинг 6.8

```
// Массив с типом данных [String] или Array<String>
let firstAr = Array(arrayLiteral: "a", "b", "c") // ["a", "b", "c"]
type(of: firstAr) // Array<String>.Type
// Массив с типом данных [Int] или Array<Int>
let 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:

```
let имяМассива: Array<T> = литерал_массива
```

Краткая форма записи:

```
let 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

```
let arrayOne: Array<Character> = ["a", "b", "c"]
let arrayTwo: [Int] = [1, 2, 5, 8, 11]
```

6.3. 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 6.9 показан пример создания копии исходного массива с последующей заменой одного из его элементов.

Листинг 6.9

```
// исходный массив
let 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.

6.4. 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 6.10.

Листинг 6.10

```
/* объявляем массив с пустым значением
  с помощью переданного значения */
let emptyArray: [String] = [] // []
 /* объявляем массив с пустым значением
  с помощью специальной функции */
let anotherEmptyArray = [String]() // []
```

В результате создаются два пустых массива `emptyArray` и `anotherEmptyArray` с уже инициализированными значениями (хотя и не содержащими элементов).

6.5. 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;
- каждая соответствующая пара элементов эквивалентна (имеют одни и те же типы данных и значения).

Рассмотрим пример сравнения двух массивов (листинг 6.11).

Листинг 6.11

```
/* 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 возникнуть ошибка. На то есть две причины:

- С помощью литерала массива, как вы узнаете далее, тоже 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 является 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 6.12.

Листинг 6.12

```
// создаем три массива
let charsOne = ["a", "b", "c"]
let charsTwo = ["d", "e", "f"]
let charsThree = ["g", "h", "i"]
// создаем новый 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.

6.6. 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 6.13.

Листинг 6.13

```
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), говоря о том, что это *массив массивов*.

Для доступа к элементу многомерного массива необходимо указывать несколько индексов (листинг 6.14).

Листинг 6.14

```
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`.

6.7. 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 6.15).

Листинг 6.15

```
let 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 6.16).

Листинг 6.16

```
let emptyArray: [Int] = []
emptyArray.count // 0
emptyArray.isEmpty // true
```

Вы можете использовать свойство `count`, чтобы получить требуемые элементы массива (листинг 6.17).

Листинг 6.17

```
var numArray = [1, 2, 3, 4, 5]
// количество элементов в массиве
let 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 6.18).

Листинг 6.18

```
let subArray = numArray.suffix(3) // [3, 4, 5]
```

Свойства `first` и `last` возвращают первый и последний элементы массива (листинг 6.19).

Листинг 6.19

```
// 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 6.20).

Листинг 6.20

```
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 6.21).

Листинг 6.21

```
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 6.22).

Листинг 6.22

```
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 6.23).

Листинг 6.23

```
numArray // [2, 3, 4, 5]
// удаляем последний элемент
numArray.dropLast() // [2, 3, 4]
// удаляем три первых элемента
let 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 6.24).

Листинг 6.24

```
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 6.25).

Листинг 6.25

```
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 6.26.

Листинг 6.26

```
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 6.27).

Листинг 6.27

```
// исходная неотсортированная коллекция
let unsortedArray = [3, 2, 5, 22, 8, 1, 29]
// метод sorted() возвращает отсортированную последовательность
// при этом исходный массив не изменяется
let 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 о замыканиях (closure).

> **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.

Вам необходимо придерживаться этого правила в том числе и при создании собственных конструкций при разработке на Swift.

Метод `randomElement()` позволяет получить случайный элемент массива (листинг 6.28).

Листинг 6.28

```
let moneyArray = [50, 100, 500, 1000, 5000]
let randomMoneyElement = moneyArray.randomElement()
```

6.8. Срезы массивов (ArraySlice)

При использовании некоторых из описанных ранее свойств и методов возвращается не массив, а значение некого типа данных `ArraySlice`. Это происходит, например, при получении части массива с помощью оператора диапазона или при использовании методов `dropFirst()` и `dropLast()` (листинг 6.29).

Листинг 6.29

```
// исходный массив
let 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)
let 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 лишние копии одних и тех же данных.

Тип `ArraySlice` требует от вас максимальной внимательности, и по этой причине 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 будет продолжать храниться в памяти, так как ссылка на его элементы все еще существует в параметре типа `ArraySlice`.

Операции с ArraySlice

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 аргумента передается коллекция типа `ArraySlice` (листинг 6.30).

Листинг 6.30

```
type(of: slice) // ArraySlice<Int>.Type
let arrayFromSlice = Array(slice)
type(of: arrayFromSlice) // Array<Int>.Type
```

Так же тип данных возвращаемого методами `dropFirst()` и `dropLast()` значения можно изменить, если явно указать тип данных параметра, которому инициализируется результат (листинг 6.31).

Листинг 6.31

```
let newArray: Array<UInt> = arrayOfNumbers.dropLast()
type(of: newArray) // Array<UInt>.Type
```

Стоит обратить внимание на то, что индексы `ArraySlice` соответствуют индексам исходной коллекции, то есть они не обязательно начинаются с 0 (листинг 6.32).

Листинг 6.32

```
// исходный массив
arrayOfNumbers // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
// его срез, полученный в одном из предыдущих листингов
slice // [5, 6, 7]
// отдельный элемент
arrayOfNumbers[5] // 6
slice[5] // 6
```

6.9. Где использовать массивы

Описываемый механизм	Где используется
Массивы	Используются для хранения переменного множества однотипных значений с сохранением порядка их добавления. Позволяют хранить элементы с идентичными значениями. **Пример:** Хранение полученных пользователем очков во время прохождения уровня для получения в конце итогового результата. `var scores: [Int] = [1, 6, 10, 2, 55, 100]`

Глава 7. Множества (Set)

Наряду с массивами множества являются частью важнейшего механизма, позволяющего работать с группой однотипных значений как с отдельным.

7.1. Введение во множества

Множество (Set) — это неупорядоченная коллекция уникальных элементов. В отличие от массивов, у элементов множества нет четкого порядка следования, важен лишь факт наличия некоторого значения в множестве. Определенное значение элемента может существовать в нем лишь единожды, то есть каждое значение в пределах одного множества должно быть уникальным. Возможно, в русскоязычной документации по языку Swift вы встречали другое название множеств — *наборы*.

Исходя из определения множества 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 отдельное множество, чтобы избежать дублирования блюд. В вашем множестве уже 4 элемента: «хлеб», «овощи», «тушенка» и «вода». Третий друг чуть позже сообщает, что готов взять мясо и овощи. Однако при попытке поместить в множество элемент «овощи» возникнет исключительная ситуация, поскольку данное значение в множестве уже присутствует. И правильно, вам совершенно не нужен второй комплект овощей.

Варианты создания множества

Множество создается с помощью *литерала множества*. 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 множества, должно иметь хешируемый тип данных.

Литерал множества возвращает множество, состоящее 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 множества 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 множества.
>
> Для упорядочивания у каждого значения вычисляется цифровой хеш. Для каждого возможного значения хеш с большой долей вероятности будет уникальным, так как он выступает своеобразным индексом, но при этом недоступен разработчику. Хеш используется только для внутренней сортировки значений.
>
> В связи с этим не каждый тип данных может использоваться для элементов множества. Чтобы тип данных имел такую возможность, он должен соответствовать требованиям протокола `Hashable`. Все фундаментальные типы поддерживают этот протокол.

При создании множества необходимо явно указать, что создается именно множество. Если переменной передать литерал множества, то Swift распознает в нем литерал массива и вместо множества будет создан массив. По этой причине для создания множества необходимо использовать один из следующих способов:

- явно указать тип данных множества с использованием конструкции `Set<T>`, где `T` указывает на тип значений элементов создаваемого множества, и передать литерал множества в качестве инициализируемого значения.

  Пример

  ```
  let mySet: Set<Int> = [1,5,0]
  ```

- неявно задать тип данных с помощью конструкции `Set` и передать литерал множества *с элементами* в качестве инициализируемого значения.

  Пример

  ```
  let 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 множества, а аргумент `arrayLiteral` содержит список элементов.

  Пример

  ```
  let mySet = Set<Int>(arrayLiteral: 5,66,12)
  ```

- использовать функцию `Set(arrayLiteral:)`, где аргумент `arrayLiteral` содержит список элементов.

  Пример

  ```
  let mySet = Set(arrayLiteral: 5,66,12)
  ```

Тип данных множества — `Set<T>`, где `T` определяет тип данных элементов множества и должен быть хешируемым типом (`Hashable`).

> **ПРИМЕЧАНИЕ** Для создания неизменяемого множества используйте оператор `let`, в ином случае — оператор `var`.

В листинге 7.1 продемонстрированы все доступные способы создания множеств.

Листинг 7.1

```
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 множества уникальных значений. При этом в области вывода порядок значений может не совпадать с определенным в литерале. Это связано с тем, что множества — неупорядоченные коллекции элементов.

7.2. Пустое множество

Пустое множество, то есть множество, значение которого не имеет элементов (по аналогии с пустым массивом), создается с помощью пустого литерала множества `[]` либо вызова функции `Set<T>()` без входных параметров, где `T` определяет тип данных элементов множества. Вы также можете передать данный литерал с целью уничтожения всех элементов изменяемого множества (то есть в качестве хранилища используется переменная, а не константа). Пример приведен в листинге 7.2.

Листинг 7.2

```
// создание пустого множества
let emptySet = Set<String>()
// множество со значениями
var setWithValues: Set<String> = ["хлеб", "овощи"]
// удаление всех элементов множества
setWithValues = []
setWithValues // Set([])
```

7.3. Базовые свойства и методы множеств

Множество — это неупорядоченная коллекция, элементы которой не имеют индексов. Для взаимодействия с его элементами используются специальные методы.

Так, для создания нового элемента множества применяется метод `insert(_:)`, которому передается создаваемое значение. Обратите внимание, что оно должно соответствовать типу множества (листинг 7.3).

Листинг 7.3

```
// создаем пустое множество
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 во множестве.

Для удаления элемента из множества 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 множества (листинг 7.4).

Листинг 7.4

```
// создание множества 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 множества
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 об опциональных типах данных.
>
> 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 множество вполне может иметь тип `Set<Bool>`. Если бы данный метод вернул `false`, то было бы логично утверждать, что из множества был удален элемент с именно таким значением.
>
> В свою очередь, получив `nil`, можно однозначно сказать, что искомый элемент отсутствует во множестве.

Проверка факта наличия значения во множестве осуществляется методом `contains(_:)`, который возвращает значение типа `Bool` в зависимости от результата проверки (листинг 7.5).

Листинг 7.5

```
musicStyleSet = ["Jazz", "Hip-Hop", "Rock", "Funk"]
// проверка существования значения во множестве
musicStyleSet.contains("Funk") // true
musicStyleSet.contains("Pop") // false
```

Для определения количества элементов во множестве вы можете использовать свойство `count`, возвращающее целое число (листинг 7.6).

Листинг 7.6

```
musicStyleSet.count //4
```

Операции со множествами

Множества в Swift подобны множествам в математике. Два или более множества могут содержать пересекающиеся и непересекающиеся между собой значения. Swift позволяет получать эти группы значений.

В листинге 7.7 создаются три различных целочисленных множества (рис. 7.1). Одно из множеств содержит четные числа, второе — нечетные, третье — те и другие.

Рис. 7.1. Три множества целочисленных значений

Листинг 7.7

```
// множество с нечетными цифрами
let oddDigits: Set = [1, 3, 5, 7, 9]
// множество с четными цифрами
let evenDigits: Set = [0, 2, 4, 6, 8]
// множество со смешанными цифрами
let differentDigits: Set = [3, 4, 7, 8]
```

Во множествах `oddDigits`, `evenDigits` и `differentDigits` существуют как уникальные для каждого из них, так и общие элементы.

Для каждой пары множеств можно произвести следующие операции (рис. 7.2):

- получить все общие элементы (`intersection(_:)`);
- получить все непересекающиеся (не общие) элементы (`symmetricDifference(_:)`);
- получить все элементы обоих множеств (`union(_:)`);
- получить разницу элементов, то есть элементы, которые входят в первое множество, но не входят во второе (`subtracting(_:)`).

При использовании метода `intersection(_:)` возвращается множество, содержащее значения, общие для двух множеств (листинг 7.8).

Листинг 7.8

```
let inter = differentDigits.intersection(oddDigits)
inter // {3, 7}
```

Для получения всех непересекающихся значений служит метод `symmetricDifference(_:)`, представленный в листинге 7.9.

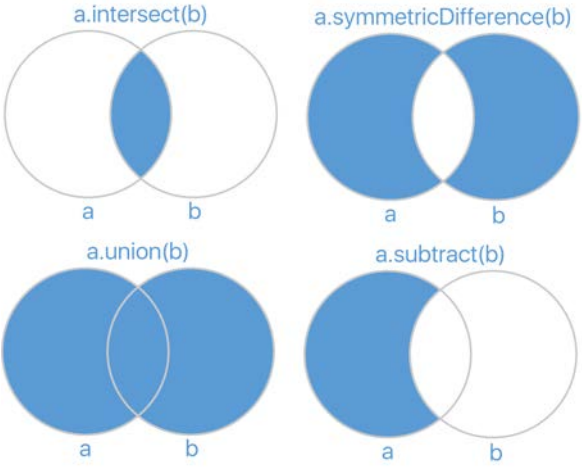

Рис. 7.2. Операции, проводимые со множествами

Листинг 7.9

```
let exclusive = differentDigits.symmetricDifference(oddDigits)
exclusive //{4, 8, 1, 5, 9}
```

Для получения всех элементов из обоих множеств (их объединения) применяется объединяющий метод `union(_:)`, как показано в листинге 7.10.

Листинг 7.10

```
let union = evenDigits.union(oddDigits)
union //{8, 4, 2, 7, 3, 0, 6, 5, 9, 1}
```

Метод `subtracting(_:)` возвращает все элементы первого множества, которые не входят во второе (листинг 7.11).

Листинг 7.11

```
let subtract = differentDigits.subtracting(evenDigits)
subtract // {3, 7}
```

Отношения множеств

В листинге 7.12 созданы три множества: `aSet`, `bSet` и `cSet`. В них присутствуют как уникальные, так и общие элементы. Их графическое представление показано на рис. 7.3.

Листинг 7.12

```
let aSet: Set = [1, 2, 3, 4, 5]
let bSet: Set = [1, 3]
let cSet: Set = [5, 6, 7, 8]
```

Множество `aSet` — это надмножество для `bSet`, так как включает в себя все элементы из `bSet`. В то же время множество `bSet` — это подмножество для `aSet`, так как все элементы `bSet` существуют и в `aSet`. Множества `cSet` и `bSet` являются непересекающимися, так как у них нет общих элементов, а множества `aSet` и `cSet` — пересекающиеся, так как имеют общие элементы.

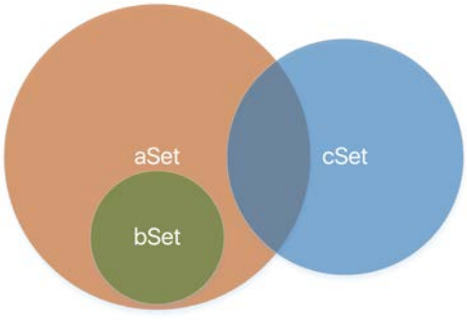

Рис. 7.3. Три множества значений с различными отношениями друг с другом

Два множества считаются эквивалентными, если у них один и тот же комплект элементов. Эквивалентность множеств проверяется с помощью оператора эквивалентности (`==`), как показано в листинге 7.13.

Листинг 7.13

```
// создаем копию множества
let copyOfBSet = bSet
/* во множествах bSet и copyOfBSet одинаковый состав
 элементов. Проверим их на эквивалентность */
bSet == copyOfBSet // true
```

Метод `isSubset(of:)` определяет, является ли одно множество подмножеством другого, как `bSet` для `aSet`. При этом он возвращает `true`, даже если множества равны (листинг 7.14).

Листинг 7.14

```
let aSet: Set = [1, 2, 3, 4, 5]
let bSet: Set = [1, 3]
bSet.isSubset(of: aSet) // true
```

Метод `isSuperset(of:)` вычисляет, является ли множество надмножеством для другого, как `aSet` для `bSet`. При этом он возвращает `true`, даже если множества равны (листинг 7.15).

Листинг 7.15

```
let aSet: Set = [1, 2, 3, 4, 5]
let bSet: Set = [1, 3]
aSet.isSuperset(of: bSet)  // true
```

Метод `isDisjoint(with:)` определяет, существуют ли в двух множествах общие элементы, и в случае их отсутствия возвращает `true` (листинг 7.16).

Листинг 7.16

```
bSet.isDisjoint(with: cSet) // true
```

Методы `isStrictSubset(of:)` и `isStrictSuperset(of:)` определяют, является множество подмножеством или надмножеством, не равным указанному множеству (листинг 7.17).

Листинг 7.17

```
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 множество. При этом будет возвращен массив, в котором все элементы расположены по возрастанию (листинг 7.18).

Листинг 7.18

```
let setOfNums: Set = [1,10,2,5,12,23]
let sortedArray = setOfNums.sorted()
sortedArray // [1, 2, 5, 10, 12, 23]
type(of: sortedArray) // Array<Int>.Type
```

7.4. Где использовать множества

Описываемый механизм	Где используется
Множества	Используются для хранения переменного множества уникальных однотипных значений. Порядок добавления элементов не сохраняется, важен лишь сам факт наличия элемента в множестве. **Пример:** • Идентификаторы забаненных пользователей. `var banedUsers: Set<Int> = [132, 345, 12, 45]` `// проверка того, забанен ли пользователь` `banedUsers.contains(132) // true`

Глава 8. Словари (Dictionary)

Продолжая свое знакомство с коллекциями (`Collection`), мы разберем, что такое словари, для чего они предназначены, чем отличаются и что имеют общего с другими видами коллекций. Кроме этого, мы узнаем, как их можно использовать при создании программ.

8.1. 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 быть единого типа 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 множества, 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 во множествах, порядок следования элементов не важен.

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 8.1.

Листинг 8.1

```
let dictionary = ["one":"один", "two": "два", "three":"три"]
dictionary // ["one": "один", "two": "два", "three": "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 (определяющие значения).

Пример

```
Dictionary(dictionaryLiteral: (100, "Сто"), (200, "Двести"), (300, "Триста"))
```

Создание словаря с помощью Dictionary(uniqueKeysWithValues:)

Еще одним способом создания словаря, который мы рассмотрим, будет использование функции `Dictionary(uniqueKeysWithValues:)`, позволяющей создать словарь на основе коллекции однотипных кортежей.

В листинге 8.2 приведен пример создания словаря из массива кортежей с помощью данной функции.

Листинг 8.2

```
// 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 8.3).

Листинг 8.3

```
// 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 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 официальной документации от Apple.

8.2. Тип данных словаря

Словари, как и другие виды коллекций, имеют обозначение собственного типа данных, у которого 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 8.4.

Листинг 8.4

```
// Словарь с типом данных [Int:String]
let 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:

```
let имяСловаря: Dictionary<T1,T2> = литерал_словаря
```

Краткая форма записи:

```
let имяСловаря: [T1:T2] = литерал_словаря
```

В обоих случаях объявляется словарь, ключи элементов которого должны иметь тип T1, а значения — T2.

Пример

```
let dictOne: Dictionary<Int,Bool> = [100: false, 200: true, 400: true]
let dictTwo: [String:String] = ["Jonh":"Dave", "Eleonor":"Green"]
```

8.3. 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 8.5).

Листинг 8.5

```
var countryDict = ["RUS": "Россия", "BEL": "Белоруссия", "UKR": "Украина"]
// получаем значение элемента
var countryName = countryDict["BEL"]
countryName // "Белоруссия"
// изменяем значение элемента
countryDict["RUS"] = "Российская Федерация"
countryDict // ["RUS": "Российская Федерация", "BEL": "Белоруссия",
               "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 8.6, при установке нового значения данный метод возвращает старое значение или `nil`, если значения по переданному ключу не существует.

Листинг 8.6

```
var oldValueOne = countryDict.updateValue("Республика Беларусь", forKey: "BEL")
// в переменной записано старое измененное значение элемента
oldValueOne // "Белоруссия"
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` передается 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 8.7).

Листинг 8.7

```
countryDict["TUR"] = "Турция"
countryDict // ["BEL": "Республика Беларусь", "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 8.8).

Листинг 8.8

```
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 следующих глав.
>
> ```
> let someDict = [1: "one", 3: "three"]
> someDict[2] // nil
> type(of: someDict[2]) // Optional<String>.Type
> ```

8.4. Пустой словарь

Пустой словарь не содержит элементов (как и пустое множество, и пустой массив). Чтобы создать пустой словарь, необходимо использовать литерал без элементов. Для этого служит конструкция `[:]` или функция `Dictionary<типКлючей:типЗначений>()` без аргументов (листинг 8.9).

Листинг 8.9

```
let emptyDictionary: [String:Int] = [:]
let anotherEmptyDictionary = Dictionary<String,Int>()
```

С помощью конструкции `[:]` также можно уничтожить все элементы словаря, если проинициализировать ее словарю в качестве значения (листинг 8.10).

Листинг 8.10

```
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.

8.5. Базовые свойства и методы словарей

Словари, как и массивы со множествами, имеют большое количество свойств и методов, наиболее важные из которых будут рассмотрены в этом разделе.

Свойство `count` возвращает количество элементов в словаре (листинг 8.11).

Листинг 8.11

```
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 8.12).

Листинг 8.12

```
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 8.13).

Листинг 8.13

```
// все ключи словаря countryDict
let keys = countryDict.keys
type(of: keys) // Dictionary<String, String>.Keys.Type
keys // Dictionary.Keys(["UKR", "RUS", "EST"])
```

```
// все значения словаря countryDict
let values = countryDict.values
type(of: values) // Dictionary<String, String>.Values.Type
values // Dictionary.Values(["Украина", "Эстония", "Российская Федерация"])
```

При обращении к свойствам `keys` и `values` Swift возвращает не массив или множество, а значение специального типа данных `Dictionary<ТипКлюча,ТипЗначения>.Keys` и `Dictionary<ТипКлюча,ТипЗначения>.Values`.

Не пугайтесь столь сложной записи. Как неоднократно говорилось, в Swift используется огромное количество различных типов, у каждого из которых свое предназначение. В данном случае указанные типы служат для доступа к ключам и значениям исходного словаря. При этом они являются полноценными коллекциями (соответствуют требованиям протокола `Collection`), а значит, могут быть преобразованы в массив или множество (листинг 8.14).

Листинг 8.14

```
let keysSet = Set(keys)
keysSet // {"UKR", "RUS", "EST"}
let valuesArray = Array(values)
valuesArray // ["Эстония", "Украина", "Российская Федерация"]
```

8.6. Вложенные типы

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` (например, `var a: Values = ...`), так как глобально такого типа нет. Он существует только в контексте типа `Dictionary`.

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 множество 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 перебираете его элементы).

Не переживайте, если данный материал оказался для вас сложным, вы обязательно разберетесь в нем, создавая свои приложения на Swift. В дальнейшем, углубляясь в разработку и создание собственных типов, вы сможете вернуться к этому описанию и прочитать его более осознанно. Сейчас важно запомнить лишь то, что свойства `keys` и `values` возвращают коллекции элементов, которые могут быть преобразованы в массив или множество.

8.7. Где использовать словари

Описываемый механизм	Где используется
Словари	Используются для хранения переменного множества однотипных значений с кастомными индексами. Порядок добавления элементов не сохраняется. **Пример:** Хранение столиц стран. `let capitals = ["Russia": "Moscow", "USA": "Washington", "Belarus": "Minsk"]` Хранение соответствия цифры и слова. `let numbers = ["one": 1, "two": 2, "three": 3]` Хранение соответствия символов при шифровании. `let symbolsCrypt = ["a": "z", "b": "x", "c": "w"]`

Глава 9. Строка — коллекция символов (String)

Во второй главе книги мы познакомились со строковыми типами `String` и `Character`, позволяющими работать с текстовыми данными в приложениях, и получили первые и основные знания о стандарте Юникод. Напомню, что строки состоят из отдельных символов, а для каждого символа существует одна или несколько кодовых точек (уникальные последовательности чисел из стандарта Юникод, соответствующие символу). Кодовые точки могут быть использованы для инициализации текстовых данных в составе юникод-скаляров (служебных конструкций `\u{}`). В этой главе будет подробно рассказано о том, как функционируют строковые типы в Swift.

9.1. Character в составе String

На концептуальном уровне строка в Swift — это сохраненная в памяти последовательность символов, представленная как коллекция. Да, именно коллекция! `String` соответствует требованиям протокола `Collection` и является коллекцией, подобной массивам, множествам 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 9.1) или осуществить их перебор с помощью оператора `for-in` (с ним мы познакомимся несколько позже).

Листинг 9.1

```
let str = "Hello!"
str.count // 6
```

Константа `str` имеет строковое значение, состоящее из `6` символов, что видно в том числе по выводу свойства `count`. Возможность использования 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, какой 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 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 в Swift такой подход неожиданно приведет к ошибке (листинг 9.2).

Листинг 9.2

```
str[2] // error: 'subscript' is unavailable: cannot subscript String with an Int
```

В чем проблема? Почему такой простой вариант доступа не может быть использован в современном языке программирования? Чтобы ответить на этот вопрос, потребуется вновь поговорить о стандарте Юникод и о работе с ним в Swift.

9.2. 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 9.3 параметру типа `Character` инициализируется значение через юникод-скаляр.

Листинг 9.3

```
let char: Character = "\u{E9}"
char // "é"
```

Символ `é` (латинская е со знаком ударения) представлен в данном примере с использованием кодовой точки `E9` (или `233` в десятичной 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 9.4).

Листинг 9.4

```
let 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 9.5).

Листинг 9.5

```
char == anotherChar // true
```

Выглядит очень странно, согласны? Дело в том, что в константе `anotherChar` содержится комбинированный символ, состоящий из двух кодовых точек и, 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 9.6 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 9.6

```
let thumbsUp = "\u{1f44d}" // "👍"
let blackSkin = "\u{1f3fd}" // "🏽"
let 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 9.7 создаются две строки, содержащие данный символ, первая из них содержит непосредственно сам символ é, а вторая — комбинацию из латинской e и знака ударения.

Листинг 9.7

```
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 коллекция символов, каждый из которых является графем-кластером. Стоит отметить, что время, необходимое для выполнения подсчета количества элементов в строке, растет линейно с увеличением количества этих элементов (букв). Причиной этому является то, 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 Юникод, 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.

9.3. Строковые индексы

Почему Swift не позволяет использовать целочисленные индексы для доступа к отдельным графем-кластерам в строке? На самом деле разработчикам этого языка не составило бы никакого труда слегка расширить тип `String` и добавить соответствующую функциональность. Это же программирование, тут возможно все! Но такой возможности они лишили нас 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 9.8).

Листинг 9.8

```
let name = "e\u{301}lastic" // "élastic"
let 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 символ в составе графем-кластера. Теперь в константе `index` хранится индекс первой буквы, и его можно использовать точно так же, как индексы других коллекций (листинг 9.9).

Листинг 9.9

```
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 9.10)?

Листинг 9.10

```
type(of: index) // String.Index.Type
```

Тип строкового индекса — `String.Index`: это тип данных `Index`, вложенный в `String` (определенный в данном пространстве имен). 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 9.11).

Листинг 9.11

```
let 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 в качестве аргумента `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 9.12).

Листинг 9.12

```
let lastCharIndex = name.index(before: indexLastChar)
name[lastCharIndex] // "c"
```

Метод `index(after:)` позволяет получить индекс последующего символа (листинг 9.13).

Листинг 9.13

```
let 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 9.14).

Листинг 9.14

```
let 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. 9.1).

Рис. 9.1. 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 9.15 показано, что количество элементов строки и значение, возвращаемое свойством `unicodeScalars`, отличаются, так как в составе строки есть сложный графем-кластер (состоящий из двух символов).

Листинг 9.15

```
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.

9.4. Подстроки (Substring)

В Swift присутствует тип данных `Substring`, описывающий подстроку некоторой строки. `Substring` для `String` — это как `ArraySlice` для `Array`. При получении подстроки возвращается значение типа `Substring`, ссылающееся на ту же область памяти, что и оригинальная строка, а это позволяет экономить ресурсы компьютера.

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 9.16).

Листинг 9.16

```
let abc = "abcdefghijklmnopqrstuvwxyz"
// индекс первого символа
let firstCharIndex = abc.startIndex
```

```
// индекс четвертого символа
let fourthCharIndex = abc.index(firstCharIndex, offsetBy:3)
// получим подстроку
let subAbc = abc[firstCharIndex...fourthCharIndex]
subAbc // "abcd"
type(of: subAbc) // Substring.Type
```

В результате выполнения кода в константе `subAbc` будет находиться значение типа `Substring`, включающее в себя первые четыре символа строки `abc`.

Подстроки обладают той же функциональностью, что и строки. Но при необходимости вы всегда можете использовать функцию `String(_:)` для преобразования подстроки в строку (листинг 9.17).

Листинг 9.17

```
type( of: String(subAbc) ) // String.Type
```

Так же 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 9.18).

Листинг 9.18

```
let subStr = abc[fourthCharIndex...]
subStr // "defghijklmnopqrstuvwxyz"
```

На этом наше знакомство со строками окончено. Уверен, что оно было интересным и познавательным. Советую самостоятельно познакомиться со стандартом Юникод в отрыве от изучения Swift. Это позволит еще глубже понять принципы его работы и взаимодействия с языком.

Часть III

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 и базовых 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.

Глава 10. 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 любом языке, благодаря которому вы сможете написать сложные функциональные приложения. В данной главе рассмотрены механизмы, которые присутствуют практически во всех языках и без которых невозможно создание любой, даже самой простой программы.

Операторы управления можно разделить на две группы:

- **Операторы ветвления**, определяющие порядок и необходимость выполнения блоков кода.
- **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.

10.1. 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 возвращено `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 10.1 показан пример использования утверждений.

Листинг 10.1

```
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 успешно завершила свою работу")
```

Консоль

```
Программа успешно завершила свою работу
```

В приведенном примере проверяется возраст дракона, записанный в переменную `dragonAge`. С помощью двух утверждений программа успешно завершит работу, только если он 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 10.2).

Листинг 10.2

```
var dragonAge = 220
assert( dragonAge <= 235, strName+strEmpty+strOld )
assert( dragonAge >= 225, strName+strEmpty+strYoung )
print("Программа успешно 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 релизных версиях проектов. В основном они предназначены для отладки разрабатываемых программ или тестирования кода в Playground.

10.2. 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, различающиеся 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 10.3 приведен пример использования оператора условия. В нем проверяется значение переменной `logicVar`.

Листинг 10.3

```
// 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 10.4).

Листинг 10.4

```
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 10.5).

Листинг 10.5

```
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, то Xcode сообщит об ошибке (листинг 10.6).

Листинг 10.6

```
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 10.7).

Листинг 10.7

```
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:

- `logicVar == false`.
- `false == false` — заменили переменную ее значением.
- `true` — два отрицательных значения идентичны друг другу.

Во втором варианте он отличается:

- `!logicVar` .
- `!false` — заменили переменную ее значением.
- `true` — *НЕ ложь* является *истиной*.

ПРИМЕЧАНИЕ Если вы хотите освежить в памяти материал по логическим выражениям, 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 10.8).

Листинг 10.8

```
// 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 10.9).

Листинг 10.9

```
let a = 1054
let 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 10.10 проверяется истинность значений двух переменных.

Листинг 10.10

```
// 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 или две переменные истинны
```

Так как проверяемое выражение возвращает `true`, если хотя бы один из операндов равен `true`, то 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 10.11).

Листинг 10.11

```
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 10.12).

Листинг 10.12

```
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 10.13).

Листинг 10.13

```
// 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, я говорил 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 10.14).

Листинг 10.14

```
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 10.15).

Листинг 10.15

```
// 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 10.16).

Листинг 10.16

```
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

```
let y = ( x > 100 ? 100 : 50 )
```

В данном примере в переменной y будет инициализировано значение одного из выражений (`100` или `50`).

В листинге 10.17 показан пример использования тернарного оператора условия. В нем сопоставляются значения констант для того, чтобы вывести на отладочную консоль соответствующее сообщение.

Листинг 10.17

```
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 10.18 показан пример того, что возвращенное оператором значение может быть использовано в составе другого выражения.

Листинг 10.18

```
// 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`.

ПРИМЕЧАНИЕ Отдельные выражения внутри проверяемого выражения могут быть отделены не только логическими операторами || и &&, но и с помощью запятой (`,`). Она с логической точки зрения работает как &&, то есть если одно из условий `false`, то общий результат проверки также будет `false`.

```
let a = Int.random(in: 1...100)
let b = Int.random(in: 1...100)

// вариант 1 (с И)
if a > 50 && b > 50 {
    print("a and b > 50")
}

// вариант 2 (с запятой)
if a > 50, b > 50 {
    print("a and b > 50")
}
```

При этом использование запятой имеет несколько особенностей:

- Вычисление значения выражений происходит по порядку, и если одно из выражений не истинно (`false`), то последующие подвыражения вычисляться не будут.
- Результаты вычисления каждого подвыражения могут быть использованы в последующих подвыражениях.

В следующем примере используются опционалы, изучению которых будет посвящена следующая глава. В нем результаты вычислений первого и второго подвыражений используются в третьем.

```
if let a = Int("43"), let b = Int("45"), a < b {
    print("a < b")
}
```

10.3. 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 10.19).

Листинг 10.19

```
// оценка
let 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.

Более удобно решать эту задачу можно 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 предел оператора. Таким образом, если вам в дальнейшем потребуется изменить принцип вывода сообщения (например, вы станете использовать другую функцию), то достаточно будет внести правку в одном месте кода вместо шести (листинг 10.20).

Листинг 10.20

```
let userMark = 4
// переменная для хранения сообщения
let message: String
```

```
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 10.21 приведен пример определения множества, в которое попадает заданное число. При этом используются операторы диапазона без использования метода `contains(_:)`, как было у оператора `if`.

Листинг 10.21

```
let 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 10.22).

Листинг 10.22

```
let 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 10.23).

Листинг 10.23

```
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 10.24).

Листинг 10.24

```
let 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 находится 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 10.25).

Листинг 10.25

```
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 совпадает значение, указанное после `case`, и условие после `where` вернет `true`.

Связывание значений

Представьте ситуацию, что ваше руководство окончательно сошло с ума 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 вы 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 10.26).

Листинг 10.26

```
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:

- Вследствие длинного имени кортежа выражение не очень удобно читать и воспринимать.
- 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. После этого данный параметр можно использовать после `where` и в теле case-блока.

В листинге 10.27 происходит связывание значения второго элемента кортежа `dragonCharacteristics`.

Листинг 10.27

```
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 10.28).

Листинг 10.28

```
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 10.29).

Листинг 10.29

```
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, конструкция завершит свою работу без ошибок, просто передав управление в default-блок, где находится `break`.

В листинге 10.30 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 10.30

```
let 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 10.31).

Листинг 10.31

```
let 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.

10.4. 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 10.32).

Листинг 10.32

```
// 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 будет проигнорирован и не 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 10.33).

Листинг 10.33

```
// 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 10.34 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 10.34

```
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 10.35 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 10.35

```
let 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
```

10.5. 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 последовательность (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` — это локальный для тела оператора 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 10.36 показан пример использования оператора `for`, в котором производится подсчет суммы всех элементов массива, состоящего из целочисленных значений.

Листинг 10.36

```
// 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 10.37) или строку (листинг 10.38).

Листинг 10.37

```
for number in 1...5 {
    print(number)
}
```

Консоль

```
1
2
3
4
5
```

Листинг 10.38

```
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 10.39).

Листинг 10.39

```
// 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.

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 10.40).

Листинг 10.40

```
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 10.41).

Листинг 10.41

```
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 10.42).

Листинг 10.42

```
var countriesAndBlocks = ["Россия": "СНГ", "Франция": "ЕС"]
for (countryName, _) in countriesAndBlocks {
    print("страна – \(countryName)")
}
for (_, orgName) in countriesAndBlocks{
    print("организация – \(orgName)")
}
```

Помимо этого, в случае, если требуется получить последовательность, состоящую только из ключей или значений словаря, можно воспользоваться свойствами `keys` и `values` — и передать результат их работы в оператор `for` (листинг 10.43).

Листинг 10.43

```
countriesAndBlocks = ["Россия": "ЕАЭС", "Франция": "ЕС"]
for countryName in countriesAndBlocks.keys {
    print("страна – \(countryName)")
}
for orgName in countriesAndBlocks.values {
    print("организация – \(orgName)")
}
```

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 10.44).

Листинг 10.44

```
print("Несколько фактов обо мне:")
let 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, идущие 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 10.45).

Листинг 10.45

```
// коллекция элементов от 1 до 10 с шагом 3
let intNumbers = [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 10.46).

Листинг 10.46

```
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 10.47).

Листинг 10.47

```
for i in stride( from: 1, to: 10, by:3 ) {
    // тело оператора
}
```

Параметр `i` будет получать значения `1`, `4` и `7`.

В листинге 10.48 приведен пример вычисления суммы всех нечетных чисел от 1 до 1000 с помощью функции `stride(from:through:by:)`.

Листинг 10.48

```
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, 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 10.49).

Листинг 10.49

```
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 10.50 показаны два блока: первый с использованием двух операторов, а второй с использованием `for` и `where`. При этом обе конструкции функционально идентичны.

Листинг 10.50

```
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 экономичные пути решения поставленных задач.

Многомерные коллекции в конструкции for-in

Если перед вами стоит задача обработки многомерных коллекций, то вы можете с легкостью организовать это, а именно вкладывать конструкции `for-in` друг в друга.

В листинге 10.51 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 10.51

```
// словарь с результатами игр
let 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
```

Константа `resultsOfGames` — «словарь массивов» с типом `[String: [String]]`. Параметр `teamName` — локальный 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 и игнорирует следующий за ним код тела оператора.

В листинге 10.52 представлена программа, в которой переменная поочередно принимает значения от 1 до 10, причем когда значение нечетное, оно выводится на консоль.

Листинг 10.52

```
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 10.53 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 10.53

```
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 что 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.

Также вы можете пойти по другому пути и выбрать случайное число, используя диапазоны (листинг 10.54). В этом случае в подгрузке библиотеки Foundation нет необходимости.

Листинг 10.54

```
for i in 1... {
    let randNum = Array<Int>(0...100).randomElement()
    if randNum == 5 {
        print("Итерация номер \(i)")
        break
    }
}
```

В этом примере одним выражением создается массив всех целочисленных элементов от 0 до 100, после чего с помощью метода `randomElement()` возвращается случайный из них.

Также вы можете использовать оператор `break`, чтобы прервать выполнение внешнего цикла. Для этого в Swift используются метки (листинг 10.55).

Листинг 10.55

```
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.

10.6. 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`.

Далее в книге мы подробно познакомимся с `guard` и рассмотрим примеры его использования.

10.7. Где использовать операторы управления

Описываемый механизм	Где используется
Утверждения	Используются в процессе отладки и тестирования приложений. Данный механизм будет подробно описан в будущих книгах при рассмотрении вопросов тестирования приложений.
Оператор `if`	Используется для выбора исполняемого блока кода в зависимости от значения тестируемого выражения, в котором могут присутствовать любые логические операторы сравнения (`<`, `>`, `<=`, `>=`), а также оператор проверки на эквивалентность (`==`). **Пример:** Проверка статуса пользователя и текущего времени. `if user.isOnline == true && 9...18.contains(currentHour) {` `// пользователь онлайн в рабочее время – уволить` `}`
Оператор `switch`	Используется для выбора исполняемого блока кода в зависимости от значения тестируемого выражения. Вы должны иметь конкретный список значений, которое может принять выражение. **Пример:** Определение месяца. `switch currentMonths {` `case .january:` `// ...` `case .february:` `// ...` `case .march:` `// ...` `default:` `// ...` `}`
Операторы `while` и `repeat while`	Используются для многократного выполнения блока кода, когда количество итераций заранее неизвестно и зависит от определенного условия. **Примеры:** Проведение необходимого количества уровней в игре. `var currentLevel = 1` `let finalLevel = 5` `// с использованием while` `while currentLevel <= finalLevel {` `// начало нового уровня` `// тут может находиться код, обеспечивающий прохождение` `// уровня`

Описываемый механизм	Где используется
	`` currentLevel += 1 } // игра окончена // с использованием repeat-while currentLevel = 1 repeat { // начало нового уровня // тут может находиться код, обеспечивающий прохождение // уровня currentLevel += 1 } while currentLevel <= finalLevel // игра окончена ```
Оператор `for`	Используется для многократного выполнения блока кода, когда количество итераций заранее известно. **Примеры:** Перебор элементов последовательности. ``` for element in [1, 2, 3, 4] { // при каждой новой итерации // element содержит очередной элемент из диапазона 1...10 } for char in "строка с текстом" { // при каждой новой итерации // char содержит очередной символ строки } ``` Выполнение блока кода необходимое количество раз. ``` for _ in 1...10 { // код в теле конструкции будет выполнен 10 раз } ```
Оператор `guard`	Используется, когда необходимо гарантировать, что тестируемое выражение будет выполнено (вернет `true`), в противном случае код после оператора выполнен не будет и произойдет выход из текущей области видимости (завершится функция, прервется выполнение цикла и др.). В большинстве случаев позволяет избежать многоуровневых вложенностей и сделать код более читабельным и простым в обслуживании. **Примеры:** Наиболее часто используется для извлечения опционального значения, так как последующий код может использовать извлеченное значение. Если использовать оператор `if`, то значение будет доступно только в соответствующем блоке оператора. ``` func shapeAnalizator(shape: Shape) { // вариант с использованием if if let sides = shape.sides, sides > 2 { ```

Описываемый механизм	Где используется
	`` // sides доступна } // sides недоступна // вариант с использованием guard guard let sides = shape.sides, sides > 2 else { return } // sides доступна } ``` Проверка условия, обязательного для работы функции. ``` func getTitleBy(id idIntDataBase: UInt) -> String? { // проверим существование записи по данному id guard let dataBase.getRowBy(id: idIntDataBase) else { return nil } // ... } ```

Выбор между if и switch

Операторы `if` и `switch` выполняют схожие задачи: позволяют запустить определенный блок кода в зависимости от значения тестируемого выражения. Время от времени перед вами будет возникать вопрос, какую из доступных конструкций желательно использовать в конкретном случае. При этом обычно задача может быть решена как с помощью `if-else`, так и с помощью `switch-case`.

Отдавайте предпочтение `switch`, если:

- Вы проверяете эквивалентность тестируемого выражения значению из конкретного набора, то есть вы знаете частично или полностью весь список возможных значений.
- У вас есть две или более ветки с программным кодом, по которым программа может продолжить свою работу.

Нет смысла использовать `switch`, если у вас есть всего один конкретный вариант значения тестируемого выражения (листинг 10.56).

Листинг 10.56

```
// неверный подход
switch userStatus {
    case .online:
    // …
    default:
    // …
}
```

```
// лучше провести проверку с помощью if
if userStatus == .online {
    // …
} else {
    // …
}
```

Отдавайте предпочтение `if`, если:

- Вы проверяете тестируемое выражение не просто на эквивалентность некоторым значениям, а решаете логическое выражение с одним или с множеством условий. Не забывайте, что оператор `switch` с помощью ключевого слова `where` позволяет указать дополнительное логическое выражение (листинг 10.57).

Листинг 10.57

```
// неверный подход
switch userStatus {
    case _ where a > b:
    // …
}
// лучше провести проверку с помощью if
if a > b {
    // …
}
```

Стоит также упомянуть о некоторых дополнительных положительных сторонах использования `switch-case` перед `if-else`.

Во-первых, используя `switch`, вы уменьшаете вероятность ошибки, так как тестируемое выражение должно быть написано лишь единожды. В то же время `if` требует, чтобы вы писали его в каждом блоке.

Во-вторых, оператор `switch` выполняется быстрее. Компьютер строит специальную таблицу переходов и заранее знает, при каком условии выражения какая ветка кода должна быть выполнена. В то же время оператор `if` рассчитывает условия постепенно, одно за другим. Так, последнее условие всегда требует наибольшей работы, так как перед ним вычисляются все предыдущие.

В-третьих, оператор `switch` проще в сопровождении и изменении. Вы никогда не пропустите default-блок, в то время как можете с легкостью забыть else-блок. Также проще добавить очередной case-блок и не допустить ошибок, чем писать новое условие в составе существующего оператора `if`.

Глава 11. 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 есть также опциональные типы данных. Это одно из важных нововведений языка, которое, вероятно, не встречалось вам ранее.

Все переменные и константы, которые мы определяли до этого момента, всегда имели некоторое конкретное значение.

Когда вы объявляете какой-либо параметр, например `var name: String`, то с ним ассоциируется определенное значение, например `Владимир`, 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.

11.1. 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, так как 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, когда изучали коллекции.

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 11.1).

Листинг 11.1

```
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, а 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 11.2).

Листинг 11.2

```
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 либо быть целым числом, либо отсутствовать полностью. Тип `Int` является базовым для этого опционала, то есть 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 нечисловое значение. Так, если в `Int(_:)` передается значение типа `Double`, то нет никакой необходимости возвращать опционал, так как при любом значении `Double` оно может быть преобразовано в `Int` (чего 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`).

```
let x1 = Int("12")
type(of: x1) // Optional<Int>.Type
let 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 11.3 приведен пример объявления переменной опционального типа, основанного на `Character`.

Листинг 11.3

```
let 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 11.4 показан пример объявления опционала с использованием сокращенного синтаксиса.

Листинг 11.4

```
var xCoordinate: Int? = 12
```

В любой момент значение опционала может быть изменено на `nil`. Это можно сделать как и при объявлении параметра, так и потом (листинг 11.5).

Листинг 11.5

```
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 11.6).

Листинг 11.6

```
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 11.7).

Листинг 11.7

```
// опциональная переменная с установленным значением
var optionalVar = Optional("stringValue") // "stringValue"
// уничтожаем значение опциональной переменной
optionalVar = nil // nil
type(of: optionalVar) // Optional<String>.Type
```

Так как в функцию `Optional(_:)` в качестве аргумента передано значение типа `String`, то возвращаемое ею значение имеет опциональный строковый тип данных `String?` (или `Optional<String>`, что является синонимами).

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 11.8 приведен пример объявления опционального кортежа.

Листинг 11.8

```
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 11.9).

Листинг 11.9

```
let tupleWithoptelements: (Int?, Int) = (nil, 100)
tupleWithoptelements.0 // nil
tupleWithoptelements.1 // 100
```

11.2. Извлечение опционального значения

Важно отметить, что нельзя производить прямые операции между значениями опционального и базового типов данных, так как это приведет к ошибке (листинг 11.10).

Листинг 11.10

```
let a: Int = 4
let 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 11.11.

Листинг 11.11

```
var optVar: Int? = 12
var intVar = 34
let result = optVar! + 34 // 46

// проверяем тип данных извлеченного значения
type(of: optVar!) // Int.Type
```

Константа `optVar` — это опционал. 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 11.12).

Листинг 11.12

```
let optString: String? = "Vasiliy Usov"
let 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. 11.1 показано, что происходит, когда производится попытка извлечения несуществующего значения.

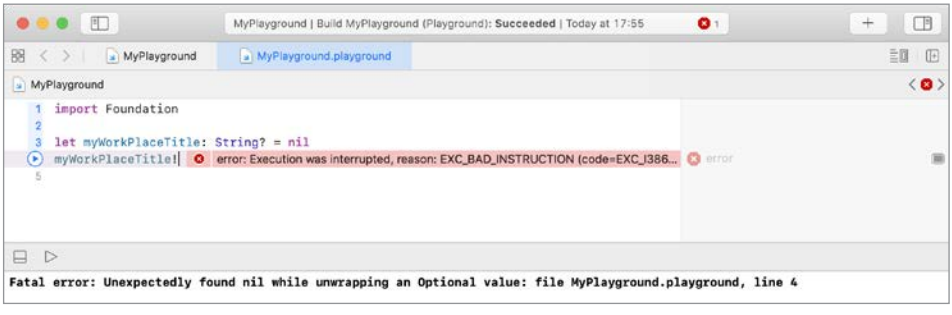

Рис. 11.1. Ошибка при извлечении несуществующего опционального значения

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 сомнения, 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 11.13).

Листинг 11.13

```
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, чревато плохими отзывами пользователей).

11.3. 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 11.14).

Листинг 11.14

```
let optOne: UInt? = nil
let 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 11.15 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 11.15

```
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.

11.4. 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 является локальным. Вне этого блока объект просто не виден.

В листинге 11.16 показан пример использования опционального связывания.

Листинг 11.16

```
let 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 11.17).

Листинг 11.17

```
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 значений 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. 11.2).

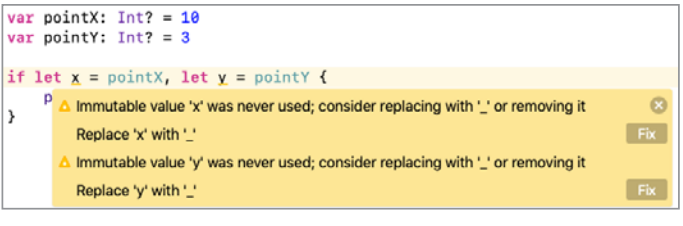

Рис. 11.2. Предупреждение от Xcode

Ранее мы уже неоднократно встречались с нижним подчеркиванием, позволяющим игнорировать определенные элементы или значения. 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 11.18).

Листинг 11.18

```
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 11.19).

Листинг 11.19

```
if let x = pointX, x > 5 {
    print("Точка очень далеко от вас ")
}
```

Консоль

```
Точка очень далеко от вас
```

11.5. 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 11.20).

Листинг 11.20

```
/* 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. На то есть три причины:

- 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 такой ситуации, заранее создав переменную `coins`, в которую будет извлечено значение опционала. Данную переменную необходимо использовать в обоих случаях вместо вызова функции `Int(_:)` (листинг 11.21).

Листинг 11.21

```
let 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 11.22).

Листинг 11.22

```
let 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 более сложных приложений для мобильных или стационарных устройств данный подход позволит получать вполне ощутимые результаты.

11.6. 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 в объявленный параметр.

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 11.23 показан пример использования оператора объединения с `nil` для извлечения значения.

Листинг 11.23

```
let 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 11.24.

Листинг 11.24

```
let 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.

11.7. Где использовать опциональные значения

Описываемый механизм	Где используется
Опционалы	Используется, когда параметр может иметь некоторое значение или не иметь его вовсе. **Примеры:** Загрузка записи из базы данных по его ID может либо вернуть эту запись, либо вернуть `nil`, указывающий на то, что такой записи нет в базе. `func getObjectBy(id: UInt) -> Object? {` `  // код загрузки данных из базы` `}` Получение файла по его имени. При отсутствии файла с таким именем возвращается `nil`. `func getFileBy(name: UInt) -> File? {` `  // код загрузки файла` `}` Ресурс большого объема, который в будущем заменится на `nil`, чтобы освободить память.

Описываемый механизм	Где используется
	Указатель на наличие/отсутствие ошибки в результате запроса на сервер. `let error: Error? = getErrorFromServerQuery()` `guard let error = error else {` `  print("Ошибок нет")` `}` `print("Ошибка \(error.code)")`
	Свойства класса, которые будут установлены уже после его инициализации. К примеру, свойства класса `ViewController` будут связаны с элементами на сцене (IBOutlet) уже после того, как произойдет инициализация. `class ViewController: UIViewController {` `  var textArea: UITextArea!` `  var button: UIButton!` `  // ...` `}`

Глава 12. Функции

Мы уже неоднократно встречались с функциями и 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.

12.1. Введение в функции

Знакомство с функциями мы начнем с описания их свойств и характеристик.

Функция:

- группирует программный код в единый контейнер;
- имеет собственное имя, с помощью которого может быть многократно вызвана с возможностью передачи аргументов;
- создает отдельную область видимости внутри себя, в результате чего все созданные в теле функции параметры недоступны извне;
- может принимать входные параметры;
- может возвращать значение как результат исполнения сгруппированного в ней кода;
- имеет собственный функциональный тип данных;
- может быть записана в параметр (переменную или константу) и в таком виде передана.

Сложно? Это только кажется! И вы убедитесь в этом, когда начнете создавать функции самостоятельно.

На вопрос, 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 хоть 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
let apple = Apple()
// используем соковыжималку
let 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) -> ТипВозвращаемогоЗначения {
    // тело функции
}
```

`имяФункции` — имя объявляемой функции, по которому она сможет быть вызвана. К функциям применимы те же правила именования, что и к параметрам.

`входные параметры` — список входных параметров функции с указанием их имен и типов.

`ТипВозвращаемогоЗначения` — 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 параметры функции. Список параметров заключается в круглые скобки и состоит из разделенных запятыми элементов. Каждый отдельный элемент описывает один параметр и состоит из имени и типа этого параметра, разделенных двоеточием. Параметры позволяют передать в функцию значения, которые ей требуются для корректного выполнения возложенных на нее задач. Количество 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.

Обратите внимание, что при объявлении функции в скобках указываются **входные параметры**, а при вызове функции передаются **аргументы**.

```
// входные параметры a и b
func myFirstFunc(a: Int, b: String) { ... }
// аргументы a и b
myFirstFunc(a: 54, b: "слово")
```

Это два близких, но отличающихся понятия. Вам необходимо запомнить, что при вызове функции указываются аргументы, а внутри функции используются параметры.

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 ключевого слова Void (это Good Practice), например:

```
func myFirstFunc(
a: Int,
b: String
) -> Void
```

- с помощью пустых скобок (), например:

```
func myFirstFunc(
a: Int,
b: String
) -> ()
```

- 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 {
    let word = "Swift"
    return String(a) + b + word
}
```

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.

При этом если тело функции состоит из одного выражения, значение которого должно быть возвращено, то оператор `return` может быть опущен.

```
func myFirstFunc(
    a: Int,
    b: String
) -> String {
    String(a) + b
}
```

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 {
    String(a) + b
}
```

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 12.1).

Листинг 12.1

```
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.

12.2. Входные параметры и возвращаемое значение

Функция может принимать входные 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 значений и возвращать полученный результат в виде значения типа `Int`. Правильным подходом будет объявление функции, генерирующей данные действия. В качестве входного параметра 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 параметры

Реализуем описанную выше задачу, но при этом исключим из нее требование возвращать результат сложения. Пусть результат операции выводится на отладочную консоль (листинг 12.2).

Листинг 12.2

```
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 параметры должны иметь значения, поэтому попытка вызвать функцию, передав в нее лишь один аргумент или не передав их вовсе, завершится ошибкой.

ПРИМЕЧАНИЕ Типы и имена аргументов вам подскажет Xcode, а точнее, механизм автодополнения кода.

Внешние имена входных параметров

Аргументы `a` и `b` функции `sumTwoInt(a:b:)` используются как при вызове функции, так и в ее теле (там они называются входными параметрами). Swift позволяет указать внешние имена параметров, которые будут использоваться при вызове функции (листинг 12.3).

Листинг 12.3

```
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 аргументов, но при этом сократить количество 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 12.4).

Листинг 12.4

```
func sumTwoInt(_ a: Int, _ b: Int) {
    print("Результат операции - \(a+b)")
}
sumTwoInt(10, 12)
```

ПРИМЕЧАНИЕ Внешние имена могут быть заданы для произвольных параметров, 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:

- должен быть указан тип возвращаемого значения;
- должен быть использован оператор `return` в теле функции с возвращаемым значением в качестве операнда.

ПРИМЕЧАНИЕ Напоминаю, что оператор `return` не обязателен в случае, когда тело функции состоит из одного выражения.

Так как результат операции сложения — целое число, в качестве типа данных выходного значения необходимо указать `Int` (листинг 12.5).

Листинг 12.5

```
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 различных и независимых параметра! Все, что объявляется в теле функции, является локальным для нее и уничтожается после завершения ее работы. Таким образом, в теле 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 параметров

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 12.6).

Листинг 12.6

```
func returnMessage(code: Int, message: String) -> String {
    var mutableMessage = message
    mutableMessage += String(code)
    return mutableMessage
}
let 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 параметров позволяет получать доступ к изменяемому значению только в пределах тела самой функции. Для того чтобы была возможность модификации параметров 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 12.7 обеспечивает обмен значениями двух внешних параметров.

Листинг 12.7

```
func changeValues(_ a: inout Int, _ b: inout Int) -> Void {
    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 изменить 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 параметра

Вы можете использовать возвращаемое некоторой функцией значение в качестве значения входного параметра другой функции. Самое важное, чтобы тип возвращаемого значения функции совпадал с типом входного параметра.

В листинге 12.8 используется объявленная ранее функция `returnMessage (code:message:)`, возвращающая значение типа `String`.

Листинг 12.8

```
// 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 параметра для `print(_:)`.

Входной параметр с переменным числом значений

В некоторых ситуациях необходимо, чтобы функция получала неизвестное заранее число однотипных значений. Мы уже встречались с таким подходом при использовании `Array(arrayLiteral:)`, когда заранее неизвестно, сколько элементов будет содержать параметр `arrayLiteral`. Такой тип входного параметра называется *вариативным*.

Вариативный параметр обозначается в списке входящих параметров с указанием оператора диапазона `...` сразу после типа. Значения аргумента при вызове функции задаются через запятую.

Рассмотрим пример из листинга 12.9. 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 12.9

```
func printRequestString(codes: Int...) -> Void {
    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 параметров.

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 12.10 функция принимает на вход код ответа сервера и в зависимости от того, к какому диапазону относится переданный код, возвращает кортеж с его описанием.

Листинг 12.10

```
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 12.11).

Листинг 12.11

```
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 параметра

Напомню, что все входные параметры 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 параметра, то оно будет использовано в теле функции; если вы не передали значение параметра, для него будет использовано значение по умолчанию. Значение по умолчанию указывается при объявлении функции в списке входных параметров для каждого параметра отдельно.

Доработаем объявленную ранее функцию `returnMessage(code:message:)` таким образом, чтобы была возможность не передавать значение параметра `message`. Для этого укажем значение по умолчанию (листинг 12.12).

Листинг 12.12

```
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 параметра `message`. Это стало возможным благодаря установке значения по умолчанию `"Код - "` в списке входных параметров.

Тело функции `returnMessage(code:message:)` написано не самым лучшим способом, так как, по сути, оно может состоять лишь из одного выражения, а следовательно, и лишиться оператора `return`. В листинге 12.13 показан оптимальный вариант данной функции.

Листинг 12.13

```
func returnMessage(code: Int, message: String = "Код - ") -> String {
    message + String(code)
}
returnMessage(code: 300) // "Код - 300"
```

> **ПРИМЕЧАНИЕ** Функция, которая принимает не более одного параметра, — это замечательно.
>
> Функции с двумя или тремя параметрами — у вас должны быть серьезные причины для того, чтобы реализовать ее.
>
> Функция с четырьмя и более параметрами — отличный повод пересмотреть структуру вашего кода.
>
> Каждая функция, которую вы реализуете, должна выполнять четкую и конкретную задачу. Не стоит создавать функции-комбайны, которые выполняют несколько задач сразу. Хорошо структурированный код позволяет вам с удобством читать его, искать необходимые функциональные блоки и оперативно вносить правки.

12.3. 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.

Простой функциональный тип

Если функция ничего не принимает и не возвращает, то ее тип указывается круглой скобкой и ключевым словом `Void` после стрелки:

```
() -> Void
```

> **ПРИМЕЧАНИЕ** Также возможен вариант с двумя парами круглых скобок, разделенных стрелкой `() -> ()`, но он не является хорошей практикой программирования.

В листинге 12.14 приведен пример функции с типом `() -> Void`, то есть не имеющей ни входных, ни выходных параметров.

Листинг 12.14

```
func printErrorMessage() {
    print("Произошла ошибка")
}
```

В первых скобках функционального типа всегда описываются типы данных входных параметров, а после стрелки 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 следующий функциональный тип:

```
() -> () -> Void
```

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 параметров, 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 параметра. Далее приведен пример, в котором функция принимает на вход целое число и другую функцию, а возвращает логическое значение.

```
(Int, (Int) -> Void) -> Bool
```

Функция, которая передается в качестве входного параметра, имеет тип `(Int) -> Void`, то есть она сама принимает целочисленное значение, но не возвращает ничего.

12.4. 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 12.15 объявлена функция `returnPrintTextFunction()`, которая возвращает значение функционального типа `() -> Void`, то есть другую функцию.

Листинг 12.15

```
// функция вывода текста на консоль
func printText() {
    print("Функция вызвана")
}
// функция, которая возвращает функцию
func returnPrintTextFunction() -> () -> Void {
    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 `() -> Void`, а сама она хранит в себе функцию, которую можно вызывать, указав имя хранилища с круглыми скобками после него. На рис. 12.1 отображено справочное окно, описывающее параметр `newFunctionInLet`, в котором хранится функция `printText()`.

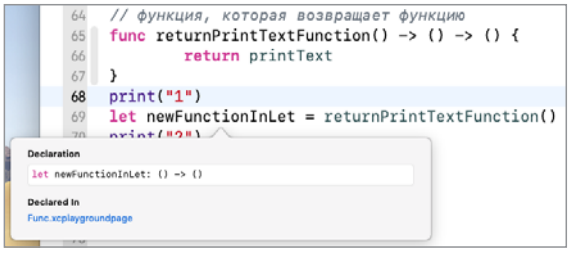

Рис. 12.1. Справочное окно для константы, хранящей функцию в качестве значения

Обратите внимание на вывод на отладочной консоли. Так как строка `Функция вызвана` находится между шагами 2 и 3, а не между 1 и 2, можно сделать вывод 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 параметров. Переданную таким образом функцию можно будет использовать в теле той функции, в которую она была передана. Для этого необходимо указать корректный функциональный тип входного параметра и в качестве его значения указать имя передаваемой функции.

Напишем функцию `generateWallet(walletLength:)`, случайным образом генерирующую 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 вычисляет и возвращает сумму всех банкнот в кошельке (листинг 12.16).

Листинг 12.16

```
// функция генерации случайного массива банкнот
func generateWallet(walletLength: Int) -> [Int] {
    // существующие типы банкнот
```

```
    let typesOfBanknotes = [50, 100, 500, 1000, 5000]
    // массив банкнот
    var wallet: [Int] = []
    // цикл генерации массива случайных банкнот
    for _ in 1...walletLength {
        wallet.append( typesOfBanknotes.randomElement()! )
    }
    return wallet
}
// функция подсчета денег в кошельке
func sumWallet(
    banknotesFunction 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(banknotesFunction: generateWallet, walletLength: 20) // 6900
```

Значение в области результатов, вероятно, будет отличаться от того, что показано в примере. Это связано с использованием метода `randomElement()`, который возвращает случайный элемент коллекции `typesOfBanknotes`.

Функция `generateWallet(walletLength:)` создает массив банкнот такой длины, которая передана ей в качестве входного параметра. В массиве `typesOfBanknotes` содержатся все доступные (разрешенные) номиналы банкнот. Суть работы функции такова: банкнота случайным образом изымается из массива `typesOfBanknotes`, после чего 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 `banknotesFunction` функции `sumWallet(banknotesFunction:walletLength:)` указан функциональный тип `(Int) -> [Int]`. Он соответствует типу функции `generateWallet(walletLength:)`.

При вызове `sumWallet(banknotesFunction:walletLength:)` необходимо указать лишь имя передаваемой функции без фигурных скобок.

Чего мы добились таким образом? Того, что функция `sumWallet(banknotesFunction: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 банкнот, после чего передать ее в качестве аргумента в `sumWallet(banknotesFunction:walletLength:)`.

В следующей главе вы увидите всю мощь, которую дают нам входные и выходные параметры 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 следующих 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 параметров 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

получено именно в тот момент, когда к нему произойдет обращение, а не в тот момент, когда функции переданы в виде 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.

12.5. 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 12.17).

Листинг 12.17

```
func oneStep(
    coordinates: inout (Int, Int),
    stepType: String
) -> Void {
    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 параметра `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 (входных и выходных параметрах).

В листинге 12.18 представлены функции, которые могут сосуществовать одновременно в одной области видимости.

Листинг 12.18

```
func say(what: String) -> Void {}
func say(what: Int) -> Void {}
```

У данных функций одно и то же имя `say(what:)`, один и тот же тип возвращаемого значения, но различные типы входных параметров. В результате Swift определяет обе функции как различные и позволяет им сосуществовать одновременно. Это связано с тем, что функциональный тип первой функции `(String) -> Void`, а второй — `(Int) -> Void`.

Рассмотрим пример из листинга 12.19. Представленные в нем функции также могут сосуществовать одновременно.

Листинг 12.19

```
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 12.20).

Листинг 12.20

```
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 12.21).

Листинг 12.21

```
let resultString: String = cry() // "one"
let 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 12.22.

Листинг 12.22

```
func countdown(firstNum num: Int) -> Void {
    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.

12.6. Где использовать функции

Описываемый механизм	Где используется
Функции	Используются, когда необходимо сгруппировать блок кода для его многократного использования.

Функции — это один из важнейших элементов языка, однозначно вы будете прибегать к их возможностям в каждой программе (в том числе в виде методов или замыканий).

Пример:

Подсчет суммы двух чисел.

```
func sumOf(_ a: Int, and b: Int) -> Int {
    return a+b
}
sumOf(2, and: 4) // 6
```

Получение имени пользователя из базы данных по его почтовому адресу.

```
func getUserBy(email: String) ->  User {
    // код загрузки пользователя из базы данных (или иного
    // хранилища)
}
```

Глава 13. Замыкания (closure)

Как объясняется в документации к языку Swift, *замыкания* (`closures`) — это организованные блоки с определенной функциональностью, 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.

13.1. 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 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 — безымянные функции.

13.2. Введение в безымянные функции

Как вы уже знаете, переменная и константа может 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) -> ТипВозвращаемогоЗначения in
    // тело замыкающего выражения
}
```

`входные_параметры` — список аргументов замыкания с указанием их имен и типов.

`ТипВозвращаемогоЗначения` — 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 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 банкноту определенного номинала. Перед нами стоит задача отбора банкнот в кошельке по различным условиям. Для каждого условия может быть создана отдельная функция, принимающая на вход массив `wallet` и возвращающая отфильтрованную коллекцию.

В листинге 13.1 показано, каким образом может быть реализована функция отбора всех сторублевых купюр.

Листинг 13.1

```
// 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 banknote in wallet {
        if banknote == 100 {
            returnWallet.append(banknote)
        }
    }
    return returnWallet
}
// вызов функции отбора купюр с 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 13.2).

Листинг 13.2

```
func handleMore1000(wallet: [Int]) -> [Int] {
    var returnWallet = [Int]()
    for banknote in wallet {
        if banknote >= 1000 {
            returnWallet.append(banknote)
        }
    }
    return returnWallet
}
// вызов функции отбора купюр с достоинством более или равно 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:

- реализовать всю функциональность отбора купюр в пределах одной функции, а в качестве аргумента передавать условие;
- реализовать всю функциональность 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:

- Функция с именем `handle`, принимающая массив-кошелек и условие отбора (в виде имени функции) в качестве аргументов и возвращающая массив отобранных купюр. В теле функции будут поочередно проверяться элементы входного массива на соответствие переданному условию.
- Функция с именем `compare100`, принимающая на вход значение очередного элемента массива-кошелька, производящая сравнение с целым числом 100 и возвращающая логический результат этой проверки.
- Функция с именем `compareMore1000`, аналогичная `compare100`, но производящая проверку на соответствие целому числу 1000.

В листинге 13.3 показана реализация описанного алгоритма.

Листинг 13.3

```
// единая функция формирования результирующего массива
func handle(wallet: [Int], closure: (Int) -> Bool) -> [Int] {
    var returnWallet = [Int]()
    for banknote in wallet {
        if closure(banknote) {
            returnWallet.append(banknote)
        }
    }
    return returnWallet
}
// функция сравнения с числом 100
func compare100(banknote: Int) -> Bool {
    return banknote == 100
}
// функция сравнения с числом 1000
func compareMore1000(banknote:Int) -> Bool {
    return banknote >= 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 многим и многим условиям (найти все полтинники; все купюры достоинством менее 1000 рублей; все купюры, которые без остатка делятся на 200, и т. д.). Условий отбора может быть великое множество. 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 13.4 показано, каким образом это может быть реализовано.

Листинг 13.4

```
// отбор купюр достоинством выше 1000 рублей
// аналог передачи compareMore1000
handle(wallet: wallet, closure: { (banknote: Int) -> Bool in
    return banknote >= 1000
})
// отбор купюр достоинством 100 рублей
// аналог передачи compare100
handle(wallet: wallet, closure: { (banknote: Int) -> Bool in
    return banknote == 100
})
```

Входной параметр `closure` имеет функциональный тип `(Int)->Bool`, а значит, передаваемая безымянная функция должна иметь тот же тип данных, что мы 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` возвращается 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 параметра.

В результате такого подхода необходимость в существовании функций `compare100(banknote:)` и `compareMore1000(banknote:)` 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 1000 рублей и больше.

13.3. 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 аргумента (листинг 13.5).

Листинг 13.5

```
// отбор купюр достоинством выше 1000 рублей
handle(wallet: wallet, closure: { banknote in
    return banknote >= 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 13.6).

Листинг 13.6

```
handle(wallet: wallet, closure: { banknote in banknote >= 1000})
```

Сокращенные имена параметров

В случае, когда замыкание состоит из одного выражения, можно опустить указание входных параметров (все до ключевого слова `in`, включая само слово). При этом доступ к входным параметрам 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 13.7).

Листинг 13.7

```
handle(wallet: wallet,
    closure: {$0>=1000})
```

Здесь `$0` — это входной параметр `banknote` аргумента `closure` в функции `handle(wallet:closure:)`.

Вынос замыкания за скобки

Если входной параметр функции 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 13.8).

Листинг 13.8

```
handle(wallet: wallet){ $0 >= 1000 }
```

Эта возможность особенно полезна, когда замыкание, передаваемое в качестве аргумента функции, является многострочным. В листинге 13.9 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 13.9

```
handle(wallet: wallet) { banknote in
    for number in Array(arrayLiteral: 100,500) {
        if number == banknote {
            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:
>
> ```
> let successBanknotes = handle(wallet: wallet) { [100,500].contains($0) }
> successBanknotes // [100, 100, 100, 500, 100]
> ```

Вынос нескольких замыканий за скобки

Начиная с версии 5.3, в Swift появилась возможность вынести за скобки не одно, а все замыкания, находящиеся в конце списка аргументов. Предположим, что вы написали функцию, осуществляющую запрос на сервер (листинг 13.10).

Листинг 13.10

```
func networkQuery(url: String, success: (String) -> (), error: (Int) -> ()) {
    // код запроса на сервер
}
```

В качестве аргументов в функцию передаются URL-адрес и два замыкания: первое будет вызвано в случае успешного окончания запроса, а второе — в случае ошибки. При вызове функции `networkQuery` вы можете использовать как стандартный синтаксис, указывая замыкания прямо в списке аргументов, так и упрощенный, вынеся оба замыкания за скобки (листинг 13.11).

Листинг 13.11

```
// классический вариант
networkQuery(url: "https://weather.com", success: { data in }, error:
                  {errorCode in })

// новый вариант
networkQuery(url: "https://weather.com") { data in
    // ...
} error: { errorCode in
    // ...
}
```

Данная возможность будет использоваться вами очень часто при работе с фреймворком SwiftUI.

13.4. Безымянные функции в параметрах

В листинге 13.12 показан пример инициализации замыкания в параметр `closure`. При этом у параметра явно указан функциональный тип (ранее в примерах он определялся неявно).

Листинг 13.12

```
let closure: () -> Void = {
    print("Замыкающее выражение")
}
closure()
```

Консоль

```
Замыкающее выражение
```

Так как данное замыкающее выражение не имеет входных параметров и возвращаемого значения, то его тип равен `() -> Void`. 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 параметры и тип выходного значения (листинг 13.13).

Листинг 13.13

```
// передача в функцию строкового значения
let closurePrint: (String) -> Void = { text in
    print(text)
}
closurePrint("Text")
```

Консоль

```
Text
```

```
// передача в функцию целочисленных значений
// с осуществлением доступа через краткий синтаксис $0 и $1
var sum: (_ numOne: Int, _ numTwo: Int) -> Int = {
    $0 + $1
}
sum(10, 34) // 44
```

ПРИМЕЧАНИЕ Ключевое слово `return` не используется в замыкании `sum`, так как его тело состоит из одного выражения.

Входные параметры замыкания не должны иметь внешних имен. По этой причине в первом случае указание имени вообще отсутствует, а во втором используется знак нижнего подчеркивания.

13.5. 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 13.14 массив `array` сортируется 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 13.14

```
let 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 именами сокращенной формой.

В результате получится выражение, приведенное в листинге 13.15. 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 13.15

```
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 13.16).

Листинг 13.16

```
sortedArray = array.sorted(by: <)
sortedArray // [1, 4, 8, 44, 50, 51, 81, 101, 277]
```

Надеюсь, вы приятно удивлены потрясающими возможностями Swift!

13.6. 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 13.17. Существуют два параметра, `a` и `b`, которые не передаются в качестве 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 13.17

```
var a = 1
var b = 2
let closureSum : () -> Int = {
    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 13.18).

Листинг 13.18

```
var a = 1
var b = 2
let closureSum : () -> Int = { [a, b] in
    a + b
}
closureSum() // 3
    a = 3
    b = 4
closureSum() // 3
```

Замыкание, хранящееся в константе `closureSum`, складывает значения переменных `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 параметры родительской функции.

Рассмотрим пример из листинга 13.19.

Листинг 13.19

```
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 параметров и возвращающее целочисленное значение.

Функция `makeIncrement(forIncrement:)` использует два параметра:

- `runningTotal` — переменную типа `Int`, объявленную в теле функции. Именно ее значение является результатом работы всей конструкции;
- `amount` — входной параметр, 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 13.20.

Листинг 13.20

```
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).

13.7. 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 13.21.

Листинг 13.21

```
var incrementByFive = makeIncrement(forIncrement: 5)
var copyIncrementByFive = incrementByFive
```

В данном примере используется функция `makeIncrement(forIncrement:)`, объявленная ранее. Напомню, она возвращает замыкание 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 13.22).

Листинг 13.22

```
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.

13.8. 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 13.23.

Листинг 13.23

```
var arrayOfNames = ["Helga", "Bazil", "Alex"]
func printName(nextName: String ) {
    print(nextName)
}
printName(nextName: arrayOfNames.remove(at: 0))
```

Консоль

```
Helga
```

При 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 13.24).

Листинг 13.24

```
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 параметру `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 в виде значения (без фигурных скобок).

Для того чтобы реализовать автозамыкание, требуется, чтобы выполнялись следующие требования:

- Входной параметр должен иметь функциональный тип.

  В примере, приведенном ранее, параметр `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 аргумента функция возвращает значение типа `String` точно так же, как определено функциональным типом входного параметра.

- 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 13.25).

Листинг 13.25

```
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:)`. Входные параметры `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.

13.9. Выходящие (сбегающие) 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 являются 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 13.26).

Листинг 13.26

```
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 13.27).

Листинг 13.27

```
func addNewClosureInArray(_ newClosure: ()->Int) {
    arrayOfClosures.append(newClosure) // ошибка
}
```

Xcode сообщит вам об ошибке. И на то есть две причины:

- Замыкание — это тип-ссылка (reference type), 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 (сбегающим). Для этого перед описанием функционального типа данного параметра укажите атрибут `@escaping`, после чего вы сможете передать в функцию `addNewClosureInArray(_:)` произвольное замыкание (листинг 13.28).

Листинг 13.28

```
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 параметр, то допускаются оба варианта.

ПРИМЕЧАНИЕ Если вы передаете замыкание в виде параметра, то можете использовать модификатор `inout` вместо `@escaping`.

13.10. Где использовать замыкания

Описываемый механизм	Где используется
Замыкания	Используются, когда необходимо сгруппировать блок кода для его передачи или многократного использования. В отличие от функций, замыкания не имеют имени. **Примеры:** Наиболее часто замыкания используются в качестве обработчика завершения (completion handler). При таком подходе они передаются для обработки результатов выполнения другого действия после его завершения. К примеру, вам необходимо организовать функцию для осуществления запросов к серверу в Сети. При этом в зависимости от того, какую информацию вы загружаете, вам потребуется по-разному ее обработать. `// загрузка новостей с сервера` `webServer.request(url: "https://swiftme.ru/news", completion:` `{answerData in` `for item in answerData.items {` `// сохранение новостей в базе данных` `saveToDataBase(item)` `}` `})` `// загрузка лого с сервера` `webServer.request(url: " https://swiftme.ru/logoImage",` `completion: {answerData in` `// отображение лого на экране устройства` `showLogo.image(answerData)` `})` Метод `webServer.request` совершает запрос к серверу, при этом в качестве аргументов ему передается адрес, а также замыкание, которое обрабатывает полученный ответ. Таким образом, для любого запроса можно использовать один и тот же метод (`request`), именно он будет нести ответственность за корректность выполнения данной операции. Но обработка ответа будет осуществляться в соответствии с логикой, переданной в замыкании. Еще один пример использования замыкания в качестве completion handler — создание `UIAlertAction` (кнопка на всплывающем уведомлении в iOS). `// создание UIAlertController` `let alert = UIAlertController(` `title: "Alert",`

Описываемый механизм	Где используется

```
    message: "Volume is \(volume)",
    preferredStyle: .alert
)

// создание "кнопки" для UIAlertController
// при этом передается замыкание, которое
// будет выполнено при нажатии на эту кнопку
let okAction = UIAlertAction(
    title: "Cancel",
    style: .cancel,
    handler:{ (action) in
        resultLabel.text = "Действие выполнено"
    })

// размещение кнопки на UIAlertController
alert.addAction(okAction)
```

Другим популярным примером использования замыканий являются функции высшего порядка. Это такие функции, которые в качестве аргументов принимают другие функции (или замыкания).

```
// передача замыкания в метод filter
let array = [1, 2, 3, 4, 5]
let smallerThanThree = array.filter { $0 < 3 }
// передача замыкания в метод sorted()
[1,5,1,6,12].sorted() { $0 < $1 }
```

Глава 14. 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 вы, вероятно, привыкли заниматься обработкой этих типов с помощью циклов, однако сейчас это не всегда оправданно. Советую вам активно использовать описанные функции.

14.1. 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. 14.1).

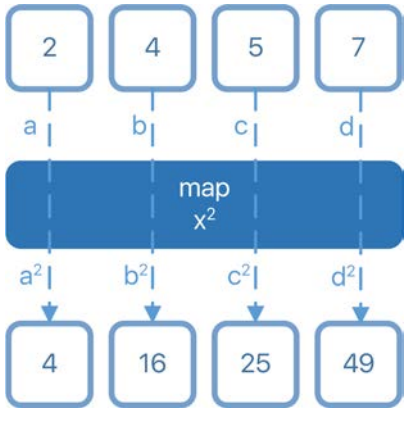

Рис. 14.1. Принцип работы метода map

Рассмотрим пример, описанный в листинге 14.1.

Листинг 14.1

```
let 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.

> **ПРИМЕЧАНИЕ** В данном примере используется сокращенное имя параметра, а именно `$0`. Данная тема была изучена в главе, посвященной замыканиям. 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 параметр того же типа, что и элементы обрабатываемой коллекции, а также один выходной параметр. Если не использовать сокращенный синтаксис, то вызов метода будет выглядеть следующим образом:
>
> ```
> let array = [2, 4, 5, 7]
> var newArray = array.map({
>     (value: Int) -> Int in
>     return value
> })
> ```
>
> Замыкание никак не изменяет входной параметр, просто возвращает его.

Оптимизируем замыкание:

- сократим код перед ключевым словом `in`, так как передаваемое замыкание имеет всего один входной параметр;
- уберем круглые скобки, так как метод `map(_:)` имеет один входной параметр;
- 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 так, чтобы `map(_:)` возводил каждый элемент в квадрат (листинг 14.2).

Листинг 14.2

```
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 ли-

стинге 14.3 количество элементов массивов `intArray` и `boolArray` одинаково, но тип элементов различается (`Int` и `Bool` соответственно).

Листинг 14.3

```
let intArray = [1, 2, 3, 4]
let 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, к примеру, в листинге 14.4 показан пример создания многомерного массива на основе базового.

Листинг 14.4

```
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 14.5 показан пример перевода расстояния, указанного в милях, в километры.

Листинг 14.5

```
let milesToDest = ["Moscow":120.0,"Dubai":50.0,"Paris":70.0]
let kmToDest = milesToDest.map { name,miles in [name:miles * 1.6093] }
kmToDest // [["Dubai": 80.465], ["Paris": 112.651], ["Moscow": 193.116]]
```

14.2. Метод mapValues(_:)

Метод `mapValues(_:)` позволяет обработать значения каждого элемента словаря, при этом ключи элементов останутся в исходном состоянии (листинг 14.6).

Листинг 14.6

```
let mappedCloseStars = ["Proxima Centauri": 4.24, "Alpha Centauri A": 4.37]
let 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.

14.3. Метод flatMap(_:)

Метод `flatMap(_:)` отличается от `map(_:)` тем, что всегда возвращает плоский одномерный массив. Так, пример, приведенный в листинге 14.4, но с использованием `flatMap(_:)`, вернет одномерный массив (листинг 14.7).

Листинг 14.7

```
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 попадающие под некоторое условие значения (листинг 14.8).

Листинг 14.8

```
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]
```

14.4. Метод compactMap(_:)

Метод `compactMap(_:)` позволяет произвести те же действия, что и `map(_:)`, разница лишь в реакции на ситуацию, когда преобразование не может быть произведено. В листинге 14.9 показан пример преобразования массива строковых значений в массив значений типа `Int`.

Листинг 14.9

```
let stringArray = ["1", "2", "3", "four", "5"]
let intFromStringArray = stringArray.map() { Int($0) }
intFromStringArray // [1, 2, 3, nil, 5]
```

Как видно из значения константы `intFromStringArray`, при неудачной попытке преобразования `String` в `Int` в результирующий массив помещается специальное ключевое слово `nil`.

Если воспользоваться методом `compactMap(_:)`, то все неуспешные преобразования будут проигнорированы и исключены из результата (листинг 14.10).

Листинг 14.10

```
let arrayWitoutNil = stringArray.compactMap() { Int($0) }
arrayWitoutNil // [1, 2, 3, 5]
```

14.5. Метод filter(_:)

Метод `filter(_:)` используется, когда требуется отфильтровать элементы коллекции по определенному правилу (рис. 14.2).

В листинге 14.11 показана фильтрация всех целочисленных элементов исходного массива, которые делятся на 2 без остатка, то есть всех четных чисел.

Листинг 14.11

```
let numArray = [1, 4, 10, 15]
let even = numArray.filter{ $0 % 2 == 0 }
even // [4, 10]
```

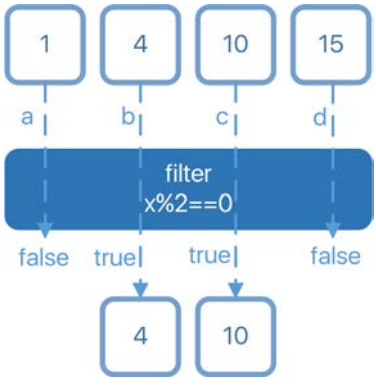

Рис. 14.2. Принцип работы метода filter

Помимо массивов, можно производить фильтрацию других типов коллекций. В листинге 14.12 показана фильтрация элементов словаря `starDistanceDict`.

Листинг 14.12

```
let 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]
```

14.6. 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. 14.3 и листинг 14.13).

Листинг 14.13

```
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`.

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 параметров (листинг 14.14).

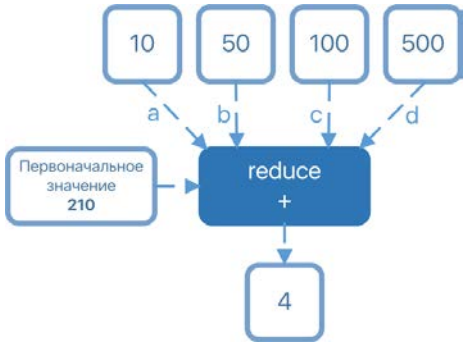

Рис. 14.3. Принцип работы метода reduce

Листинг 14.14

```
let multiTotal = cash.reduce(210, { $0 * $1 })
multiTotal // 5250000000
let totalThree = cash.reduce(210, {a,b in a-b})
totalThree // -450
```

Чтобы было легче запомнить: если в `reduce` в качестве второго аргумента передан оператор, то он просто расставляется между элементами коллекции:

```
cash.reduce(210, +) = 210 + 10 + 50 + 100 + 500
```

14.7. 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 14.15.

Листинг 14.15

```
let collectionOne = [1, 2, 3]
let collectionTwo = [4, 5, 6]
let 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 следующих главах, а со временем даже научитесь создавать собственные. Но настоящая магия начинается тогда, когда вы перестаете бояться таких конструкций и понимаете, что они значат и откуда появляются. Это неминуемо, если вы будете старательно учиться и пытаться делать чуть больше, чем сказано в книге.

14.8. 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 количество сторон неизвестно.

Реализуем функцию с именем `countSidesOfShape` (листинг 14.16).

Листинг 14.16

```
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 14.17).

Листинг 14.17

```
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 14.18).

Листинг 14.18

```
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 выполняет 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 до того, как перейдете к выполнению кода функции.

14.9. Оператор отложенных действий defer

Оператор `defer` откладывает выполнение определенного в его теле кода до момента выхода из области видимости, в которой он был использован (например, после окончания выполнения функции).

В листинге 14.19 показан пример использования `defer` в теле функции.

Листинг 14.19

```
func someFunction() {
    defer {
        print("action in defer")
    }
    defer {
        print("another action in defer")
    }
    print("action in function")
}
someFunction()
```

Консоль

```
action in function
another action in defer
action in defer
```

Как видно из примера, отложенные действия были выполнены после того, как функция завершила свою работу, и что важно, они выполнялись в обратном порядке: сперва блок последнего оператора `defer`, затем предпоследнего и т. д.

Вы можете использовать `defer` для выполнения любых отложенных действий, очистки и удаления использованных ресурсов, закрытия файлов, логирования и т. д.

Глава 15. 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.

15.1. 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 обращении 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.

15.2. 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 15.1.

Листинг 15.1

```
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.

15.3. 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 15.2).

Листинг 15.2

```
let baseCollection = [1,2,3,4,5,6]
let myLazyCollection = baseCollection.lazy
type(of:myLazyCollection) // LazySequence<Array<Int>>.Type
let collection = myLazyCollection.map{$0 + 1}
type(of:collection) // LazyMapSequence<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 константе `collection`.

Вся прелесть такого подхода в том, что вы можете увеличивать цепочки вызовов, но при этом лишнего расхода ресурсов не будет (листинг 15.3).

Листинг 15.3

```
let resultCollection = [1,2,3,4,5,6].lazy.map{$0 + 1}.filter{$0 % 2 == 0}
Array(resultCollection) // [2, 4, 6]
```

Часть IV

ВВЕДЕНИЕ В РАЗРАБОТКУ ПРИЛОЖЕНИЙ

Позади большое количество материала, и нам пора сделать небольшой перерыв. В этой части книги вы создадите свои первые полноценные приложения в Xcode.

Неважно, помните ли вы все, что мы 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 16. Консольное приложение «Сумма двух чисел»
- ✓ Глава 17. Консольная игра «Отгадай число»

Глава 16. 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 пройденный материал, то результат гарантирован! Именно такого подхода мне и хочется придерживаться. Из-за ограничений на размер 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.

16.1. Обзор интерфейса Xcode

Прежде чем переходить к созданию полноценного приложения, рассмотрим некоторые 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. 16.1).

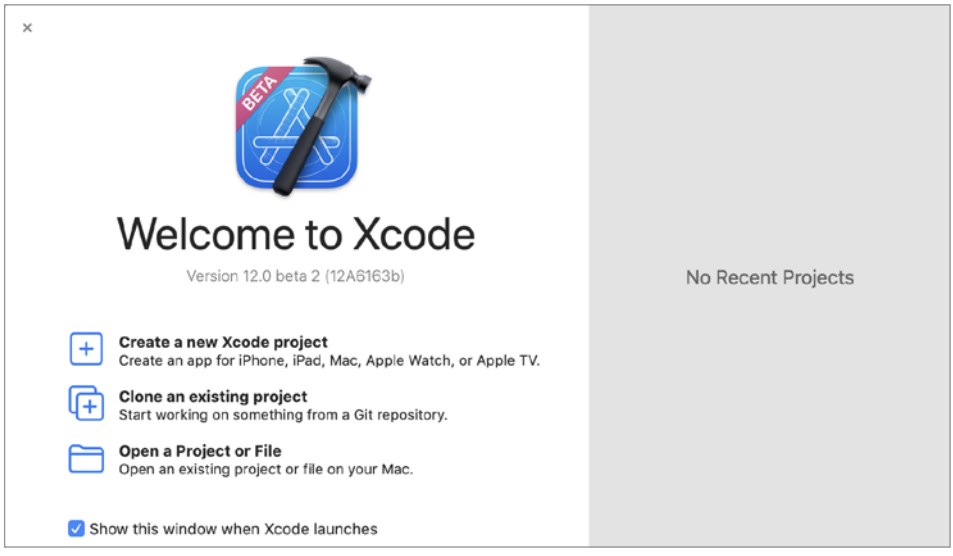

Рис. 16.1. Стартовое окно Xcode

Как говорилось ранее, стартовое окно служит для создания новых проектов и доступа к созданным ранее. До этого мы работали исключительно с Xcode Playground. Теперь вам необходимо начать процесс создания полноценного Xcode-проекта.

- Выберите пункт Create a new Xcode project.

Перед вами появится окно выбора шаблона приложения (рис. 16.2).

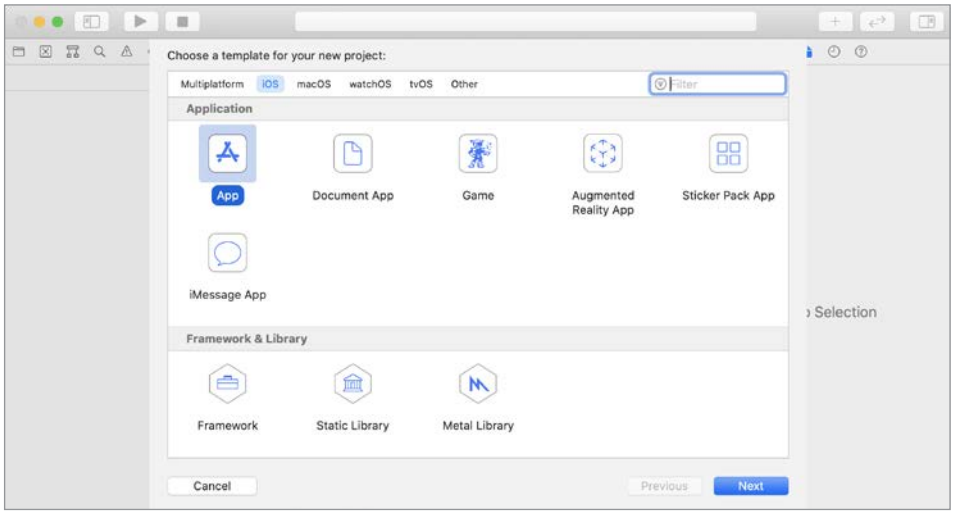

Рис. 16.2. Окно выбора шаблона проекта

В верхней части окна вам доступен выбор платформы, под которой будет функционировать приложение (рис. 16.3).

Рис. 16.3. Выбор платформы

В настоящий момент список состоит из следующих пунктов:

- Multiplatform — разработка кросс-платформенного приложения.
- iOS — разработка приложения под iPhone и iPad.
- macOS — разработка приложения под настольные и мобильные персональные компьютеры iMac и Macbook.
- watchOS — разработка приложения под смарт-часы Apple Watch.
- tvOS — разработка приложения под телевизионную приставку Apple TV.
- Other — разработка других типов приложений.

В этой главе мы создадим приложение для настольной операционной системы macOS.

➤ Выберите пункт macOS.

После выбора операционной 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.

➤ Выберите шаблон Command Line Tool.

Шаблон Command Line Tool как раз и используется при создании приложений, функционирующих в среде командной строки.

➤ Нажмите Next.

В следующем окне потребуется ввести идентификационные данные и указать первичные настройки проекта (рис. 16.4).

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 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 идентифици-

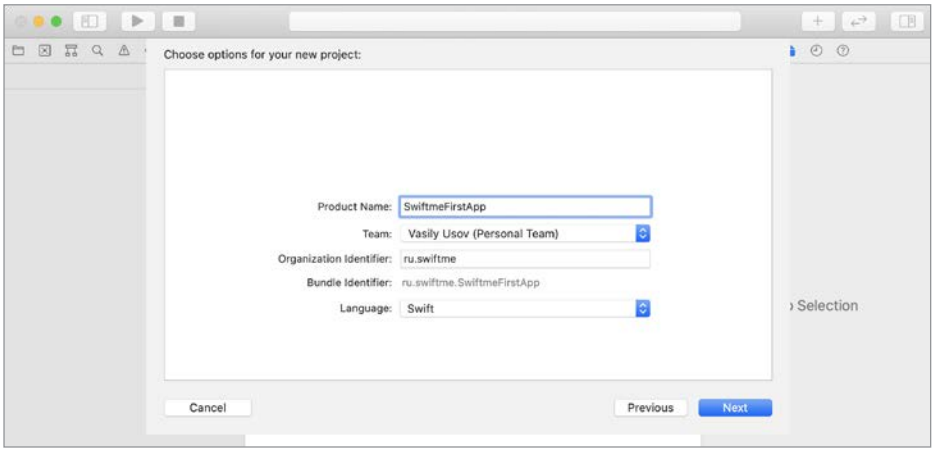

Рис. 16.4. Окно первичной настройки приложений

ровать разработчика. Organization Identifier должен быть уникальным среди всех разработчиков, размещающих свои приложения в AppStore. Введите произвольное значение.

- Bundle Identifier — данная строка генерируется автоматически на основе Organization Identifier и Product Name. Это уникальный идентификатор вашего приложения. Он чувствителен к регистру.
- Language — язык программирования, на котором будет написано ваше приложение. Вам следует выбрать Swift.

➤ Нажмите Next.

➤ В появившемся окне укажите место сохранения проекта.

Интерфейс и элементы управления Xcode-проектом

Перед вами откроется рабочая среда Xcode (рис. 16.5).

Интерфейс Xcode-проекта отличается от уже знакомого вам Xcode Playground. Его рабочая среда предоставляет доступ ко всей функциональности, с помощью которой создаются проекты и осуществляется управление ими. Она автоматически подстраивается под решаемую задачу. На рис. 16.5 изображен один из множества вариантов внешнего вида рабочей среды. Со временем вы узнаете обо всех возможностях этой программы.

Рабочая среда Xcode состоит из пяти основных рабочих областей (рис. 16.6):

- **Toolbar** — панель инструментов.
- **Navigator** — панель навигации, позволяющая производить навигацию по различным элементам проекта.

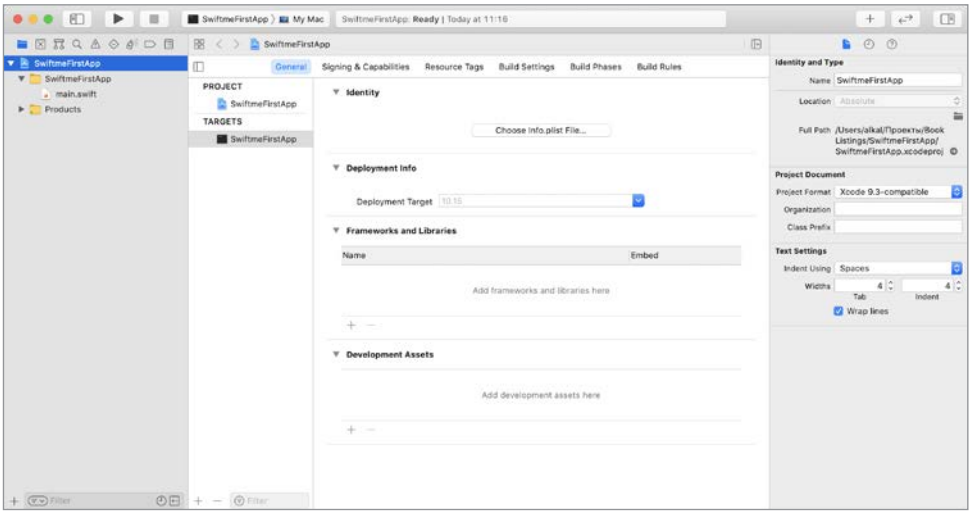

Рис. 16.5. Рабочая среда Xcode

- **Inspectors** — панель настроек, отображающая свойства активного элемента.
- **Project editor** — редактор и основная рабочая площадка проекта.
- **Debug area** — панель отладки (в настоящий момент скрыта).

ПРИМЕЧАНИЕ Значения некоторых полей вашей рабочей среды Xcode и среды, изображенной на рис. 16.6, могут отличаться.

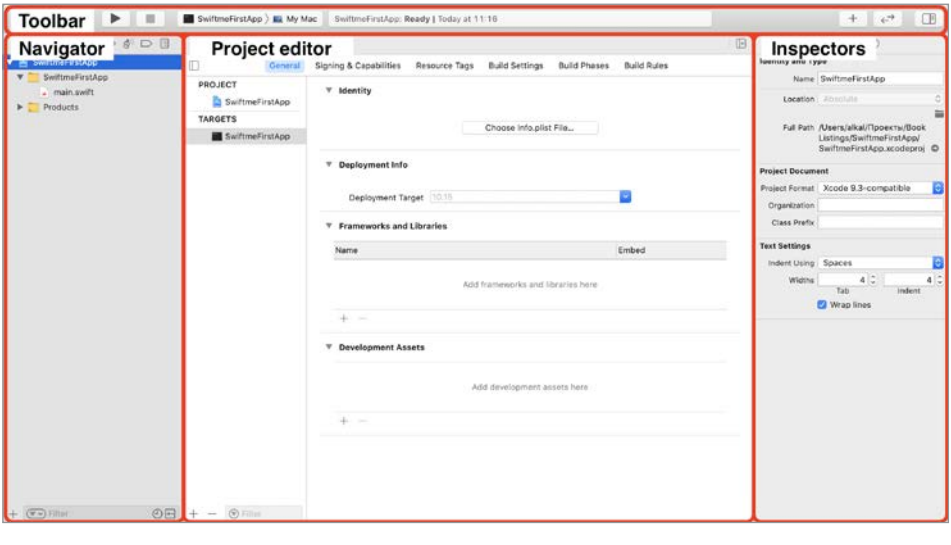

Рис. 16.6. Основные области рабочей среды Xcode

Панель инструментов (Toolbar) находится в верхней части рабочей среды Xcode (рис. 16.7). На ней расположены кнопки управления проектом и рабочей средой.

Рис. 16.7. Кнопки в правой части Toolbar

- Кнопка скрывает и отображает панель навигации (Navigator), с помощью которой можно переходить между файлами, а также между другими категориями вашего проекта.
- С помощью кнопок можно запускать и прерывать сборку проекта.
- Большой прямоугольник посередине панели инструментов называется «Панель статуса» (Statusbar). В нем отображаются все действия и процессы, которые происходят с проектом. Например, когда вы запускаете проект, в данной области отображается каждый шаг сборки приложения, а также возникающие ошибки.
- Кнопка открывает доступ к библиотекам сниппетов, объектов и медиаресурсов. Мы воспользуемся ее возможностями позже.
- Кнопка включает режим Code Review, который превращает рабочую область в своеобразную машину времени и позволяет работать с предыдущими версиями файлов проекта. Эта функция доступна при наличии настроенной системы контроля версий.
- Кнопка позволяет отобразить и скрыть панели Navigator и Inspectors. Ее основным назначением является добавление в проект функциональных и графических элементов, просмотр и модификация различных атрибутов, а также доступ к справочной информации.

Для отображения пока еще скрытой области отладки (Debug Area) вы можете нажать сочетание клавиш CMD+SHIFT+C или выбрать в главном меню пункт View > Debug Area > Activate Console (рис. 16.8).

16.2. Подготовка к разработке приложения

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

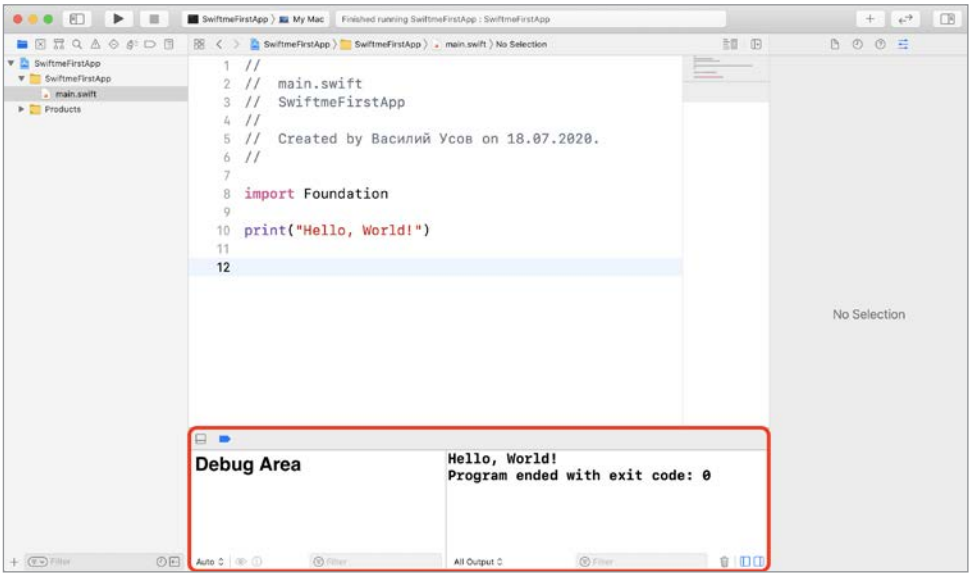

Рис. 16.8. Область отладки Xcode-проекта

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, расположенной в левой части рабочей среды (рис. 16.9).

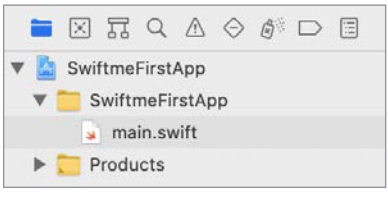

Рис. 16.9. Область навигации

С помощью пиктограмм, расположенных в верхней части области навигации, вы можете переключаться между девятью различными разделами навигации вашего проекта. По умолчанию Navigator отображает Project navigator (навигатор проекта). Неважно, какие ресурсы входят в состав вашего проекта: файлы с исходным кодом, модели данных, списки настроек, раскадровки (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.

ПРИМЕЧАНИЕ У любой программы, написанной на Swift, есть так называемая *точка входа*, то есть файл с исходным кодом, который будет первым загружен и проанализирован компилятором. Это касается как консольных приложений, так и приложений под iOS. В данном случае точкой входа является файл main.swift, и если вы переименуете его, то при запуске проекта получите сообщение об ошибке. О точке входа в iOS-приложение мы поговорим во второй книге.

Верхний элемент представленного дерева в Project navigator имеет название, совпадающее с названием проекта SwiftmeFirstApp. Он определяет ваш проект в целом. Выделив его, вы можете приступить к конфигурированию приложения, или, другими словами, перейти к его настройкам. В следующих главах мы познакомимся с некоторыми из них.

Помимо файла, в состав проекта входят две папки:

- SwiftmeFirstApp — 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 SwiftmeFirstApp (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 скомпилирован.

Теперь щелкните по файлу main.swift. Вы увидите, что Project Editor изменится: в нем отобразится редактор кода. Также изменится и панель Inspectors.

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 Inspectors 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.

16.3. 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 Debug Area.

В данный момент вывод на консоли Xcode пуст, так как приложение еще ни разу не было запущено и не производило каких-либо выводов.

- В Navigator выберите файл main.swift.

Если взглянуть на код, написанный в редакторе (листинг 16.1), то вы увидите директиву `import` и функцию `print(_:)`.

Листинг 16.1

```
import Foundation
print("Hello, World!")
```

Напомню, что функция `print(_:)` предназначена для вывода текстовой информации на консоль.

Чтобы приложение осуществило вывод информации на консоль, оно должно быть запущено. Для управления работой проекта в левой части Toolbar расположены специальные кнопки (рис. 16.10).

Рис. 16.10. Кнопки управления работой приложения

Кнопка с изображением стрелки, называющаяся Build and run, активирует сборку приложения с его последующим запуском.

Кнопка с изображением квадрата 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. 16.11). Спустя несколько секунд на консоли (в области отладки) вы увидите вывод вашего консольного приложения, который осуществила функция `print(_:)` (рис. 16.12).

Рис. 16.11. Statusbar отображает текущее состояние приложения

Теперь, если обратиться к Project navigator, можно увидеть, что в папке Products файл SwiftmeFirstApp 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 Inspectors. Убедитесь, что активирован раздел File Inspector ( ). Атрибут Full Path указывает путь на расположение активного ресурса, в данном случае это скомпилированное консольное приложение (рис. 16.13). 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. 16.14).

Рис. 16.12. Вывод на консоль

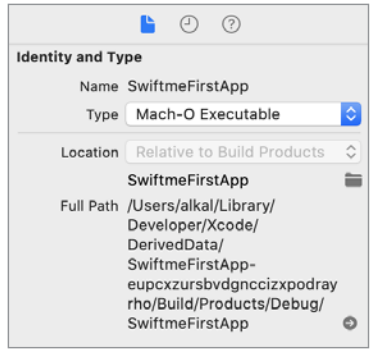

Рис. 16.13. Панель атрибутов файла SwiftmeFirstApp

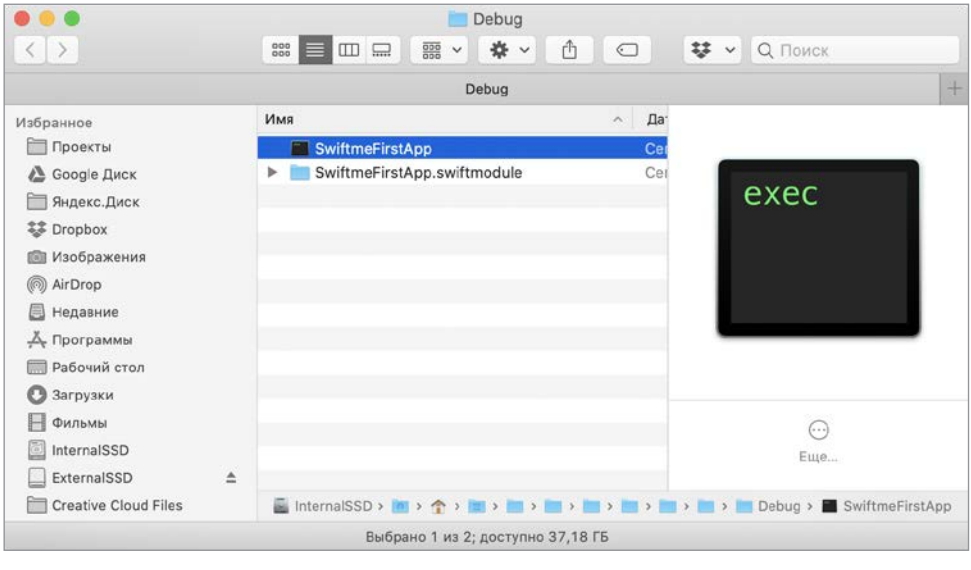

Рис. 16.14. Расположение программы SwiftmeFirstApp

Запустите приложение, дважды щелкнув по нему в Finder, и вы увидите результат его выполнения в Терминале (рис. 16.15).

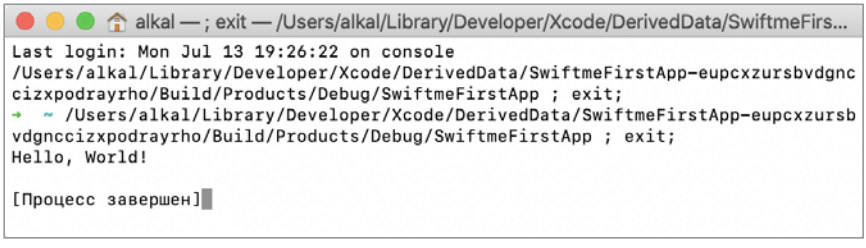

Рис. 16.15. Результат работы программы SwiftmeFirstApp

16.4. 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 данную функциональность. В Project navigator щелкните на файле main.swift и удалите из него все содержимое (комментарии удалять не обязательно).

Напомню, что разрабатываемая программа должна запрашивать у пользователя два значения, 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).

- Добавьте в файл main.swift код из листинга 16.2.

Листинг 16.2

```
print("Введите первое значение")
// получение первого значения
var a = readLine()
print("Введите второе значение")
// получение второго значения
var b = readLine()
```

С помощью данного кода у пользователя будут запрашивать значения параметров `a` и `b`.

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.

Добавим в проект новый файл:

- Правой кнопкой мыши щелкните на папке SwiftmeFirstApp в Navigator.
- В появившемся меню выберите пункт New File.
- В появившемся окне (рис. 16.16) выберите Swift File, нажмите Next и введите имя `func` для создаваемого файла.

После нажатия Create файл появится в составе проекта в Navigator.

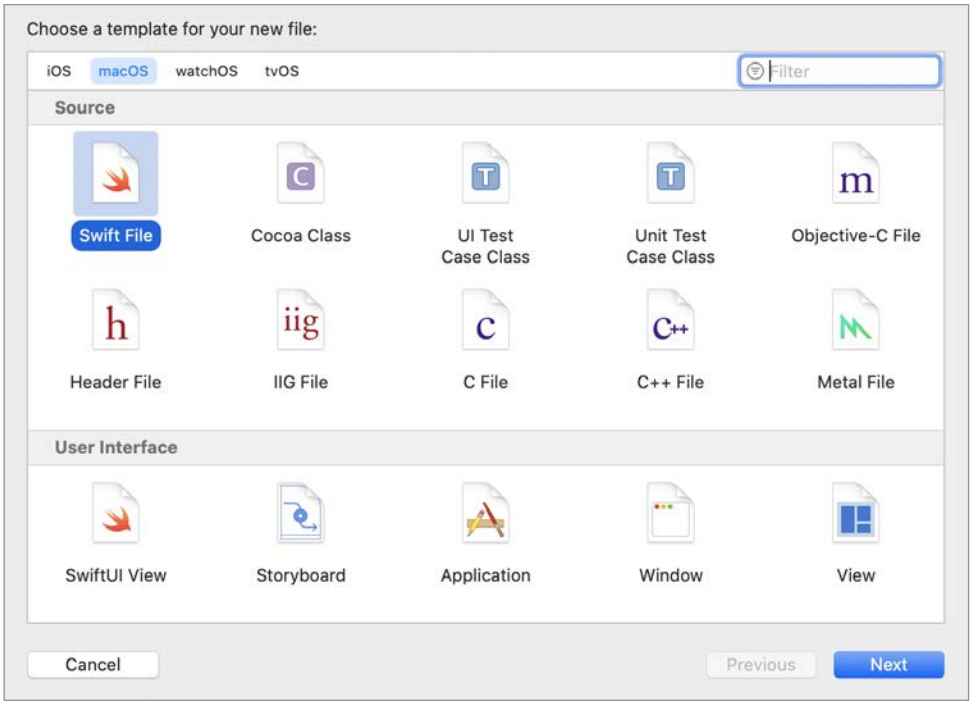

Рис. 16.16. Создание файла func.swift

- В Navigator выберите файл func.swift.

В данном файле мы реализуем функцию `sum(_:_:)`, предназначенную для сложения двух чисел. Входные параметры 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 (рис. 16.17):

- (1) На панели Toolbar нажмите кнопку Library (+), после чего отобразится всплывающее окно библиотеки.
- (2) В появившемся окне нажмите кнопку Show the Snippets Library ({}).

- (3) С помощью поля поиска отфильтруйте сниппеты по слову `func`.
- (4) Среди отобразившихся элементов дважды щелкните на Function Statement.

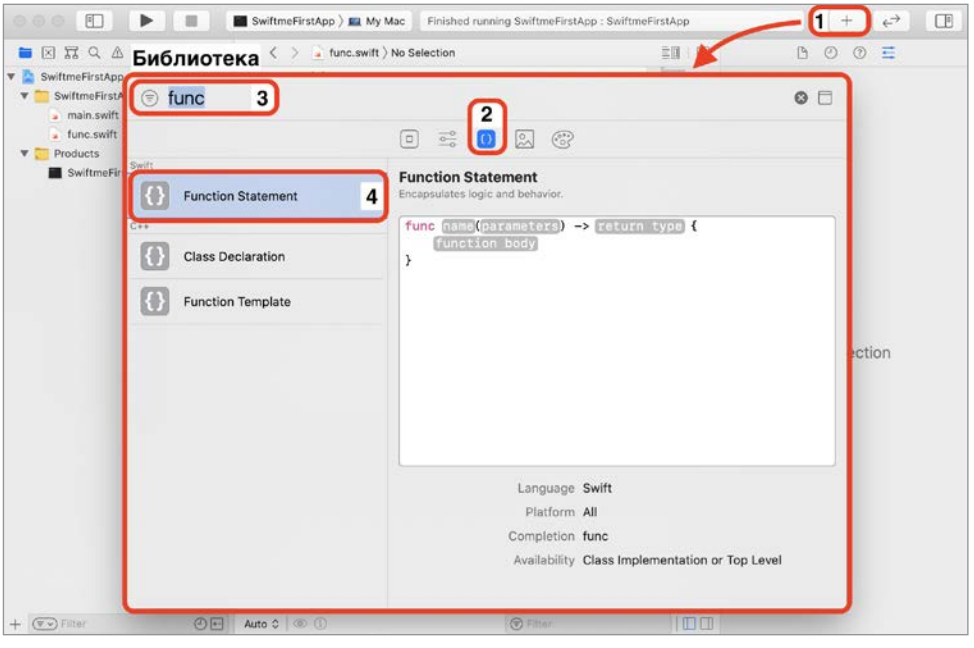

Рис. 16.17. Работа с библиотекой кодовых сниппетов

В результате этих действий в области редактора появится шаблон функции, в котором потребуется заполнить несколько полей (рис. 16.18).

```
import Foundation
func name(parameters) -> return type {
    function body
}
```

Рис. 16.18. Кодовый сниппет функции

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 16.3).

Листинг 16.3

```
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 операнда 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 восклицательный знак.

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 16.4.

Листинг 16.4

```
let result = sum(a, b)
print("Результат сложения - \(result)")
```

Поздравляю, ваша программа готова! Вы можете запустить ее и с помощью консоли в Debug Area попробовать произвести сложение двух чисел (рис. 16.19).

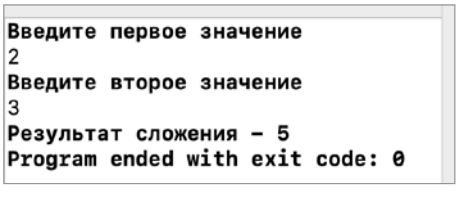

Рис. 16.19. Работа консольного приложения в области отладки

Осталось лишь сохранить программу в виде автономного приложения. Вы уже знаете один способ получения доступа к нему: открыть его в Finder с помощью атрибута Full Path. Но есть еще один способ: в Project navigator в папке Product выбрать файл SwiftmeFirstApp, зажать Option (Alt) и перетащить его в произвольное место на вашем компьютере (например, на рабочий стол или в программы).

Написанная программа не идеальна, она будет вести себя некорректно, например, при вводе букв в качестве запрашиваемых значений, да и вообще то, что наличие значений в опционалах не проверяется, — ужасная практика, но вы исправите это самостоятельно при выполнении домашней работы.

Глава 17. Консольная игра «Угадай число»

Консоль замечательно подходит для создания простых в реализации приложений с минимальной функциональностью. В этой главе мы создадим игру «Угадай 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 — больше, меньше или равно ему. Игровой процесс продемонстрирован на рис. 17.1.

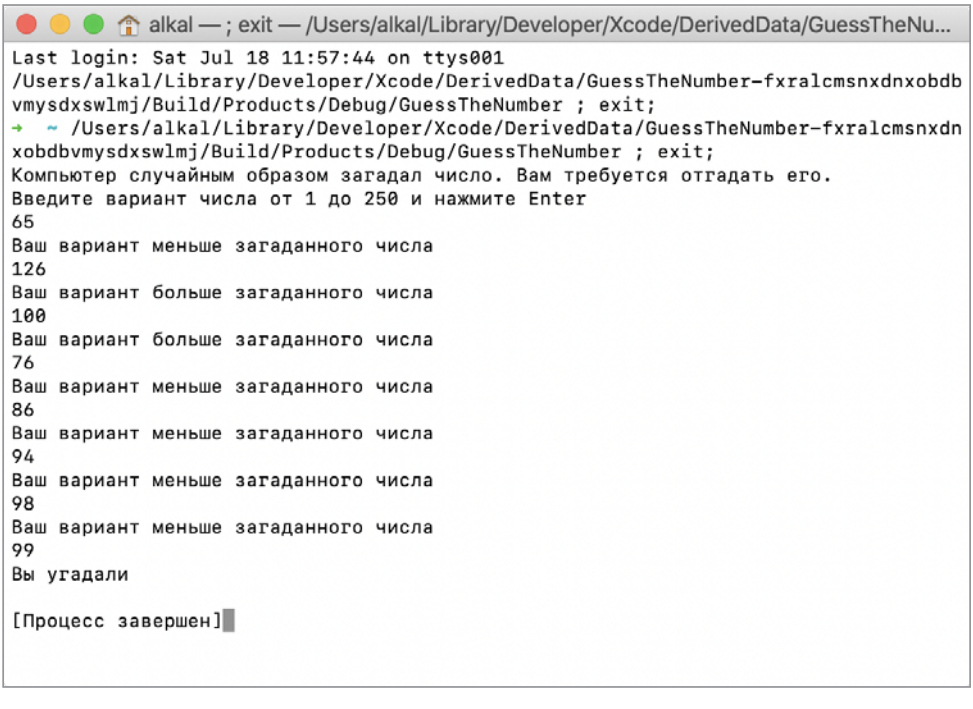

Рис. 17.1. Игра «Угадай число»

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 одинаковы, то работа программы завершается.
- Если числа различны, то происходит переход к шагу 2.

17.1. Код приложения «Угадай число»

Создайте в Xcode новый консольный проект с именем Unknown Number. 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 17.1).

Листинг 17.1

```
// генерация случайного числа
let randomNumber = UInt8.random(in: 1...250)
```

Теперь константа `randomNumber` содержит случайно сгенерированное число.

Несколько слов об оптимизации приложения. В общем случае любая оптимизация — это поиск компромисса, обычно между памятью и процессорным временем устройства. Обратите внимание, что в листинге 17.1 в качестве типа данных константы `randomNumber` используется `UInt8`. Если не определить тип данных самостоятельно, то Swift автоматически определит его как `Int`, а это 64 бита памяти вместо 8 бит. 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. Тем не менее порой вы можете пойти на некоторые траты ресурсов в угоду читабельности кода, но я советую вам думать об оптимизации при написании любого кода.

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 17.2).

ПРИМЕЧАНИЕ Случайные числа, генерируемые большинством языков программирования (в том числе и Swift), в действительности являются псевдослучайными! Для их вычисления используются алгоритмы, где нет непредсказуемых составляющих. 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 представленный способ генерации вполне подходит для программы практически любого уровня сложности.

Листинг 17.2

```
import Foundation
// генерация случайного числа
let randomNumber = UInt8.random(in: 1...250)
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 испытайте себя в борьбе с искусственным интеллектом.

17.2. Устраняем ошибки приложения

Написанная нами программа не совершенна. Создавая ее, мы пошли по пути наименьшего сопротивления и допустили несколько довольно серьезных ошибок. Выделим основные проблемы программы:

- 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 там, где это может привести к аварийному завершению.
- Многократное приведение к типу `UInt8`.

Если вы чувствуете в себе силы самостоятельно исправить эти ошибки — дерзайте. В любом случае ознакомьтесь с решением, которое предлагаю вам я (листинг 17.3).

Листинг 17.3

```
print("Компьютер случайным образом загадал число. Вам требуется отгадать его.")

// Случайное число
let randomNumber = UInt8.random(in: 1...250)
print("Введите вариант числа от 1 до 250 и нажмите Enter")
// флаг-указатель на победу
var isWin = false
repeat {
    // попытка преобразования введенного значения к UInt8
    guard let userNumber = UInt8(readLine() ?? "") else {
        print("Вы ввели некорректное число. Попробуйте снова")
        continue
    }
    // проверка введенного числа
    if userNumber < randomNumber{
        print("Ваш вариант меньше загаданного числа")
    } else if userNumber > randomNumber {
        print("Ваш вариант больше загаданного числа")
    } else {
        print("Вы угадали")
        isWin = true
    }
} while !isWin
```

Часть V

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 позволяет вам создавать и собственные объектные типы. Для этого существует четыре механизма: перечисления (`enum`), структуры (`struct`), классы (`class`) и протоколы (`protocol`). 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.

Глава 18. Введение в объектно-ориентированное и протокол-ориентированное программирование

Парадигма программирования — это совокупность идей, определяющих стиль написания программного кода. Другими словами, парадигма говорит, каким образом необходимо организовать свой программный код и как его организация будет влиять на ход вычислений при решении задач вашим приложением.

Две наиболее популярные парадигмы, используемые при разработке на Swift, это:

- объектно-ориентированное программирование (ООП);
- протокол-ориентированное программирование (ПОП).

Обе парадигмы не являются взаимоисключающими, то есть вы можете применять их одновременно. Более того, они построены на базе одних и тех же понятий (экземпляр, класс, структура, протокол, расширение) и одних и тех же принципов (инкапсуляция, наследование и полиморфизм).

Сейчас вам предстоит познакомиться с новыми для вас объектными возможностями Swift. В дальнейшем эти знания помогут вам правильно понять, что такое ООП и ПОП и как эти возможности языка используются в рамках той или иной парадигмы. Знание и понимание парадигм будет приходить постепенно в ходе изучения материала не только этой книги, но и других книг по теории программирования.

18.1. Экземпляры

В оставшейся части книги большая часть времени работы с кодом будет уходить на создание и изменение таких языковых конструкций, как протоколы, перечисления, структуры и классы.

С понятием протокола вы уже познакомились в самом начале книги. Напомню, что, по сути, это категории объектов, которые предъявляют некоторые требования. Так, если тип данных реализует протокол, то это значит, что он реализует

его требования. После рассмотрения перечислений, структур и классов вы очень подробно изучите и протоколы, а также научитесь создавать собственные.

В ходе изучения материала вы успели поработать не только с протоколами, но и со структурами! Например, тот же целочисленный тип данных `Int` — это структура. Перечисления, структуры и классы называются **объектными типами**, каждый из них имеет уникальные возможности и свою область применения. Именно они делают Swift таким, какой он есть.

Для любого объектного типа может быть создан *экземпляр*, то есть значение данного типа данных.

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 параметра (переменной или константы) для данных определенного типа. Например упомянутый ранее `Int` — это тип данных, созданный с помощью структуры, а переменная `myInteger`, хранящая в себе значение этого типа, — экземпляр данной структуры.

ПРИМЕЧАНИЕ Напомню, что понятия «объект» и «экземпляр» — это, по сути, синонимы.

В главе, посвященной последовательности и коллекции, мы уже говорили об экземплярах, и даже приводили несколько примеров: цифра 5 — это экземпляр типа данных `Int`.

Вы уже умеете создавать и использовать экземпляры фундаментальных типов и в скором времени научитесь использовать перечисления, структуры, классы и протоколы для создания собственных типов данных. При работе с объектными типами вы сможете в полной мере использовать свое воображение.

Рассмотрим пример.

Представьте, что вы разрабатываете программу, которая оперирует таким понятием, как «здание». Зданий в вашей программе очень много: большие и маленькие, одноэтажные и многоэтажные, с гаражом и с крыльцом и т. д. Чтобы управлять всем этим зоопарком, с помощью структуры (`struct`) вы создаете тип данных `Building`. Данный тип не является каким-то конкретными домом. Это шаблон, на основе которого может быть создан конкретный дом (как `Int` — шаблон для целого числа). Вы можете создать переменную (или константу) типа `Building` и проводить с ее помощью необходимые операции.

```
var cottage: Building = Building()
```

Сама структура `Building` не имеет конкретных черт, так как еще неизвестно, какой определенный объект реального (или нереального) мира он будет определять. Так, мы не знаем, сколько в здании будет этажей, сколько комнат, какой у него будет адрес и т. д. Характеристики объектных типов называются *свойствами*, с ними мы встречались неоднократно. Для переменной `cottage` значения этих свойств могли бы быть следующими:

```
cottage.floorsNumber = 2
cottage.roomsNumber = 6
cottage.buildingArea = 210
```

Свойства представляют собой хранилища данных, то есть это те же самые переменные и константы, но с особым порядком, доступом и областью видимости: они доступны только через доступ к экземпляру. Иначе говоря, сперва вы получаете доступ к экземпляру, а уже потом (через точку) — к его свойству.

Помимо свойств, у экземпляра могут быть определены методы. Методы, с ними мы также неоднократно встречались, — это функции, которые определены внутри объектных типов. Класс `Building` мог бы иметь метод «открыть дверь»:

```
// откроем дверь, передав ее идентификатор (у здания может быть много дверей)
cottage.openDoorWith(identifier:)
```

Таким образом, создавая экземпляр, мы можем наполнять его свойства информацией и использовать его методы. И свойства, и методы определяются типом данных.

18.2. 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 (что дать ему на вход и что можно получить на выходе). В состав модулей входят исходные файлы.

Исходный файл — отдельный файл, содержащий программный код.

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 уже встречались: вспомните про `Foundation` (его полное название — `Cocoa's Foundation Framework`), который содержит большое количество функциональных механизмов, позволяющих расширить возможности ваших программ.

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.

18.3. Пространства имен

Пространство имен (namespace) — это именованный фрагмент программ. Оно имеет одно очень важное свойство — скрывает свою внутреннюю реализацию и не позволяет получить доступ к объектам внутри себя 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.

Понятие пространства имен очень близко рассмотренному ранее понятию **области видимости**. Простейшим примером ограниченной области видимости может служить функция. Все переменные, объявленные в пределах функции, вне ее недоступны. Но при этом функция не является пространством имен, так как не позволяет получить доступ к объектам внутри себя извне.

Пространства создаются при использовании *перечислений, структур* и *классов*. Именно их изучением мы и займемся в этой главе. Также к пространствам имен относятся модули. Одни пространства имен могут включать другие: так, модуль `UIKit`, ориентированный на разработку iOS-приложений, в своем коде выполняет импорт модуля `Cocoa's Foundation Framework`.

18.4. API Design Guidelines

Одной из главных проблем предыдущих версий Swift была нестандартизированность и неоднозначность написания имен функциональных элементов. Каждый разработчик сам определял, как 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 бросить, 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 API Design Guideline — 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 API Design Guidelines язык Swift был очень изменчив. Ниже приведены некоторые наиболее важные правила.

- 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`).

В процессе создания собственных типов мы будем придерживаться рекомендаций API Design Guidelines.

Глава 19. Перечисления

Перейдем к изучению механизмов создания объектных типов данных, и в первую очередь рассмотрим простейший из них — перечисления.

19.1. Синтаксис перечислений

Перечисление (enum) — 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 19.1). 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 19.1

```
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 19.2). Массив содержит все возможные значения, которые доступны в программе. При необходимости происходит получение требуемого элемента массива.

Листинг 19.2

```
let currencyUnit: [String] = ["Rouble", "Euro"]
let euroCurrency = currencyUnit[1]
```

В действительности это довольно хороший способ ведения списков возможных значений. И его положительные свойства заканчиваются ровно там, где 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 в Swift не предусмотрели его. Значительно улучшить ситуацию позволяет использование перечислений вместо массивов или фундаментальных типов данных.

Перечисление — это тип данных, содержащий множество альтернативных значений, каждое из которых может быть проинициализировано некоторому параметру.

СИНТАКСИС

```
enum ИмяПеречисления {
    case значение1
    case значение2
    // другие значения
}
```

`Значение` — значение очередного члена перечисления.

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.

Попробуем использовать перечисления для работы с типами денежных единиц (листинг 19.3). Перечисление, подобно массиву из предыдущего примера, содержит список значений, одно из которых может быть проинициализировано параметру.

Листинг 19.3

```
enum CurrencyUnit {
    case rouble
    case euro
}
```

Несколько членов перечисления можно писать в одну строку через запятую (листинг 19.4).

Листинг 19.4

```
enum CurrencyUnit {
    case rouble, euro
}
```

После создания нового перечисления одно из его значений (членов перечисления) может быть присвоено параметру (переменной или константе), для этого используется специальный синтаксис.

СИНТАКСИС

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 параметра.

```
let имяПараметра: ИмяПеречисления =.значение
```

- Полный синтаксис. При этом тип определяется неявно

```
let имяПараметра = ИмяПеречисления.значение
```

В дальнейшем для изменения значения переменной, указывающей на член перечисления, можно использовать сокращенный синтаксис, так как тип параметра уже определен.

```
имяПараметра = .значение
```

Имя перечисления выступает в качестве типа данных параметра. Далее доступ к значениям происходит уже без указания его имени.

В листинге 19.5 показаны примеры создания параметров и инициализации им членов перечисления `CurrencyUnit`.

Листинг 19.5

```
var roubleCurrency: CurrencyUnit = .rouble
var otherCurrency = CurrencyUnit.euro
// поменяем значение одного из параметров на другой член перечисления
otherCurrency = .rouble
```

В результате создаются две переменные 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 константах `currency1` и `currency2` из следующего примера не эквивалентны:
>
> ```
> let currency1 = CurrencyUnit.rouble
> let currency2 = "rouble"
> ```

19.2. 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 указанием имен и типов, разделенных двоеточием. Набор ассоциированных параметров может быть уникальным для каждого отдельного члена.

Создадим новое усовершенствованное перечисление `AdvancedCurrencyUnit`, основанное на `CurrencyUnit`, но имеющее ассоциированные параметры, с помощью которых можно указать список стран, где данная валюта используется, а также краткое наименование валюты (листинг 19.6).

Листинг 19.6

```
enum AdvancedCurrencyUnit {
    case rouble(countries: [String], shortName: String)
    case euro(countries: [String], shortName: String)
}
```

> **ПРИМЕЧАНИЕ** При описании ассоциированных параметров не обязательно указывать их имена, можно обойтись лишь типами данных.
>
> ```
> enum AdvancedCurrencyUnit {
>     case rouble([String], String)
>     case euro([String], String)
> }
> ```

Оба члена перечисления содержат одинаковые ассоциированные параметры. Параметр `countries` является массивом, так как валюта может использоваться не в одной, а в нескольких странах: например, евро используется на территории Европейского союза.

Теперь, чтобы создать переменную или константу типа `AdvancedCurrencyUnit`, необходимо указать значения для всех ассоциированных параметров (листинг 19.7).

Листинг 19.7

```
let euroCurrency: AdvancedCurrencyUnit =  .euro(countries: ["German", "France"],
shortName: "EUR")
```

Теперь в константе `euroCurrency` хранится член `euro` со значениями двух ассоциированных параметров.

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 19.8).

Листинг 19.8

```
// 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`.

19.3. 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 19.9).

Листинг 19.9

```
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 19.10).

Листинг 19.10

```
let 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.

19.4. Оператор switch для перечислений

Для анализа и разбора значений перечислений можно использовать оператор `switch`.

Рассмотрим пример из листинга 19.11, в котором анализируется значение переменной типа `AdvancedCurrencyUnit`.

Листинг 19.11

```
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`).

19.5. 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» намного лучше отражает их предназначение.

Указание связанных значений

Для задания связанных значений членов перечисления необходимо указать тип данных самого перечисления, соответствующий значениям членов, и определить значения для каждого отдельного члена перечисления (листинг 19.12).

Листинг 19.12

```
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 различны 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). Конечно же, можно указать эти значения самостоятельно. Например, в листинге 19.13 представлено перечисление, содержащее список планет Солнечной системы в порядке удаленности от Солнца.

Листинг 19.13

```
enum Planet: Int {
      case mercury = 1, venus, earth, mars, jupiter, saturn, uranus, neptune,
pluto = 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 `pluto` 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 19.14).

Листинг 19.14

```
let iAmHappy = Smile.joy
iAmHappy.rawValue // ":)"
```

В результате использования свойства rawValue мы получаем исходное значение члена joy типа String.

19.6. 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 19.15 показан пример использования инициализатора перечисления.

Листинг 19.15

```
let 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, вернется 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.

19.7. Свойства в перечислениях

Свойство в перечислении — это хранилище, аналогичное переменной или константе, объявленное в пределах перечисления, доступ к которому осуществляется через экземпляр перечисления. В Swift существует определенное ограничение для свойств в перечислениях: в качестве их значений не могут выступать фиксированные значения-литералы, только 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 19.16). За основу возьмем перечисление `Smile` и создадим вычисляемое перечисление, которое возвращает связанное с текущим членом перечисления значение.

Листинг 19.16

```
enum Smile: String {
    case joy = ":)"
    case laugh = ":D"
    case sorrow = ":("
    case surprise = "o_O"
    // вычисляемое свойство
    var description: String { return self.rawValue }
}
let mySmile: Smile = .sorrow
mySmile.description // ":("
```

Вычисляемое свойство должно быть объявлено как переменная (`var`). В противном случае (если используете оператор `let`) вы получите сообщение об ошибке.

С помощью оператора `self` внутри замыкания вы получаете доступ к текущему члену перечисления, при этом *его использование не является обязательным*. Тем не менее данный оператор будет активно использоваться вами при разработке приложений. С ним мы познакомимся подробнее уже в ближайших разделах.

19.8. 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 19.17).

Листинг 19.17

```
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.

19.9. 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 текущий 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 19.18).

Листинг 19.18

```
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 {
        // использование self перед rawValue не является обязательным
        return 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, при этом использование `self` не является обязательным. Swift из контекста понимает, к какому именно параметру вы пытаетесь обратиться.

При необходимости `self` позволяет выполнить анализ перечисления внутри самого перечисления с помощью конструкции `switch self {}`, где значениями являются члены перечисления.

Оператор `self` используется внутри перечисления в двух случаях:

- для доступа к текущему члену перечисления;
- для доступа к свойствам и методам перечисления.

При этом, начиная со Swift 5.3, во втором случае использование `self` не является обязательным и данный оператор может быть опущен. Мы видели это ранее в методе `descriptionRawValue`. Это правило актуально не только для перечислений, но и для структур и классов, с которыми мы познакомимся в следующих главах.

19.10. 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 функциональностью 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 19.19). Для простоты оставим только две операции: сложение и вычитание.

Листинг 19.19

```
enum ArithmeticExpression {
    // операция сложения
    case addition(Int, Int)
    // операция вычитания
    case substraction(Int, Int)
}
let 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 19.20).

Листинг 19.20

```
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., 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 19.21. В данном примере вычисляется значение выражения `20 + 10 – 34`.

Листинг 19.21

```
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 evaluate( valueLeft ) + evaluate( valueRight )
            case .subtraction( let valueLeft, let valueRight ):
                return evaluate( valueLeft ) - evaluate( valueRight )
        }
    }
}

let 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` принимает на вход 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:

```
hardExpr.evaluate(hardExpr)
```

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.

> **ПРИМЕЧАНИЕ** Обратите внимание, что обращение к методу `evaluate` внутри самого метода в приведенном примере происходит без использования ключевого слова `self`. Тем не менее при желании вы всегда можете использовать его.

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.

19.11. Где использовать перечисления

Описываемый механизм	Где используется
Перечисление	Используется, когда необходимо сгруппировать множество альтернативных взаимосвязанных значений. Перечисления позволяют значительно повысить безопасность вашего кода. **Пример:** Выбор между мужским и женским полом. `enum Gender { case male case female }` Указатель на направление движения. `enum Movement { case left case right case up case down }` Описание транзакции на бирже. `enum TradeOperation { case buy(name: String, amount: Decimal) case sell(name: String, amount: Decimal) }` Физические константы. `enum Constant: Double { case π = 3.14159 case e = 2.71828 case λ = 1.30357 }`

Глава 20. Структуры

Перечисления (`enum`) — это ваш входной билет в объектно-ориентированное и протокол-ориентированное программирование. Теперь пришло время познакомиться с еще более функциональными и интересными конструкциями — структурами (`struct`).

Вы уже давно используете структуры при написании кода, так как все фундаментальные типы данных — это структуры. Они в некоторой степени похожи на перечисления и во многом сходны с классами (с ними мы познакомимся в следующей главе).

20.1. 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 20.1).

Листинг 20.1

```
var playerVasiliy = (name: "Василий", victories: 10)
```

Такой подход, конечно же, решает поставленную задачу, но если количество характеристик будет увеличиваться, то кортеж станет чересчур сложным.

Исходя из этого, нам требуется механизм, позволяющий гибко описывать даже самые сложные сущности, и который учитывает все возможные параметры. Структуры как раз и являются таким механизмом. Они позволяют создать «скелет», или, иными словами, «шаблон» 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.

ПРИМЕЧАНИЕ Объявляя структуру, вы определяете новый тип данных.

Объявим структуру, которая будет описывать сущность «игрок в шахматы» (листинг 20.2).

Листинг 20.2

```
struct ChessPlayer {}
```

Теперь у вас появился новый тип данных `ChessPlayer`. Пока еще он не имеет совершенно никакой практической ценности, у него нет свойств и методов. Тем не менее уже можно объявить новый параметр данного типа и проинициализировать ему значение (листинг 20.3).

Листинг 20.3

```
let playerOleg = ChessPlayer()
type(of: playerOleg) // ChessPlayer.Type
```

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 определена структура, то есть то, к какому пространству имен она относится. Вы можете не обращать на префикс никакого внимания, это не влияет на процесс разработки.

20.2. Свойства в структурах

Объявление свойств

Созданная структура `ChessPlayer` пуста — она не описывает ни одной характеристики игрока. Для повышения информативности и ценности структуры в нее могут быть добавлены свойства (подобно тем, что встречались нам при изучении перечислений).

СИНТАКСИС

```
struct ИмяСтруктуры {
    var свойство1: ТипДанных
    let свойство2: ТипДанных
    // остальные свойства и методы
}
```

`свойство: Any` — свойство структуры, может быть произвольного типа данных

Свойство может быть представлено как в виде переменной, так и в виде константы. Количество свойств в структуре не ограничено.

Добавим в структуру `ChessPlayer` два свойства, описывающие имя и количество побед (`name` и `victories`) (листинг 20.4).

Листинг 20.4

```
struct ChessPlayer {
    var name: String
    var victories: 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 20.5).

Листинг 20.5

```
let playerHarry = ChessPlayer.init(name: "Гарри Поттер", victories: 32)
```

В результате будет создан новый экземпляр структуры `ChessPlayer`, 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.

Имя инициализатора (`init`) может быть опущено (листинг 20.6). Точно такой же подход вы видели при создании значений любых фундаментальных типов.

Листинг 20.6

```
var harry = ChessPlayer(name: "Гарри", victories: 32)
```

Значения свойств по умолчанию

Для свойств могут быть заданы значения по умолчанию (листинг 20.7). При этом такие свойства могут быть опущены при создании нового значения данного типа (Swift автоматически создает новый инициализатор).

Листинг 20.7

```
struct ChessPlayer {
    var name: String
    var victories: UInt = 0
}
```

На рис. 20.1 изображены два разных инициализатора, доступных при создании экземпляра типа `ChessPlayer`:

- первый требует указать значение только для свойства `name`, так как для него не задано значение по умолчанию;

- второй требует указать оба значения.

Рис. 20.1. Два инициализатора в окне автодополнения

> **ПРИМЕЧАНИЕ** Если инициализатор вообще не требует передачи каких-либо значений, то он называется *пустым*. Он доступен, когда структура не имеет свойств или для всех свойств заданы значения по умолчанию.

Значения по умолчанию указываются при объявлении свойств точно так же, как вы указываете значение любой переменной или константы. При этом вы можете указывать или не указывать значения по умолчанию для каждого свойства в отдельности.

В листинге 20.8 показан пример создания параметров типа `ChessPlayer` с передачей значения свойств и без нее (через пустой инициализатор).

Листинг 20.8

```
struct ChessPlayer {
    var name: String = "Игрок"
    var victories: UInt = 0
}
// передаем значения для обоих свойств
var playerJohn = ChessPlayer(name: "Джон", victories: 32)
// используем значения по умолчанию с помощью пустого инициализатора
var unknownPlayer = ChessPlayer()
```

20.3. 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 `ChessPlayer` с двумя свойствами, для доступа к которым необходимо указать имя параметра и через точку (`.`) имя свойства (листинг 20.9). Таким образом, вы сперва получаете доступ к структуре (пространству имен), а затем к самому свойству.

Листинг 20.9

```
playerJohn.name    // "Джон"
unknownPlayer.name // "Игрок"
```

Данный способ доступа обеспечивает не только чтение, но и изменение значения свойства экземпляра структуры (листинг 20.10).

Листинг 20.10

```
playerJohn.victories // 32
playerJohn.victories += 1
playerJohn.victories // 33
```

ПРИМЕЧАНИЕ Если свойство структуры или ее экземпляр указаны как константа (`let`), при попытке изменения значения свойства Xcode сообщит об ошибке.

20.4. Собственные инициализаторы

Как говорилось ранее, инициализатор — это специальный метод, который носит имя `init`. Если вас не устраивают инициализаторы, которые создаются для структур автоматически, вы можете определить собственные.

ПРИМЕЧАНИЕ Автоматически созданные встроенные инициализаторы удаляются при объявлении первого собственного инициализатора.

Вам необходимо придерживаться требования: «после того, как будет создан экземпляр структуры, все ее свойства должны иметь значения». 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 — это 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`. Это связано с тем, что аргумент функции имеет то же самое имя (`name`), что и свойства структуры. В случае, если бы аргумент и свойства имели различные имена, то использование ключевого `self` не являлось бы обязательным.

Создадим инициализатор для структуры `ChessPlayer`, который принимает значение только для свойства `name` (листинг 20.11).

Листинг 20.11

```
struct ChessPlayer {
    var name: String = "Игрок"
    var victories: UInt = 0
    // инициализатор
```

```
    init(name: String){
        self.name = name
    }
}
var playerHelgaPotaki = ChessPlayer(name: "Ольга")
playerHelgaPotaki.victories // 0
// следующий код вызовет ошибку
// структура больше не имеет встроенных инициализаторов
var newPlayer = ChessPlayer()
```

Инициализатор принимает значение только для свойства `name`, при этом свойству `victories` 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 (Value type). 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 = ChessPlayer(name: "Олег")
var olegLapin = olegMuhin
```

20.5. 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.

Реализуем метод `description()`, который выводит справочную информацию об игроке в шахматы на консоль (листинг 20.12).

Листинг 20.12

```
struct ChessPlayer {
    var name: String = "Игрок"
    var victories: UInt = 0
    init(name: String) {
        self.name = name
    }
    func description() {
        print("Игрок \(name) имеет \(victories) побед")
    }
}
```

```
var andrey = ChessPlayer(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 метода необходимо указать модификатор `mutating`.

Создадим метод `addVictories(count:)`, который будет изменять значение свойства `victories` (листинг 20.13).

Листинг 20.13

```
struct ChessPlayer {
    var name: String = "Игрок"
    var victories: UInt = 0
    init(name: String) {
        self.name = name
    }
    func description() {
        print("Игрок \(name) имеет \(victories) побед")
    }
    mutating func addVictories( count: UInt = 1 ) {
        victories += count
    }
}
var harold = ChessPlayer(name: "Гарольд")
harold.victories // 0
harold.addVictories ()
harold.victories // 1
harold.addVictories(count: 3)
harold.victories // 4
```

ПРИМЕЧАНИЕ Структура может изменять значения свойств только в том случае, если экземпляр структуры хранится в переменной.

Глава 21. Классы

Классы, наравне со структурами, являются очень функциональными конструкциями, которые используются при разработке приложений. В этой главе мы рассмотрим их основные возможности и отличия от других объектных типов.

Классы довольно сильно похожи на структуры, они точно так же позволяют создать программный образ любой сущности, наделить ее свойствами и методами. Но у этих конструктивных элементов есть несколько ключевых особенностей.

Тип. Класс — это ссылочный тип (reference type). Экземпляры класса передаются по ссылке, а не копированием.

Изменяемость. Экземпляр класса может изменять значения своих свойств, объявленных как переменная (`var`), даже если сам экземпляр хранится в константе (`let`). При этом использовать ключевое слово `mutating` для методов не требуется.

Наследование. Два класса могут быть в отношении «родительский — дочерний» друг к другу. При этом подкласс наследует и включает в себя все характеристики (свойства и методы) суперкласса и при необходимости может быть дополнительно расширен. Наследование подробно рассматривается в главе 24.

Инициализатор (конструктор). Класс имеет только пустой встроенный инициализатор `init()`, который не требует передачи значения входных параметров для их инициализации свойствам.

Деинициализатор (деструктор). 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.

Наверняка у вас возник вопрос о том, зачем же Swift предоставляет нам столь похожие элементы, как структуры и классы. У каждого из них на самом деле своя область применения, со временем вы научитесь определять наиболее подходящие конструкции для решения возникающих задач.

21.1. Синтаксис классов

Объявление классов очень похоже на объявление структур, но вместо слова `struct` используется `class`.

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 в дальнейшем.

21.2. Свойства 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 инициализации значений свойств требуется разрабатывать инциализаторы 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 `Chessman`, а потом с помощью структуры `Chessman`. Использование класса для моделирования 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 ссылочный тип) 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 составе стандарта Unicode присутствуют необходимые символы).

Координаты будут служить не только для определения местоположения фигуры на шахматной доске, но и для определения факта ее присутствия. Если фигура взята 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 21.1 приведен код класса, описывающий сущность «шахматная фигура».

Листинг 21.1

```
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
        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.

ПРИМЕЧАНИЕ Обратите внимание, что для доступа к свойству `figureSymbol` мы не используем `self`, но при желании вы можете его добавить. Использование `self` для доступа к свойствам `type` и `color` обусловлено тем, что входные параметры инициализатора имеют те же самые имена, что и свойства класса.

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 соответствуют `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 21.2).

Листинг 21.2

```
// 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 21.3).

Листинг 21.3

```
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
        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.

21.3. 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 не может быть использована непосредственно для игры. Это связано с тем, что механизмы, позволяющие установить фигуру на игровое поле и динамически ее перемещать, еще не разработаны.

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, решающих следующие задачи (листинг 21.4):

- установка координат фигуры (при выставлении на поле или при движении);
- снятие фигуры с игрового поля (окончание партии или взятие фигуры).

Листинг 21.4

```
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
        figureSymbol = figure
    }

    // установка координат
    func setCoordinates(char c:String, num n: Int) {
        coordinates = (c, n)
    }

    // взятие фигуры
    func kill() {
        coordinates = nil
    }
}
let 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 к этой задаче мы скоро приступим.

21.4. 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 21.5).

Листинг 21.5

```
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
        figureSymbol = figure
    }

    init(type: ChessmanType, color: ChessmanColor, figure: Character,
                coordinates: (String, Int)) {
        self.type = type
        self.color = color
        figureSymbol = figure
        setCoordinates(char: coordinates.0, num: coordinates.1)
    }

    func setCoordinates(char c:String, num n: Int){
        coordinates = (c, n)
    }

    func kill() {
        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`.

21.5. 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 21.6).

Листинг 21.6

```
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
        figureSymbol = figure
    }

    init(type: ChessmanType, color: ChessmanColor, figure:
        Character, coordinates: (String, Int)) {
        self.type = type
        self.color = color
        figureSymbol = figure
        setCoordinates(char: coordinates.0, num:coordinates.1)
    }

    func setCoordinates(char c:String, num n: Int) {
        coordinates = (c, n)
    }

    func kill() {
        coordinates = nil
    }
}
```

Структуры `ChessmanColor` и `ChessmanType` теперь являются вложенными в класс `Chessman` и существуют только в пределах области видимости данного класса. Мы с вами уже неоднократно встречались с подобным подходом.

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. В следующем примере (листинг 21.7) мы получаем доступ к одному из членов перечисления `ChessmanType`, объявленного в контексте класса `Chessman`.

Листинг 21.7

```
let linkToEnumValue = Chessman.ChessmanType.bishop
```

Глава 22. Свойства

Свойства — неотъемлемая часть объектных типов. В этой главе мы более подробно разберем их возможности.

22.1. Типы свойств

Свойства — это параметры, объявленные в пределах объектного типа данных. Они позволяют хранить или вычислять значения.

По типу 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 в инициализаторе (метод с именем `init`);
- получить значение по умолчанию в случае, если при создании экземпляра ему не передается никакое значение;
- изменить значение в процессе использования экземпляра.

Мы уже многократно создавали хранимые свойства ранее, например, при реализации класса `Chessman`.

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. В нем будут 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 22.1).

Листинг 22.1

```
class Person {
    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 firstName + " " + secondName
    }
}
var me = Person(name:"Егор", secondName:"Петров")
me.wholeName
```

Экземпляр класса `Person` 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 не потребуется значение какого-либо свойства.

> **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 22.2.

Листинг 22.2

```
me.wholeName // "Егор Петров"
me.secondName = "Иванов"
me.wholeName // "Егор Петров"
```

Несмотря на то что значение свойства `secondName` было изменено, значение ленивого свойства `wholeName` осталось прежним.

Методы типов данных в некоторой степени тоже являются ленивыми: они вычисляют значение при обращении к ним, но делают это каждый раз. Если внимательно посмотреть на структуру класса `Person`, 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 22.3).

Листинг 22.3

```
class Person {
    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 = Person(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 организовать с использованием скобок.

При этом свойство wholeName, хранящее в себе замыкание, будет возвращать актуальное значение каждый раз при обращении к нему. То есть данное ленивое хранимое свойство называется lazy-by-name.

Почему необходимо использовать lazy для свойства wholeName? Обратите внимание, что при получении значения свойства wholeName происходит обращение к элементам этого же объектного типа (с помощью ключевого слова self). Такой подход доступен только для ленивых хранимых свойств: если убрать lazy, то Xcode сообщит об ошибке:

```
error: Cannot find 'self' in scope
```

В данном случае использование self является обязательным, так как происходит захват ссылки на объект (при необходимости об этом вам сообщит сам Xcode). Более подробно о захвате ссылок на объекты мы поговорим в главе, посвященной управлению памятью в Swift.

Вычисляемые свойства

Также существует иной способ создать параметр, значение которого будет вычисляться при каждом доступе к нему. Для этого можно использовать уже знакомые нам по перечислениям 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 при каждом обращении к ним.

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 `Person` вычисляемым (листинг 22.4).

Листинг 22.4

```
class Person {
    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 = Person(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.

22.2. 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, необходимо использовать специальный синтаксис.

СИНТАКСИС

```
var имяСвойства: ТипДанных {
    get {
        // тело геттера
        return возвращаемоеЗначение
    }
    set (ассоциированныйПараметр) {
        // тело сеттера
    }
}
```

`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, 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*;
- *радиус*.

Но нам также требуется третья характеристика: *длина окружности*, а она напрямую зависит от радиуса и может быть вычислена по специальной формуле. Необходимо учесть, что в процессе работы с экземпляром и *радиус*, и *длина окружности* могут изменяться. Но при изменении одной величины также должна измениться и другая. То есть оба свойства в любой момент времени должны возвращать корректное значение.

Для решения данной задачи свойство, описывающее длину окружности, сделаем вычисляемым и содержащим геттер и сеттер (листинг 22.5).

Листинг 22.5

```
struct Circle {
    var coordinates: (x: Int, y: Int)
    var radius: Float
    var perimeter: Float {
        get {
            // высчитаем значение, используя текущее значение радиуса
            return 2.0 * 3.14 * radius
        }
        set(newPerimeter) {
            // изменим текущее значение радиуса
            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, структура `Circle` может выглядеть как в листинге 22.6.

Листинг 22.6

```
struct Circle {
    var coordinates: (x: Int, y: Int)
    var radius: Float
    var perimeter: Float {
        get {
return 2.0 * 3.14 * radius
        }
        set {
            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 значений свойств. 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 установкой нового значения хранимого свойства, либо сразу после нее.

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 наблюдатель `willSet` в данный аргумент 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 функциональность, выводящую при изменении радиуса окружности информацию об этом на консоль (листинг 22.7).

Листинг 22.7

```
struct Circle {
    var coordinates: (x: Int, y: Int)
    // var radius: Float
    var radius: Float {
        willSet (newValueOfRadius) {
            print("Вместо значения \(radius) будет установлено
                  значение \(newValueOfRadius)")
        }
        didSet (oldValueOfRadius) {
            print("Значение \(oldValueOfRadius) изменено на \(radius)")
        }
    }
    var perimeter: Float {
        get {
            return 2.0 * 3.14 * radius
        }
        set {
            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 будет установлено значение 5.0
Значение 10.0 изменено на 5.0
```

22.3. 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 SomeEnumeration {
     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 22.8). Класс `AudioChannel` моделирует аудиоканал, у которого есть два параметра:

- максимально возможная громкость ограничена для всех каналов в целом;
- текущая громкость ограничена максимальной громкостью.

Листинг 22.8

```
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.

Глава 23. Сабскрипты

Сабскрипты уже встречались вам ранее, в частности, при работе с массивами и словарями. Они использовались для доступа к отдельным значениям коллекций по их индексу. Однако сабскрипты позволяют значительно упростить работу со структурами и классами.

23.1. Назначение сабскриптов

С помощью сабскриптов структуры и классы можно превратить в некое подобие коллекций. У вас есть возможность 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.

23.2. 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 аргументов.

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 23.1 показан код класса `GameDesk`, описывающего шахматную доску.

Листинг 23.1

```
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`, он 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 помощью сабскрипта (листинг 23.2). 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 23.2

```
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, то перед доступом к свойству `coordinates` возвращенной шахматной фигуры необходимо выполнить извлечение опционального значения.

Теперь мы имеем возможность установить фигуры на шахматную доску с помощью метода `setChessman(chess:coordinates:)` и получить информацию об отдельной ячейке с помощью сабскрипта.

Мы можем расширить функциональность сабскрипта, добавив в него сеттер, позволяющий устанавливать фигуру на новые координаты (листинг 23.3).

Листинг 23.3

```
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"..."H"
        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.

Сабскрипты 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 различными способами. 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 `Chessman`.

Глава 24. Наследование

Один из принципов методологии объектно-ориентированного программирования называется *наследованием*. Он подразумевает то, что классы могут быть созданы не с нуля, а на базе уже существующих классов. При этом все свойства, методы и сабскрипты старшего класса включаются в состав нового.

При наследовании старший класс называется суперклассом (базовым или родительским), а новый (созданный на его основе) — подклассом (производным или дочерним).

> **ПРИМЕЧАНИЕ** Наследование доступно только при работе с классами (class). Структуры не поддерживают наследование.

24.1. 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 24.1). Данный класс описывает сущность «четвероногое животное». Дополнительно объявляется дочерний класс `Dog`, описывающий сущность «собака». Все характеристики класса `Quadruped` применимы и к классу `Dog`, поэтому их можно наследовать.

Листинг 24.1

```
// 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 подклассов.

Доступ к наследуемым характеристикам

Доступ к наследуемым элементам родительского класса в производном классе реализуется так же, как и к собственным элементам данного производного класса, то есть с использованием ключевого слова `self`. В качестве примера в класс `Dog` добавим метод, выводящий в консоль кличку собаки. Кличка хранится в свойстве `name`, которое наследуется от класса `Quadruped` (листинг 24.2).

Листинг 24.2

```
class Dog: Quadruped {
    func bark() {
        print("woof")
    }
    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.

24.2. Переопределение наследуемых элементов

Подкласс 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 дочернем классе 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 24.3).

Листинг 24.3

```
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 родительским вне зависимости от того, является он дочерним для другого класса 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 данная функциональность уже реализована в одноименном родительском методе. Перепишем реализацию метода `bark()` таким образом, чтобы избежать дублирования кода (листинг 24.4).

Листинг 24.4

```
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 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 подкласса 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 24.5).

Листинг 24.5

```
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 под-

класса. 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.

Подкласс 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.

24.3. 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.

24.4. Подмена экземпляров классов

Наследование, помимо всех перечисленных возможностей, позволяет заменять требуемые экземпляры определенного класса экземплярами одного из подклассов.

Рассмотрим пример из листинга 24.6. Объявим массив элементов типа `Quadruped` и добавим в него несколько элементов.

Листинг 24.6

```
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`.

24.5. 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 24.7).

Листинг 24.7

```
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 указывает 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, в случае неудачи, вызывает ошибку.

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 24.8).

Листинг 24.8

```
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 ему становится доступным метод bark().

Глава 25. Контроль доступа

В процессе разработки программного кода Swift предоставляет вам возможность определять уровень доступа (контролировать доступ) к объектным типам и их элементам (свойствам и методам). Благодаря этому вы всегда можете быть уверены, что при работе с классом, структурой или перечислением случайно не обратитесь к «техническому» элементу, использование которого напрямую может привести к нежелательным последствиям.

Чтобы лучше понять, что же такое «контроль доступа», рассмотрим следующий пример. Предположим, что вы разработали класс `UserManager`, который управляет учетными записями пользователей приложения. Данный класс состоит из множества свойств и методов (рис. 25.1).

При этом некоторые из методов являются «техническими», или, иными словами, «внутренними», которые вызываются из других методов данного класса. При этом их использование извне может привести к неожиданным и не всегда корректным результатам. В таком случае было бы логично запретить к ним доступ напрямую, то есть сделать их приватными. А остальные элементы, доступ к которым должен быть, сделать публичными (рис. 25.2).

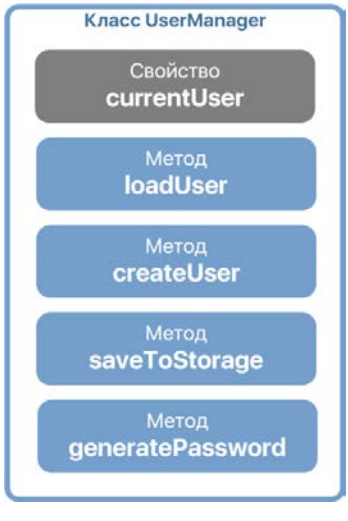

Рис. 25.1. Состав класса UserManager

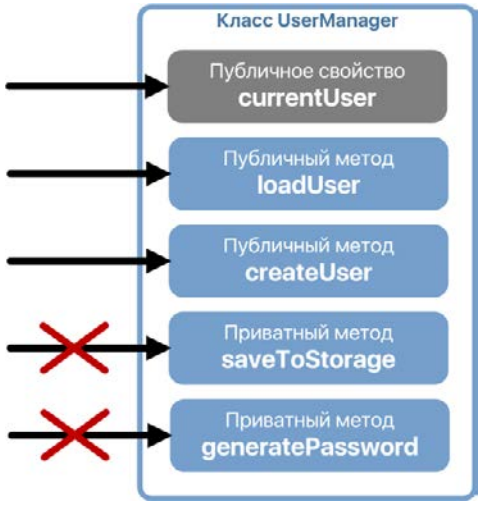

Рис. 25.2. Уровни доступа к элементам класса UserManager

При таком подходе вы всегда сможете контролировать, какие именно элементы класса будут доступны и на каком уровне. Так, теперь попытка доступа к методу `saveToStorage` не в теле данного объектного типа вызовет ошибку (листинг 25.1).

Листинг 25.1

```
let userManager = UserManager()
userManager.generatePassword() // Ошибка
```

При этом у вас все еще остается возможность обращаться к приватным методам внутри других методов класса `UserManager`.

Для контроля доступа используются специальные модификаторы, которые указываются перед объявлением объектного типа, свойства и метода. Всего в Swift доступны пять модификаторов:

open — открытый

Наименее ограничивающий уровень доступа. Позволяет использовать элемент (объектный тип, свойство или метод) без каких-либо ограничений.

Данный модификатор применяется исключительно к классам и их элементам:

- класс, имеющий уровень доступа `open`, может иметь подклассы внутри модуля, где он определен, и в модулях, в которые он импортирован;
- элементы класса, имеющие уровень доступа `open`, могут быть переопределены в подклассе в том модуле, где он объявлен, а также в модулях, в которые он импортирован.

public — публичный

Подобно `open`, данный модификатор позволяет использовать элемент без ограничений. Однако в случае с классами есть некоторые особенности:

- класс, имеющий уровень доступа `public` (или более строгий), может иметь подклассы только в том модуле, где был объявлен;
- элементы класса, имеющие уровень доступа `public` (или более строгий), могут быть переопределены (с помощью оператора `override`) в подклассе только в том модуле, где объявлен сам класс.

Для структур данный модификатор является наивысшим (наименее ограничивающим), так как структуры не поддерживают наследование, а соответственно, и модификатор `open`.

internal — внутренний

Данный модификатор используется в случаях, когда необходимо ограничить использование объекта модулем. Таким образом, объект будет доступен во

всех исходных файлах модуля, исключая его использование за пределами модуля.

fileprivate — приватный в пределах файла

Данный модификатор позволяет использовать объект только в пределах данного исходного файла.

private — приватный (частный)

Данный модификатор позволяет использовать объект только в пределах конструкции, в которой он объявлен. Например, объявленное в классе свойство или метод не будут доступны извне или в его расширениях.

По умолчанию все объекты имеют уровень доступа `internal`. Для того чтобы изменить его, необходимо явно указать требуемый уровень.

Вернемся к примеру с классом `UserManager`. В соответствии с указанными выше требованиями, синтаксис его объявления мог бы выглядеть так, как показано в листинге 25.2.

Листинг 25.2

```
open class UserManager {
   // публичное свойство
    public var currentUser: User
   // публичный метод
    public func loadUser(byLogin login: String) -> User {
        // ...
    }
    // публичный метод
    public func createUser(withLogin login: String) {
        // ...
    }
    // приватный метод
    private func saveToStorage() {
        // ...
    }
    // приватный метод
    private func generatePassword() {
        // ...
    }
}
```

Запомните: если предполагается, что уровень доступа к объекту `internal`, то можно его не указывать, так как по умолчанию для любого объекта назначен именно этот уровень.

При определении уровня доступа элементов необходимо запомнить следующее правило:

В составе объекта с более строгим уровнем доступа не могут находиться элементы с менее строгим уровнем доступа.

Рассмотрим три примера, демонстрирующие данное правило.

1. В составе класса, имеющего уровень доступа `public`, не могут быть объявлены свойства и методы, имеющие уровень доступа `open`.
2. Уровень доступа к кортежу определяется наиболее строгим элементом, включенным в кортеж. Так, если вы скомпонуете кортеж из двух разных типов, один из которых будет иметь уровень доступа `internal`, а другой — `private`, то результирующим уровнем доступа всего кортежа будет `private`, то есть самый строгий.
3. Подобно кортежу, уровень доступа к функции определяется самым строгим уровнем типов аргументов функции и типа возвращаемого значения.

Рассмотрим пример из листинга 25.3.

Листинг 25.3

```
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 (так как отсутствие модификатора равносильно `internal`) (листинг 25.4).

Листинг 25.4

```
private func someFunction() -> (SomeInternalClass, SomePrivateClass) {
   // тело функции
}
```

Что касается перечислений, стоит обратить внимание на то, что каждый член перечисления получает тот же уровень доступа, который установлен для самого перечисления.

Глава 26. 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.

26.1. Псевдоним Any

Благодаря псевдониму `Any` можно создавать хранилища неопределенного типа данных. Объявим массив, который может содержать элементы произвольных типов (листинг 26.1).

Листинг 26.1

```
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 даже значение функционального типа `(String)->String`. Таким образом, 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 26.2).

Листинг 26.2

```
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).

26.2. 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 Touch`. Данный фреймворк написан на Objective-C, а этот язык не имеет массивов с явно указанными типами.

Объявим массив экземпляров с помощью псевдонима `AnyObject` (листинг 26.3).

Листинг 26.3

```
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 26.4).

Листинг 26.4

```
for object in someObjects {
    let animal = object as! Dog
    print(animal.type)
}
```

Консоль

```
dog
dog
dog
```

Для сокращения записи вы можете выполнить приведение типа для преобразования всего массива целиком (листинг 26.5).

Листинг 26.5

```
for object in someObjects as! [Dog] {
    print(animal.type)
}
```

> **ПРИМЕЧАНИЕ** В предыдущих примерах показан код, извлекающий опциональные значения с помощью оператора !, а это, напомню, не является безопасным способом. Это сделано сознательно, чтобы вы акцентировали внимание именно на работе с `AnyObject`.

Так, код листинга 26.4 в «безопасном» исполнении мог бы выглядеть следующим образом:

```
for object in someObjects {
    guard let animal = object as? Dog else {
        continue
    }
    print(animal.type)
}
```

А код листинга 26.5 так:

```
for object in (someObjects as? [Dog]) ?? [] {
    print(animal.type)
}
```

Глава 27. Инициализаторы и деинициализаторы

Инициализатор (конструктор) — это специальный метод, выполняющий подготовительные действия при создании экземпляра объектного типа данных. Инициализатор срабатывает при создании экземпляра, а при его удалении вызывается деинициализатор.

27.1. Инициализаторы

Инициализатор выполняет установку начальных значений хранимых свойств и различных настроек, 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 27.1).

Листинг 27.1

```
class Dog: Quadruped {
    override init() {
        super.init()
        self.type = "dog"
    }

    convenience init(text: String) {
         self.init()
         print(text)
    }

    func bark() {
        print("woof")
    }

    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 и выводит текст на консоль.

ПРИМЕЧАНИЕ Вспомогательный инициализатор может вызывать назначенный 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.

На рис. 27.1. представлены все три правила

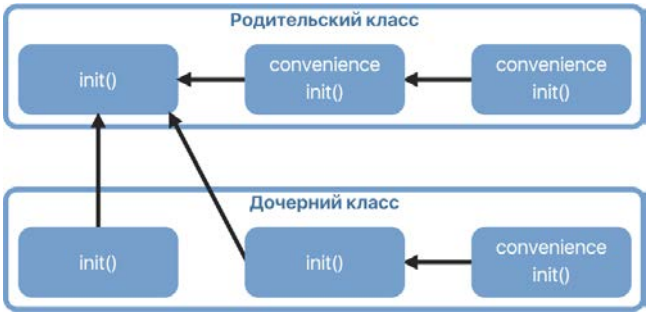

Рис. 27.1. Отношения между инициализаторами

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), который 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 27.2).

Листинг 27.2

```
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 меньше или равно нулю, 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 27.3. 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 27.3

```
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 27.4).

Листинг 27.4

```
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.

27.2. Деинициализаторы

Деинициализатор (деструктор) — это специальный метод, который автоматически вызывается во время уничтожения экземпляра класса. Вы не можете вызвать деинициализатор самостоятельно. Один класс может иметь не более одного деинициализатора.

> **ПРИМЕЧАНИЕ** Деинициализаторы являются отличительной особенностью классов и недоступны для структур.

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 используется 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 27.5).

Листинг 27.5

```
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` в консоль выводится соответствующее сообщение, так как данная функциональность находится в наследуемом от суперкласса проваливающемся инициализаторе. В конце программы мы удаляем созданный экземпляр, передав ему в качестве значения `nil`. При этом вывод в консоль показывает, что первым выполняется деинициализатор подкласса, потом — суперкласса.

Глава 28. Управление памятью в Swift

Управление памятью (memory management) — это одна из тем теоретических основ программирования. Конечно, даже совершенно ничего не понимая в том, как происходит работа по выделению и освобождению памяти вашего компьютера, вы сможете писать вполне работоспособные программы. Однако:

1. Недостаток знаний может привести к большим проблемам, а именно к утечкам памяти и, возможно, последующему падению производительности, критическим ошибкам и аварийному завершению ваших приложений.
2. Если вы будете проходить собеседование на должность Junior Swift Developer, вас могут спросить о том, а что же такое стек (stack), чем он отличается от кучи (heap) и как в них хранятся значения различных типов данных. И вам ни в коем случае нельзя ударить в грязь лицом.

ПРИМЕЧАНИЕ Утечка памяти — это программная ошибка, приводящая к излишнему расходованию оперативной памяти.

28.1. Что такое управление памятью

Любое приложение в процессе функционирования использует оперативную память. Очень важно, чтобы этот процесс, с одной стороны, был максимально быстр и не заметен для пользователя, а с другой — чтобы ресурсы своевременно освобождались, не превращая все доступное пространство в некое подобие «кладбища», где хранятся объекты, которые никогда не будут использоваться в дальнейшем.

Основную работу по распределению и очистке памяти Swift совместно с операционной системой (например, iOS или macOS) производит самостоятельно, без участия разработчика. Вы просто пишете код, компилируете его, и он работает. При этом все создаваемые в приложении значения корректно и быстро записываются в память и извлекаются из нее, не требуя от вас каких-либо специальных знаний.

Я бы хотел подробнее рассказать вам об этих процессах, опуститься на уровень ниже и разобраться с тем, как в оперативной памяти организовано хранение данных.

Как вы знаете, память представляет собой множество ячеек. В первых главах книги мы уже знакомились с ее упрощенной структурой. Тогда был показан пример, в котором каждая ячейка представлялась некоей структурой, способной хранить одно конкретное значение. В действительности все несколько сложнее.

Типовая ячейка оперативной памяти построена на основе полупроводниковых электронных компонентов (транзисторов и конденсаторов), объединенных в группу. Она способна хранить значения размером в 1 байт, то есть 8 бит (8 нулей и единиц). На одной планке памяти расположены миллиарды таких ячеек, и для навигации каждая из них имеет уникальный порядковый номер, который является ее физическим адресом (рис. 28.1). По данному адресу ячейка может быть найдена во всем множестве ячеек, доступных в памяти.

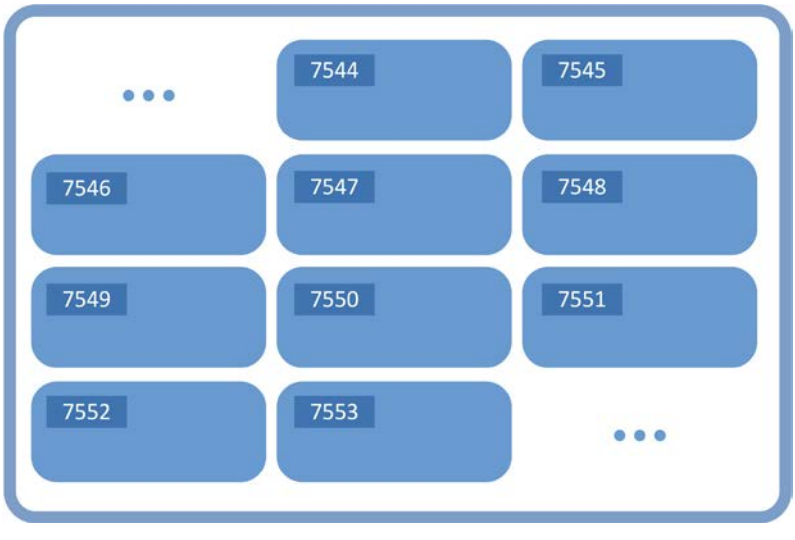

Рис. 28.1. Схематичный вид оперативной памяти

От того, какое именно значение сохраняется в памяти, зависит количество ячеек, которое оно будет занимать. Так, значение типа `UInt16` займет всего 2 ячейки, так как для него требуется 2 байта, то есть 16 бит (2^{16} = 65 535 — максимальное значение, которое может храниться в параметре типа `UInt16`).

Но для того, чтобы программы могли взаимодействовать с этими физическими ячейками, необходим подходящий механизм, которым является виртуальная память. *Виртуальная память* объединяет первичную (оперативная память) и вторичную (жесткий диск или SSD для хранения файлов подкачки) память в единое хранилище со сквозным адресным пространством.

Как только в операционной системе запускается новый процесс (например, ваше приложение), ему выделяется персональный участок виртуальной памяти, который будет использоваться в процессе функционирования. Благодаря наличию

виртуальной памяти приложения защищены от воздействия на участки друг друга. Работая с памятью в своем приложении, вы взаимодействуете только с теми ячейками, которые были выделены именно вам.

Выделенная вашему приложению память логически делится на три области:

1. **Статическая память**, в которой размещается код приложения, различные библиотеки, метаданные и глобальные переменные, необходимые для работы.
2. **Автоматическая память**, в которой хранятся все локальные для текущей области видимости параметры.
3. **Динамическая память**, в которой память выделяется динамически по запросу.

Поговорим подробнее о каждой из указанных областей и рассмотрим несколько примеров.

Статическая память

В процессе компиляции ваше приложение переводится в машинный код, то есть код, понятный компьютеру. А в процессе загрузки этот код, вместе с различными библиотеками, загружается в статическую область памяти, откуда по мере необходимости вызывается. Статическая память выделяется один раз еще до появления приложения на экране и существует все время, пока приложение работает.

Автоматическая память

Автоматическая память работает на основе структуры данных стек (stack), то есть по принципу «последним пришел — первым вышел» (*Last in, first out,* или LIFO). Объекты, записанные в стек, похожи на стопку тарелок: последняя поставленная тарелка не дает убрать те, что находятся ниже, пока данная тарелка сама не будет убрана.

В стеке хранятся все локальные (относящиеся к данной области видимости) значения. То есть как только создается новая переменная, она помещается в стек, следующая переменная помещается поверх нее, и т. д. А при выходе из данной области видимости все созданные значения последовательно удаляются из стека.

> **ПРИМЕЧАНИЕ** Стек — это лишь способ хранения параметров и их значений. Это не значит, что у вас при этом есть доступ только к верхнему элементу стека. Ваше приложение всегда хранит ссылки на адреса всех параметров, и вы при необходимости можете получить доступ к любому из них (в соответствии с их областью видимости) для их чтения или изменения. Но удаляются элементы стека последовательно.

В Swift в стеке хранятся значимые типы данных (value type), а также указатели на ссылочные типы (reference type). Работая со стеком, вы не встретитесь с какими-либо проблемами излишнего использованием памяти: его элементы своевременно создаются и удаляются.

Рассмотрим пример, приведенный в листинге 28.1.

Листинг 28.1

```
// структура, описывающая сущность "Приложение"
struct Application {
    var name: String
}

// функция – точка входа в приложение
func main(testMode: Bool) {
    let app = Application(name: "Calculator")
    // ... последующие операции внутри функции
}

// начало работы программы
main(testMode: true)
```

Структура `Application` и функция `main(testMode:)` после загрузки будут находиться в статической памяти. В ходе работы приложения по мере необходимости они будут вызваны оттуда.

Как только в программе создается новая область видимости (scope) — в данном случае вызывается функция `main`, — в стеке создается новый фрейм (специальный блок, объединяющий несколько элементов одной области видимости), в котором выделяется место под локальные для этой области видимости параметры и их значения (рис. 28.2).

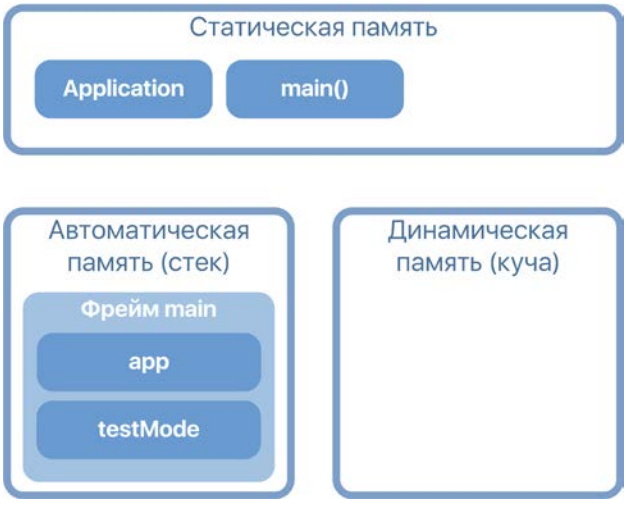

Рис. 28.2. Структура областей памяти

Функция `main` имеет два локальных параметра: входной аргумент `testMode` и константу `app`. Тип данных `Application` создан с помощью структуры (`struct`),

следовательно, это значимый тип (value type). Как говорилось выше, значения `value type` хранятся в стеке.

Теперь объявим еще одну функцию и вызовем ее уже внутри `main` (листинг 28.2).

Листинг 28.2

```
// функция, производящая загрузку ресурсов
func loadResources(forApp: Application) {
    let appPath = "./bin/\(forApp.name)/"
    // ... последующие операции внутри функции
}

// функция – точка входа
func main(testMode: Bool) {
    let app = Application(name: "Calculator")
    loadResources(forApp: app)
    // ... последующие операции внутри функции
}

// начало работы программы
main(testMode: true)
```

Теперь работа с памятью будет включать следующие шаги (рис. 28.3):

1. В момент вызова функции `main` в стеке создается новый фрейм, в который помещаются значения локальных параметров `testMode` и `app`.
2. В момент вызова функции `loadResources(forApp:)` в стеке создается второй фрейм, в который помещаются значения локальных параметров `forApp` и `appPath`.
3. Как только функция `loadResources` завершает работу, из стека последовательно удаляются все объекты, относящиеся к данной области видимости.
4. Как только функция `main` завершает работу, из стека также удаляются все объекты, связанные с ней.

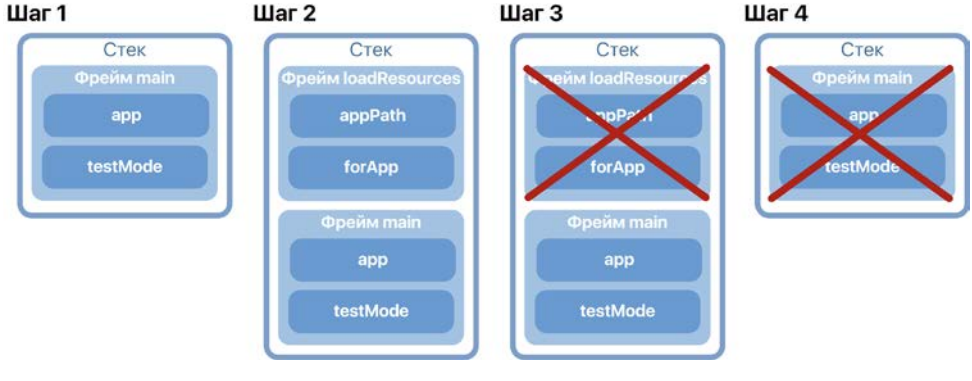

Рис. 28.3. Состав стека

Аргумент `forApp` функции `loadResources`, а также локальный параметр `appPath` — это `value type`, а значит, они также сохраняются в стеке.

Приложение знает, где находится вершина стека (оно хранит адрес верхнего значения в стеке), а фреймы хранят информацию о предыдущем фрейме. Благодаря этому работа со стеком очень быстрая, практически мгновенная. При необходимости удалить фрейм у стека просто изменяется указатель на верхний элемент (на шаге 3 вместо `appPath` верхним элементом стал `app`).

Динамическая память

Но что будет, если вместо структуры `Application` использовать одноименный класс, ведь класс — это reference type, его значения передаются по ссылке, а не копированием, как в случае со структурой. Для хранения значений ссылочных типов используется динамическая память.

Динамическая память — это область, память в которой выделяется по запросу приложения. Данная область также называется **кучей** (heap). Объекты в куче не упорядочены, программа при необходимости захватывает память требуемого объема и помещает туда значение. При этом в стеке создается запись, в которой сохраняется ссылка на объект в куче.

Вместо структуры `Application` из предыдущего листинга используем одноименный класс, при этом объявим в нем деинициализатор и добавим несколько вызовов функции `print`, чтобы отследить, когда именно создается и удаляется экземпляр (листинг 28.3).

Листинг 28.3

```
class Application {
    var name: String

    init(name: String) {
        print("создание экземпляра")
        self.name = name
    }

    deinit {
        print("уничтожение экземпляра")
    }
}

func loadResources(forApp: Application) {
    print("начало функции loadResources")
    let appPath = "./bin/\(forApp.name)/"
    // ... последующие операции внутри функции
    print("завершение функции loadResources")
}

// функция – точка входа
func main(testMode: Bool) {
```

```
    print("начало функции main")
    let app = Application(name: "Calculator")
    loadResources(forApp: app)
    // ... последующие операции внутри функции
    print("завершение функции main")
}

// начало работы программы
main(testMode: true)
```

Консоль

```
начало функции main
создание экземпляра
начало функции loadResources
завершение функции loadResources
завершение функции main
уничтожение экземпляра
```

Обратите внимание на консоль: самое интересное, что объект типа `Application` уничтожается лишь в самом конце, после того, как функция `main` завершает свою работу.

Разберем порядок работы кода по шагам (рис. 28.4):

1. В момент вызова функции `main` в стеке создается новый фрейм, в который помещаются значения локальных параметров. При этом значение типа `Application` (это ссылочный тип) сохраняется в куче, а в стеке создается запись со ссылкой на него.
2. В момент вызова функции `loadResources(forApp:)` в стеке создается второй фрейм, в который помещаются значения локальных параметров. При этом значение типа `Application` передано по ссылке, а не копированием, поэтому в стеке создается новая запись, содержащая ссылку на уже существующее значение в куче.
3. Как только функция `loadResources` завершает работу, из стека последовательно удаляются все объекты, относящиеся к данной области видимости, включая ссылку на объект типа `Application`, хранящийся в куче. Но сам объект `Application` в куче не удаляется.
4. Как только функция `main` завершает работу, из стека также удаляются все объекты, связанные с ней, включая ссылку на объект типа `Application`, хранящийся в куче.
5. Объект типа `Application` в куче больше не имеет входящих ссылок, а значит, может быть удален. Это происходит в автоматическом режиме с помощью встроенного в Swift механизма ARC (Automatic Reference Counting), с которым мы познакомимся в конце главы.

Все проблемы, связанные с излишним использованием памяти в ваших приложениях, случаются только из-за неправильного использования ссылок на объекты

в куче; чуть позже мы искусственно создадим утечку памяти, после чего рассмотрим способы избежать этого.

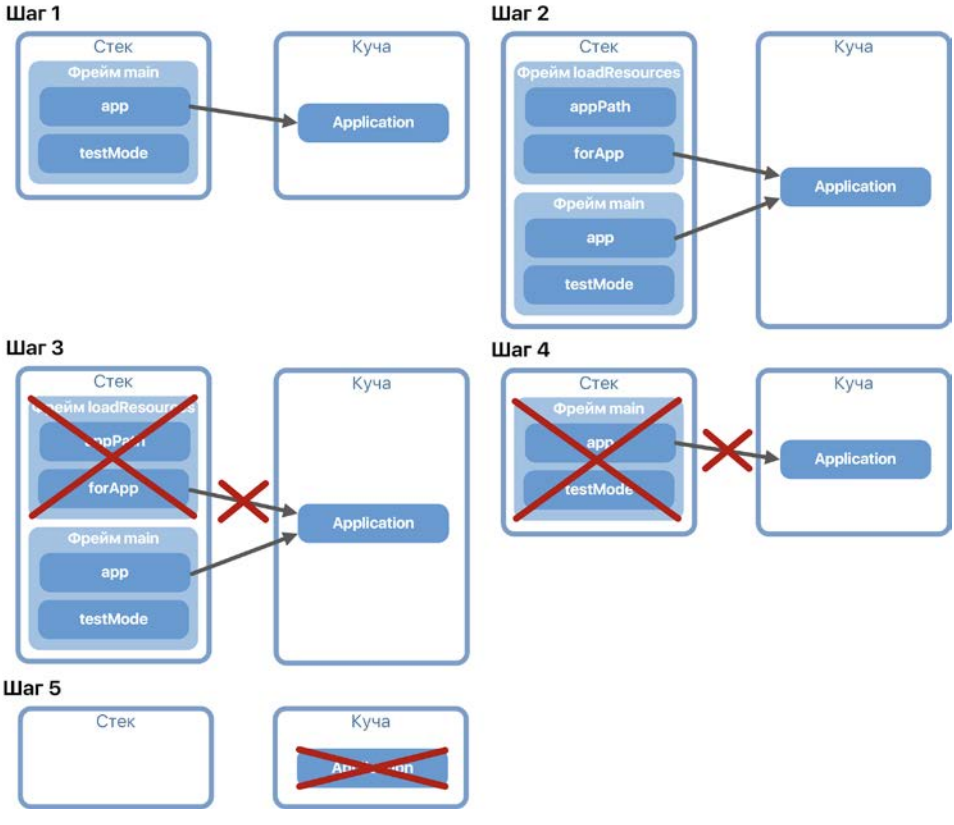

Рис. 28.4. Состав областей памяти

28.2. Уничтожение экземпляров

Стандартный цикл жизни экземпляров объектных типов состоит из следующих шагов:

1) выделение памяти (в стеке или в куче);
2) инициализация (выполнение кода в методе `init`);
3) использование;
4) деинициализация (выполнение кода в методе `deinit`, актуально только для классов);
5) освобождение памяти.

Как говорилось в предыдущем разделе, Swift принимает решение об удалении значения в памяти, основываясь на том, когда происходит выход из области видимости (например, завершается функция). Когда речь идет о стеке, то просто последовательно удаляются все значения, находящиеся в соответствующем фрейме. Значимые типы практически никогда не создают каких-либо проблем с памятью, так как хранятся в стеке.

Но как обстоит ситуация с кучей? Каким образом Swift принимает решение о том, что экземпляры классов могут быть удалены? Для этого специальный механизм ARC ведет подсчет ссылок на каждый объект в куче, и в случае, если их количество равняется нулю, объект удаляется из памяти.

Количество ссылок на экземпляр

Рассмотрим пример из листинга 28.4.

Листинг 28.4

```
class House {
    var title: String
    var owner: Human?

    init(title: String, owner: Human? = nil) {
        print("Дом \(title) создан")
            self.title = title
            self.owner = owner
        return
    }

    deinit{
        print("Дом \(title) уничтожен")
    }
}

class Human {
    var name: String

    init(name: String) {
        print("Владелец \(name) создан")
        self.name = name
        return
    }

    deinit {
        print("Владелец \(name) уничтожен")
    }
}
```

Класс `House` описывает сущность «Дом», а класс `Human` — сущность «Человек». Человек может быть владельцем дома, в этом случае он указывается в свойстве `owner`. Предположим, что некий человек является владельцем двух домов (листинг 28.5).

Листинг 28.5

```
// создаем область видимости
if true { // шаг 1
    let houseOwner = Human(name:"Василий")

    if true { // шаг 2
        let house = House(title:"Частный дом", owner: houseOwner)
    } // шаг 3

    // шаг 4
    let secondHouse = House(title: "Квартира", owner: houseOwner)
} // шаг 5
```

Консоль

```
Владелец Василий создан
Дом Частный дом создан
Дом Частный дом уничтожен
Дом Квартира создан
Дом Квартира уничтожен
Владелец Василий уничтожен
```

> **ПРИМЕЧАНИЕ** Конструкция `if` в данном примере используется только для того, чтобы создать новую область видимости, при выходе из которой должны удаляться записанные в память объекты.

Рассмотрим по шагам, как именно работает данный код (рис. 28.5):

1. В куче выделяется память под новый объект ссылочного типа, описывающий владельца (Василий). В стеке создается новый фрейм, в котором выделяется память под константу `houseOwner`, содержащую указатель на объект «Василий» в куче.
2. В куче создается новый объект типа `House`, содержащий в свойстве `owner` ссылку на уже существующий объект типа `Human` (передается в качестве аргумента инициализатора). В стеке создается новый фрейм (#2), в котором выделяется память для константы `house`, содержащей ссылку на объект типа `House` (Частный дом) в куче.
3. Происходит выход из области видимости (вложенный оператор `if`), вследствие чего в стеке удаляется фрейм (#2), а также его элементы и ссылка на объект `House` (Частный дом) в куче. Данный объект в куче также автоматически удаляется, так как у него больше нет входящих ссылок. Следовательно, удаляется ссылка на объект типа `Human`, которая хранилась в свойстве `owner`.
4. Обратите внимание, что при этом объект типа `Human` не удаляется, так у него все еще существует входящая ссылка из стека.
5. В куче создается новый объект типа `House` (Квартира). В стеке, в уже существующем фрейме (#1) (так как это та же самая область видимости), создается параметр `secondHouse`, содержащий ссылку на объект в куче.
6. Оператор `if` завершает свою работу, а значит, происходит выход из соответствующей области видимости. По этой причине последовательно удаляются

элементы фрейма #1 в стеке. После этого удаляется объект `House` (Квартира), так как на него больше нет ссылок. В последнюю очередь удаляется объект типа `Human` (Василий), так как последняя ссылка на него исчезла вместе с `House` (Квартира).

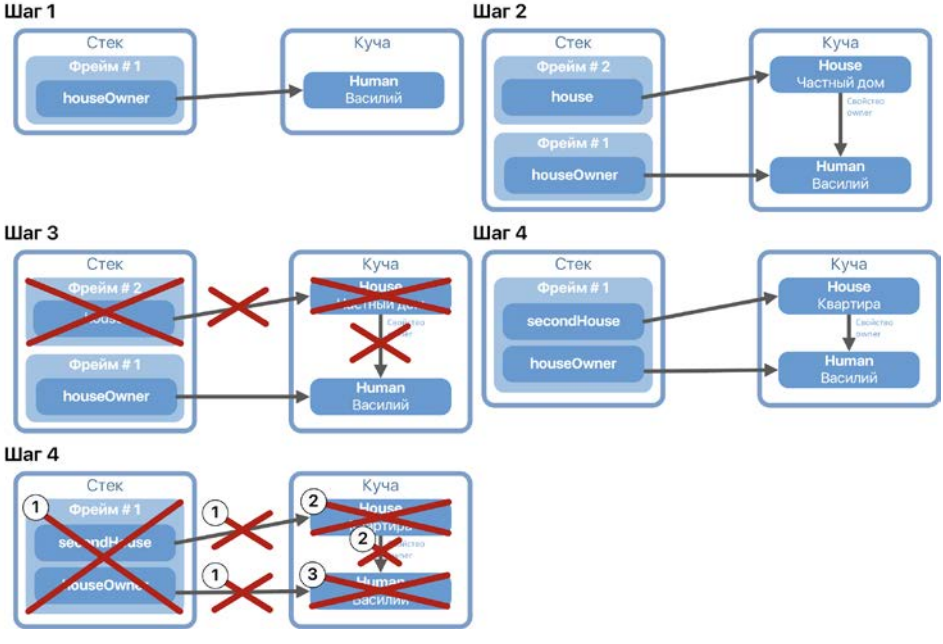

Рис. 28.5. Схема использования памяти

28.3. Утечки памяти и ARC

Одна из наиболее серьезных проблем, которая может возникнуть в процессе разработки приложений, — это утечка памяти. Некоторые утечки практически не заметны, а из-за некоторых вы лишаетесь значительного объема доступной памяти. А это, в свою очередь, чревато падением производительности и экстренным завершением приложения.

При утечке памяти Swift не может принять своевременное решение об удалении объектов в куче. В результате этого в памяти появляются «бесхозные» записи, которые никогда больше не будут использоваться и будут находиться там до завершения работы приложения.

Пример утечки памяти

Рассмотрим ситуацию, при которой может возникнуть утечка памяти. Изменим объявленные ранее классы `House` и `Human` так, чтобы `Human` также мог хранить

ссылки на дома, чьим владельцем он является (листинг 28.6). То есть создадим возможность создания кольцевых ссылок, когда объекты взаимно указывают друг на друга.

Листинг 28.6

```
class Human {
    var name: String
    var houses: [House] = []

    init(name: String) {
        print("Владелец \(name) создан")
        self.name = name
        return
    }

    func add(house: House) {
        self.houses.append(house)
    }

    deinit {
        print("Владелец \(name) уничтожен")
    }
}
```

С помощью метода `add(house:)` мы можем установить дополнительную связь между домом и его владельцем: если раньше главным объектом была сущность «Дом», так как содержала ссылку на владельца, то теперь связь двусторонняя.

Рассмотрим пример из листинга 28.7.

Листинг 28.7

```
// создаем область видимости
if true { // шаг 1
    let houseOwner = Human(name:"Василий")

    if true { // шаг 2
        let house = House(title:"Частный дом", owner: houseOwner)
        houseOwner.add(house: house)
    } // шаг 3

    // шаг 4
    let secondHouse = House(title: "Квартира", owner: houseOwner)
    houseOwner.add(house: secondHouse)
} // шаг 5
```

Консоль

```
Владелец Василий создан
Дом Частный дом создан
Дом Квартира создан
```

Если посмотреть на вывод на консоли, то вы увидите, что все три объекта (`houseOwner`, `house` и `secondHouse`) были созданы. Но хотя код и завершил работу

без ошибок, они так и не были удалены (деинициализатор ни в одном из них не был вызван). Это пример утечки памяти: происходил выход из областей видимости, но локальные параметры не удалялись.

Для того чтобы разобраться, в чем причина такого поведения, рассмотрим работу кода по шагам (рис. 28.6):

1. В куче выделяется память под новый объект ссылочного типа, описывающий владельца (Василий). В стеке создается новый фрейм, в котором выделяется память под константу `houseOwner`, содержащую указатель на данный объект.
2. В куче создается новый объект `house`, который в свойстве `owner` содержит ссылку на уже существующий объект типа `Human`. В стеке создается новый фрейм (#2), в котором выделяется память для константы `house`, содержащей ссылку на объект тип `House` (Частный дом). При этом в свойство `houses` параметра `houseOwner` добавляется ссылка на созданный «Частный дом».

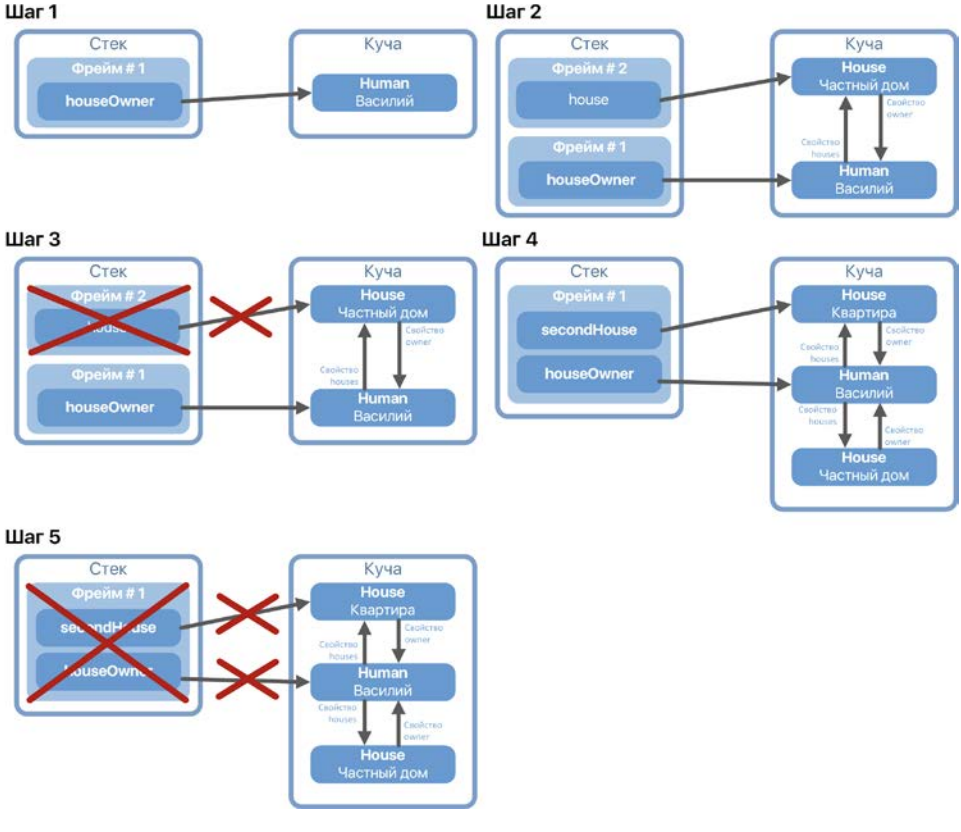

Рис. 28.6. Схема использования памяти

3. Происходит выход из области видимости (вложенный оператор `if`), вследствие чего в стеке удаляется фрейм (#2) и все его элементы. Но самое интересное, что объект типа `House` (Частный дом) в куче не может быть удален, так как на него все еще ссылается объект `houseOwner`. И несмотря на то что был произведен выход из области видимости, объект остается «висеть» в памяти.
4. В куче создается новый объект типа `House` (Квартира). В стеке в уже существующем фрейме (#1) создается объект `secondHouse`, содержащий ссылку на объект в куче. При этом создается взаимная ссылка от параметра `houseOwner` к созданному объекту.
5. Оператор `if` завершает свою работу, а значит, происходит выход из соответствующей области видимости. По этой причине последовательно удаляются элементы фрейма #1 в стеке, но ни один из элементов в куче не может быть удален, так как они имеют взаимные ссылки друг на друга.

Подобная ситуация называется циклом сильных ссылок (retain cycle). В этом случае Swift просто не может решить, какую из ссылок можно уничтожить, а какую нет. В результате все объекты имеют входящие ссылки и не удаляются из памяти.

Это типичный пример утечки памяти в приложениях.

Сильные (strong), слабые (weak) и бесхозные (unowned) ссылки

Все создаваемые по умолчанию ссылки на объекты являются сильными (strong). Если объект имеет хотя бы одну входящую сильную ссылку, то он просто не может быть удален.

Для решения показанной ранее ситуации с утечкой памяти в Swift используются слабые ссылки. Такой вид ссылок не мешает удалению объекта, то есть если объект в куче имеет входящие слабые ссылки, но не имеет входящих сильных ссылок, то будет удален, как только для этого возникнут подходящие условия (произойдет выход из области видимости).

Для того чтобы пометить ссылку как сильную, вам не нужно использовать специальные ключевые слова. Для слабых ссылок предназначен модификатор `weak`, который указывается перед объявлением свойства, содержащего ссылку:

```
weak var имяСвойства: ОпциональныйТипДанных?
```

Данный модификатор говорит о том, что по усмотрению Swift значение свойства может быть переведено в `nil`, то есть уничтожено.

Внесем изменения в класс `House` и сделаем ссылку на владельца (свойство `owner`) слабой (`weak`) (листинг 28.8).

Листинг 28.8

```
class House {
    var title: String
    weak var owner: Human?

    init(title: String, owner: Human? = nil) {
        print("Дом \(title) создан")
        self.title = title
        self.owner = owner
        return
    }

    deinit {
        print("Дом \(title) уничтожен")
    }
}
```

Теперь, если вновь запустить код из листинга 28.7, то на консоли вы увидите следующее:

Консоль

```
Владелец Василий создан
Дом Частный дом создан
Дом Квартира создан
Владелец Василий уничтожен
Дом Частный дом уничтожен
Дом Квартира уничтожен
```

Все объекты уничтожены, а это значит, что никаких утечек памяти больше нет. Разберем листинг 28.7 по шагам (рис. 28.7).

1. В куче выделяется память под новый объект ссылочного типа, описывающий владельца (Василий). В стеке создается новый фрейм, в котором выделяется память под константу `houseOwner`, содержащую ссылку на объект «Василий».
2. В куче создается новый объект типа `House`, содержащий в свойстве `owner` **слабую** ссылку на уже существующий объект типа `Human`. В стеке создается новый фрейм (#2), в котором выделяется память для константы `house`, содержащей ссылку на объект тип `House` (Частный дом). При этом в свойство `houses` параметра `houseOwner` добавляется ссылка на созданный «Частный дом».
3. Происходит выход из области видимости (вложенный оператор `if`), вследствие чего в стеке удаляется фрейм (#2) и все его элементы. При этом объект типа `House` (Частный дом) в куче не может быть удален, так как на него ссылается объект `houseOwner`.
4. В куче создается новый объект типа `House` (Квартира). В стеке, в уже существующем фрейме (#1), создается объект `secondHouse`, содержащий **слабую** ссылку на объект в куче. При этом создается взаимная ссылка от параметра `houseOwner` к созданному объекту.

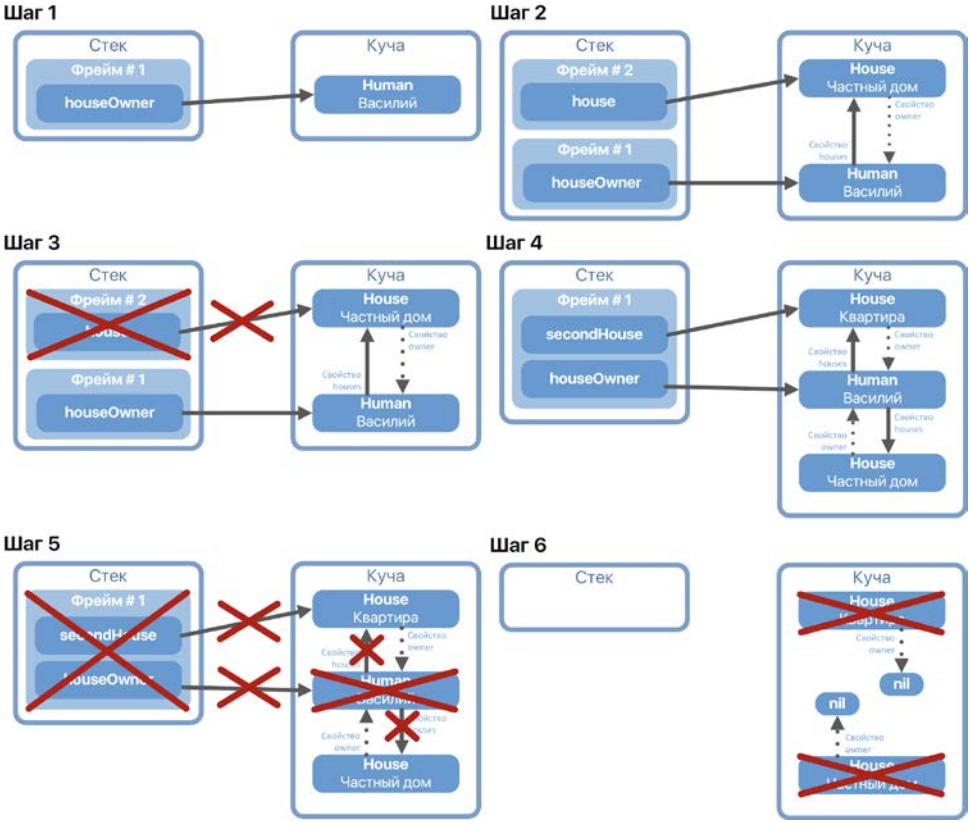

Рис. 28.7. Схема использования памяти

5. Оператор `if` завершает свою работу, и происходит выход из соответствующей области видимости. По этой причине последовательно удаляются элементы фрейма #1 в стеке. В куче все еще находятся три объекта, взаимно ссылающиеся друг на друга. Swift видит данную ситуацию и изменяет значение, хранящееся по слабой ссылке (свойства `owner` в типах `House`) на `nil`, тем самым уничтожая объект типа `Human`.
6. Два оставшихся объекта типа `House` теперь не имеют входящих сильных ссылок, а значит, могут быть безболезненно удалены, что и происходит.

Использование модификатора `weak` привносит некоторые особенности:

- Так как значение свойства переводится в `nil`, оно должно быть опциональным. В нашем примере свойство `owner` имеет опциональный тип `Human?`.
- Так как свойство *изменяет* свое значение (на `nil`), оно не может быть константой.

Если необходимо сделать слабую ссылку на экземпляр неопционального типа данных, то используется модификатор `unowned` вместо `weak` (листинг 28.9). Такие ссылки называются **бесхозными**.

Листинг 28.9

```
class House {
    var title: String
    unowned var owner: Human

    init(title: String, owner: Human) {
        print("Дом \(title) создан")
        self.title = title
        self.owner = owner
        return
    }

    deinit {
        print("Дом \(title) уничтожен")
    }
}
```

Указанные ключевые слова можно использовать только для хранилища конкретного экземпляра класса. Вы не можете указать слабую ссылку на массив экземпляров или на кортеж, состоящий из экземпляров класса. В рассмотренном примере именно по этой причине мы не могли определить ссылку от значения типа `Human` к `House` как слабую, свойство `houses` — это массив сущностей «Дом».

Automatic Reference Counting (ARC)

Automatic Reference Counting (ARC) — это специальный механизм автоматического подсчета ссылок. Именно он подсчитывал количество входящих сильных ссылок на каждый объект и при возможности удалял неиспользуемые экземпляры. Всю «магию», о которой мы говорили в данной главе, реализует именно ARC. Как только последняя ссылка на экземпляр будет удалена и произойдет выход из ее области видимости, ARC незамедлительно вызовет деинициализатор и уничтожит объект.

ARC делает работу со Swift еще более удобной.

28.4. Ссылки в замыканиях

В процессе изучения материала мы неоднократно говорили о том, что value type передается копированием, а reference type — по ссылке. И это утверждение истинно практически в 100% случаев, но если на одном из будущих собеседований вас спросят: «А может ли value type передаваться по ссылке?» и вы ответите: «Нет! Не может!», то, вероятно, не пройдете его. В этом разделе

я покажу вам простейший пример того, что и значимые типы могут быть переданы по ссылке.

ПРИМЕЧАНИЕ На самом деле вы уже знаете этот материал, но до этого при его рассмотрении я не акцентировал внимание именно на таком поведении значимых типов.

Взгляните на листинг 28.10.

Листинг 28.10

```
var a = 2
let f = { print(a) }
f()
a = 3
f()
```

Консоль

```
2
3
```

Это пример уже известного вам захвата переменных замыканиями. В нем мы создаем параметр типа `Int` и выводим его значение на консоль с помощью замыкания `f`. Обратите внимание на то, что `Int` — это структура, следовательно, value type, значение которого должно передаваться копированием. Но при этом замыкание `f` всегда выводит актуальное значение параметра. Это говорит о том, что внутри замыкания используется не копия параметра `a`, а ссылка на него!

Это и есть пример того, как значимый тип данных может быть передан по ссылке. Запомните, что **замыкания всегда захватывают значения по ссылке**, а не копированием! Даже значение value type!

Такое поведение также может быть для вас неочевидным и стать источником проблем, в частности, в результате несвоевременного удаления объектов, захваченных в замыканиях. Но и в этом случае можно использовать модификаторы `weak` и `unowned` для изменения типа ссылки для входных аргументов замыканий.

Рассмотрим пример из листинга 28.11, в котором создается пустое опциональное замыкание и в зоне ограниченной области видимости (в теле оператора `if`) ему передается значение (тело замыкания).

Листинг 28.11

```
class Man {
    var name = "Человек"
    deinit {
        print("Объект удален")
    }
}

var closure : (() -> Void)?
```

```
if true {
    let man = Man()
    closure = {
        print(man.name)
    }
    closure!()
}
print("Программа завершена")
```

Консоль

```
Человек
Программа завершена
```

Так как условный оператор ограничивает область видимости переменной `man`, содержащей экземпляр класса `Man`, то на первый взгляд кажется, что данный объект должен быть удален вместе с окончанием условного оператора. Однако по выводу на консоль видно, что экземпляр создается, однако 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, то 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 28.12).

Листинг 28.12

```
if true {
    let man = Man()
    closure = { [unowned man] in
        print(man.name)
    }
    closure!()
}
print("Программа завершена")
```

Консоль

```
Человек
Объект удален
Программа завершена
```

В данном примере используется модификатор `unowned`, поскольку объектный тип не является опционалом. Захватываемый параметр `man` является локальным для замыкания и условного оператора, поэтому Swift без проблем может самостоятельно принять решение об уничтожении хранящейся в нем ссылки.

При необходимости указать тип ссылки для нескольких входных аргументов это нужно делать перед каждым из них отдельно (листинг 28.13).

Листинг 28.13

```
let closureWithSomeArgs = { [unowned human, weak house] in
    // тело замыкания
}
```

Тема управления памятью в Swift является достаточно интересной, а что самое важное, она может натолкнуть вас на самостоятельное изучение дополнительного материала — что же такое на самом деле ячейка памяти, чем отличаются физические и виртуальные адреса, где размещаются стек и куча и многое другое. Очень хорошо, если в процессе обучения вы выписывали возникающие вопросы, чтобы в дальнейшем самостоятельно разобраться в них.

Глава 29. Опциональные цепочки

Опционалы — очень полезный и важный элемент Swift, и если вы достаточно хорошо освоите его, то уже не сможете обойтись без него ни в одной программе. В этой главе мы поговорим об особенностях доступа к значениям объектных типов, которые являются опционалами, а также к их свойствам и методам.

29.1. Доступ к свойствам через опциональные цепочки

Рассмотрим пример из листинга 29.1. Перед вами два класса: `Residence`, описывающий сущность «Место жительства», и `Person`, описывающий «Персону».

Листинг 29.1

```
class Person {
    // резиденция данной персоны
    var residence: Residence?
}

class Residence {
    // количество комнат в резиденции
    var rooms = 1
}
```

Свойство `residence` в классе `Person` может содержать значение типа `Residence` (а может не содержать его, так как это опционал), но по умолчанию при создании нового экземпляра оно является `nil` (листинг 29.2).

Листинг 29.2

```
var man = Person()
man.residence // nil
```

В дальнейшем этому свойству может быть проинициализировано определенное значение, а позже оно может быть вновь удалено, и так многократно (листинг 29.3).

Листинг 29.3

```
man.residence = Residence()
man.residence = nil
```

Проблема в том, что в любой момент времени вы не можете точно знать, содержит ли свойство `residence` конкретное значение или находится ли в нем `nil`. Чтобы избежать ошибок доступа, необходимо проверить значение и только в случае его наличия проводить с ним операции. Например, это можно сделать с помощью опционального связывания (листинг 29.4).

Листинг 29.4

```
if let manResidence = man.residence {
    // действия с manResidence
}
```

Но что, если вложенность опциональных свойств окажется многоуровневой? В этом случае потребуется извлекать значение на каждом этапе, создавая вложенные друг в друга конструкции, что, в свою очередь, значительно усложнит навигацию по вашему коду. В качестве примера реализуем тип `Room`, описывающий сущность «Комната» (листинг 29.5). При этом укажем его в качестве типа данных свойства `rooms` в классе `Residence`.

Листинг 29.5

```
class Person {
    var residence: Residence?
}

class Residence {
    // перечень комнат в резиденции
    var rooms:[Room]?
}

struct Room {
    // площадь комнаты
    let square: Int
}

var man = Person()

// для доступа к значению типа Room
// необходимо выполнить два опциональных связывания
if let residence = man.residence {
    if let rooms = residence.rooms {
        // действия с коллекцией типа [Room]
    }
}
```

ПРИМЕЧАНИЕ Обратите внимание, что свойство `rooms` класса `Residence` теперь является опциональной коллекцией. Ранее оно имело целочисленный тип данных и определяло количество комнат в резиденции.

Как видите, для доступа к свойству `rooms` и хранящейся в нем коллекции требуется строить вложенные конструкции опционального связывания. И чем сложнее будут ваши программы, тем более нагроможденным будет код.

Для решения данной проблемы предназначены *опциональные цепочки*. Они позволяют в одном выражении написать полный путь к требуемому элементу, при этом после каждого опционала необходимо ставить знак вопроса (`?`).

В листинге 29.6 показан пример доступа к свойству `rooms` с использованием опциональной цепочки.

Листинг 29.6

```
// 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
// получаем доступ к комнатам с помощью опциональной цепочки
if let rooms = man.residence?.rooms {
    // действия с коллекцией типа [Room]
}
```

С помощью опциональной цепочки происходит последовательный доступ к каждому элементу, и в случае, если какое-то из значений отсутствует, возвращается `nil`. Таким образом, если в свойстве `residence` не будет значения (`nil`), то операция опционального связывания не будет выполнена, не вызвав при этом каких-либо ошибок.

Вы можете использовать опциональные цепочки для вызова свойств, методов и сабскриптов любого уровня вложенности. Это позволяет «пробираться» через свойства внутри сложных моделей вложенных типов и проверять возможность доступа к их элементам. Например, вы могли бы осуществить доступ следующим образом:

```
man.residence?.kitchen?.table?.move()
```

И если значение отсутствует хотя бы в одном элементе, то результатом выражения будет `nil`.

29.2. Установка значений через опциональные цепочки

Использование опциональных цепочек не ограничивается получением доступа к свойствам и методам. Они также могут использоваться для инициализации значений.

В листинге 29.7 показан пример того, как с помощью опциональной цепочки можно передать значение.

Листинг 29.7

```
let room1 = Room(square: 15)
let room2 = Room(square: 25)
man.residence?.rooms = [room1, room2]
```

Если при доступе к `rooms` значение свойства `residence` будет `nil`, программа не вызовет ошибку, а только не выполнит операцию инициализации.

29.3. Доступ к методам через опциональные цепочки

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 29.8).

Листинг 29.8

```
class Residence {
    var rooms:[Room]?
    func roomsCount() -> Int {
        if let rooms = self.rooms {
            return rooms.count
        } else {
            return 1
        }
    }
}
```

Для вызова данного метода также можно воспользоваться опциональной последовательностью (листинг 29.9).

Листинг 29.9

```
man.residence?.roomsCount() // 2
```

29.4. Доступ к сабскриптам через опциональные цепочки

Опциональные цепочки также могут быть использованы и для доступа к сабскриптам. В листинге 29.10 показан пример доступа к первому элементу коллекции типа `[Room]?`. При этом если в коллекции отсутствует значение, выражение вернет `nil`.

Листинг 29.10

```
let firstRoom = man.residence?.rooms?[0]
type(of:firstRoom) // Room?
```

Такой способ доступа можно использовать и для инициализации значений (листинг 29.11).

Листинг 29.11

```
man.residence?.rooms?[0] = Room(square: 35)
man.residence?.rooms?[0].square // 35
```

Глава 30. Протоколы

В процессе изучения основ разработки на языке Swift мы уже неоднократно обращались к понятию протокола. Вам уже хорошо известны `Equatable`, `Comparable` и многие другие протоколы, благодаря которым типы данных делятся на категории. Эта и несколько следующих глав продолжат знакомить вас с этим элементом языка, и вы научитесь самостоятельно реализовывать и использовать их.

30.1. Понятие протокола

В самом простом случае **протокол** — это перечень требований, которым должен удовлетворять тип данных, соответствующий ему. В более сложных случаях протокол также может содержать не просто требования наличия свойств и методов, но и их конкретную реализацию (об этом будет подробно рассказано в главах про расширения и протокол-ориентированное программирование). Если у вас есть опыт разработки на других языках, то, вполне вероятно, вы встречались с понятием *интерфейса*, или *контракта*, которые являются синонимами протокола.

> **ПРИМЕЧАНИЕ** Если к типу данных применяется протокол, это означает, что:
>
> - *тип данных подписан на протокол,*
>
> или
>
> - *тип данных реализует требования протокола,*
>
> или
>
> - *тип данных принимает протокол к реализации,*
>
> или
>
> - *тип данных соответствует протоколу.*

В протоколе может содержаться перечень свойств, методов и сабскриптов, которые должны быть реализованы в объектном типе, принимающем его к реализации. Другими словами, протоколы содержат требования к наличию определенных элементов внутри типа данных.

Рассмотрим синтаксис объявления протоколов.

СИНТАКСИС

```
protocol ИмяПротокола {
    // тело протокола
}
```

Для объявления нового протокола используется ключевое слово `protocol`, после которого указывается его имя (в верхнем верблюжьем регистре).

Любой объектный тип данных может быть подписан на протокол, неважно, используете вы перечисления (enum), структуры (struct) или класс (class). Для того чтобы принять протокол к исполнению, необходимо написать его имя через двоеточие сразу после имени объектного типа данных:

```
struct ИмяПринимающегоТипа: ИмяПротокола {
    // тело структуры
}
```

После подписки на протокол тип данных обязан выполнить все его требования, то есть реализовать свойства, методы и сабскрипты, описанные в протоколе. При этом тип может быть подписан на произвольное количество протоколов.

```
enum ИмяПринимающегоТипа: ИмяПротокола, ИмяДругогоПротокола {
    // тело перечисления
}
```

Если класс не просто принимает протоколы, но и наследует другой класс, то имя родительского класса необходимо указать первым, а за ним через запятую — список протоколов:

```
class ИмяПринимающегоКласса: ИмяСуперКласса, ИмяПротокола, ИмяДругогоПротокола {
    // тело класса
}
```

30.2. Требуемые свойства

В протоколе может содержаться требование реализации одного или нескольких свойств (в том числе свойств типа, указываемых с помощью ключевого слова `static`). При этом для каждого свойства в протоколе указывается:

- название;
- тип данных;
- требования доступности и изменяемости.

В листинге 30.1 приведен пример объявления протокола.

Листинг 30.1

```
protocol SomeProtocol {
    var mustBeSettable: Int { get set }
    var doesNotNeedToBeSettable: Int { get }
}
```

Протокол `SomeProtocol` имеет требования реализации двух свойств. Таким образом, если тип данных подпишется на протокол `SomeProtocol`, то в нем потребуется реализовать данные свойства, при этом:

- Первое свойство должно иметь название `mustBeSettable`, а второе — `doesNotNeedToBeSettable`.

- Тип данных обоих свойств — `Int`.
- Свойство `mustBeSettable` должно быть доступно как для чтения, так и для изменения, то есть в нем должны быть геттер и сеттер.
- Свойство `doesNotNeedToBeSettable`, в свою очередь, должно иметь как минимум геттер.

Требования доступности и изменяемости определяются с помощью конструкций `{ get }` и `{ get set }`. В первом случае у свойства должен быть как минимум геттер, а во втором — и геттер и сеттер. В случае, если свойству определено требование `{ get set }`, то оно не может быть вычисляемым «только для чтения» или константой.

Протокол определяет минимальные требования к типу, то есть тип данных обязан реализовать все, что описано в протоколе, но он может не ограничиваться этим набором. Так, для свойства `doesNotNeedToBeSettable` из предыдущего листинга может быть реализован не только геттер, но и сеттер (в протоколе содержится требование реализации геттера).

В листинге 30.2 показан пример реализации типом данных `SomeStruct` требований протокола `SomeProtocol`.

Листинг 30.2

```
struct SomeStruct: SomeProtocol {
    var mustBeSettable: Int
    let doesNotNeedToBeSettable: Int
    // дополнительный метод, не описанный в протоколе
    func getSum() -> Int {
        return self.mustBeSettable + self.doesNotNeedToBeSettable
    }
}
```

Тип данных `SomeStruct` полностью соответствует описанному ранее протоколу `SomeProtocol`, но при этом содержит дополнительный метод `getSum()`, который возвращает сумму свойств.

Для указания в протоколе требования к реализации свойства типа необходимо использовать модификатор `static` (листинг 30.3).

Листинг 30.3

```
protocol AnotherProtocol {
    static var someTypeProperty: Int { get }
}
```

Если тип данных подписывается на протокол `AnotherProtocol`, то в нем обязательно должно быть реализовано свойство типа `someTypeProperty` (листинг 30.4).

Листинг 30.4

```
struct AnotherStruct: SomeProtocol, AnotherProtocol {
    var mustBeSettable: Int
    let doesNotNeedToBeSettable: Int
    static var someTypeProperty: Int = 3
```

```
    func getSum() -> Int {
        return self.mustBeSettable
            + self.doesNotNeedToBeSettable
            + AnotherStruct.someTypeProperty
    }
}
```

Структура `AnotherStruct` принимает к реализации два протокола: `SomeProtocol` и `AnotherProtocol`. Это значит, что в ней должны быть реализованы все элементы обоих протоколов.

30.3. Требуемые методы

Помимо свойств, протокол может содержать требования к реализации одного или нескольких методов. Для требования реализации метода типа необходимо использовать модификатор `static`, а для изменяющего метода — `mutating`.

> **ПРИМЕЧАНИЕ** Если вы указали ключевое слово `mutating` перед требованием метода, то указывать его при реализации метода в классе уже не нужно. Данное ключевое слово требуется только при реализации структуры.

В листинге 30.5 показан пример объявления протокола, содержащего требование к реализации методов.

Листинг 30.5

```
protocol RandomNumberGenerator {
    var randomCollection: [Int] { get set }
    func getRandomNumber() -> Int
    mutating func setNewRandomCollection(newValue: [Int])
}
```

Протокол `RandomNumberGenerator` содержит требования реализации свойства `randomCollection` и двух методов: `getRandomNumber()` и `setNewRandomCollection(newValue:)`.

При реализации методов в объектном типе необходимо в точности соблюдать все требования протокола: имя метода, наличие или отсутствие входных аргументов, тип возвращаемого значения и модификаторы.

В листинге 30.6 показан пример создания структуры и класса, принимающих протокол `RandomNumberGenerator`.

Листинг 30.6

```
struct RandomGenerator: RandomNumberGenerator {
    var randomCollection: [Int] = [1,2,3,4,5]

    func getRandomNumber() -> Int {
        return randomCollection.randomElement()
    }
```

```
    mutating func setNewRandomCollection(newValue: [Int]) {
        self.randomCollection = newValue
    }
}

class RandomGeneratorClass: RandomNumberGenerator {
    var randomCollection: [Int] = [1,2,3,4,5]

    func getRandomNumber() -> Int {
        if let randomElement = randomCollection.randomElement() {
            return randomElement
        }
        return 0
    }

    // не используется модификатор mutating
    func setNewRandomCollection(newValue: [Int]) {
        self.randomCollection = newValue
    }
}
```

Оба объектных типа идентичны в плане функциональности, но имеют некоторые описанные выше различия в реализации требований протокола.

30.4. Требуемые инициализаторы

Протокол может предъявлять требования к реализации инициализаторов. При этом в классах можно реализовать назначенные (designated) или вспомогательные (convenience) инициализаторы. В любом случае перед объявлением инициализатора в классе 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 данного протокола (листинг 30.7).

Листинг 30.7

```
protocol Named {
    init(name: String)
}

class Person: Named {
    var name: String
    required init(name: String) {
        self.name = name
    }
}
```

30.5. Протокол в качестве типа данных

Протокол может выступать в качестве типа данных, то есть в определенных случаях вы можете писать не имя конкретного типа, а имя протокола, которому должно соответствовать значение. Рассмотрим несколько случаев.

Протокол, указывающий на множество типов

Протокол может выступать в качестве указателя на множество типов данных. С его помощью определяется требование к значению: оно должно иметь тип данных, соответствующий указанному протоколу. С этим подходом мы уже неоднократно встречались в процессе изучения материала книги, когда название протокола (например, `Hashable`) указывало на целую категорию типов.

Рассмотрим пример из листинга 30.8.

Листинг 30.8

```
func getHash<T: Hashable>(of value: T) -> Int {
    return value.hashValue
}
```

> **ПРИМЕЧАНИЕ** Приведенный в листинге код, возможно, покажется вам сложным, так как в нем используются не изученные ранее конструкции. Такой прием называется универсальным шаблоном (generic) и будет подробно рассмотрен в следующих главах.

Функция `getHash` может принять любое значение типа `T`, где тип `T` должен соответствовать протоколу `Hashable`. Таким образом, данный протокол указывает на целое множество типов данных. Так, вы можете передать в нее значение любого хешируемого типа и получить значение свойства `hashValue` (листинг 30.9).

Листинг 30.9

```
getHash(of: 5)
getHash(of: "Swift")
```

Протокол и операторы as? и as!

Операторы `as?` и `as!` уже знакомы вам — мы рассматривали их, когда изучали классы. Напомню, что эти операторы производят попытку приведения указанного *до* оператора значения к указанному *после* оператора типу данных.

Рассмотрим следующий пример: предположим, что у нас есть коллекция, которая может хранить элементы произвольных типов данных. При этом вы помещаете в нее значения как фундаментальных, так и собственных типов (листинг 30.10).

Листинг 30.10

```
protocol HasValue {
    var value: Int { get set }
}
```

```
class ClassWithValue: HasValue {
    var value: Int
    init(value: Int) {
        self.value = value
    }
}

struct StructWithValue: HasValue {
    var value: Int
}

// коллекция элементов
let objects: [Any] = [
    2,
    StructWithValue(value: 3),
    true,
    ClassWithValue(value: 6),
    "Usov"
]
```

Типы данных `StructWithValue` и `ClassWithValue` подписаны на протокол `HasValue`. Значения этих типов вперемешку со значениями других типов помещены в коллекцию `objects`.

Предположим, что теперь вам необходимо перебрать все элементы коллекции, выбрать из них те, что соответствуют протоколу `HasValue`, и вывести для них на консоль значение свойства `value`. Эту задачу позволит нам выполнить оператор `as?` (листинг 30.11).

Листинг 30.11

```
for object in objects {
    if let elementWithValue = object as? HasValue {
        print("Значение \(elementWithValue.value)")
    }
}
```

Консоль

```
Значение 3
Значение 6
```

Оператор `as?` пытается преобразовать значение к типу данных `HasValue` (протокол выступает в качестве типа). В случае успеха он выводит значение соответствующего свойства на консоль, а в случае неудачи возвращает `nil`.

Протокол и оператор is

Вы можете использовать протоколы совместно с оператором `is` для проверки соответствия типа данных значения этому протоколу. В листинге 30.12 приведен пример, использующий коллекцию элементов различных типов.

Листинг 30.12

```
for object in objects {
    print(object is HasValue)
}
```

Консоль

```
false
true
false
true
false
```

Если проверяемый элемент соответствует протоколу, то при проверке соответствия возвращается значение `true`, и `false` — в ином случае.

30.6. Наследование протоколов

Протокол может наследовать один или более других протоколов. При этом в него могут быть добавлены новые требования поверх наследуемых — тогда тип, принявший протокол к реализации, будет вынужден выполнить требования всех протоколов в иерархии. При наследовании протоколов используется тот же синтаксис, что и при наследовании классов.

Работа с наследуемыми протоколами показана в листинге 30.13.

Листинг 30.13

```
import Foundation
protocol GeometricFigureWithXAxis {
    var x: Int { get set }
}

protocol GeometricFigureWithYAxis {
    var y: Int { get set }
}

protocol GeometricFigureTwoAxis: GeometricFigureWithXAxis,
GeometricFigureWithYAxis {
    let distanceFromCenter: Float { get }
}

struct Figure2D: GeometricFigureTwoAxis {
    var x: Int = 0
    var y: Int = 0
    var distanceFromCenter: Float {
        let xPow = pow(Double(self.x), 2)
        let yPow = pow(Double(self.y), 2)
        let length = sqrt(xPow + yPow)
        return Float(length)
    }
}
```

Протоколы `GeometricFigureWithXAxis` и `GeometricFigureWithYAxis` определяют требование на наличие свойства, указывающего на координату объекта на определенной оси. В свою очередь, протокол `GeometricFigureTwoAxis` объединяет требования двух вышеназванных протоколов, а также вводит дополнительное свойство. В результате структура `Figure2D`, принимающая к реализации протокол `GeometricFigureTwoAxis`, должна иметь все свойства, описанные во всех трех протоколах.

30.7. Классовые протоколы

Вы можете ограничить применение протокола исключительно на классы, запретив его использование для структур и перечислений. Для этого после имени протокола через двоеточие необходимо указать ключевое слово `class`, после которого могут быть определены родительские протоколы.

В листинге 30.14 приведен пример создания классового протокола `SubProtocol`.

Листинг 30.14

```
protocol SuperProtocol { }
protocol SubProtocol: class, SuperProtocol { }

class  ClassWithProtocol: SubProtocol { } // корректно
struct StructWithProtocol: SubProtocol { } // ошибка
```

> **ПРИМЕЧАНИЕ** Протоколы в листинге выше не имеют каких-либо элементов. Это сделано умышленно с целью акцентировать внимание именно на способе объявления протокола и создания на его основе объектного типа.

30.8. Композиция протоколов

В случаях, когда протокол выступает в качестве указателя на множество типов данных, бывает удобнее требовать, чтобы тип данных используемого значения соответствовал не одному, а нескольким протоколам. В этом случае можно пойти двумя путями:

1. Создать протокол, который подписывается на два родительских протокола, и использовать его в качестве указателя на тип данных.
2. Использовать *композицию протоколов*, то есть комбинацию нескольких протоколов.

СИНТАКСИС

```
Протокол1 & Протокол2 ...
```

Для композиции необходимо указать имена входящих в нее протоколов, разделив их оператором & (амперсанд).

В листинге 30.15 приведен пример, в котором два протокола комбинируются в единое требование.

Листинг 30.15

```
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)
```

Консоль

```
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.

Глава 31. Расширения

Расширения (*extension*) позволяют добавить новую функциональность к уже существующему объектному типу (классу, структуре, перечислению) или протоколу. Таким образом вы можете расширять возможности любых типов, даже тех, что входят в состав Swift, то есть таких типов данных, доступ к исходным кодам которых у вас отсутствует. Расширения — это прекрасный и очень полезный функциональный элемент языка!

С помощью расширений вы можете:

- добавить **вычисляемые** свойства экземпляра и **вычисляемые** свойства типа (`static`) к объектному типу;
- добавить методы экземпляра и методы типа к объектному типу;
- добавить новые инициализаторы, сабскрипты и вложенные типы;
- подписать тип данных на протокол и обеспечить выполнение его требований.

Расширения могут добавлять новую функциональность к типу, но не могут изменять существующую. 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.

31.1. Вычисляемые свойства в расширениях

Расширения могут добавлять вычисляемые свойства экземпляра и вычисляемые свойства типа в уже существующий объектный тип.

ПРИМЕЧАНИЕ Расширения могут добавлять только новые вычисляемые свойства. При попытке добавить хранимые свойства или наблюдателей свойств происходит ошибка.

Рассмотрим следующий пример. Ваша программа оперирует расстояниями и использует для этого значения типа `Double`. По умолчанию такое значение определяет расстояние в метрах, но вам требуется организовать оперативный перевод метров в другие единицы измерения расстояний. Для этого расширим тип данных `Double` и добавим в него несколько специальных вычисляемых свойств (листинг 31.1).

Листинг 31.1

```
extension Double {
    var asKM: Double { return self / 1000.0 }
    var asM: Double { return self }
    var asCM: Double { return self * 100.0 }
    var asMM: Double { return self * 1000.0 }
}
```

Теперь при необходимости перевести расстояние из метров в другие единицы мы просто воспользуемся одним из свойств (листинг 31.2).

Листинг 31.2

```
let length: Double = 25 // 25 метров
length.asKM // расстояние 25 метров в километрах - 0.025
length.asMM // расстояние 25 метров в миллиметрах – 25000
```

Применение геттеров и сеттеров для вычисляемых свойств позволит использовать их возможности по максимуму (листинг 31.3).

Листинг 31.3

```
extension Double {
    var asFT: Double {
        get {
            return self / 0.3048
        }
        set(newValue) {
            self = newValue * 0.3048
        }
    }
}

var distance: Double = 100 // расстояние 100 метров
distance.asFT // расстояние 100 метров в футах - 328.08 фута
// установим расстояние в футах, но оно будет сохранено в метрах
distance.asFT = 150 // 45.72 метра
```

31.2. Методы в расширениях

Расширения могут добавлять в объектные типы не только свойства, но и методы. Рассмотрим пример из листинга 31.4, в котором расширяется тип `Int`. В нем

появился новый 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 31.4

```
extension Int {
    func repetitions(task: () -> ()) {
        for _ in 0..<self {
            task()
        }
    }
}
3.repetitions {
    print("Swift")
}
```

Консоль

```
Swift
Swift
Swift
```

Для изменения свойств перечислений (`enum`) и структур (`struct`) не забывайте использовать модификатор `mutating`. В листинге 31.5 в расширении объявляется 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 31.5

```
extension Int {
    mutating func squared() {
        self = self * self
    }
}
var someInt = 3
someInt.squared() // 9
```

31.3. Инициализаторы в расширениях

Благодаря расширениям появляется возможность добавлять новые инициализаторы к существующему объектному типу. Так, вы можете расширить типы, например, 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 приведет к ошибке.

В качестве примера создадим структуру `Line`, описывающую линию на плоскости. С помощью расширения реализуем инициализатор типа `Double`, принимающий на

вход экземпляр данной структуры и устанавливающий значение, соответствующее длине линии (листинг 31.6).

Листинг 31.6

```
import Foundation

struct Line {
    var pointOne: (Double, Double)
    var pointTwo: (Double, 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
```

Импортированная в первой строке листинга библиотека Foundation обеспечивает доступ к математическим функциям `sqrt(_:)` и `pow(_:_:)` (вычисление квадратного корня 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, чтобы к завершению инициализации каждое из свойств расширяемого типа имело определенное значение. Например, если вы напишете расширение уже для типа `Line`, но в нем проинициализуете значение только для одного из свойств этого типа, это приведет к ошибке.

31.4. Сабскрипты в расширениях

Помимо свойств, методов и инициализаторов, расширения позволяют создавать новые сабскрипты.

Создаваемое в листинге 31.7 расширение типа `Int` реализует новый сабскрипт, который позволяет получить определенную цифру собственного значения экземпляра. В сабскрипте указывается номер позиции цифры, которую необходимо вернуть.

Листинг 31.7

```
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.

31.5. Расширения протоколов

Как вы знаете, расширения могут применяться к любым объектным типам. В этом разделе я бы хотел отдельно рассмотреть возможности совместного использования расширений и протоколов.

Подпись объектного типа на протокол

С помощью расширений у вас есть возможность подписать существующий тип на определенный протокол. Для этого в расширении после имени типа данных через двоеточие необходимо указать список новых протоколов. В листинге 31.8 тип данных `Int` подписывается на протокол `TextRepresentable`, который требует наличия метода `asText()`.

Листинг 31.8

```
protocol TextRepresentable {
    func asText() -> String
}

extension Int: TextRepresentable {
    func asText() -> String {
        return String(self)
    }
}

5.asText() // "5"
```

С помощью расширения мы добавляем требование о соответствии типа протоколу, при этом обязательно указывается реализация метода.

Расширение протоколов и реализации по умолчанию

Расширения могут расширять не только конкретные типы данных, но и протоколы. При этом они позволяют указать реализацию по умолчанию для любого метода этого протокола (листинг 31.9).

Листинг 31.9

```
protocol Descriptional {
    func getDescription() -> String
}

// расширение протокола и указание реализации метода по умолчанию
extension Descriptional {
    func getDescription() -> String {
        return "Описание объектного типа"
    }
}

// подпишем класс на протокол
class myClass: Descriptional {}
// вызовем метод
print(myClass().getDescription())
```

Консоль

```
Описание объектного типа
```

Несмотря на то что в классе `myClass` отсутствует реализация метода `getDescription()`, при его вызове не появляются сообщения об ошибках. Это связано с тем, что протокол имеет реализацию метода по умолчанию, описанную в его расширении.

> **ПРИМЕЧАНИЕ** Реализация методов по умолчанию для протоколов доступна только при использовании расширений. Вы не можете наполнить метод непосредственно при объявлении протокола.

При этом вы всегда можете переопределить реализацию по умолчанию непосредственно в самом объектном типе (листинг 31.10).

Листинг 31.10

```
class myStruct: Descriptional {
    func getDescription() -> String {
        return "Описание структуры"
    }
}
myStruct().getDescription() // "Описание структуры"
```

> **ПРИМЕЧАНИЕ** Реализация методов по умолчанию, а также множественное наследование протоколов являются основой методологии протокол-ориентированного программирования, о котором уже упоминалось ранее в книге.

С помощью расширений вы можете в том числе расширять и сами протоколы.

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 реализации.

Напишем расширение для реализованного ранее протокола `TextRepresentable` (листинг 31.11).

Листинг 31.11

```
extension TextRepresentable {
    func about() -> String {
        return "Данный тип поддерживает протокол TextRepresentable"
    }
}

// Тип Int уже подписан на протокол TextRepresentable
5.about()
```

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`.

Знание и правильное использование протоколов — очень важный навык, так как Swift является первым протокол-ориентированным языком программирования. В следующей главе мы поговорим о преимуществах, которые приносит использование протоколов.

Глава 32. Протокол-ориентированное программирование

В этой главе я бы хотел поговорить о методологии разработки приложений «Протокол-ориентированное программирование» (ПОП), а также рассмотреть несколько сопутствующих вопросов. Глава станет для вас введением в ПОП. По мере того как вы будете накапливать опыт разработки, вы сможете самостоятельно вернуться к темам, рассмотренным в данной главе, и взглянуть на них уже под другим углом.

32.1. Важность использования протоколов

Протокол (или *интерфейс*) — это понятие, которое относится не только к области программирования. Вы могли встречать протоколы (интерфейсы) повсюду, начиная от работы в Сети (сетевые протоколы — HTTP, FTP, SSL и т. д.) до повседневной жизни. Так, например, электрическая розетка — это также определенный интерфейс, с которым у вас есть возможность взаимодействовать только определенным образом (вставить вилку в розетку). И если в вилке расстояние между контактами будет больше или количество контактов будет другое, то взаимодействия не произойдет. Таких примеров множество: пульт от телевизора, духовка, кухонный нож — каждый из них имеет свой интерфейс, или стандарт, через который вы взаимодействуете с данным объектом. И стоит отойти от правил, требуемый результат не будет достигнут. Другой пример: вы нажимаете на кнопку включения PS5, и вам неважно, как именно эта команда будет отработана, важен лишь результат — консоль перешла в активный режим.

Интерфейс — это стандарт, описывающий порядок взаимодействия с объектом. Иными словами, это набор требований, которым должен удовлетворять объект, чтобы с ним можно было производить взаимодействие установленным образом.

> **ПРИМЕЧАНИЕ** Зачастую хакеры занимаются поиском уязвимостей в протоколах, то есть стараются найти способы взаимодействия, не предусмотренные стандартом. Так, если вместо вилки вы вставите в розетку пальцы, то в некотором роде можете почувствовать себя хакером, так как используете интерфейс не по назначению.

> На всякий случай уточню: это была шутка, не стоит искать уязвимости подобного рода, особенно в физических интерфейсах.

Протокол (`protocol`) в Swift подобен интерфейсу в реальной жизни: с его помощью вы можете определить, какие элементы доступны вам при взаимодействии с тем или иным объектом, не вдаваясь в конкретную реализацию. К примеру, вы работаете с объектом типа `Car`, подписанным на протокол `Drive`, в котором есть требования реализации метода `driveForward(withSpeed:)`, а значит, данный метод доступен и в самом объекте.

Есть три важнейшие причины использования протоколов. В протоколах обеспечивается:

1) целостность типов данных;
2) инкапсуляция;
3) полиморфизм.

Целостность типов данных

Использование протоколов обеспечивает целостность типов данных. Другими словами, если некоторый объектный тип подписан на протокол, то с уверенностью можно утверждать, что в нем реализованы требования данного протокола.

Сам по себе тип не обязан иметь какую-либо функциональность, но протокол, в свою очередь, обязывает его. И если в протоколе есть требование реализовать конкретное свойство или метод, то оно однозначно будет доступно при работе с типом. Например, если тип реализует требования протокола `Hashable`, то вы сможете получить конкретное уникальное целочисленное значение при доступе к свойству `hashValue`, так как в протоколе есть соответствующее требование.

Инкапсуляция

Инкапсуляция — это подход, при котором данные и методы для работы с этими данными объединяются в единую сущность, при этом скрывая свою внутреннюю реализацию.

Рассмотрим пример. Сущность «Телевизор» в реальном мире объединяет данные (телевизионных программ) и методы (переключение программ, изменение громкости и т. д.). При этом вы совсем ничего не знаете, как все это функционирует, вы лишь используете интерфейс телевизора (нажимаете кнопки на пульте) и получаете результат. Таким образом, можно сказать, что в данном объекте инкапсулируются данные и методы.

В листинге 32.1 приведен пример того, как бы мог выглядеть телевизор в виде программного кода.

Листинг 32.1

```
// сущность ТВ-шоу
class TVShow {
    // ...
}

// протокол, описывающий функциональность работы с ТВ-шоу
protocol DisplayShow {
    func getShowsList() -> [TVShow]
    func changeShowTo(show: TVShow) -> Void
    var currentShow: TVShow { get }
    // дополнительные требования ...
}

// протокол, описывающий функциональность управления звуком
protocol ChangeVolume {
    var currentVolume: UInt8 { get set }
    func increase(by: UInt8) -> Void
    func decrease(by: UInt8) -> Void
    // дополнительные требования...
}

// сущность Телевизор принимает протоколы к исполнению
struct TV: DisplayShow, ChangeVolume {
    // реализация методов и свойств протоколов ...
}
```

Если бы структура `TV` не принимала протокол `ChangeVolume`, то у нас не было бы гарантии, что она реализует управление громкостью. Если бы структура `TV` не принимала протокол `DisplayShow`, то у нас не было бы гарантии, что она реализует показ телевизионных передач.

Таким образом, если всегда начинать разработку класса или структуры с протоколов, то сначала вы определяетесь с тем, что должны делать ваши программные сущности, а уже потом создаете их реализацию.

Полиморфизм

Полиморфизм — это подход, предусматривающий возможность взаимодействия с различными типами данных единым образом в отсутствие информации о конкретном типе данных объекта.

Например, мы могли бы разработать структуру, описывающую DVD-плеер, и так как данное устройство может управлять звуком, то используем при этом объявленный ранее протокол `ChangeVolume` (листинг 32.2).

Листинг 32.2

```
struct DVDPlayer: ChangeVolume {
    // реализация методов и свойств протоколов ...
}
```

Теперь предположим, что мы разработали универсальный пульт, который может менять громкость любого устройства, имеющего интерфейс управления звуком (соответствует протоколу `ChangeVolume`) (листинг 32.3).

Листинг 32.3

```
struct UniversalManager {
    var currentDevice: ChangeVolume
    func increaseVolume(by: UInt8) -> Void {
        self.currentDevice.increase(by: by)
    }
    // ...
}

// начнем работу с DVD-плеером
let manager = UniversalManager(currentDevice: DVDPlayer())
manager.increaseVolume(by: 1)

// переключимся на работу с телевизором
manager.currentDevice = TV()
manager.increaseVolume(by: 5)
```

Таким образом, вы можете настроить значение типа `UniversalManager` хоть на работу с `TV`, хоть на работу с `DVDPlayer`. В этом и есть суть полиморфизма. Наш универсальный пульт может взаимодействовать с любым объектом, не имея информации о его конкретном типе.

Конечно, можно добиться подобного результата и без протоколов, с помощью наследования классов. Но Swift не поддерживает множественное наследование классов, зато приветствует его для протоколов.

ПРИМЕЧАНИЕ Множественное наследование — это наследование от двух и более родителей.

32.2. Протокол-ориентированное программирование

В этой книге я не ставил перед собой целей обучить вас подходам объектно-ориентированного и протокол-ориентированного программирования (ООП и ПОП). Для того чтобы глубоко понять их и начать применять, потребуется получить опыт разработки реальных приложений на Swift в среде Xcode. Тем не менее это не помешает нам кратко ознакомиться с некоторыми элементами этих методологий.

Основные принципы ООП — это:

- наследование;
- инкапсуляция;
- полиморфизм.

С каждым из этих понятий мы уже неоднократно встречались в процессе изучения материала книги. При этом ПОП основан на ООП, но на данный момент не имеет четкого и конкретного описания. Впервые об этой методологии упомянули на конференции WWDC 15, где Swift был назван первым протокол-ориентированным языком в истории программирования. Но зачем? Почему Swift не может в полной мере использовать старый добрый объектно-ориентированный подход? На то есть две основные причины.

Причина № 1. В Swift отсутствует множественное наследование.

Классы могут наследоваться только от одного родителя, а структуры вообще не могут иметь родительские объектные типы. Но при этом у вас есть возможность подписывать классы и структуры на множество протоколов.

Вернемся к рисунку из главы «Последовательности и коллекции» (рис. 32.1, схема перевернута для лучшего восприятия). На схеме приведены протоколы, которым соответствует тип данных `Int`. Данный тип одновременно выполняет требования множества протоколов: и `Hashable`, и `Equatable`, и `Comparable` и т. д.

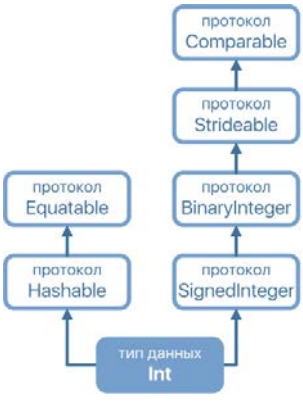

Рис. 32.1. Соответствие типа Int протоколам

Если провести глубокий анализ, то станет понятно, что соответствия фундаментальных типов и протоколов — это гигантская паутина. Каждый тип данных соответствует множеству протоколов, которые, в свою очередь, также наследуют другие протоколы и т. д. Таким образом, Swift позволяет использовать множественное наследование, но только при работе с протоколами.

Причина № 2. Протоколы позволяют содержать реализацию по умолчанию.

Как уже неоднократно говорилось, в Swift протокол может быть не просто набором требований, но также содержать их реализацию. Таким образом, мы можем создать множество протоколов, при необходимости указать реализацию по умолчанию и при создании объектного типа подписать его на данные протоколы, с ходу обеспечив тип функциональной составляющей. Не написав ни строчки кода в теле объектного типа, вы наделяете его возможностями!

С помощью протоколов принцип «наследование» из ООП в Swift реализуется иначе:

- В ООП классы наследуют друг друга, и один дочерний класс может иметь множество родительских классов.
- В ПОП (в Swift) класс (или структура) наследует множество протоколов, которые, возможно, помимо самих требований, содержат еще и их собственную реализацию.

Суть протокол-ориентированного программирования состоит в том, что в первую очередь описываются протоколы, а уже потом объектные типы, принимающие их к реализации.

Рассмотрим еще один пример.

Предположим, что вам необходимо описать сущность «Утка» в своей программе. В первую очередь выделим категории, в которые данная сущность потенциально может входить вместе с другими сущностями. Утка умеет летать — значит, входит в категорию «Летающие», умеет плавать — значит, входит в категорию «Плавающие», и умеет ходить, значит, входит в категорию «Ходячие». Основываясь на этом, создадим три протокола и структуру, описывающую сущность «Утка» (листинг 32.4).

Листинг 32.4

```
protocol Walking {}
protocol Flying {}
protocol Swimming {}
struct Duck: Walking, Flying, Swimming {}
```

Прелесть такого подхода в том, что если вам потребуется создать другую сущность, к примеру «Пингвин», то вы с легкостью подпишете ее только на те протоколы (то есть включите только в те группы), которым данная сущность должна соответствовать (листинг 32.5)

Листинг 32.5

```
struct Penguin: Walking, Swimming {}
```

О том, насколько хорош подход ПОП, вы сможете понять позже, уже после изучения основ программирования на Swift. Сейчас мне важно, чтобы вы получили ознакомительные сведения об этой методологии и в дальнейшем при чтении книг и статей не впадали в ступор, а имели определенное представление.

32.3. Где использовать class и struct

Как мы уже выяснили, первый элемент любой сущности — это протокол. Именно с него необходимо начинать разработку. На основе протоколов создаются объектные типы данных, и для этого вам доступны перечисления (`enum`), классы (`class`) и структуры (`struct`). С предназначением перечислений мы разобрались ранее в книге, но что делать в остальных случаях? Что использовать для реализации каждой конкретной сущности?

Классы и структуры — очень гибкие конструкции Swift. Наполняя свой код функциональными возможностями, вы непременно будете встречаться с ситуацией, в которой неясно, какими средствами лучше решать возникшую задачу. При этом

любая из них может быть решена как с использованием классов, так и с использованием структур. В этом разделе мы поговорим с вами о выборе между `class` и `struct` при написании программного кода.

Правила выбора между классом и структурой

При создании очередной сущности в вашем приложении придерживайтесь следующих правил:

Правило № 1

Отдавайте предпочтение структурам, нежели классам.

Правило № 2

Используйте классы только тогда, когда вам действительно нужны их особенности.

Звучит довольно просто и, вероятно, совершенно непонятно. Но что скрывается за этими правилами?

Ответ на вопрос «`class` или `struct`?» кроется в отличиях между этими конструкциями. Какое самое важное отличие вы знаете? Структуры — это value type (значимые типы), а классы — это reference type (ссылочные типы).

Рассмотрим пример. В листинге 32.6 созданы класс и структура, описывающие пользователя.

Листинг 32.6

```
class UserClass {
    var name: String
    init(name: String) {
        self.name = name
    }
}

struct UserStruct {
    var name: String
}
```

При создании параметров типа `UserClass` и `UserStruct` мы получим одинаковые по своей сути экземпляры, содержащие одноименные свойства (листинг 32.7).

Листинг 32.7

```
var userByClass = UserClass(name: "Vasiliy")
var userByStruct = UserStruct(name: "Vasiliy")
```

Но мы почувствуем разницу, когда проинициализируем значения `userByClass` и `userByStruct` новым параметрам (листинг 32.8).

Листинг 32.8

```
var newUserByClass = userByClass
var newUserByStruct = userByStruct
```

Если вы не пропустили главу, посвященную управлению памятью, то уже должны знать, чем именно отличаются ссылочные типы от значимых, и должны догадаться, что произойдет. В первом случае мы получим два параметра (`userByClass` и `newUserByClass`), которые указывают на один и тот же объект в памяти. Во втором случае мы получим два независимых значения (`userByStruct` и `newUserByStruct`) (рис. 32.2).

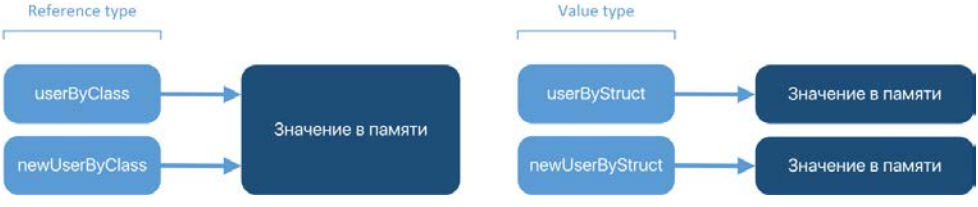

Рис. 32.2. Ссылочные и значимые типы

Данная особенность является основой выбора между структурой и классом. Тем не менее правило № 1 говорит о том, что мы должны предпочесть структуры классам. Но почему?

> **ПРИМЕЧАНИЕ** Стоит отметить, что у утверждения «структуры — это value type» есть одно исключение. При захвате значения структуры внутри замыкания значение по факту передается по ссылке, а не копируется, если иное не указано явно. Мы говорили об этом в главе об управлении памятью.

Причина выбрать структуру № 1. Безопасность

Структуры безопаснее классов. Множественные копии объекта, в отличие от множественных ссылок, не способны привести к случайному, неконтролируемому изменению данных. Представьте, что вы используете классы для некоторой сущности в своей программе, а значит, передаете ее значение по ссылке. Существует вероятность, что в один из моментов значение данной сущности будет изменено, а это, в свою очередь, может привести к ошибкам в другой части программы, где работа должна вестись со старой, неизмененной версией экземпляра. Особенно это важно при работе в среде мультипоточности (multitheading).

> **ПРИМЕЧАНИЕ** Swift поддерживает мультипоточность, благодаря которой у вас есть возможность исполнять программный код в несколько потоков, то есть выполнять несколько задач одновременно. К примеру, в одном потоке вы можете отправить запрос к удаленному серверу, и чтобы не замораживать графический интерфейс приложения в ожидании ответа, в другом потоке продолжить обработку действий пользователя (нажатие кнопок, свайпы и т. д.).
>
> Мультипоточность — это очень интересная, но довольно серьезная тема, с которой необходимо знакомиться, уже имея некоторый опыт в создании приложения. В следующих книгах мы обязательно подробно разберем ее.

Причина выбрать структуру № 2. Утечки памяти

В главе об управлении памятью нами был разобран пример утечки памяти. При использовании структур такая проблема исчезает, опять же по причине того, что значения передаются копией, а не в виде ссылки.

Причина выбрать структуру № 3. Раздутые объектные типы

Swift не поддерживает множественное наследование. Таким образом, у вас отсутствует возможность создать класс — наследник нескольких родительских классов. И как уже отмечалось, для структур наследование вообще недоступно. Это побуждает создавать огромные раздутые типы-комбайны, которые одновременно способны решать большое количество задач. С точки зрения Best Practice (лучшие практики) это плохой подход к разработке. Каждая отдельная сущность должна решать строго определенную задачу. Создание «комбайнов» является плохим приемом программирования.

> **ПРИМЕЧАНИЕ** Во второй книге мы подробнее поговорим о дроблении зон ответственности с целью создания компактных классов и структур, отвечающих за решение одной конкретной задачи.

При использовании приемов протокол-ориентированного программирования и создании протоколов, наполненных не просто требованиями к реализации свойств и методов, а непосредственно их реализацией, у вас появляется первоклассный аналог множественного наследования. Это может оказаться очень удобным способом наращивания функциональности ваших сущностей.

Причина выбрать структуру № 4. Структуры быстрее классов

Проведем простой эксперимент. Создадим аналогичные сущности с помощью класса и структуры и протестируем время выполнения 100 миллионов операций над ними (листинг 32.9).

Листинг 32.9

```
import Foundation

class ValueClass {
    var value: Int = 0
}

struct ValueStruct {
    var value: Int = 0
}

var operationArray = Array(1...100_000_000)

// опыт с классом
var startTime = Date().timeIntervalSince1970
var a = ValueClass()

for _ in operationArray {
```

```
    a.value += 1
}

var finishTime = Date().timeIntervalSince1970
print("Время выполнения операций с классом - \(finishTime-startTime)")

// опыт со структурой
startTime = Date().timeIntervalSince1970
var b = ValueStruct()

for _ in operationArray {
    b.value += 1
}

finishTime = Date().timeIntervalSince1970
print("Время выполнения операций со структурой - \(finishTime-startTime)")
```

Консоль

```
Время выполнения операций с классом - 4.297344923019409
Время выполнения операций со структурой - 0.07223701477050781
```

> **ПРИМЕЧАНИЕ** Тип `Date` ранее не рассматривался и не использовался в книге. Он предназначен для работы с датой и временем. В данном случае с его помощью получается число, описывающее количество секунд, прошедших с 1 января 1970 года, так называемое Unix-время. Для его работы требуется импортировать библиотеку Foundation.

Как вы можете видеть из вывода на консоли, выполнение операций со структурой происходит в 60 раз быстрее, чем с классом. Причина этого кроется в организации хранения и доступа к классам и структурам в памяти компьютера.

> **ПРИМЕЧАНИЕ** Вполне вероятно, что конкретное время выполнения, которое покажет ваш компьютер, будет отличаться от того, что получилось у меня. Более того, Xcode Playground на моем Mac так и не смог выполнить этот код. По этой причине мне пришлось проводить тесты на Ubuntu с установленным компилятором Swift.

Все вышесказанное не означает, что вы всегда должны использовать структуры. Классы существуют не просто ради того, чтобы они были, у них также есть своя область применения.

Причина выбрать класс № 1. Логичная передача объекта по ссылке

Стоит использовать класс в тех случаях, когда однозначно понятно, что лучше передавать объект по ссылке, нежели копировать его. Примерами могут служить файловые дескрипторы и сетевые подключения, открытые в ваших приложениях.

Также с помощью классов могут быть реализованы различные контроллеры, производящие контроль/управление объектами, когда объект — это просто менеджер, для которого нет необходимости создавать копии. Представьте, что в вашем приложении есть тип `User`, описывающий пользователя, а также тип `UserController`, производящий контроль над пользователями (значениями типа `User`).

Подумайте, что лучше использовать для каждого объектного типа:

- Для реализации типа `User` имеет смысл использовать структуру, так как нам не нужны особенности классов.
- Тип `UserController` `UserController` — это контроллер, менеджер пользователей, производящий управление учетными записями, для которого бессмысленно использовать преимущества структуры. Вполне логично, если значение данного типа всегда будет передаваться по ссылке, более того, это не приведет к каким-либо описанным выше проблемам.

В листинге 32.10 приведена реализация данных типов.

Листинг 32.10

```
struct User {
    var id: Int
}

class UserController {
    var users: [User]
    func add(user: User) {
        // ...
    }
    func removeBy(userID: Int) {
        // ...
    }
    func loadFromStorageBy(userID: Int) -> User {
        // ...
    }
    init(user: User) {
        // ...
    }
}
```

Причина выбрать класс № 2. Совместимость с Objective-C

Если на ваши плечи ляжет необходимость поддержки старых проектов, написанных на Objective-C, или использования старых библиотек, то классы станут вашим верным другом.

> **ПРИМЕЧАНИЕ** В данной книге не рассматриваются примеры совместного использования Swift и Objective-C кода.

Со временем вы поймете реальное значение и отличия value type и reference type, и это позволит вам без труда выбирать необходимые элементы языка. Тем не менее уже сейчас, при создании каждой новой сущности задавайтесь вопросом, с помощью каких конструктивных элементов Swift она должна быть реализована.

Глава 33. Разработка приложения в Xcode Playground

Среда Xcode Playground прекрасно подходит для тестирования программного кода, так как для этого не требуется создания полноценного проекта. Ее возможности не ограничиваются выводом данных на консоль и в область результатов. С помощью дополнительных библиотек Playground позволяет работать даже с графическими и интерактивными элементами.

33.1. UIKit и SwiftUI

Библиотеки `UIKit` и `SwiftUI` станут вашими верными спутниками в процессе дальнейшего обучения. Изучение каждой из них будет интересным, полным открытий процессом, но потребует вдумчивости и максимальной отдачи.

> **ПРИМЕЧАНИЕ** Информация о UIKit и SwiftUI, приведенная в данной книге, является ознакомительной. Более подробному изучению этих библиотек посвящены следующие книги серии.

`UIKit` — это фреймворк, обеспечивающий ключевую инфраструктуру, необходимую для создания и функционирования iOS-приложений. `UIKit` содержит большое количество объектных типов данных, благодаря которым функционируют приложения, строится их графический интерфейс, обрабатывается анимация и события (например, прикосновение к экрану устройства или получение push-уведомления), рассчитывается физическое взаимодействие предметов и многое другое. Это важнейшая и совершенно незаменимая библиотека, которую вы будете использовать при разработке каждого приложения.

Несмотря на такой широкий перечень обязанностей, одной из наиболее используемых возможностей `UIKit` является создание графического интерфейса приложений: разработка его отдельных экранов и правил перехода между ними, размещение кнопок, надписей и картинок и т. д. Все это позволяет делать `UIKit`.

> **ПРИМЕЧАНИЕ** Обратите внимание, что UIKit используется только при разработке приложений под iOS. В рамках операционной системы macOS работает фреймворк AppKit, но его рассмотрение выходит за рамки учебного материала книги.

`SwiftUI` — это новая библиотека от Apple, которая пришла на замену `UIKit` в части, касающейся создания графического интерфейса. Таким образом, iOS-приложения

все еще функционируют на основе `UIKit`, но графическая составляющая может быть разработана как средствами `UIKit`, так и средствами `SwiftUI` либо при их совместном использовании.

Библиотека `SwiftUI` предлагает совершенно новый способ создания интерфейса приложений. Основным отличием `SwiftUI` от `UIKit` (в части построения графического интерфейса) является декларативный подход вместо императивного. В следующих главах мы более детально рассмотрим отличия `UIKit` и `SwiftUI` и разберемся, что же такое декларативный подход.

> **ПРИМЕЧАНИЕ** Довольно часто у участников нашего сообщества в Telegram возникает вопрос о том, что изучать: UIKit или SwiftUI. Запомните, что вы не сможете избежать изучения UIKit, так как в отличие от SwiftUI, этот фреймворк отвечает не только за построение графического интерфейса. Перечень его функций намного шире. Более того, SwiftUI поддерживается, начиная только с iOS 13.
>
> Актуальная ссылка на чат сообщества в мессенджере Telegram всегда размещена на сайте https://swiftme.ru.

33.2. Разработка интерактивного приложения

Теперь мы перейдем к разработке небольшого интерактивного приложения прямо в Xcode Playground, которое позволит нам попробовать `UIKit` в деле. Приложение будет отображать прямоугольную площадку с размещенными на ней цветными шариками. При перемещении шариков (с помощью мыши) будет обрабатываться физика их столкновения друг с другом и с границами площадки. Для создания такого графического интерактивного интерфейса будет использоваться `UIKit`.

> **ПРИМЕЧАНИЕ** Данный playground-проект может быть назван приложением лишь условно, так как вы не сможете провести его компиляцию и сохранить в виде отдельного исполняемого файла. Для его запуска всегда потребуется открывать Xcode Playground.

- Создайте новый playground-проект типа `Blank`. Назовите его `Balls` и удалите весь программный код, присутствующий в нем.

Библиотека PlaygroundSupport

Ранее, при работе с Xcode Playground, мы не использовали какие-либо интерактивные элементы, только выводили значения на консоль. Но благодаря Interactive playground (интерактивная игровая площадка) у нас есть возможность создания интерактивных графических элементов прямо в окне playground-проекта. Работу Interactive playground обеспечивает библиотека `PlaygroundSupport`, наиболее важным элементом которой является класс `PlaygroundPage`.

В созданный ранее проект Balls импортируйте библиотеки `PlaygroundSupport` и `UIKit` (листинг 33.1).

Листинг 33.1

```
import PlaygroundSupport
import UIKit
```

Структура проекта

В первую очередь необходимо продумать, каким образом и с помощью каких элементов языка будет реализован графический интерфейс. Поговорим об этом подробнее.

Одним из важнейших элементов `UIKit` является класс `UIView`, предназначенный для создания графических элементов. На его основе базируются все доступные по умолчанию элементы (кнопки, надписи, текстовые поля и т. д.). Таким образом, на его основе можно создать любой графический элемент, в том числе прямоугольную площадку с шариками. Стоит обратить внимание, что сущности «Шарик» и «Площадка» — это два независимых элемента, каждый из которых будет создан на основе `UIView`:

- Для реализации шарика создадим класс `Ball`, потомок класса `UIView`.
- Для реализации площадки создадим класс `SquareArea`, потомок класса `UIView`.

Любой графический элемент, основанный на `UView`, называется **отображением**, или **представлением**. В дальнейшем мы часто будем использовать эти понятия.

При создании любого элемента программы вы должны решить, с помощью каких конструкций языка он должен быть реализован (структура, класс, перечисление и т. д.). В данном случае мы делаем «выбор без возможности выбора» использовать классы для обеих сущностей, так как они будут основаны на классе `UIView`.

> **ПРИМЕЧАНИЕ** Напомню, что структуры не поддерживают наследование, и в частности, не могут быть дочерними элементами класса UIView.

Для того чтобы логически разделить элементы программы, вынесем реализацию каждой сущности в отдельный файл, так чтобы:

- в главном файле playground-проекта размещался код, отображающий графический интерфейс (с помощью класса `PlaygroundSupport`);
- в дополнительных файлах размещался код, описывающий графические элементы (классы `SquareArea` и `Ball`).

На панели Project Navigator в папке Source создайте два новых файла с именами SquareArea.swift и Ball.swift. Для этого щелкните по названию папки правой кнопкой мыши, выберите пункт New File и укажите имя создаваемого файла. В результате файлы отобразятся в составе папки Sources (рис. 33.1).

> **ПРИМЕЧАНИЕ** Xcode Playground все еще периодически работает некорректно. Если в процессе работы над проектом вы увидите ошибку `Cannot find type 'Ball' in scope` или подобную, то пишите весь программный код в одном файле.

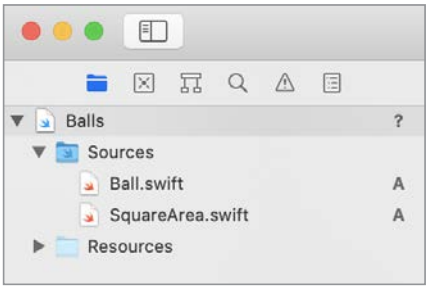

Рис. 33.1. Структура проекта в Project Navigator

Класс Ball

Реализацию программы начнем с создания шарика. В Project Navigator выделите файл Ball.swift, таким образом содержимое отобразится в центральной области Xcode playground. Вместо библиотеки `Foundation` импортируйте `UIKit`.

Реализацию сущности «Шарик» начнем с создания протокола — добавьте его реализацию в соответствии с листингом 33.2.

Листинг 33.2

```
import UIKit

protocol BallProtocol {
    init(color: UIColor, radius: Int, coordinates: (x: Int, y: Int))
}
```

> **ПРИМЕЧАНИЕ** О важности использования протоколов и о том, почему мы начали реализацию именно с протокола, а не с класса, мы уже говорили в предыдущих главах. Вероятно, для вас эта тема все еще остается не до конца понятной. Со временем, получая опыт и читая тематическую литературу, вы поймете необходимость протоколов и принципы объектно-ориентированного и протокол-ориентированного программирования.

Протокол `BallProtocol` предусматривает наличие в принимаемом типе только инициализатора, в который передаются следующие аргументы:

- `color` — определяет цвет шарика, представленный классом `UIColor`. Это встроенный в `UIKit` объектный тип, предназначенный специально для работы с цветом.
- `radius` — определяет радиус шарика в виде целого числа (тип `Int`).
- `coordinates` — определяет координаты шарика, представленные в виде кортежа.

Теперь объявим новый класс `Ball`, потомок `UIView` (листинг 33.3).

Листинг 33.3

```
public class Ball: UIView, BallProtocol {}
```

В соответствии с требованиями протокола `BallProtocol`, класс `Ball` должен иметь инициализатор. Добавьте его в класс в соответствии с листингом 33.4.

Листинг 33.4

```
required public init(color: UIColor, radius: Int, coordinates: (x:Int, y: Int)) {
    // создание графического прямоугольника
    super.init(frame:
        CGRect(x: coordinates.x,
               y: coordinates.y,
               width: radius * 2,
               height: radius * 2))
    // скругление углов
    self.layer.cornerRadius = self.bounds.width / 2.0
    // изменение цвета фона
    self.backgroundColor = color
}
```

Данный инициализатор должен быть требуемым (`required`, мы говорили об этом при изучении протоколов), а также публичным (`public`, так как по умолчанию он `internal`). В инициализаторе вызывается родительский инициализатор `init(frame:)`, который входит в состав класса `UIView`. С его помощью создается графический элемент «Прямоугольник», характеристики которого передаются в объекте типа `CGRect` (координаты левого верхнего угла, ширина и высота).

> **ПРИМЕЧАНИЕ** На данном этапе мы создали лишь программный аналог прямоугольника, который в дальнейшем будет дополнительно настроен и выведен на экран. Воспринимайте класс UIView как шаблон для создания графических элементов. С его помощью вы программно описываете элемент и в дальнейшем выводите его на экран.

Далее с помощью цепочки свойств `layer.cornerRaduis` происходит скругление углов прямоугольника, и он превращается в круг.

Свойство `backgroundColor` позволяет установить цвет создаваемого объекта.

Данный класс также должен иметь специальный инициализатор `init?(coder:)`, о чем вам, возможно, уже сказал Xcode в сообщении об ошибке. Реализуйте его, как это сделано в листинге 33.5.

Листинг 33.5

```
required init?(coder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
}
```

На этом работа с классом `Ball` окончена. В листинге 33.6 приведен его код.

Листинг 33.6

```
class Ball: UIView, BallProtocol {
    required public init(color: UIColor, radius: Int, coordinates: (
                         x:Int, y: Int)) {
        // создание графического прямоугольника
        super.init(frame:
```

```
            CGRect(x: coordinates.x,
                   y: coordinates.y,
                   width: radius * 2,
                   height: radius * 2))
        // скругление углов
        self.layer.cornerRadius = self.bounds.width / 2.0
        // изменение цвета фона
        self.backgroundColor = color
    }

    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
}
```

Класс SquareArea

Теперь перейдем к реализации класса `SquareArea`, представляющего прямоугольную площадку, по которой будут перемещаться шарики. Перейдите к файлу SquareArea.swift в Project Navigator и вместо библиотеки `Foundation` импортируйте `UIKit`.

Реализацию сущности мы вновь начнем с протокола (листинг 33.7).

Листинг 33.7

```
import UIKit

protocol SquareAreaProtocol {
    init(size: CGSize, color: UIColor)
    // установить шарики в область
    func setBalls(withColors: [UIColor], andRadius: Int)
}
```

Протокол `SquareAreaProtocol` содержит два требования:

- Инициализатор, в который передаются размеры прямоугольной области и цвет фона. Класс `CGSize` позволяет создать объект со свойствами `width` и `height`. То есть вместо того, чтобы передавать отдельно два аргумента для ширины и высоты, мы объединяем их в значение типа `CGSize`.
- Метод `setBalls(withColors:andRadius:)`, с помощью которого на площадке будут размещаться шарики. Обратите внимание, что мы не будем передавать в данный метод значения типа `Ball`, а вместо этого передадим просто коллекцию цветов, в которые должны быть окрашены шарики, и их радиус. Тип `Ball` будет использоваться уже внутри реализации метода.

Объявите класс `SquareArea` в соответствии с листингом 33.8.

Листинг 33.8

```
public class SquareArea: UIView, SquareAreaProtocol {
    // коллекция всех шариков
```

```
    private var balls: [Ball] = []
    // аниматор графических объектов
    private var animator: UIDynamicAnimator?
    // обработчик перемещений объектов
    private var snapBehavior: UISnapBehavior?
    // обработчик столкновений
    private var collisionBehavior: UICollisionBehavior
}
```

Свойство `balls` будет использоваться для хранения экземпляров типа `Ball`, то есть шариков, размещенных на площадке.

Благодаря трем следующим свойствам создается та самая интерактивность, о которой мы говорили ранее:

- Свойство `animator` типа `UIDynamicAnimator` будет использоваться для анимации движений шариков.
- Свойство `snapBehavior` типа `UISnapBehavior` будет использоваться при обработке взаимодействия пользователя с шариками (их перемещении).
- Свойство `collisionBehavior` типа `UICollisionBehavior` будет обрабатывать столкновения шариков друг с другом и с границами прямоугольной области.

Теперь реализуем два инициализатора и метод `setBalls` (его временно оставьте пустым) (листинг 33.9).

Листинг 33.9

```
required public init(size: CGSize, color: UIColor) {
    // создание обработчика столкновений
    collisionBehavior = UICollisionBehavior(items: [])
    // строим прямоугольную графическую область
    super.init(frame:
        CGRect(x: 0, y: 0, width: size.width, height: size.height))
    // изменяем цвет фона
    self.backgroundColor = color
    // указываем границы прямоугольной области
    // как объекты взаимодействия
    // чтобы об них могли ударяться шарики
    collisionBehavior.setTranslatesReferenceBoundsIntoBoundary(
                      with: UIEdgeInsets(top: 1, left: 1, bottom: 1, right: 1))
    // подключаем аниматор с указанием на сам класс
    animator = UIDynamicAnimator(referenceView: self)
    // подключаем к аниматору обработчик столкновений
    animator?.addBehavior(collisionBehavior)
}

required init?(coder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
}

public func setBalls(withColors ballsColor: [UIColor], andRadius radius: Int) {}
```

Теперь перейдите к основному файлу проекта Balls в Project Navigator и создайте экземпляр класса `SquareArea` и с помощью `PlaygroundPage` отобразите его (листинг 33.10).

Листинг 33.10

```
// размеры прямоугольной области
let sizeOfArea = CGSize(width: 400, height: 400)
// создание экземпляра
var area = SquareArea(size: sizeOfArea, color: UIColor.gray)
// установка экземпляра в качестве текущего отображения
PlaygroundPage.current.liveView = area
```

Свойство `current` класса `PlaygroundPage` возвращает текущую страницу playground-проекта. Данный проект состоит всего из одной страницы (что видно в Project Navigator), но вы всегда можете создать дополнительные с помощью кнопки «+» в левом нижнем углу. Свойству `liveView` может быть передано произвольное отображение (класс `UIView` или его потомок), которое в дальнейшем будет выведено на экран.

> **ПРИМЕЧАНИЕ** Файлы с расширением .swift не являются отдельными страницам, они входят в состав страниц. Непосредственно страницы в Project Navigator помечаются изображением листа бумаги с логотипом Swift голубого цвета.

Если сейчас запустить проект, то в правой части Xcode Playground отобразится область Live View, содержащая прямоугольник серого цвета, в котором в дальнейшем будут расположены цветные шарики (рис. 33.2).

Для указания цвета площадки передается значение `UIColor.gray`, соответствующее серому цвету. Тип `UIColor` имеет большое количество свойств: `UIColor.white` для белого, `UIColor.red` для красного и т. д.

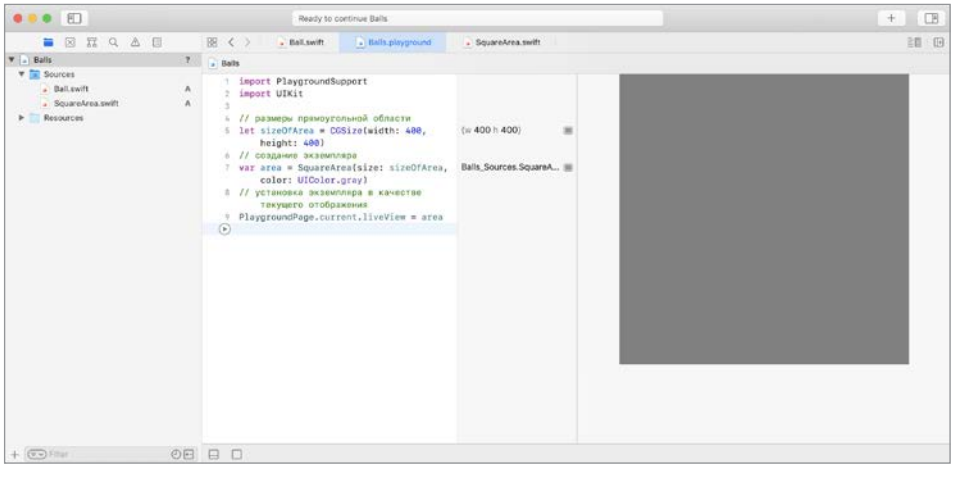

Рис. 33.2. Отображение прямоугольной площадки

ПРИМЕЧАНИЕ Xcode позволяет преобразовать некоторые ресурсы из текстового вида в графический. Так, например, описание цвета может быть представлено в виде визуального элемента прямо в редакторе кода!

Для этого вместо `UIColor.gray` укажите специальный литерал:

```
#colorLiteral(red: 0, green: 0, blue: 0, alpha: 1)
```

где с помощью `red`, `green`, `blue` определяется цвет, а с помощью `alpha` — прозрачность. После его написания этот код будет преобразован к графическому квадрату черного цвета (рис. 33.3).

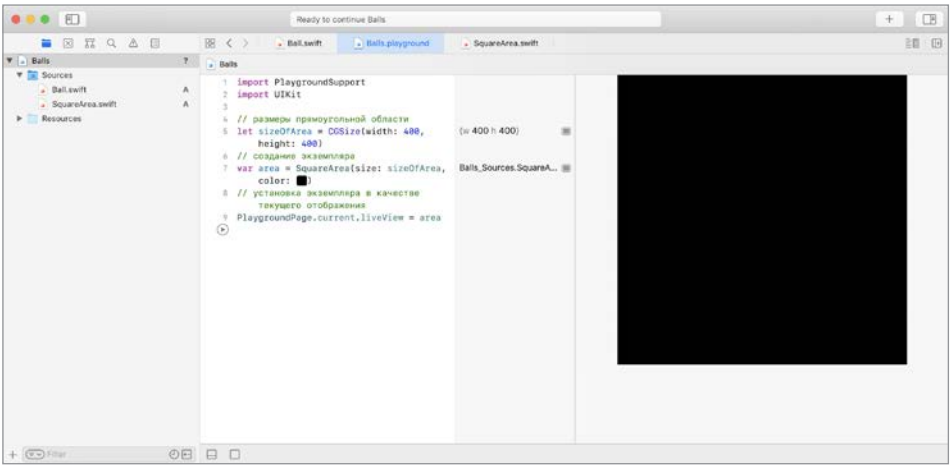

Рис. 33.3. Отображение прямоугольной площадки

Для выбора нового цвета вам необходимо дважды щелкнуть на квадрате 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 страниц playground. В файлах с исходным кодом (Balls.swift и SquareArea.swift) вся информация отображается исключительно в текстовом виде.

Вернемся к файлу SquareArea.swift. Теперь перед нами стоит задача наполнить метод `setBalls` для вывода шариков на экран (листинг 33.11).

Листинг 33.11

```
public func setBalls(withColors ballsColor: [UIColor], andRadius radius: Int) {
    // перебираем переданные цвета
    // один цвет – один шарик
    for (index, oneBallColor) in ballsColor.enumerated() {
        // рассчитываем координаты левого верхнего угла шарика
        let coordinateX = 10 + (2 * radius) * index
        let coordinateY = 10 + (2 * radius) * index
        // создаем экземпляр сущности "Шарик"
        let ball = Ball(color: oneBallColor,
                        radius: radius,
                        coordinates: (x: coordinateX, y: coordinateY))
```

```
            // добавляем шарик в текущее отображение (в состав прямоугольной
            // площадки)
            self.addSubview(ball)
            // добавляем шарик в коллекцию шариков
            self.balls.append(ball)
            // добавляем шарик в обработчик столкновений
            collisionBehavior.addItem(ball)
        }
}
```

Основная задача данного метода — создание экземпляров типа `Ball` и размещение их на прямоугольной площадке. Работа включает следующие шаги:

1. Производится перебор всех переданных цветов, так как количество шариков соответствует количеству цветов. При этом используется уже известный вам метод `enumerated()`, позволяющий получить целочисленный индекс и значение каждого элемента коллекции.
2. Рассчитываются координаты очередного шарика. Указанная формула позволяет разместить элементы каскадом, то есть каждый последующий находится правее и ниже предыдущего.

ПРИМЕЧАНИЕ Вы можете «поиграть» с кодом и попробовать собственные формулы для расчета координат шариков.

3. Создается экземпляр типа `Ball` (с помощью созданного ранее инициализатора).
4. Созданное отображение шарика включается в состав отображения прямоугольной области с помощью метода `addSubview(_:)`.

   Данный пункт я хотел бы рассмотреть чуть подробнее.

   Как вы видели ранее, чтобы отобразить графический элемент в Live view, мы инициализировали его в качестве значения свойству `liveView` текущей страницы Playground. Но данному свойству в качестве значения может быть проинициализирован только один экземпляр, а в нашем случае их пять (один `SquareArea` и четыре `Ball`). Как в таком случае отобразить все графические элементы?

   У каждого экземпляра типа `UIView` (или его потомков) есть свойство `subviews`, которое содержит *подпредставления* данного представления. То есть одно представление в своем составе может содержать другие представления. При отрисовке графического элемента на экране устройства Swift определяет, есть ли в его свойстве `subviews` элементы. И если они найдены, то также отрисовываются.

   Далее у каждого из дочерних элементов также происходит проверка значения свойства `subviews`. И так далее, пока все элементы иерархии представлений не будут выведены на экран.

Метод `addSubview(_:)` предназначен для того, чтобы включить одни представления в состав другого.

5. Созданный экземпляр добавляется в приватную коллекцию.
6. Созданный экземпляр добавляется в обработчик столкновений.

Теперь вернемся к главному файлу проекта и после создания переменной `area` добавим вызов метода `setBall`, передав в него цвета четырех шариков. После запуска проекта шарики отобразятся поверх черной площадки (рис. 33.4).

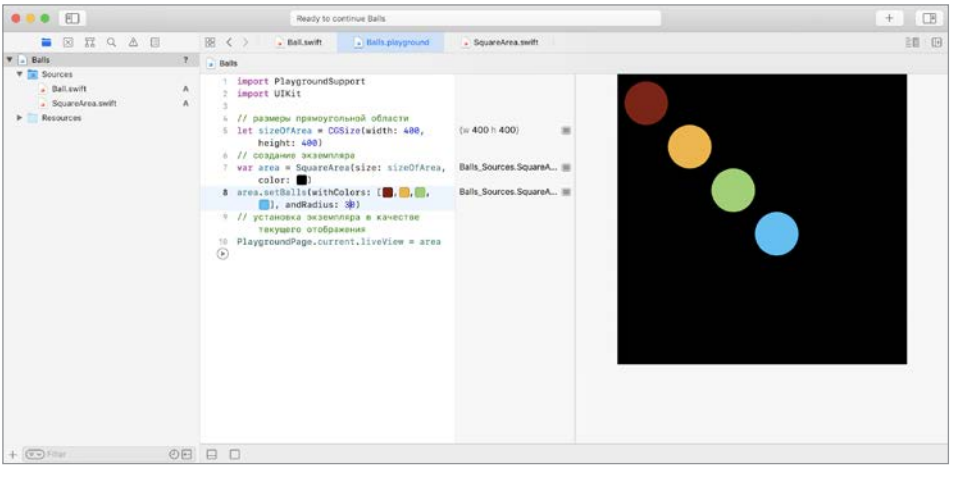

Рис. 33.4. Отображение шариков

Нам осталось реализовать возможность перемещения шариков указателем мыши, обработку их столкновений между собой и с границами площадки.

Как говорилось ранее, `UIKit` обеспечивает не только вывод графических элементов, но и обработку событий, в том числе касаний. Как только пользователь касается пальцем экрана устройства, операционная система формирует специальный объект, передающийся в приложение, который «путешествует» там и в конце концов достигает графического элемента, с которым произошло взаимодействие.

ПРИМЕЧАНИЕ Обработка событий — это одна из тем, подробно рассматриваемая во второй книге.

Класс `UVIew` позволяет добавить к графическому элементу специальные методы, обрабатывающие произошедшие с ним события касания:

```
touchesBegan(_:with:)
```

Метод срабатывает 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 классе `UIView`, поэтому при создании собственной реализации этих методов в дочерних классах (в нашем случае в `SquareArea` и `Ball`) потребуется их переопределять с использованием ключевого слова `override`. Данные методы будут реализованы только в классе `SquareArea`, так как класс `Ball` занимается исключительно отображением шариков. При нажатии на определенную точку в прямоугольной области будет определяться, соответствуют ли координаты нажатия текущему положению одного из шариков.

Реализуем метод `touchesBegan(_:with:)` (листинг 33.12).

Листинг 33.12

```
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!)
            }
        }
    }
}
```

Аргумент `touches` содержит данные обо всех текущих касаниях. Это связано с тем, что экран всех современных смартфонов поддерживает мультитач, то есть одновременное касание несколькими пальцами. В начале метода извлекаются данные о первом элементе множества `touches` и помещаются в константу `touch`.

Константа `touchLocation` содержит координаты касания относительно площадки, на которой расположены шарики.

С помощью метода `ball.frame.contains()` мы определяем, относятся ли координаты касания к какому-либо из шариков. Если находится соответствие, то в свойство `snapBehavior` записываются данные о шарике, с которым в текущий момент происходит взаимодействие, и о координатах касания.

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 будут анимированы.

Далее необходимо обрабатывать перемещение пальца. Для этого реализуем метод `touchesMoved(_:with:)` (листинг 33.13).

Листинг 33.13

```
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 33.14).

Листинг 33.14

```
public override func touchesEnded(_ touches: Set<UITouch>, with event: UIEvent?) {
    if let snapBehavior = snapBehavior {
        animator?.removeBehavior(snapBehavior)
    }
    snapBehavior = nil
}
```

Этот метод служит для решения одной очень важной задачи — очистки используемых ресурсов. После того как взаимодействие с шариком окончено, хранить информацию об обработчике поведения в `snapBehavior` уже нет необходимости.

> **ПРИМЕЧАНИЕ** Возьмите за привычку удалять ресурсы, пользоваться которыми вы уже не будете. Это сэкономит изрядное количество памяти и уменьшит вероятность возникновения ошибок.

Перейдите к странице Balls, и после запуска проекта в окне Assistant Editor вы увидите изображение четырех шариков, но теперь с помощью указателя мыши вы можете перемещать и сталкивать их друг с другом и с границами площадки!

Библиотеки `UIKit` и `Foundation` предоставляют огромное количество возможностей. Так, например, можно создать гравитационное, магнитное взаимодействие элементов, смоделировать силу тяжести или турбулентность, установить параметры скорости и ускорения. Все это позволит вам создавать поистине функциональные приложения.

Глава 34. Универсальные шаблоны (Generic)

Функции и объектные типы (перечисления, структуры и классы) предназначены для того, чтобы писать *хороший* код, который можно многократно использовать и при необходимости расширять. По большому счету, вся суть этих функциональных элементов сводится к улучшению качества кода. Действительно, зачем десять раз повторно реализовывать загрузку требуемых значений из базы данных, когда можно создать функцию и при необходимости вызывать ее. *Универсальные шаблоны* (generic, или дженерик) предназначены для того, чтобы сделать ваш код еще более качественным: вы сможете писать меньше кода, получая тот же самый результат.

> **ПРИМЕЧАНИЕ** Чтобы глубоко понять и начать применять универсальные шаблоны, потребуется время. Не ограничивайтесь изучением материала этой книги, ищите новые источники знаний и уделите дженерикам пристальное внимание, так как они способны значительно улучшить ваш код.

Универсальные шаблоны в Swift представлены тремя базовыми элементами:

- универсальные функции;
- универсальные протоколы;
- универсальные объектные типы.

В этой главе мы подробно разберем каждый из них.

34.1. Зачем нужны дженерики

С помощью универсальных шаблонов вы сможете реализовывать очень гибкие конструкции без привязки к конкретным типам данных. На самом деле вы уже давно работаете с дженериками, вероятно, даже не подозревая об этом. К ним, например, относятся массивы, множества и словари, элементами которых могут выступать значения различных типов.

Рассмотрим следующий пример.

У вас есть два параметра типа `Int`. Перед вами стоит задача написать код, меняющий значения данных параметров между собой. Каким образом лучше решить эту задачу? Вероятно, написать функцию (листинг 34.1).

Листинг 34.1

```
var first = 3
var second = 5

func change(a: inout Int, b: inout Int) {
    let tmp = first
    first = second
    second = tmp
}

change(a: &first, b: &second)
first // 5
second // 3
```

Для обмена значений переменных функция `change(a:b:)` использует сквозные параметры, в результате чего значения `first` и `second` меняются местами.

Теперь перед вами встала задача реализовать обмен значениями типа `String`. Как поступить в этом случае? Ответ очевиден: написать новую функцию, которая будет принимать значения данного типа (листинг 34.2).

Листинг 34.2

```
func change(a: inout String, b: inout String) {
    let tmp = first
    first = second
    second = tmp
}
```

Но и этого оказалось мало. Со временем вам потребовалось бы менять местами значения и других типов: `Double`, `UInt` и даже некоторых собственных структур. Для каждого из них потребуется создать собственную функцию. Несмотря на то что мы применяем механизм, позволяющий избежать дублирования кода, в итоге мы занимаемся этим самым дублированием: функции отличаются лишь типом аргументов, тело всех функций идентично друг другу.

Дженерики позволят устранить эту проблему, создав единый для всех типов данных механизм.

34.2. Универсальные функции

Наиболее удобным способом решения задачи с функцией обмена значениями может стать реализация универсальной функции, которая будет принимать на вход значения любых типов данных, после чего выполнять с ними требуемые операции.

Для того чтобы объявить универсальную функцию, после имени функции в угловых скобках необходимо указать заполнитель типа, например

```
change<T>
```

который будет использован в сигнатуре или в теле функции как указатель на пока еще не определенный тип данных. То есть с помощью заполнителя типа `<T>` вы определяете параметр типа `T`, который будет использоваться в качестве указателя на тип данных. Таким образом, в сигнатуре (и при необходимости в теле) функции вместо `Int`, `String`, `Double` и иных типов потребуется указывать `T`.

Запомните:

- `<T>` — заполнитель типа;
- `T` — параметр типа.

> **ПРИМЕЧАНИЕ** В качестве заполнителя принято указывать символ T (первая буква в слове Type). При нескольких заполнителях в пределах одного дженерика используйте T1, T2 и т. д. Но эта рекомендация не обязывает вас использовать именно T, вы можете заменить его произвольным символом или символами.

Реализуем универсальную функцию, меняющую значения входных параметров произвольных типов данных (листинг 34.3).

Листинг 34.3

```
func change<T>(a: inout T,  b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

В универсальной функции `change<T>(a:b:)` параметр типа `T` в списке аргументов заменяет указание на конкретный тип. Но при этом указано, что и `a` и `b` должны иметь один и тот же тип данных `T`.

С помощью данной функции мы заменили все реализованные ранее функции обмена значениями. Теперь при необходимости можно использовать именно ее (листинг 34.4).

Листинг 34.4

```
// используем универсальную функцию
// для строковых значений
var firstString = "one"
var secondString = "two"
change(a: &firstString, b: &secondString)

firstString // "two"
secondString // "one"

// используем универсальную функцию
// для целочисленных значений
var firstInt = 3
var secondInt = 5
change(a: &firstInt, b: & secondInt)
firstString // 5
secondInt // 3
```

ПРИМЕЧАНИЕ Обратите внимание, что при вызове универсальной функции заполнитель типа опускается. Он используется только на этапе объявления дженерика.

Заполнитель типа может состоять не из одного, а из нескольких параметров. В листинге 34.5 показан пример универсальной функции `printValues<T1,T2>(a:b:)` с параметрами типа `T1` и `T2`, которые используются для указания типа данных аргументов. Таким образом, вы можете передать в функцию как значения различных типов (например, `String` и `Double`, `Int` и `UInt`), так и одного и того же типа.

Листинг 34.5

```
func printValues<T1,T2>(a: T1, b: T2) {
    print("Значение 1 - \(a), значение 2 - \(b)")
}

printValues(a: "book", b: 15)
printValues(a: 4, b: 5)
```

Консоль

```
Значение 1 - book, значение 2 - 15
Значение 1 - 4, значение 2 - 5
```

Параметры типа могут быть использованы не только для аргументов, но и для возвращаемого значения. В листинге 34.6 показан пример простейшей универсальной функции, которая принимает значение произвольного типа данных и возвращает его. При этом параметр типа `T` используется в том числе и для указания на тип возвращаемого значения.

Листинг 34.6

```
func getSelf<T>(_ a: T) -> T {
    return a
}

getSelf(15) // 15
```

Универсальные функции в значительной мере могут улучшить ваш код, позволяя создавать универсальные конструкции. Но это лишь вершина айсберга. В следующих разделах вы познакомитесь с универсальными типами и протоколами.

34.3. Ограничения типа

К параметрам типа могут быть применены различные ограничения, уточняющие, какие именно типы данных можно использовать в том или ином случае.

Рассмотрим следующий пример.

Реализуем универсальную функцию, производящую сложение переданных ей значений и возвращающую результат этой операции (листинг 34.7).

Листинг 34.7

```
func sum<T>(_ a: T, _ b: T) -> T {
    return a + b
}
```

На первый взгляд такая функция выглядит вполне рабочей, но при компиляции вы получите сообщение об ошибке (рис. 34.1). Дело в том, что оператор + может быть использован только при сложении значений конкретных типов данных (они перечислены в тексте ошибки). Так, например, вы вполне можете сложить два значения типа `Int`, но для типа `String` эта возможность недоступна. При этом функция `sum<T>` объявлена таким образом, что входные параметры могут иметь произвольный тип, в том числе и `String` (а также любой другой).

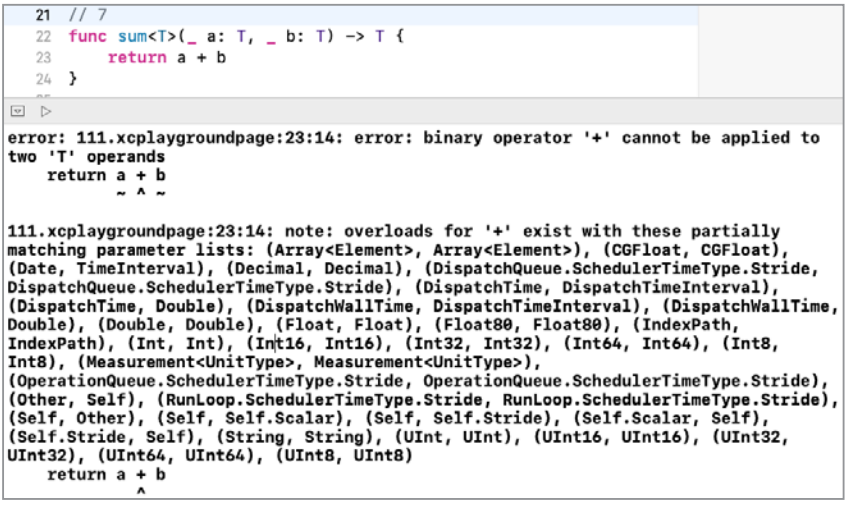

Рис. 34.1. Ошибка в ходе компиляции

Получается, что необходимо ввести ограничения, или, другими словами, указать, какие именно типы данных могут быть использованы вместо параметра `T`, таким образом исключив `String` (и другие неподходящие типы).

Вполне логично будет предположить, что функция `sum<T>` будет использоваться для числовых значений, а значит, в качестве ограничения можно использовать протокол `Numeric` (на него подписаны все соответствующие типы данных). Если в качестве ограничения указать данный протокол, то функция сможет быть использована только для числовых значений.

Ограничения типа могут быть указаны двумя способами:

1. Непосредственно в заполнителе типа, через двоеточие после параметра (листинг 34.8).

Листинг 34.8

```
func sum<T: Numeric>(_ a: T, _ b: T) -> T {
    return a + b
}

sum(1, 6) // 7
sum(1.1, 7.8) // 8.9
sum("one", "two") // вызовет ошибку
```

2. С помощью ключевого слова `where`, указываемого после сигнатуры функции (листинг 34.9).

Листинг 34.9

```
func sum<T>(_ a: T, _ b: T) -> T where T: Numeric {
    return a + b
}

sum(1, 6) // 7
sum(1.1, 7.8) // 8.9
sum("one", "two") // вызовет ошибку
```

Любой из указанных вариантов позволит использовать функцию `sum<T>(_ a: T, _ b: T)` для сложения любых числовых значений.

Рассмотрим еще один пример использования ограничения типа. В листинге 34.10 реализована функция, производящая поиск переданного значения в переданном массиве. При этом в качестве ограничения указано, что значение должно быть `Comparable`, то есть сопоставимым (при поиске элемента массива происходит сопоставление переданного значения очередному элементу).

> **ПРИМЕЧАНИЕ** Как вы уже знаете, соответствие типа данных протоколу `Comparable` гарантирует, что значения этого типа могут быть сопоставлены друг с другом с помощью оператора ==.

Листинг 34.10

```
func search<T: Comparable>(value: T, in collection: [T]) -> Bool {
    for item in collection {
        if item == value {
            return true
        }
    }
    return false
}

var array = [1,7,9,11]
search(value: 1, in: array) // true
search(value: 5, in: array) // false
```

С помощью ограничений типа вы можете уточнять область применения дженерика.

34.4. Универсальные объектные типы

Помимо универсальных функций, к дженерикам относятся и универсальные объектные типы.

Рассмотрим следующий пример: перед вами стоит задача реализовать механизм хранения целочисленных значений в виде стека. Понятие стека уже известно вам, мы рассматривали его в главе об управлении памятью. Такая структура данных характеризуется тем, что вам всегда доступен только верхний (первый) элемент стека. Все новые элементы добавляются в начало стека, а для доступа к нижележащим верхний элемент должен быть убран из стека.

Для реализации стека воспользуемся структурой, реализовав в ней свойство `items`, которое будет хранить элементы стека, а также два метода:

- метод `push(_:)` — добавляет новый элемент в начало стека;
- метод `pop()` — возвращает первый элемента стека, удаляя его оттуда.

Наличие других свойств и операций не подразумевается.

В листинге 34.11 показана реализация структуры `StackInt`, способной хранить значения типа `Int`.

Листинг 34.11

```
struct StackInt {
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.insert(contentsOf: [item], at: 0)
    }
    mutating func pop() -> Int {
        return items.removeFirst()
    }
}

var intStorage = StackInt(items: [1,2,3,4,5])
intStorage.items // [1, 2, 3, 4, 5]
intStorage.pop() // 1
intStorage.push(9)
intStorage.items // [9, 2, 3, 4, 5]
```

Эта структура имеет один большой недостаток: область ее применения ограничена типом `Int`. В случае необходимости работы с другими типами потребуется реализовывать новые структуры. Но и в этом случае на помощь приходят дженерики, а точнее, универсальные типы (листинг 34.12).

Листинг 34.12

```
struct Stack<T> {
    var items = [T]()
    mutating func push(_ item: T) {
        items.insert(contentsOf: [item], at: 0)
```

```
    }
    mutating func pop() -> T {
        return items.removeFirst()
    }
}

var genericIntStorage = Stack(items: [1,2,3,4,5])
var genericStringStorage = Stack(items: ["Bob", "John", "Alex"])
```

Для создания универсального типа вновь используется заполнитель `<T>`, который указывается после имени объектного типа. Параметр типа `T` обеспечивает использование любого типа данных для значений стека.

Ограничения доступны и при работе с универсальными типами. В листинге 34.13 показан пример стека, который может хранить только числовые значения.

Листинг 34.13

```
struct Stack<T: Numeric> {
    var items = [T]()
    mutating func push(_ item: T) {
        items.insert(contentsOf: [item], at: 0)
    }
    mutating func pop() -> T {
        return items.removeFirst()
    }
}
```

Расширения универсального типа

Универсальные типы могут быть расширены точно так же, как и обычные объектные типы данных. В этом случае в расширении (`extension`) вы можете использовать параметры типа, которые были определены в самой реализации объектного типа. Например, если бы потребовалось расширить универсальный тип `Stack`, реализовав в нем метод, заменяющий верхний элемент на переданный, то для указания типа данных значения нового элемента необходимо было использовать `T` (листинг 34.14).

Листинг 34.14

```
extension Stack {
    mutating func replaceFirst(_ newValue: T) {
        items[0] = newValue
    }
}

var genericIntStorage = Stack(items: [1,2,3,4,5])
genericIntStorage.replaceFirst(10)
genericIntStorage.items // [10, 2, 3, 4, 5]
```

Метод `replaceFirst` принимает на вход значение пока еще неизвестного типа `T` и заменяет им первый элемент стека.

34.5. Универсальные протоколы

Вернемся к примеру с универсальной функцией, производящей поиск элемента в переданном массиве (листинг 34.10). В качестве ограничения типа указано, что тип данных должен быть `Comparable` (сопоставимым), а конструкция `[T]` говорит о том, что вторым аргументом (значение, в котором производится поиск) может быть только массив (`Array`) или множество (`Set`) (литерал `[T]` соответствует только им). Но как быть, если возникла необходимость, чтобы в качестве второго параметра можно было передать диапазон?

Для решения этой задачи потребуются два параметра типа вместо одного `T`: `T1` и `T2`, где `T1` будет определять тип искомого элемента, а `T2` — тип коллекции (листинг 34.15).

Листинг 34.15

```
func search<T1, T2>(value: T1, in collection: T2) -> Bool
    where T1: Comparable, T2: Collection {
        for item in collection {
            if item == value {
                return true
            }
        }
        return false
}
```

Теперь дженерик использует два параметра типа (`T1` и `T2`), для каждого из которых определены соответствующие ограничения:

- `T1` должен быть `Comparable`.
- `T2` должен быть `Collection`.

При попытке выполнить этот код вы получите сообщение об ошибке! Но в этом нет ничего удивительного. Дело в том, что, введя отдельный параметр типа `T2` для коллекции, мы потеряли связь между типом данных искомого элемента и типом данных элементов этой коллекции. Swift не позволяет сравнивать между собой значения различных типов: например, вы не можете искать `String` в `Array<Int>`. Но использованный синтаксис функции `search` не запрещает передать конфликтующие значения.

По этой причине необходимо указать новое ограничение, которое будет говорить о том, что тип искомого элемента должен соответствовать типу элементов коллекции. Как это сделать? Обратимся к документации.

➤ Откройте справку к протоколу `Collection` и найдите раздел Associated Types (рис. 34.2).

Associated Types (ассоциированные, или связанные, типы) — это средство, с помощью которого протоколы могут стать универсальными. Ассоциированные типы содержат указания на типы данных, которые используются внутри типа, реализующего протокол. Так, например, протокол `Collection` описывает коллекцию

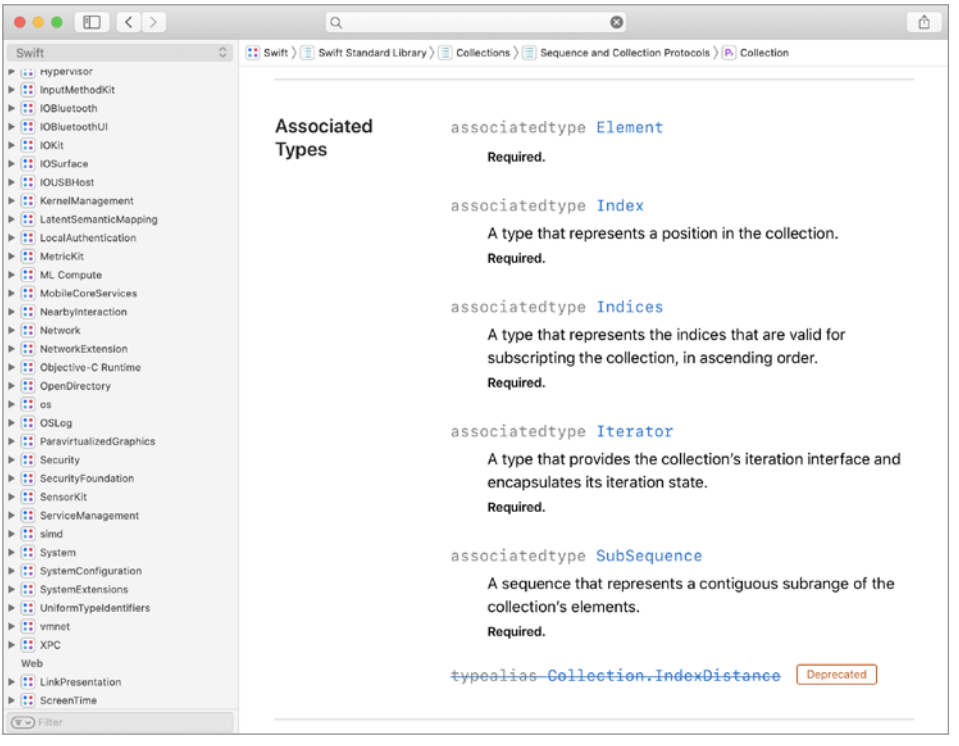

Рис. 34.2. Справка к протоколу Collection

элементов, причем заранее неизвестно, какой тип данных будут иметь эти элементы. С помощью ассоциированных типов мы можем указать некий заполнитель (аналогично заполнителю в универсальных функциях и универсальных типах), который будем использовать при описании свойств и методов внутри протокола.

Как видно на рис. 34.2, в протоколе `Collection` присутствует ассоциированный тип `Element`, который содержит информацию о типе данных элемента коллекции. Такой ассоциированный тип может быть использован и в случае с реализованной нами функцией `search` для указания связи между `T1` и `T2`. Для этого ограничения типа необходимо дополнить выражением `T1 == T2.Element`, то есть тип `T1` должен быть равен типу элементов `T2` (листинг 34.16).

Листинг 34.16

```
func search2<T1, T2>(value: T1, in collection: T2) -> Bool
    where T1: Comparable, T2: Collection, T1 == T2.Element {
        for item in collection {
            if item == value {
                return true
            }
        }
        return false
}
```

Теперь этот код не вызывает ошибок, а универсальная функция `search` может быть применена для поиска элемента в любой коллекции, выполняющей следующие требования:

- Тип данных искомого элемента соответствует протоколу `Comparable`.
- Поиск элемента производится в коллекции.
- Тип данных искомого элемента соответствует типу данных элементов коллекции.

Использование ассоциированных параметров

Протокол `Collection`, к которому мы только что обращались, является универсальным. И делают его таким ассоциированные типы данных. Далее мы рассмотрим, каким образом можно использовать ассоциированные типы при разработке собственных протоколов.

Вернемся к разработанному ранее универсальному типу, описывающему стек (листинг 34.17).

Листинг 34.17

```
struct Stack<T> {
    var items = [T]()
    mutating func push(_ item: T) {
        items.insert(contentsOf: [item], at: 0)
    }
    mutating func pop() -> T {
        return items.removeFirst()
    }
}
```

Реализуя данную структуру, мы отошли от принципов протокол-ориентированного программирования и пропустили этап создания протокола, а ведь именно с него необходимо начинать реализацию любой сущности. Но как это сделать в случае со структурой `Stack`, которая является универсальной? Свойство этой структуры использует `T`, оба метода данной структуры также используют `T`. Как написать протокол, который сможет стать основной для универсального типа? Для этого и нужны универсальные протоколы с ассоциированными типами.

Создавая протокол с помощью ключевых слов `associatedtype`, можно указать необходимые ассоциированные типы, после чего использовать их при описании требований внутри протокола.

В листинге 34.18 показано, как будет выглядеть протокол `StackProtocol`, который станет основой для универсального типа `Stack`.

Листинг 34.18

```
protocol StackProtocol {
    associatedtype ItemType
```

```
    var items: [ItemType] { get set }
    mutating func push(_: ItemType)
    mutating func pop() -> ItemType
}
```

В этом протоколе с помощью ключевого слова `associatedtype` указывается, что в принимающем данный протокол объектном типе будет использован некий пока неизвестный тип данных, обозначенный в протоколе как `ItemType`.

Таким образом, `ItemType` (аналогично `Element` в протоколе `Collection`) — это некий заполнитель, который используется, чтобы описать требования в теле протокола.

Теперь мы можем подписать структуру `Stack` на данный протокол, но при этом в ней потребуется создать псевдоним (`typealias`), указывающий, что `ItemType` — это `T` (пока неизвестный тип) (листинг 34.19).

Листинг 34.19

```
struct Stack<T>: StackProtocol {
    // псевдоним для ассоциативного типа
    typealias ItemType = T
    var items = [T]()
    mutating func push(_ item: T) {
        items.insert(contentsOf: [item], at: 0)
    }
    mutating func pop() -> T {
        return items.removeFirst()
    }
}
```

В результате мы не отошли от принципов протокол-ориентированного программирования и последовательно реализовали сущность «Стек»: сперва описали соответствующий протокол, а уже потом основанную на нем структуру.

Дальнейшая доработка сущности

Но на этом мы не остановимся и в полной мере применим возможности ПОП, перенеся функциональность методов `push` и `pop` типа `Stack` в протокол `StackProtocol` (с помощью `extension`). В листинге 34.20 показан полный код, реализующий сущность «Стек».

Листинг 34.20

```
protocol StackProtocol {
    associatedtype ItemType
    var items: [ItemType] { get set }
    mutating func push(_ item: ItemType)
    mutating func pop() -> ItemType
}

extension StackProtocol {
    mutating func push(_ item: ItemType) {
```

```
        items.insert(contentsOf: [item], at: 0)
    }

    mutating func pop() -> ItemType {
        return items.removeFirst()
    }
}

struct Stack<T>: StackProtocol {
    typealias ItemType = T
    var items: [T]
}

// Проверка работы
var myStack = Stack(items: [2,4,6,8])
myStack.pop() // 2
myStack.push(9)
myStack.items // [9, 4, 6, 8]
```

Теперь любой объектный тип, принимающий к реализации протокол `StackProtocol`, автоматически будет обладать двумя методами, обеспечивающими функционирование этой структуры данных.

На этом мы завершаем знакомство с универсальными шаблонами и их возможностями в Swift. Настоятельно рекомендую продолжить изучение самостоятельно, так как полное понимание этой темы позволит вам писать качественный и очень компактный код, который совершенно точно оценит ваш будущий работодатель.

34.6. Непрозрачные типы (Opaque types) и ключевое слово some

Представим, что перед нами стоит задача описать несколько типов транспортных средств. Решение этой задачи начнем с реализации протокола (листинг 34.21).

Листинг 34.21

```
protocol Vehicle {
    var uid: Int { get set }
}
```

Протокол `Vehicle` максимально упрощен и содержит всего одно свойство `uid`, содержащее уникальный идентификатор транспортного средства.

Основываясь на протоколе, мы сможем реализовать необходимые структуры (листинг 34.22).

Листинг 34.22

```
// Машина
struct Car: Vehicle {
    var uid: Int
```

```
}

// Грузовик
struct Truck: Vehicle {
    var uid: Int
}
```

Теперь напишем функцию, которая возвращает экземпляр одной из структур, при этом тип возвращаемого значения определим, используя протокол `Vehicle` (листинг 34.23).

Листинг 34.23

```
func getCar() -> Vehicle {
    return Car(uid: 93)
}

getCar() // Car
```

Давайте усложним код. Предположим, что каждый тип транспортного средства имеет свою систему идентификации по уникальным номерам и для свойства `uid` требуется использовать значения различных типов в различных реализациях протокола `Vehicle`. Реализуем это с помощью связанного типа (листинг 34.24).

Листинг 34.24

```
protocol Vehicle {
    // связанный тип
    associatedtype Identifier
    var uid: Identifier { get set }
}

struct Car: Vehicle {
    var uid: Int
}

struct Truck: Vehicle {
    var uid: String
}
```

Теперь структуры `Car` и `Truck` используют различные типы данных для свойства `uid`. Но внезапно Xcode сообщает нам об ошибке в функции `getCar()` (рис. 34.3).

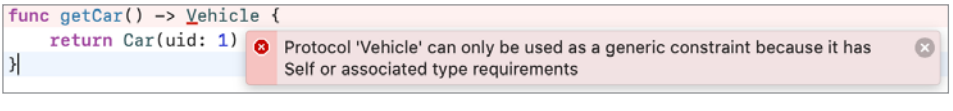

Рис. 34.3. Ошибка использования протокола Vehicle

В ошибке говорится о том, что раз протокол использует ассоциированный тип (или ключевое слово `Self`), значит, он может быть использован только в качестве ограничения в дженерике (generic). То есть мы не можем просто взять и использовать протокол в качестве указателя на возвращаемое значения.

Почему так? Разберем это по шагам:

1. Протокол `Vehicle` имеет связанный тип, а реализующие его структуры используют различные типы данных для свойства `uid` (`String` и `Int`).
2. Для того чтобы определить возвращаемое функцией `getCar()` значение, компилятор не анализирует код в ее теле. В обычных условиях тип значения должен явно определяться из сигнатуры функции.
3. В текущей реализации функция `getCar()` потенциально может вернуть значение любого типа данных, подписанного на протокол `Vehicle`. При этом заранее неизвестно, значение какого именно типа должно быть возвращено.
4. Потенциально может произойти ситуация, при которой `getCar()` возвращает значение типа `Car`, когда вы ожидаете получить значение типа `Truck` (или наоборот). Но при этом вы будете продолжать работать со свойством `uid` как со строкой (`uid` в типе `Truck` имеет тип `String`), а не с целым числом (`uid` в типе `Car` имеет тип `Int`). И это приведет к ошибке Runtime.

> **ПРИМЕЧАНИЕ** Ошибка Runtime — это критическая ошибка во время выполнения программы, которая приводит к немедленной экстренной остановке работы приложения и вылету из него. Один из вариантов получить такую ошибку — принудительное извлечение опционального значения (с помощью символа `!`), когда в нем не содержится никакого значения (`nil`).

Компилятор, основываясь на данных, которыми он обладает, не может принять решение о том, значение какого конкретно типа данных будет возвращено функцией `getCar()`.

Решение проблемы

Как же решить эту проблему? Как использовать протокол в качестве возвращаемого значения и при этом не получить сообщение об ошибке?

Вариант 1. Использовать дженерики.

В тексте ошибки (см. рис. 34.3) сказано, что протокол `Vehicle` может быть использован в дженерике в качестве ограничения. А значит, мы можем реализовать универсальную функцию, которая возвращает значение требуемого типа (листинг 34.25).

Листинг 34.25

```
func getCar<T: Vehicle>() -> T {
    return Car(uid: 22) as! T
}

let someVehicle: Car = getCar()
```

Обратите внимание, что тип возвращаемого значения определяется явно при вызове функции (`Car` после имени константы). Такой подход будет вполне рабочим ровно до тех пор, пока вы не решите проинициализировать результат вызова функции `getCar` параметру типа `Truck` (изначально вы не знаете тип данных воз-

вращаемого функцией `getCar` значения, а знаете лишь то, что оно соответствует протоколу `Vehicle`). Это приведет к ошибке Runtime, однако компиляция программы пройдет корректно (рис. 34.4).

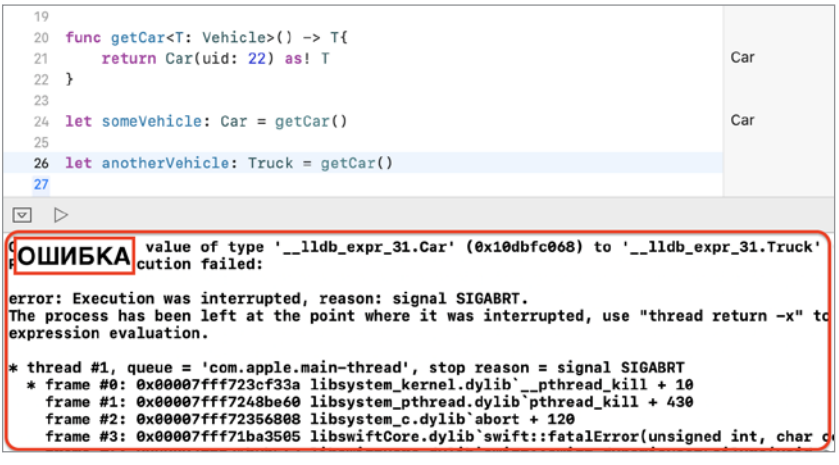

Рис. 34.4. Ошибка при использовании Generics

Использование дженерика в данном случае было бы оправданно, если бы протокол `Vehicle` не имел ассоциированных типов, а функция `getCar` могла бы вернуть значения любого подходящего типа (листинг 34.26).

Листинг 34.26

```
protocol Vehicle {
    var uid: Int { get set }
    // инициализатор
    init()
}

struct Car: Vehicle {
    var uid: Int = 0
}

struct Truck: Vehicle {
    var uid: String = ""
}

func getObject<T: Vehicle>() -> T {
    return T()
}

let car: Car = getObject() // Car
let truck: Truck = getObject() // Truck
```

В зависимости от того, какой тип данных имеет принимающий параметр, функция вернет корректное значение требуемого типа. При использовании дженериков тип данных итогового значения определяется при вызове функции.

Вариант 2. Использовать непрозрачные типы (Opaque types).

При реализации функции `getCar` тип возвращаемого значения может быть указан как протокол `Vehicle` с ключевым словом `some` (листинг 34.27).

Листинг 34.27

```
func getCar() -> some Vehicle {
    return Car(uid: 54)
}

let myCar = getCar() // Car
```

Ключевое слово `some` говорит о том, что `Vehicle` является непрозрачным типом (Opaque type), с помощью чего компилятор заранее анализирует тело функции и определяет конкретный тип возвращаемого значения.

Но в чем суть Opaque types, неужели это действительно так необходимо? Почему мы должны использовать ключевое слово `some`, если вариант с дженериком вполне работает?

Универсальные шаблоны (дженерики) и непрозрачные типы связаны между собой. Можно сказать, что непрозрачные типы — это обратные универсальные шаблоны. При использовании дженериков конкретный тип возвращаемого значения определяет тот, кто вызывает функцию, а при использовании Opaque types конкретный тип определяется реализацией функции (в ее теле) (листинг 34.28).

Листинг 34.28

```
//
// Generic
//
func getCar<T: Vehicle>() -> T{
    return Car(uid: 22) as! T
}

// Тип возвращаемого значения определяется тут
let a: Car = getCar()

//
// Opaque type
//
func getCar() -> some Vehicle {
    // Тип возвращаемого значения определяется тут
    return Car(uid: 54)
}

let b = getCar() // Car
```

В первом случае заполнитель типа (`T`) заменяется на конкретный тип данных при вызове функции. В свою очередь, при использовании Opaque type тип данных возвращаемого значения определяется в ходе анализа функции еще до ее вызова.

Другими словами, с помощью `some` мы скрываем от пользователя (того, кто вызывает функцию) информацию о конкретном типе возвращаемых данных. При этом этот тип данных определяется в теле функции. На рис. 34.5 видно, что в окне автодополнения при вызове функции указывается `Vehicle` вместо конкретного типа, однако после вызова функции в константе `b` находится значение конкретного типа `Car`.

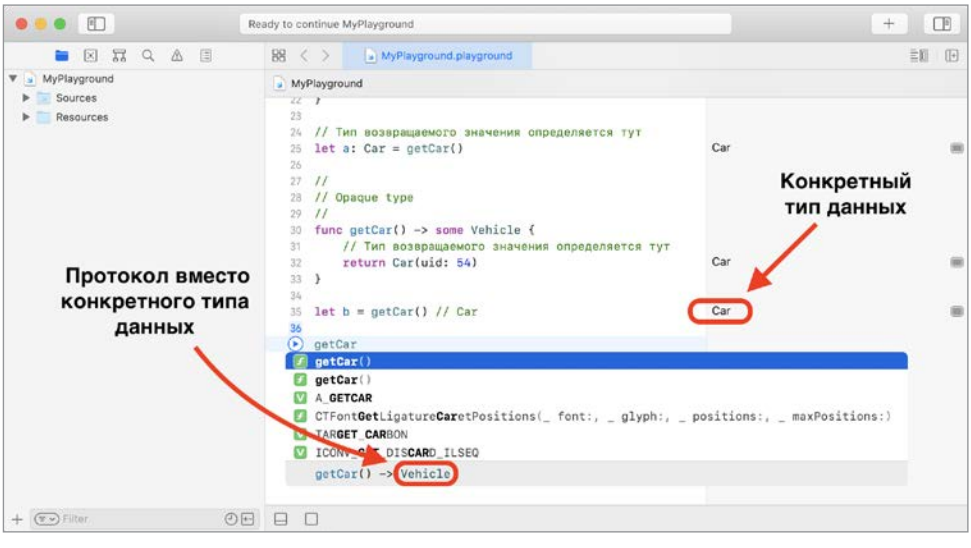

Рис. 34.5. Opaque types в действии

Функция, использующая непрозрачный тип, всегда будет возвращать значение одного конкретного типа данных. Таким образом, код, показанный в листинге 34.29, недопустим.

Листинг 34.29

```
func getCar(by number: Int) -> some Vehicle {
    switch number {
    case 1:
        return Car(uid: 55)
    default:
        return Truck(uid: "0X882S1")
    }
}

// ОШИБКА! Такой код недопустим, так как функция должна возвращать значение
конкретного типа данных, а не одного из ...
```

Обратите особое внимание на пример выше. Непрозрачные типы обеспечивают идентичность типов, то есть функция, использующая Opaque type, возвращает

значение одного конкретного (повторю: **конкретного**) типа данных, просто до ее вызова вы не знаете, какого именно.

Появление непрозрачных типов в Swift позволяет скрывать от пользователя библиотек (или фреймворков) их внутреннюю реализацию.

Как уже говорилось ранее, при использовании дженериков пользователь библиотеки (программист, использующий реализованную в библиотеке функцию) должен самостоятельно указать тип данных возвращаемого дженериком значения (листинг 34.30).

Листинг 34.30

```
func getCar<T: Vehicle>() -> T{
    return Car(uid: 22) as! T
}

let a: Car = getCar()

let b = getCar() // ОШИБКА, Тип не указан
```

Таким образом, определение типа отдается на откуп пользователю. При использовании Opaque types компилятор самостоятельно определяет тип возвращаемого значения, а пользователь просто использует функцию, ничего не зная о типе данных, который она возвращает.

Резюмируем все вышесказанное:

- Opaque types позволяют указывать тип возвращаемого функцией значения с помощью протокола, использующего связанные типы (или ключевое слово `Self`) в своей реализации.
- Opaque types позволяют определять тип возвращаемого значения внутри самой функции.

Opaque types (непрозрачные типы) — это одно из значительных нововведений Swift. В дальнейшем вы еще неоднократно встретитесь с этим механизмом, в том числе при изучении фреймворка `SwiftUI`, знакомству с которым посвящена глава 38.

Глава 35. Обработка ошибок

Контролируемое создание и последующая обработка ошибок — очень полезная возможность Swift, которую вы могли видеть и в других языках программирования. Вполне вероятно, что на начальных этапах при написании первых приложений вы не станете вдаваться в те возможности, которые описаны в этой главе. Тем не менее со временем вы поймете, что обработка ошибок способна значительно улучшить качество вашего кода.

Обработка ошибок подразумевает реагирование на возникающие в процессе выполнения программы ошибки — вы сами создаете ошибки и описываете реакцию программы на них. Но здесь слово «ошибки» стоит понимать не как синтаксические ошибки кода, а как экземпляр объектного типа (обычно перечисления), реализующего протокол `Error`, который описывает конкретный тип ошибки.

Например, ваш код производит попытку доступа к файлу — при этом возможны несколько вариантов:

- файл будет успешно прочитан;
- доступ к файлу будет запрещен по причине отсутствия прав;
- файл вовсе будет отсутствовать.

Второй и третий варианты описывают ошибки (или, иными словами, *исключительные ситуации,* или *исключения*), из-за которых невозможно выполнить требуемую операцию. При этом реагировать на каждую из ситуаций нужно определенным образом: при отсутствии прав необходимо запросить их, а при отсутствии файла — скачать его с сервера на устройство.

Встретившись с исключительной ситуацией, вы сможете создать специальный объектный тип и вынести его обработку в отдельное место в коде. Таким образом, программный код, который должен заниматься именно чтением файла, будет делать только это и не сможет выполнять задачи по обработке возникающих ошибок. Но как только возникнет ошибка, управление будет передано в специальный блок, где ошибка будет соответствующим образом обработана.

Как уже было сказано, для описания ошибок обычно используются перечисления (`enum`), соответствующие протоколу `Error`. Каждый член перечисления указывает на один конкретный тип ошибки. При обработке возникшей исключительной ситуации вы можете проанализировать экземпляр объектного типа, определить, какой именно член перечисления передан, и понять, какая ошибка возникла.

Если вернуться к примеру с чтением файла, то перечисление с ошибками могло бы выглядеть так, как показано в листинге 35.1.

Листинг 35.1

```
enum FileError: Error {
    // файл не существует
    case notExists
    // нет прав доступа
    case notRights
}
```

35.1. Выбрасывание ошибок

Рассмотрим следующий пример.

Ваша программа обеспечивает работу торгового аппарата по продаже еды. В процессе его работы могут возникнуть различные ошибки, которые не позволят совершить покупку:

- неправильный выбор;
- нехватка средств;
- отсутствие выбранного товара.

Для описания этих ошибок создадим перечисление (листинг 35.2).

Листинг 35.2

```
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```

Каждый элемент перечисления соответствует конкретной ошибке и по нему можно определить, что именно произошло: не хватило денег на счете, кончился товар или произошло что-то иное.

Сама по себе ошибка позволяет показать, что произошла какая-то нестандартная ситуация и программа не может выполняться в обычном режиме. Процесс появления ошибки называется *выбрасыванием ошибки*. Для того чтобы выбросить ошибку, необходимо воспользоваться оператором `throw`. В листинге 35.3 показан код проверки баланса счета, и в случае недостатка монет выбрасывается ошибка.

Листинг 35.3

```
// если монет не хватает
guard productPrice <= accountBalans else {
    // определяем, сколько монет не хватает
```

```
    let diff = productPrice - accountBalans
    // выбрасываем ошибку о нехватке монет
    // при этом передаем количество монет
    throw VendingMachineError.insufficientFunds(coinsNeeded: diff)
}
```

35.2. Обработка ошибок

Ошибку недостаточно выбросить, ее необходимо перехватить и корректно обработать. Swift предлагает четыре способа обработки ошибок:

- передача выброшенной ошибки в вышестоящий код;
- обработка ошибки оператором `do-catch`;
- преобразование ошибки в опционал;
- подавление выброса ошибки.

Если при вызове какой-либо функции или метода он может выбросить ошибку, то перед именем данной функции необходимо указывать ключевое слово `try`.

Теперь разберем каждый из способов обработки ошибок.

Передача ошибки

Функция (или метод), выбросившая ошибку, может не самостоятельно обрабатывать ее, а передать выше в код, который вызвал данную функцию. Для этого в сигнатуре функции после списка параметров указывается ключевое слово `throws`. В листинге 35.4 приведен пример объявления двух функций, которые передают возникающие в них ошибки выше.

Листинг 35.4

```
// перечисление, содержащее ошибки
enum CustomError: Error {
    case wrongName
}

// функция, которая выбрасывает ошибку
// и передает ее "наверх"
func funcThrowsError() throws {
    //...
    throw CustomError.wrongName
    //...
}

// функция, которая НЕ выбрасывает ошибку,
// но получает ошибку, выброшенную функцией funcThrowsError
// после чего передает ее "наверх"
func funcGetsError() throws -> String {
    //...
    try funcThrowsError()
    //...
```

```
    return "TextResult"
}

// в данной области кода
// ошибка может быть получена и обработана
try funcGetsError()
```

То есть внутри функции `funcThrowsError()` происходит выброс ошибки, но благодаря тому, что в ее сигнатуре указано ключевое слово `throws`, данная ошибка передается выше, в код функции `funcGetsError()`, которая и произвела вызов функции `funcThrowsError()`. Так как и сама функция `funcGetsError()` помечена с помощью ключевого слова `throws`, происходит дальнейшая передача ошибки «наверх», где ошибка и может быть обработана.

Если функция не помечена ключевым словом `throws`, то все возникающие внутри нее ошибки она должна обрабатывать самостоятельно.

Рассмотрим пример из листинга 35.5.

Листинг 35.5

```
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 (выбран неверный товар, товара нет в наличии или внесенных монет недостаточно для покупки) метод `vend(itemNamed:)` может выбросить ошибку, соответствующую одному из членов перечисления `VendingMachineError`. Сама реализация метода использует оператор `guard` для осуществления преждевременного выхода с помощью оператора `throw`. Оператор `throw` мгновенно изменяет ход работы программы, в результате выбранный продукт может быть куплен только в том случае, если выполняются все условия покупки.

Так как метод `vend(itemNamed:)` передает все возникающие в нем ошибки вызывающему его коду, то данный метод должен быть вызван с помощью оператора `try`, а сама ошибка обработана в родительском коде.

Реализуем функцию, которая в автоматическом режиме пытается приобрести какой-либо товар (листинг 35.6). В данном примере словарь `favoriteSnacks` содержит указатель на любимое блюдо каждого из трех человек.

Листинг 35.6

```
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 выбрасывает ошибку, она вызывает метод `vend(itemNamed:)`, который может выбросить ошибку. И в этом случае используются операторы `throws` и `try`.

Оператор do-catch

Выброс и передача ошибок вверх в конце концов должны 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 имяВызываемогоБлока
} catch шаблон1 {
```

```
    // код...
} catch шаблон2, шаблон3 {
    // код...
} catch {
    // 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 она 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 этот блок соответствует любой ошибке, а сама ошибка будет находиться в локальной переменной error.

Используем оператор `do-catch`, чтобы перехватить и обработать возможные ошибки (листинг 35.7).

Листинг 35.7

```
var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8
do {
    try buyFavoriteSnack(person: "Alice", vendingMachine: vendingMachine)
} catch VendingMachineError.InvalidSelection {
    print("Неверный выбор")
} catch VendingMachineError.OutOfStock {
    print("Товара нет в наличии")
} catch VendingMachineError.InsufficientFunds(let coinsNeeded) {
    print("Недостаточно средств, пожалуйста, внесите еще \(coinsNeeded) монет(ы)")
}
```

Консоль

```
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 `Alice`, возвращается ошибка и выводится соответствующее этой ошибке сообщение.

Преобразование ошибки в опционал

Для преобразования выброшенной ошибки в опциональное значение используется оператор `try?`. Если в этом случае выбрасывается ошибка, то значение выражения вычисляется как `nil`.

Рассмотрим пример из листинга 35.8.

Листинг 35.8

```
func someThrowingFunction() throws -> Int {
    // ...
}
let x = try? someThrowingFunction()
```

Если функция `someThrowingFunction()` выбросит ошибку, то в константе `x` окажется значение `nil`.

Подавление выброса 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 подавляет выброшенную ошибку, или, иными словами, запрещает передачу ошибки.

Рассмотрим пример из листинга 35.9.

Листинг 35.9

```
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 программа экстренно завершится.

35.3. Структуры и классы для обработки ошибок

Как было сказано в начале главы, не только перечисления, но и другие объектные типы могут быть использованы для описания возникающих исключительных ситуаций.

В качестве примера реализуем структуру `NetworkError`, соответствующую протоколу `Error` (листинг 35.10).

Листинг 35.10

```
struct NetworkError: Error {}
```

В этом случае для того, чтобы поймать ошибку с помощью конструкции `do-catch`, необходимо использовать ключевое слово `is`, чтобы сопоставить тип ошибки с объектным типом `NetworkError` (листинг 35.11).

Листинг 35.11

```
do {
    // принудительно вызываем исключительную ситуацию
    throw NetworkError()
} catch is NetworkError  {
    print("it is network error")
} catch {
   // ...
}
```

Но объектный тип, описывающий ошибку, может содержать произвольные свойства и методы (листинг 35.12).

Листинг 35.12

```
struct NetworkError: Error {
    var code: Int
    func description() -> String {
        return "it is network error with code \(code)"
    }
}

do {
    // принудительно вызываем исключительную ситуацию
    throw NetworkError(code: 404)
} catch let error as NetworkError  {
    print(error.description())
} catch {
   // ...
}
```

Обратите внимание, что в этом случае в блоке `catch` используется оператор `as` вместо `is`.

Глава 36. 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 этом случае вам потребуется самостоятельно создать свои реализации стандартных операторов или полностью новые операторы.

36.1. 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 36.1).

Листинг 36.1

```
struct Vector2D {
    var x = 0.0, y = 0.0
}
```

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 36.2.

Листинг 36.2

```
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 называется перегрузкой (overloading). С данным понятием мы уже неоднократно сталкивались в книге.

Префиксные и постфиксные операторы

Оператор сложения является бинарным инфиксным — 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 36.3).

Листинг 36.3

```
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 36.4 приведен пример реализации составного оператора присваивания-сложения для экземпляров типа `Vector2D`.

Листинг 36.4

```
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, нет нужды реализовывать его в теле данной функции. Можно 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 для сравнения двух экземпляров необходимо перегрузить данный оператор с помощью операторной функции.

В следующем примере приведена реализация оператора эквивалентности и оператора неэквивалентности (листинг 36.5).

Листинг 36.5

```
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.

36.2. 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 36.6).

Листинг 36.6

```
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 VI

ВВЕДЕНИЕ В МОБИЛЬНУЮ РАЗРАБОТКУ

Вы прошли весь предлагаемый материал и выполнили все задания, а значит, можете с гордостью заявить, что имеете навыки программирования на Swift! Но это лишь начало пути до нашей основной цели «стать профессиональным Swift-разработчиком под iOS». В дальнейшем вам предстоит научиться еще многому.

Эта глава будет введением в разработку iOS-приложений, она подготовит вас к материалу второй и последующих книг[1] по разработке приложений на Swift. Уже скоро мы попробуем в деле возможности UIKit и SwiftUI, совместно создав несколько несложных приложений. В процессе этого мы подробно разберем интерфейс Xcode и новые функциональные элементы и понятия: Storyboard, View Controller, Interface Builder, графические элементы библиотеки UIKit и многое другое.

Держитесь крепче, будет интересно!

- ✓ Глава 37. Разработка приложения с использованием UIKit
- ✓ Глава 38. Разработка приложения с использованием SwiftUI
- ✓ Глава 39. Паттерны проектирования

[1] Всю актуальную информацию о книгах серии вы всегда можете найти на сайте https://swiftme.ru.

Глава 37. Разработка приложения с использованием UIKit

UIKit — это основа любого iOS-приложения. В процессе его освоения вам потребуется пройти еще много учебного материала, так как данный фреймворк отвечает не только за создание графического интерфейса, но и за функционирование приложений в принципе.

Эта глава будет посвящена разработке первого (не будем учитывать playground-проект с шариками) мобильного приложения с использованием UIKit. Оно не будет иметь особой ценности для потенциального пользователя, но даст вам возможность попрактиковаться в создании простого графического интерфейса, программировании реакций на события (например, выполнение определенных задач при нажатии на кнопку) и взаимодействии с рабочей средой Xcode. Каждый шаг процесса разработки будет разобран максимально подробно и при необходимости подкреплен рисунками.

37.1. Создание проекта MyName

Мы приступаем к созданию вашего первого iOS-приложения, в процессе разработки которого вы познакомитесь с некоторыми базовыми понятиями и функциональными возможностями среды.

- В Xcode создайте новый проект (**не** playground). В качестве платформы выберите iOS, а в качестве шаблона — App (рис. 37.1), после чего нажмите Next.
- На экране настроек нового проекта укажите следующие значения (рис. 37.2):
    - Product Name (название проекта) — MyName.
    - Language (язык программирования) — Swift.
    - User Interface (способ создания пользовательского интерфейса) — Storyboard.
    - Галочки в пунктах Use Core Data и Include Tests должны быть сняты.
    - Все остальные поля заполните по своему усмотрению. Они были рассмотрены ранее в ходе создания консольного приложения.

ПРИМЕЧАНИЕ Выбранное в пункте User Interface значение определяет, каким образом будет создаваться интерфейс приложения: средствами UIKit (для этого должно быть выбрано значение Storyboard) или SwiftUI. При этом выбор одного варианта не отменяет возможности использовать другое средство в процессе работы над проектом. На основании значения данного пункта Xcode автоматически проводит некоторые предварительные настройки проекта.

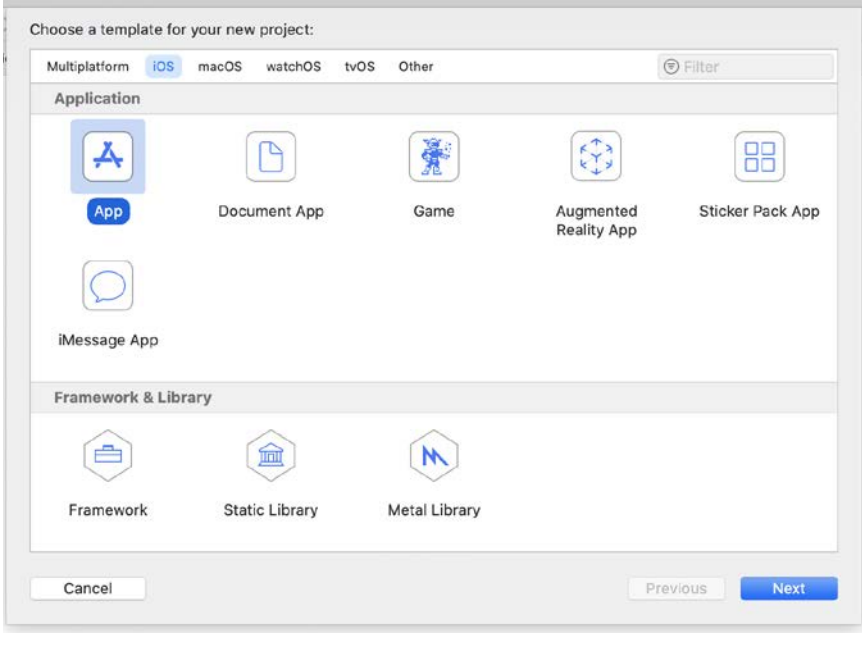

Рис. 37.1. Создание нового проекта

После подтверждения создания и сохранения проекта перед вами откроется рабочая среда Xcode. Сейчас в Project Editor отображаются настройки проекта (рис. 37.3). Напомню, что для доступа к этим настройкам в Project Navigator необходимо выбрать корневой объект дерева ресурсов (в данном проекте это элемент MyName с синей иконкой).

ПРИМЕЧАНИЕ Если вы забыли названия и предназначение рабочих областей среды разработки, то вернитесь к материалу главы, посвященной разработке первого консольного приложения.

В верхней части Project Editor расположена панель вкладок, с помощью которой можно переключаться между различными группами настроек (рис. 37.4). В настоящий момент активна вкладка General, и именно ее состав отображается в редакторе проекта. С ее помощью вы можете изменить основные настройки приложения.

Все доступные для настройки параметры во вкладке General разделены на подразделы.

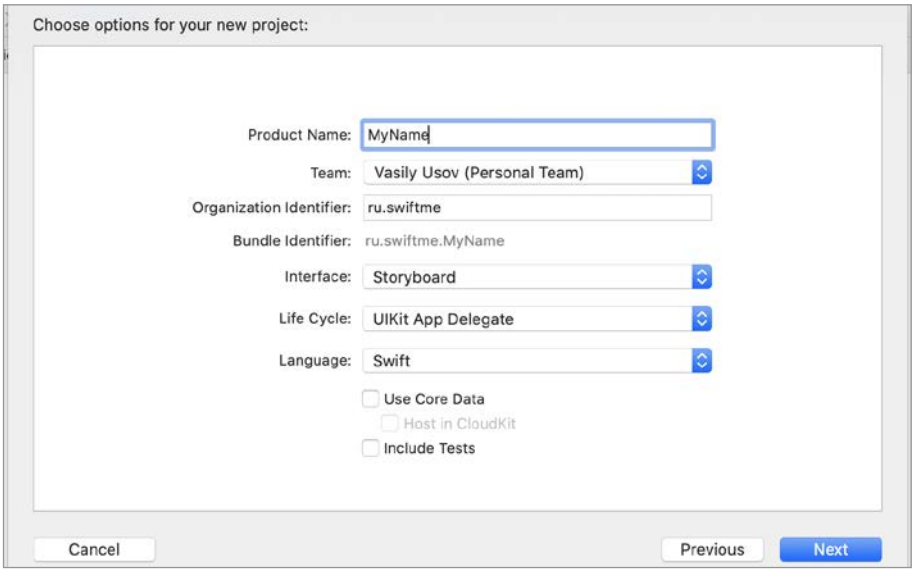

Рис. 37.2. Настройки нового проекта

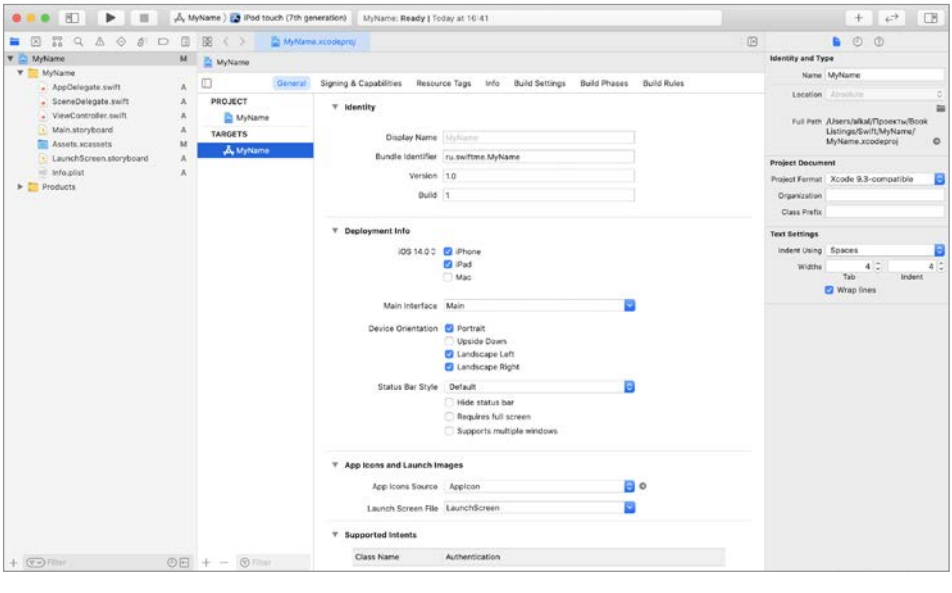

Рис. 37.3. Настройки проекта MyName

Рис. 37.4. Вкладки настроек проекта

Раздел Identify содержит следующие параметры:

- Display Name — название приложения, которое будет отображаться под иконкой установленного приложения.
- Bundle Identifier — идентификатор продукта.
- Version — версия приложения.
- Build — номер сборки приложения.

Раздел Deployment Info позволяет вам изменить перечень поддерживаемых устройств, версию операционной системы, доступную ориентацию (портретный и ландшафтный режимы) и некоторые другие параметры.

Прежде чем вы разместите новое приложение в магазине AppStore, его необходимо подписать с помощью цифровой подписи. Возможность (или невозможность) совершения этой операции указана во вкладке Signing & Capabilities. На данном этапе приложение будет тестироваться на специальном симуляторе (программном аналоге реального iPhone), поэтому нам не придется задумываться об этом.

ПРИМЕЧАНИЕ Со временем мы рассмотрим большинство вкладок и настроек, доступных на данном экране.

37.2. Структура проекта

Любой Xcode-проект состоит из множества элементов, каждый из которых предназначен для решения конкретной задачи. В этом разделе мы рассмотрим некоторые из них.

На панели Project Navigator приведен текущий состав проекта (рис. 37.5).

Проект состоит из следующих элементов:

- Файлы с расширением .swift (AppDelegate.swift, SceneDelegate.swift и ViewController.swift) содержат исходный код приложения.
- Файлы с расширением .storyboard (Main.storyboard и LaunchScreen.storyboard) предназначены для создания графического интерфейса приложения.
- Папка с расширением .xcassets предназначена для хранения различных ресурсов вроде иконки приложения, картинок и т. д.

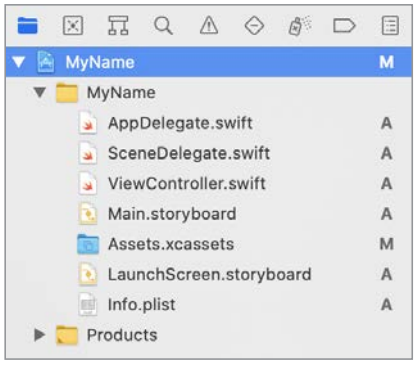

Рис. 37.5. Структура проекта MyName

- Файл с расширением .plist (расшифровывается как property list) предназначен для хранения настроек приложения. Info.plist содержит основные настройки, которые используются для запуска и функционирования приложения. В дальнейшем мы научимся создавать и использовать plist-файлы для хранения собственных данных.

Сейчас нас интересуют storyboard-файлы.

➤ В Project Navigator щелкните по файлу Main.storyboard.

Обратите внимание на то, как изменился Project Editor (рис. 37.6): перед вами открылся редактор интерфейса (Interface Builder, сокращенно IB), который обеспечивает удобный визуальный способ создания и редактирования графического интерфейса приложения. С помощью Interface Builder вы можете создавать рабочие экраны вашего приложения методом WYSIWYG (What You See Is What You Get — «что видишь, то и получишь»). Таким образом, с помощью простых движений мышки вы можете размещать графические элементы и настраивать их свойства. Способ размещения элементов в Interface Builder определяет то, как вы будете видеть их на своем устройстве в запущенном приложении.

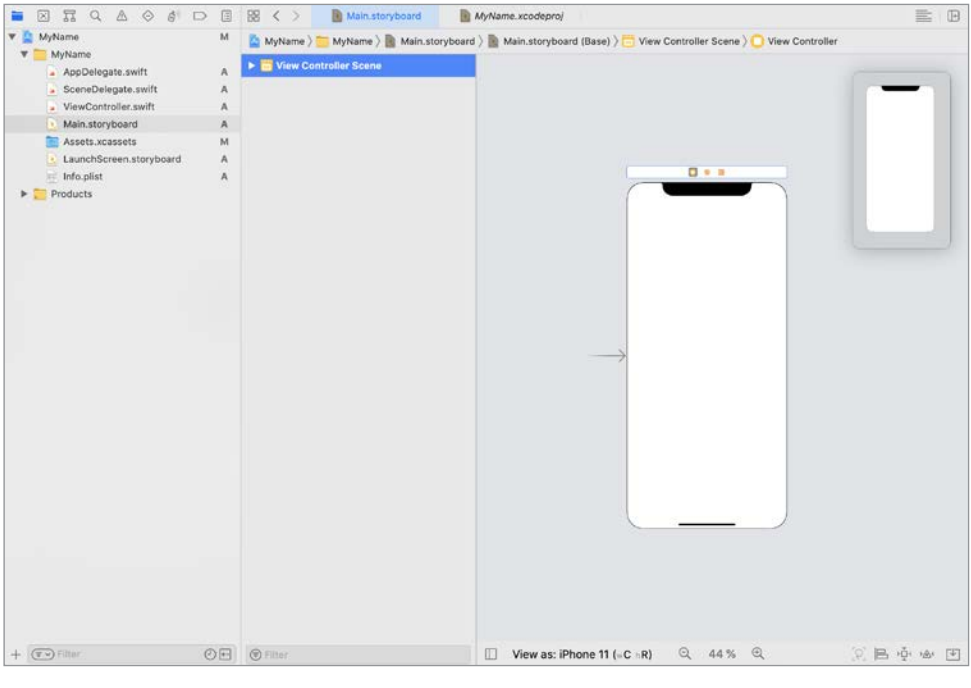

Рис. 37.6. Редактор интерфейса

ПРИМЕЧАНИЕ Вы можете изменять масштаб Storyboard и перемещаться по нему с помощью сенсорной поверхности Magic Mouse или тачпада.

Interface Builder позволяет вам создавать (другими словами, верстать) рабочие экраны приложения.

Но что такое рабочий экран? Взаимодействие с любым iOS-приложением основано на рабочих экранах. Например, сразу после загрузки приложение показывает вам экран авторизации, после ввода данных и нажатия на кнопку «Войти» вы попадаете на домашний экран, и т. д.

Рабочие экраны в контексте Interface Builder и storyboard-файлов называются сценами (scene) (рис. 37.7).

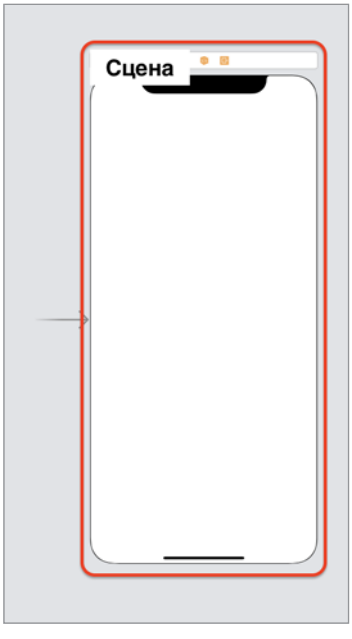

Рис. 37.7. Сцены в Storyboard

При этом в составе любой сцены можно выделить следующие элементы:

- отображения (`view`, которые программисты называют «вьюшками») графических элементов сцены (кнопок, надписей, таблиц и т. д.);
- дополнительные элементы, обеспечивающие функциональность сцены, которые будут рассмотрены чуть позже. Например, к таким элементам относятся *ограничения* (constraints), задающие правила расположения элементов на сцене.

Storyboard (раскадровка) — это совокупность сцен и связей между ними, представленная в виде файла с расширением .storyboard. Со временем в ваших раскадровках будут появляться все новые и новые сцены, описывающие новые рабочие экраны приложения. Пример storyboard со множеством сцен приведен на рис. 37.8.

Interface Builder предоставляет вам возможность создавать графический интерфейс приложений практически без написания программного кода!

Помимо Main.storyboard, в состав проекта входит файл LaunchScreen.storyboard. Он отвечает за экран загрузки приложения, то есть за графический интерфейс, который отображается после того, как пользователь запустил приложение, но до того, как приложение будет полностью загружено. Как только приложение загрузится, на экране устройства отобразится стартовая сцена (в нашем случае это единственная сцена из файла Main.storyboard). Пока мы будем работать только с Main.storyboard, а при загрузке приложения будет отображаться белый экран.

ПРИМЕЧАНИЕ Файлы с расширением .storyboard описывают интерфейс сцен приложения, а хранится эта информация в виде данных в формате XML. Чтобы убедиться в этом, щелкните правой кнопкой мышки по Main.storyboard в Project Navigator, выберите пункт Show in Finder и откройте этот файл в любом редакторе кода, например Sublime Text (рис. 37.9).

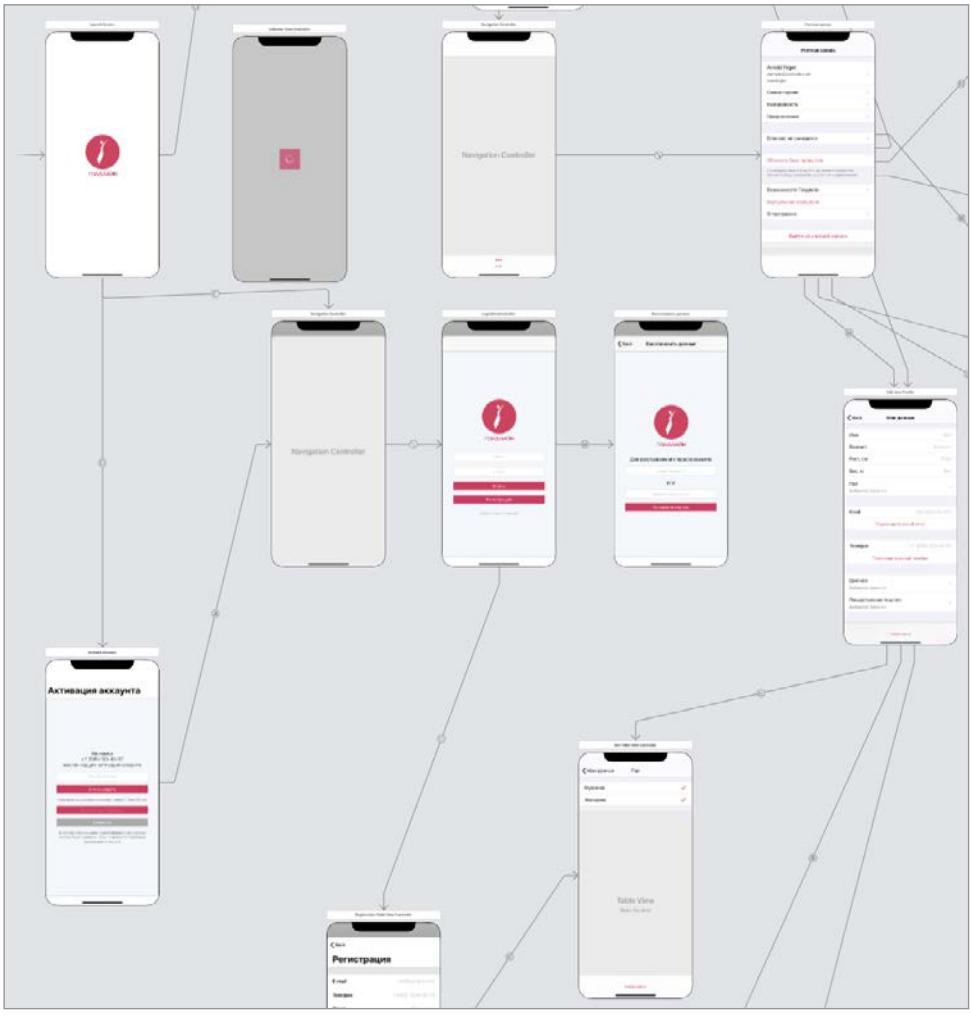

Рис. 37.8. Несколько сцен в одном storyboard-файле

Обратите внимание на Jump Bar — панель перехода, расположенную прямо над Interface Builder (рис. 37.10). Это очень полезный функциональный элемент Xcode: с его помощью без использования Project Navigator можно выбрать ресурс, который требуется отобразить в Project Editor.

Ранее на этапе создания проекта в качестве шаблона был выбран App. По умолчанию проект такого типа в файле Main.storyboard содержит одну сцену, то есть один рабочий экран (см. рис. 37.6).

В левой части Interface Builder находится панель Document outline, описывающая структуру текущего storyboard и каждой отдельной сцены в его составе (рис. 37.11).

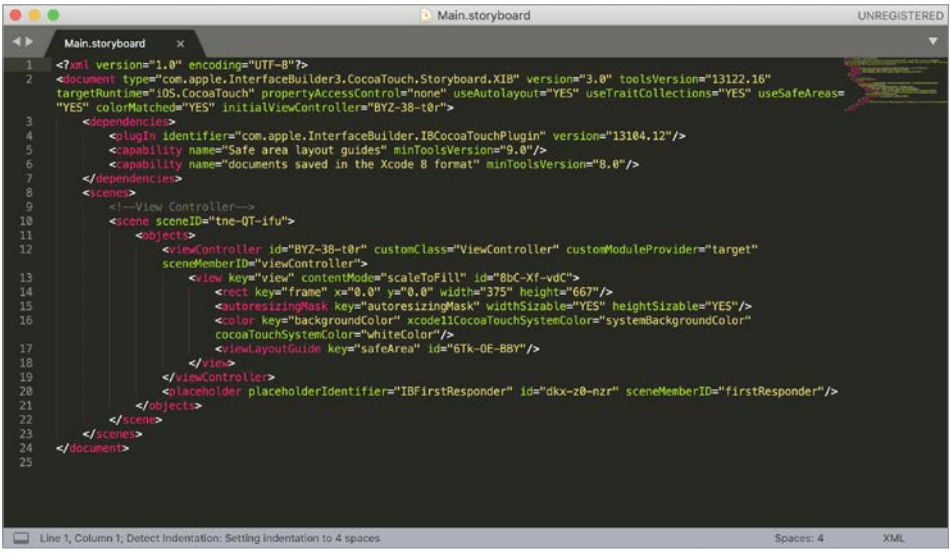

Рис. 37.9. Редактор кода и файл Main.storyboard

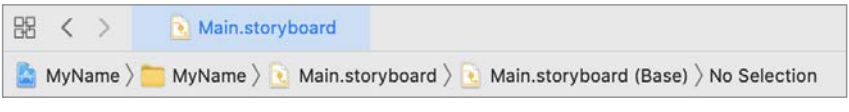

Рис. 37.10. Панель Jump Bar

Как было отмечено, сейчас файл Main.storyboard описывает всего одну сцену, которой в Document outline соответствует элемент View Controller Scene.

- Если какие-либо элементы в составе сцены в Document outline не видны (ориентируйтесь на рис. 37.11), то отобразите их с помощью серых стрелочек, расположенных слева.

Рис. 37.11. Структура storyboard

Если Project Navigator (панель в левой части Xcode) позволяет получить доступ к различным ресурсам проекта, в том числе к storyboard-файлам, то благодаря Document outline вы получаете возможность доступа к отдельным элементам storyboard и каждой сцены.

Элемент View Controller Scene, описывающий сцену, состоит из следующих элементов:

- View Controller — это своего рода менеджер сцены. Он управляет всем, что происходит со сценой и на ней: отвечает за ее загрузку на экран устройства, следит за всеми событиями и элементами. С помощью этого элемента вы будете проводить предварительную настройку сцены перед ее показом на экране

устройства, а также наполнять элементы данными (заполнять таблицы, изменять изображения, редактировать надписи и т. д.).

- View — это экземпляр класса `UIView`, который является старшим графическим элементом на данной сцене. В состав данного View могут входить другие графические элементы (например, кнопки, надписи), которые будут последовательно выводиться на экране устройства. Так, при отображении сцены View Controller сперва выводит данный View, а потом и все отображения, которые входят в него.

ПРИМЕЧАНИЕ Вы уже встречались с тем, что одни View входят в состав других, когда разрабатывали проект с шариками. Там родительским отображением (view) была прямоугольная подложка, а шарики представляли собой дочерние отображения.

➤ Выделите элемент View в Document Outline.

Xcode автоматически подсветил на сцене соответствующий данному элементу прямоугольник. Рассматривайте этот элемент как фоновый, на котором будут располагаться все будущие элементы интерфейса.

Safe Area — это специальный элемент, описывающий сцену без верхнего бара с данными о сотовой сети, заряде батареи и другой системной информацией (а также небольшую область в нижней части экрана для версий iPhone без кнопок). Safe Area используется для удобства позиционирования других графических элементов на сцене. С его помощью вам не придется задумываться о том, что кнопки и надписи заползают на поля с системной информацией.

➤ Выделите элемент Safe Area в Document Outline и посмотрите на подсвеченную на сцене область.

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, то именно оно и будет являться First responder.

Exit служит единственной цели и используется только в приложениях со множеством сцен. Когда вы создаете программу, которая перемещает пользователя между сценами, то Exit обеспечивает перемещение к одному из предыдущих экранов.

Storyboard Entry Point — это признак стартовой (или, иными словами, инициирующей) сцены приложения. Сцена, отмеченная такой стрелкой, будет отображена первой после завершения загрузки приложения.

ПРИМЕЧАНИЕ Не переживайте, если предназначение каких-либо (или всех) элементов понятно вам не в полной мере. Со временем вы поработаете с каждым из них.

Для некоторых элементов из Document outline существуют аналоги на сцене в storyboard (рис. 37.12).

➤ Поочередно щелкайте по элементам в Document outline и посмотрите, что будет подсвечиваться на изображении сцены справа.

ПРИМЕЧАНИЕ Если в верхней части вашей сцены вместо трех иконок отображается надпись View Controller, то для их отображения достаточно щелкнуть на любом элементе в Document outline.

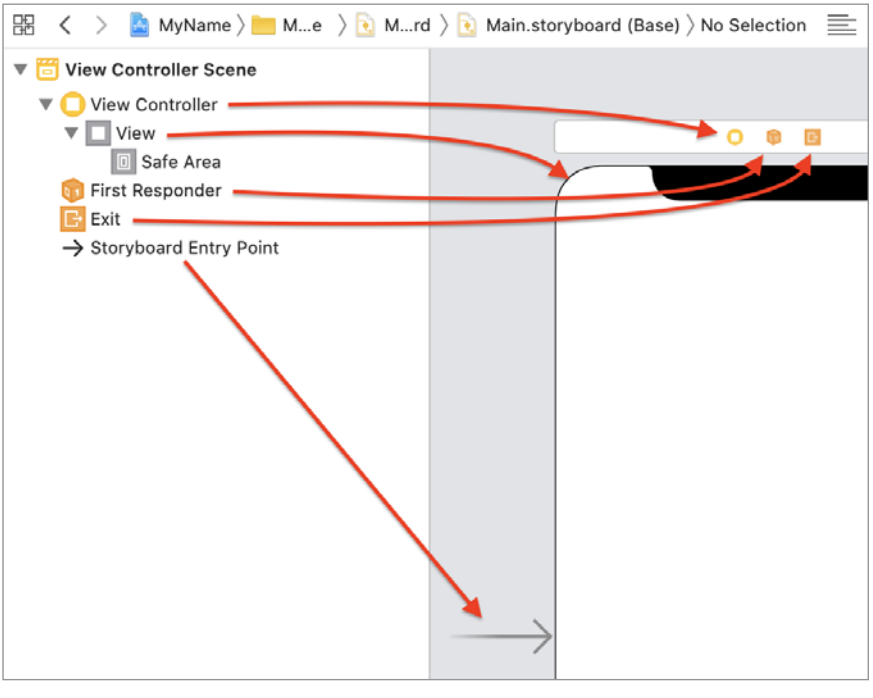

Рис. 37.12. Соответствие элементов Document outline и элементов сцены

37.3. Разработка простейшего UI

➤ Убедитесь, что в Project Navigator выделен файл Main.storyboard, а в редакторе проекта отображается Interface Builder.

При разработке приложений Xcode позволяет использовать предустановленные шаблоны графических элементов, таких как текстовое поле, надпись, ползунок, кнопка и многие другие. Для доступа к библиотеке объектов нажмите кнопку Library (+), расположенную в Toolbar, после чего перед вами откроется окно со всеми доступными элементами (рис. 37.13).

Перейдем непосредственно к использованию IB и разработке интерфейса приложения.

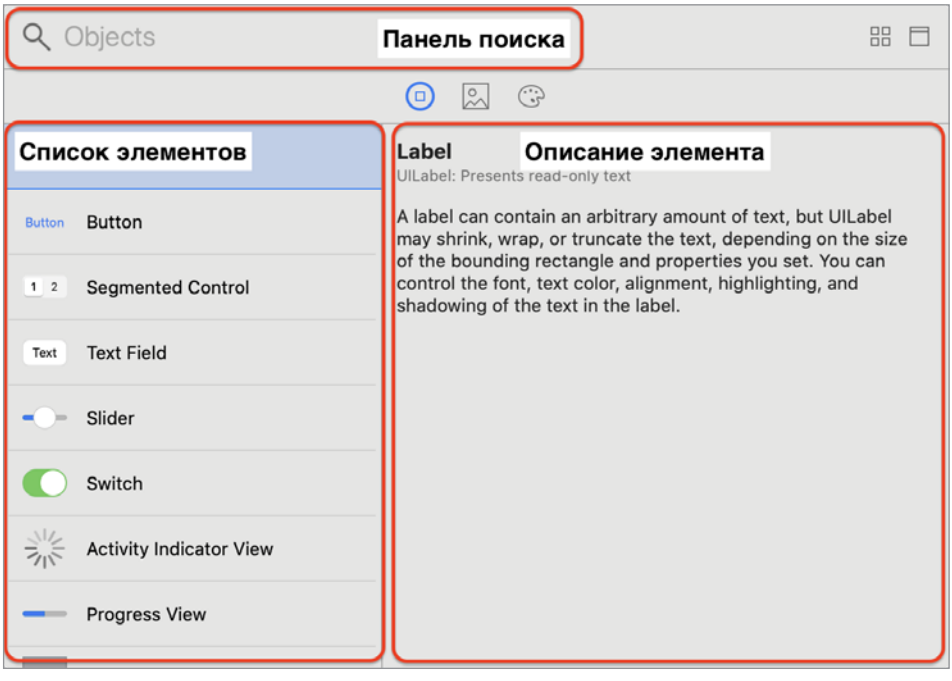

Рис. 37.13. Библиотека объектов

Шаг 1. Размещение кнопки

Первым делом на сцене необходимо разместить кнопку (рис. 37.14):

- (1) В библиотеке объектов найдите элемент Button (используйте поле поиска).
- (2) Перетащите его прямо из библиотеки в центр сцены в Interface Builder. При этом во время позиционирования на сцене для удобства будут отображаться синие вертикальная и горизонтальная линии, пересечение которых соответствует центру.
- (3) Теперь в Document outline появился новый элемент Button, соответствующий только что размещенной кнопке.

Каждый графический элемент, который вы можете разместить на сцене, входит в состав фреймворка `UIKit` и описывается конкретным классом. С одной стороны, у вас есть графическое изображение элемента, а с другой — его программное представление (класс). Любой графический элемент описывается классом, дочерним для `UIView`. `UIView` — это один из базовых классов фреймворка `UIKit`, на основе которого строятся все графические элементы, входящие в состав фреймворка. Так, только что размещенная кнопка представлена классом `UIButton`, который является дочерним по отношению к `UIView`.

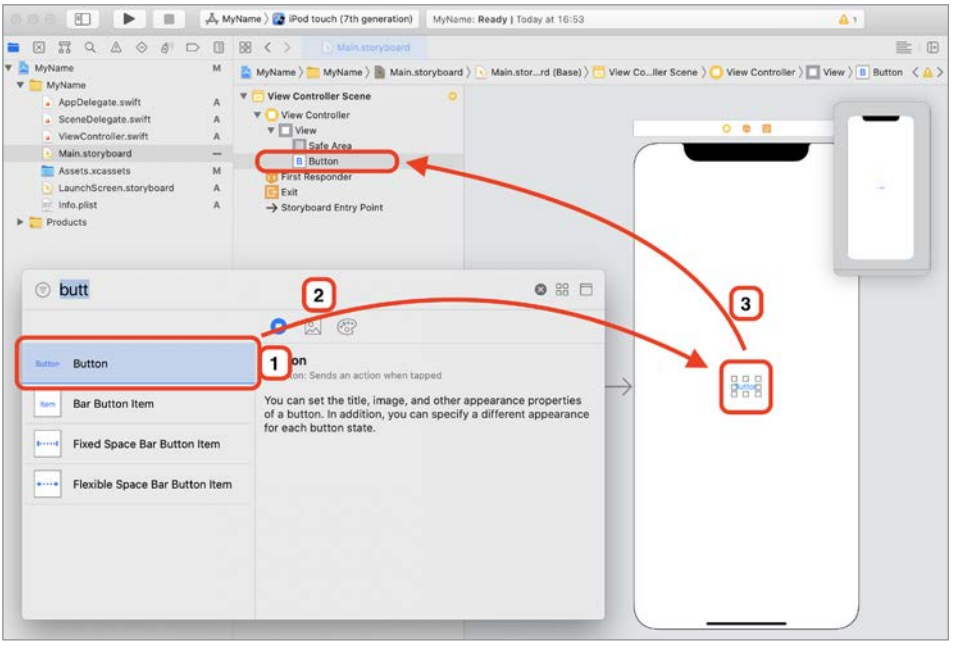

Рис. 37.14. Размещение элемента на сцене

ПРИМЕЧАНИЕ Стоит отметить, что в действительности класс UIButton наследуется от UIControl, который, в свою очередь, наследуется от UIView. Все графические элементы представляют собой довольно большую иерархию классов, каждый из которых дополняет родителя некоторыми возможностями. Но сейчас этот материал, скорее всего, сложен для понимания, а поэтому мы будем возвращаться к нему неоднократно в этой и будущих книгах.

Шаг 2. Изменение текста кнопки

Для изменения текста есть два варианта действий.

Вариант 1:

- Дважды щелкните по кнопке прямо на сцене и введите текст `"Hello World"`.
- Нажмите Enter на клавиатуре для сохранения результата.

Вариант 2 (рис. 37.15):

- (1) Выделите кнопку на сцене или в Document outline.
- (2) Откройте панель Attributes Inspector.
- (3) Введите текст `"Hello World"` в поле Title.

ПРИМЕЧАНИЕ После изменения текста вам может потребоваться повторная центровка кнопки на сцене, так как ее размер изменился.

Рис. 37.15. Изменение текста кнопки

37.4. Запуск приложения в симуляторе

Разработка iOS-проектов была бы чрезвычайно сложной, если бы в целях тестирования разрабатываемые приложения требовалось запускать только на реальных устройствах. Но в комплекте Xcode есть специальные симуляторы всех доступных на рынке устройств собственного производства. При этом вы можете тестировать проект на симуляторе любого устройства с любой операционной системой.

Список доступных по умолчанию симуляторов зависит от версии Xcode, но в любой момент вы можете посмотреть и при необходимости скачать их. Доступ к симуляторам предоставляет пункт меню Xcode -> Preferences -> Components (рис. 37.16). Уже скачанные симуляторы отмечены синей галочкой, а доступные для загрузки — серой стрелочкой.

> **ПРИМЕЧАНИЕ** Каждый из симуляторов занимает довольно большой объем пространства, поэтому старайтесь не тратить место попусту, устанавливая симуляторы, которые не будете использовать.

При создании консольного приложения вы уже научились производить запуск приложения. В текущем проекте панель инструментов также содержит набор кнопок для запуска сборки и осуществления принудительной остановки запущенного приложения. При этом дополнительно имеется возможность выбрать устройство, на симуляторе которого будет запущено приложение (рис. 37.17). Вопрос тестирования приложений на различных симуляторах становится очень актуальным в связи с тем, что сегодня приложения функционируют на большом числе устройств с различным разрешением и версией операционной системы.

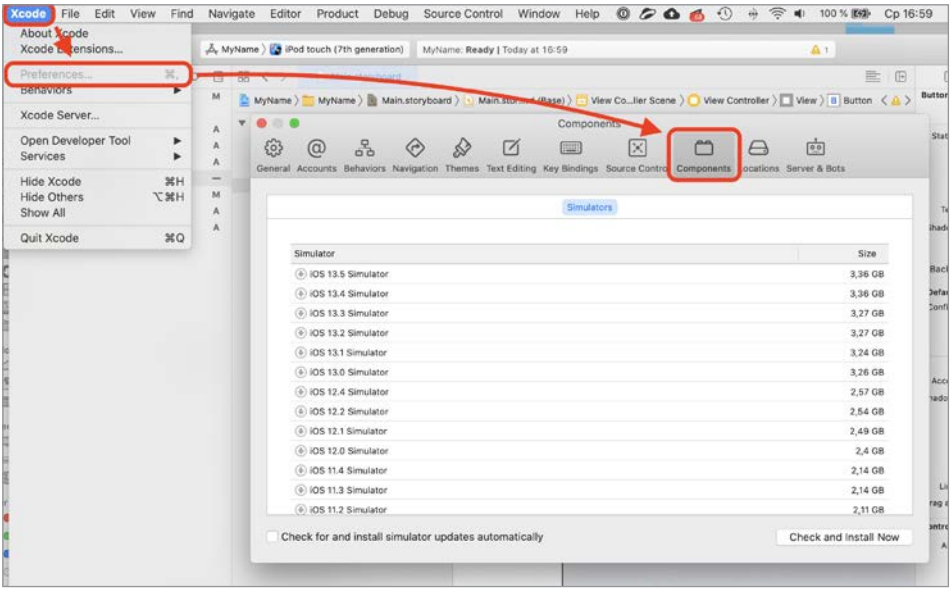

Рис. 37.16. Список симуляторов

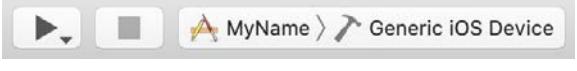

Рис. 37.17. Кнопки запуска проекта

- В качестве устройства выберите один из доступных iPhone таким образом, чтобы выбранная версия устройства соответствовала той, что указана в нижней части Interface Builder (рис. 37.18).
- Запустите приложение с помощью кнопки Build and run.

После успешной компиляции и запуска симулятора ваше приложение отобразится на его экране, а в центре будет размещена кнопка с текстом синего цвета (рис. 37.19). Щелчки по кнопке не дадут эффекта, так как пока мы не запрограммировали какие-либо действия для нее.

- Остановите запущенное приложение с помощью кнопки Stop the running scheme or application и на панели инструментов.
- Выберите другое устройство в списке доступных симуляторов. При этом сделайте так, чтобы разрешение экрана данной версии iPhone отличалось от выбранной ранее. К примеру, если вы до этого запускали проект на iPhone 11 или iPhone Xr, то теперь выберите iPhone 7 или другую модель. При этом значения выбранного симулятора должно отличаться от модели, указанной в нижней части Interface Builder.

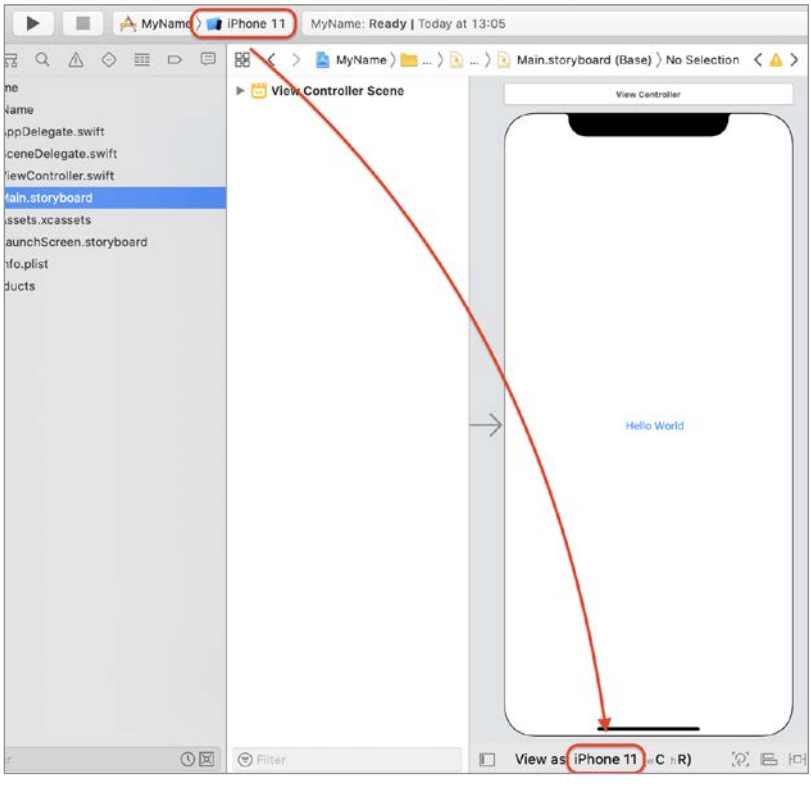

Рис. 37.18. Выбор версии симулятора

- Повторно запустите сборку проекта.

На этот раз кнопка Hello World будет расположена не в центре, а немного ниже! И на первый взгляд это удивительно (рис. 38.20). Чтобы понять, в чем проблема, вернемся в Xcode. Вспомните, что в нижней части Interface Builder также указывается версия устройства. Она определяет, для экрана какого конкретно устройства производится верстка интерфейса. Так, размещая кнопку на сцене, мы позиционировали ее в центре экрана iPhone 11 (или другой версии, указанной в нижней части Interface Builder). При этом для кнопки были зафиксированы конкретные координаты (отступ от верхнего левого угла в пикселах, а если быть точным, в пойнтах). При запуске программы на другой версии iPhone координаты были сохранены, но они уже не соответствовали центру (так как разрешение iPhone 7 отличается от iPhone 11).

Xcode имеет специальные механизмы, позволяющие одновременно верстать сцены сразу под большое количество устройств. Таким образом, вам потребуется задавать не конкретные координаты для каждого разрешения, а общие типовые ограничения, которым должен соответствовать графический элемент на любом

запущенном устройстве. Знакомству с этими механизмами будут посвящены несколько глав следующей книги.

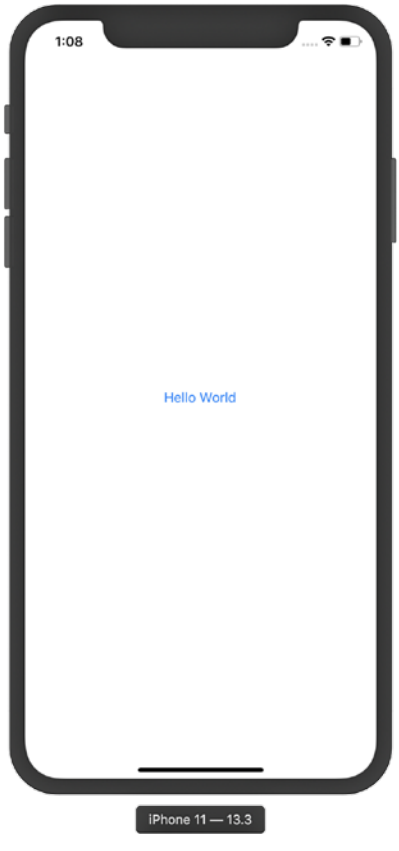

Рис. 37.19. Окно симулятора

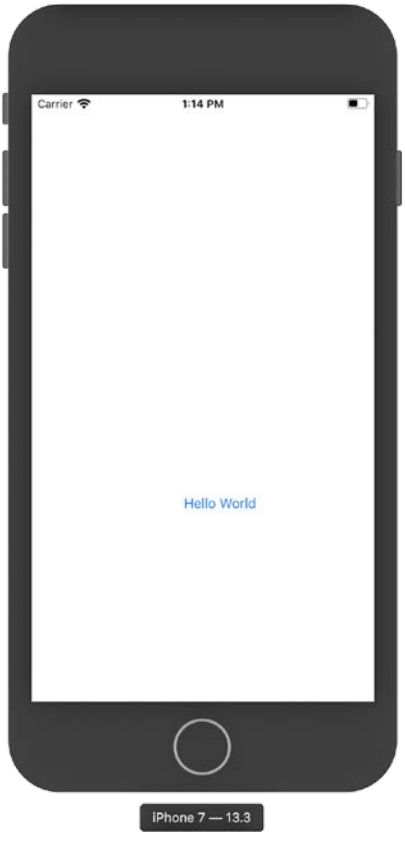

Рис. 37.20. Запуск приложения на симуляторе

Сейчас же сделаем так, чтобы кнопка отображалась в центре на любом устройстве, на котором будет запущено приложение (рис. 37.21):

- (1) Выделите кнопку на сцене.
- (2) Нажмите кнопку Align в нижней части Interface Builder.
- (3) В открывшемся окне выделите пункты Horizontally in Container и Vertically in Container.
- Нажмите кнопку Add 2 Constraints.

Вы только что добавили ограничение (constraint), которое заставит кнопку автоматически позиционироваться в центре экрана любого запущенного устройства

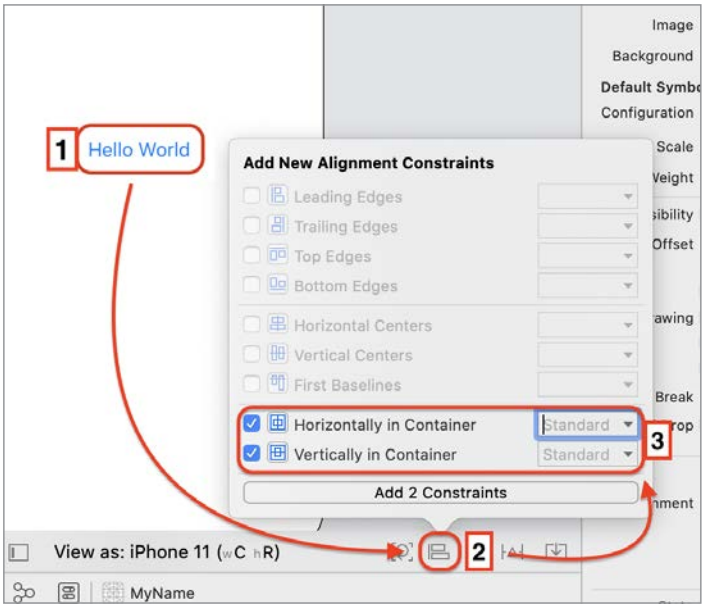

Рис. 37.21. Запуск приложения на симуляторе

(любой версии iPhone с любым разрешением). Ограничения — довольно сложная, но при этом очень интересная тема, изучению которой будет посвящено большое количество учебного материала в будущих книгах. С помощью ограничений можно определять различные правила позиционирования элементов: отступы друг от друга, отступы от границ экрана, выравнивание и многое другое.

- Запустите приложение на симуляторах различных устройств, тем самым убедившись, что установленное ограничение работает и кнопка всегда располагается в центре экрана.

Теперь перейдем к наполнению приложения функциональными возможностями, а именно к программированию действий по нажатию кнопки.

37.5. View Controller сцены и класс UIViewController

В состав фреймворка `UIKit` входит класс `UIViewController`. На его основе разрабатываются дочерние классы, которые связываются со сценами в storyboard, таким образом создавая связь между кодом и графическим представлением. Когда вы создаете новую сцену, то должны самостоятельно создать новый класс-потомок `UIViewController` и связать его с элементом View Controller на сцене (отображается в составе сцены в Document Outline). Так как при создании проекта мы выбрали шаб-

лон App, то он уже содержит одну сцену и связанный с ней класс `ViewController`, который можно найти в файле ViewController.swift.

- В Project Navigator щелкните на файле ViewController.swift, после чего его содержимое отобразится в Project Editor.

В коде файла присутствует класс `ViewController`, потомок `UIViewController` (листинг 37.1).

Листинг 37.1

```
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.
    }
}
```

Класс `ViewController` связан с элементом View Controller на сцене. Именно с его помощью мы и будем управлять сценой. Чтобы увидеть эту связь, выполните следующие действия (рис. 38.22):

- (1) Выделите файл Main.storyboard в Project Navigator.
- (2) Выделите элемент View Controller в составе сцены в Document Outline.
- (3) Откройте панель Identity Inspector.
- (4) В поле Class данного элемента будет указан класс `ViewController`, который как раз и описан в файле ViewController.swift.

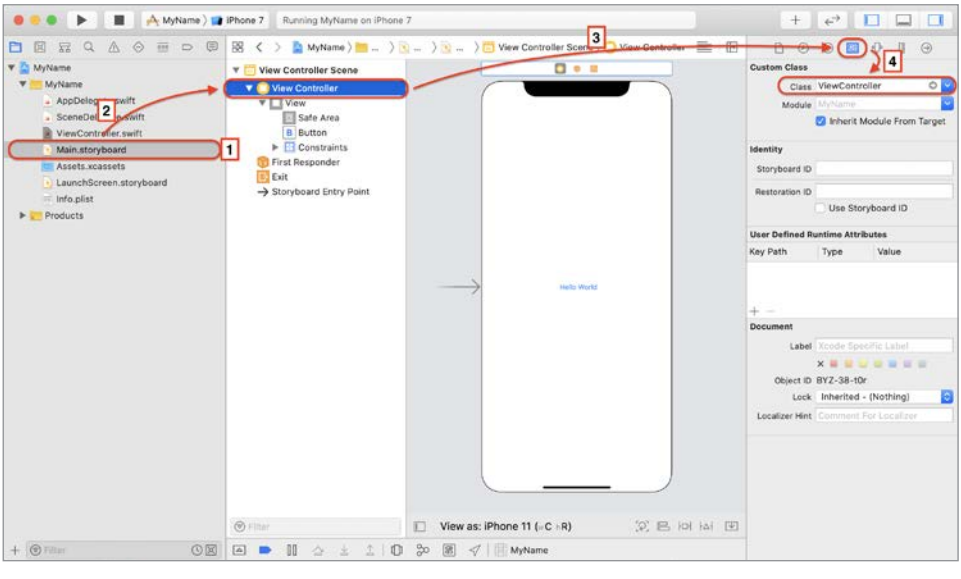

Рис. 37.22. Связь объекта View Controller и класса ViewController

Работая над сложными проектами, вы будете создавать большое количество различных сцен (а значит, и элементов View Controller), классов и связей между ними.

Вернемся к файлу ViewController.swift. Обратите внимание, что в верхней строчке файла происходит импорт фреймворка `UIKit` (листинг 37.2).

Листинг 37.2

```
import UIKit
```

`UIKit` 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.

Класс `UIViewController` входит в состав `UIKit` и доступен в коде только потому, что произведен импорт данного фреймворка. Если из кода исключить строку `import UIKit`, то Xcode сообщит об ошибке доступа к объявленному классу `UIViewController`. Одной из функций `UIViewController` (и его потомков) является обновление содержимого отображений сцены, с которой он связан. Обновление может происходить по разным причинам, к примеру, в ответ на действия пользователей (например, была нажата кнопка на сцене, после чего интерфейс приложения стал темным).

Родительский класс `UIViewController` имеет большое количество методов, которые могут быть переопределены в потомках. Каждый из этих методов служит для решения конкретной задачи. Со многими из таких методов вы познакомитесь в процессе обучения разработке в Xcode.

Так, например, метод `viewDidLoad()`, описанный в классе `ViewController`, позволяет выполнить программный код после того, как сцена загружена в память устройства и готова отобразиться на его экране (но до отображения). Несмотря на то что `viewDidLoad()` не содержит какой-либо код, а только вызывает одноименный родительский метод, при необходимости он может быть расширен для реализации требуемой функциональности.

> **ПРИМЕЧАНИЕ** Метод `viewDidLoad()` будет подробно рассмотрен в следующей книге в главе, описывающей жизненный цикл `View Controller`.

Сейчас же в данном методе нет необходимости, поэтому уберем его из кода, оставив пустой класс `ViewController` (листинг 37.3).

Листинг 37.3

```
class ViewController: UIViewController {}
```

37.6. 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 расширения возможностей сцены необходимо работать именно с этим классом. Объявим в нем новый метод `showMessage()`, использующий внутри своей реализации функцию `print(_:)`, выводящую строку `"you pressed Hello World button"` (листинг 37.4).

ПРИМЕЧАНИЕ Не забывайте, что вы можете использовать кодовые сниппеты для упрощения создания кодовых конструкций.

Листинг 37.4

```
class ViewController: UIViewController {
     func showMessage(){
          print("you pressed Hello World button")
     }
}
```

Для того чтобы появилась возможность создать связь между методом класса и элементом сцены, используется определитель типа `@IBAction` (разработчики называют его *экшеном*, или *экшн-методом*), который указывается перед определением метода. Добавим `@IBAction` к методу `showMessage()` (листинг 37.5).

Листинг 37.5

```
@IBAction func showMessage(){
     print("you press Hello World button")
}
```

ПРИМЕЧАНИЕ Обратите внимание, что в редакторе кода слева от метода `showMessage()` вместо номера строки теперь отображается круг.

С помощью ключевого слова `@IBAction` отмечаются методы, которые могут быть связаны с графическими элементами на сцене. То есть, например, мы можем связать кнопку на сцене с вызовом метода таким образом, чтобы при нажатии на нее вызывался данный метод.

ПРИМЕЧАНИЕ IBAction расшифровывается как Interface Builder Action, то есть действие, к которому можно обратиться из Interface Builder.

Так как метод `showMessage()` помечен с помощью экшена, мы можем организовать его вызов по событию «нажатие на кнопку». Для этого выполните следующие действия:

- Откройте файл Main.storyboard.
- Удерживайте клавишу Control на клавиатуре и с помощью мыши потяните кнопку сцены на желтый кружок, символизирующий View Controller, после чего отпустите (рис. 37.23). В процессе должна отображаться линия синего цвета.
- В появившемся окне выберите элемент showMessage, находящийся в разделе Sent Events (рис. 37.24).

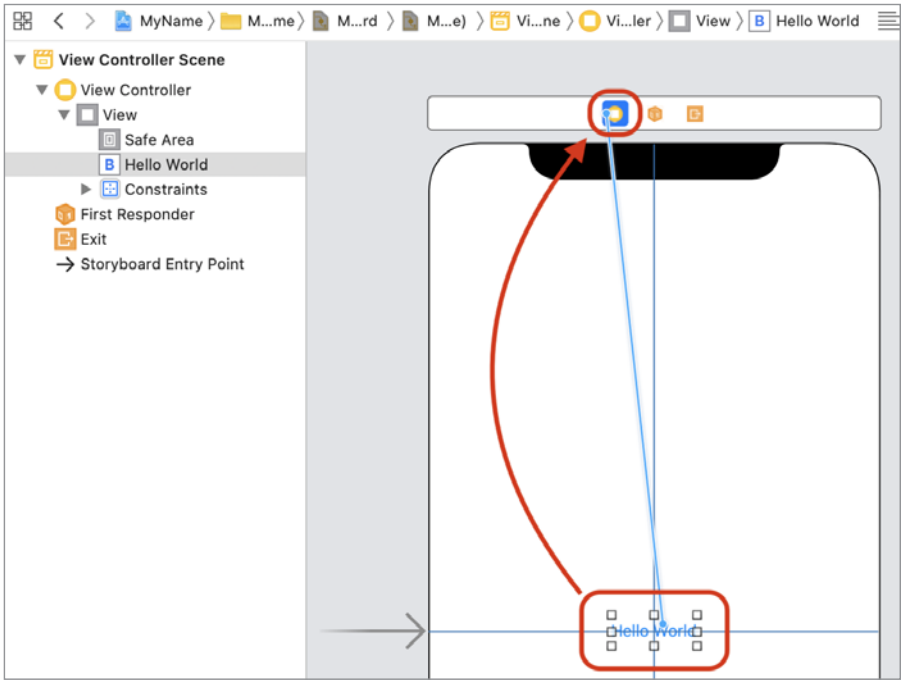

Рис. 37.23. Создание связи между кнопкой на сцене и методом showMessage. Перемещение кнопки

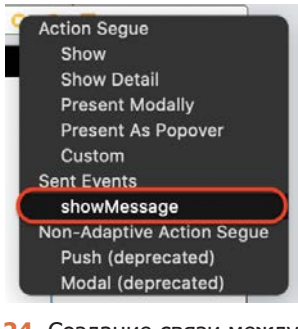

Рис. 37.24. Создание связи между кнопкой на сцене и методом showMessage. Выбор элемента

ПРИМЕЧАНИЕ Метод `showMessage()` находится в выпадающем списке и доступен для связывания, так как помечен ключевым словом `@IBAction`.

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. 37.25).

- Произведите запуск приложения на симуляторе и проверьте работоспособность созданной связи.

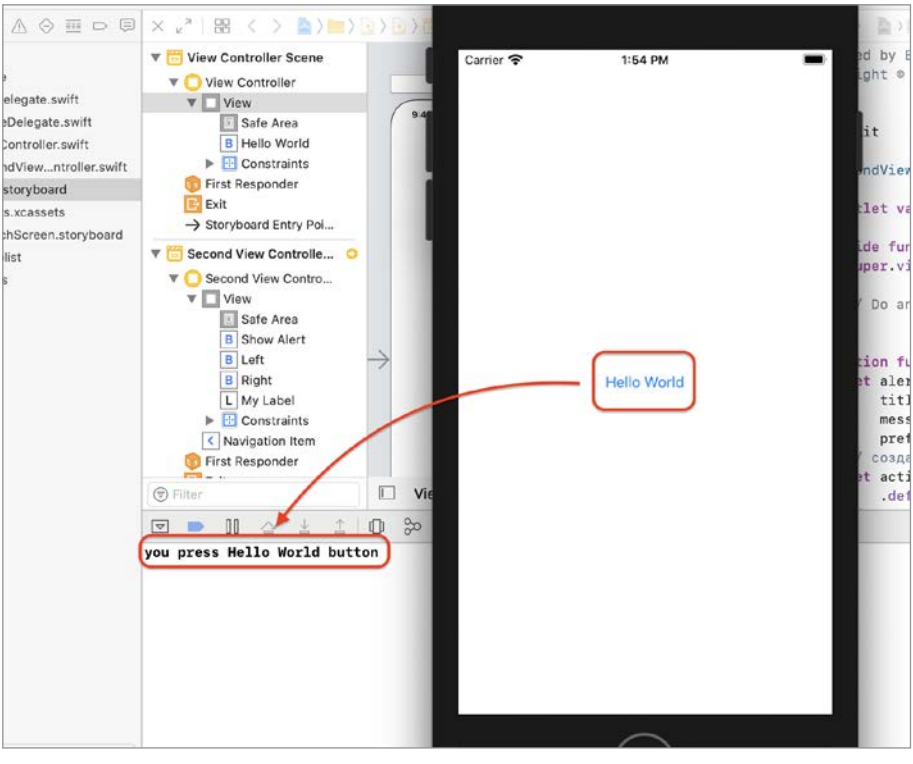

Рис. 37.25. Вывод на консоль после нажатия на кнопку

37.7. Создание дополнительной сцены

Xcode предоставляет вам множество способов создания сцен и организации перехода от одной сцены к другой. Вы можете делать это программным кодом или использовать возможности Interface Builder. На первых порах использование IB является наиболее предпочтительным способом, так как он более легкий в освоении. В этом разделе мы займемся созданием второй сцены и организуем переход к ней по нажатии кнопки.

- На сцене под кнопкой Hello World разместите новую кнопку и поменяйте ее текст на «Show Next Screen».
- В библиотеке объектов найдите элемент View Controller и перетащите его на storyboard, разместив справа от существующей сцены (рис. 37.26).

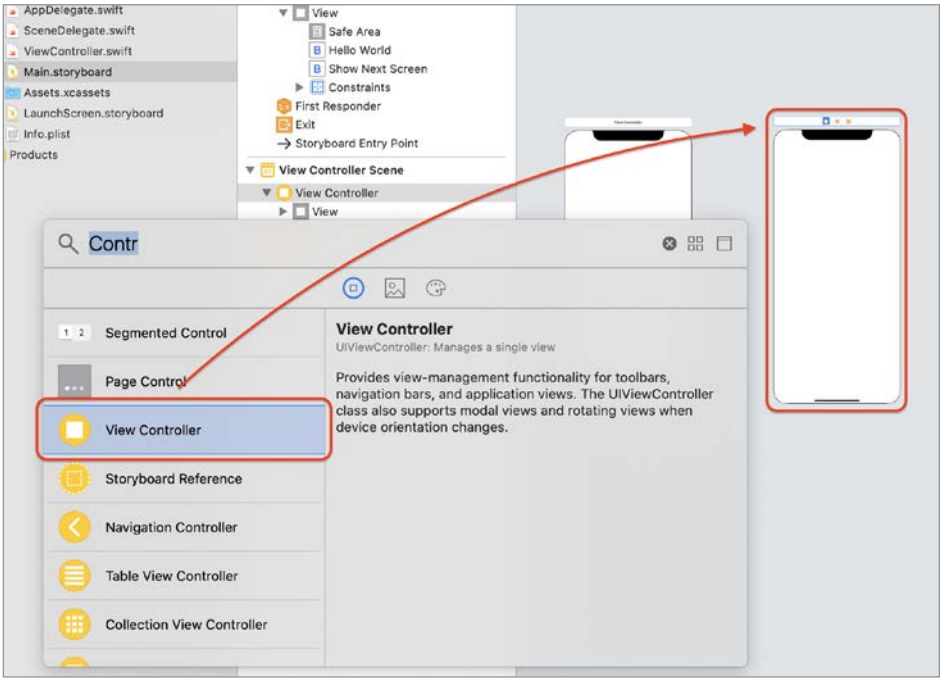

Рис. 37.26. Создание новой сцены

Теперь storyboard проекта содержит две сцены, и мы можем организовать переход между ними. Но сперва внесем во вторую сцену отличительную особенность, изменим цвет ее фона (рис. 37.27):

- (1) В Document outline выделите элемент View новой сцены.
- (2) Откройте панель Attributes Inspector.
- (3) Поменяйте значение свойства background на произвольное (например, на оранжевый цвет).

Нам осталось только добавить переход от первой сцены ко второй:

- Зажмите клавишу Control на клавиатуре и с помощью мыши перетяните кнопку Show Next Screen на новую сцену. При этом сама кнопка останется на месте, а от нее будет тянуться синяя линия.
- Когда новая сцена подсветится синим, отпустите кнопку мыши, а в появившемся меню выберите Show в разделе Action Seque (рис. 37.28).

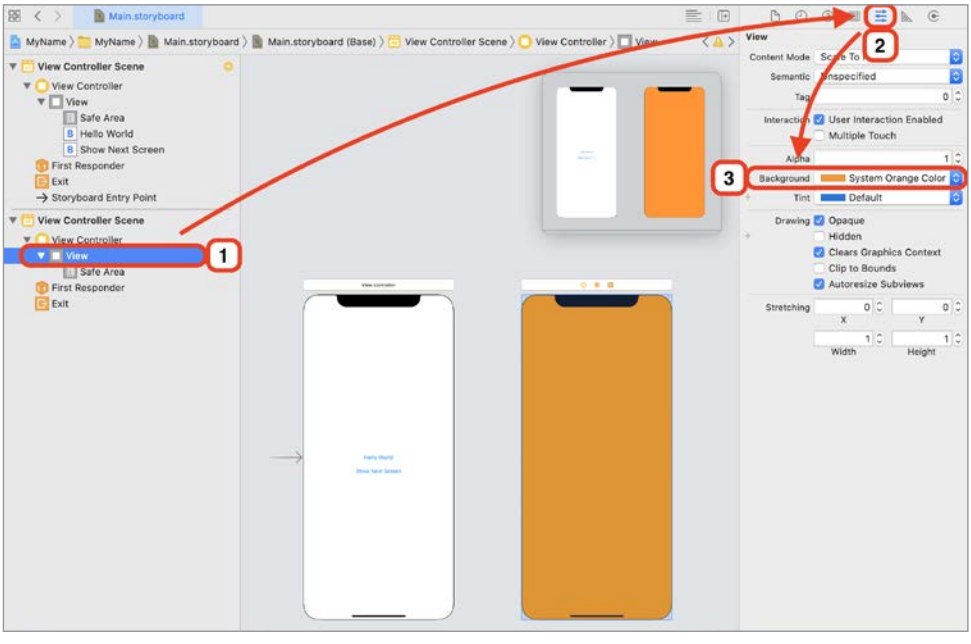

Рис. 37.27. Изменение фонового цвета сцены

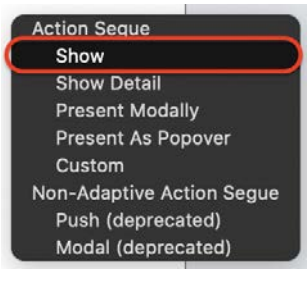

Рис. 37.28. Создание перехода между сценами

Переход между сценами создан! Обратите внимание, что на storyboard появилась серая стрелка, идущая от одной сцены к другой, а в Document Outline теперь отображается новый элемент Show segue to View Controller (рис. 37.29).

- Запустите приложение на симуляторе и попробуйте осуществить переход к новой сцене.

Если вы запускаете приложение на симуляторе с iOS 13 (или выше), то новая сцена появляется в виде так называемой карточки (предыдущая сцена в верхней части экрана слегка выглядывает). В этом случае для возвращения к первому экрану достаточно просто смахнуть новую сцену вниз (рис. 37.30).

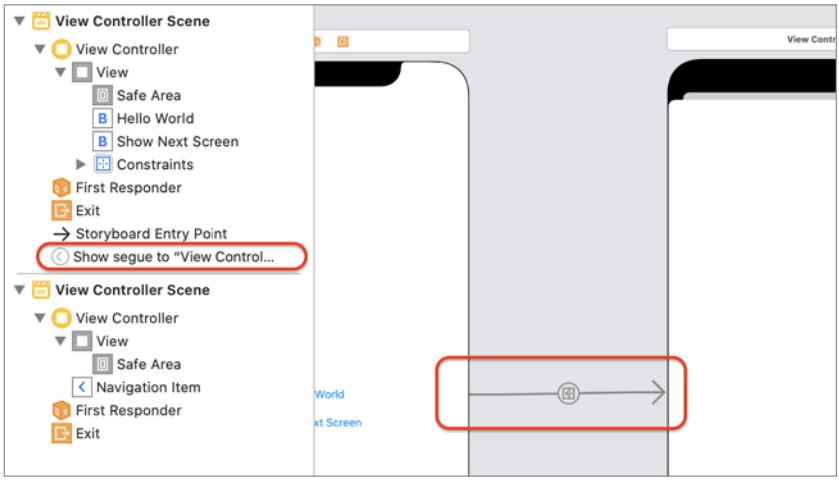

Рис. 37.29. Созданный переход между сценами

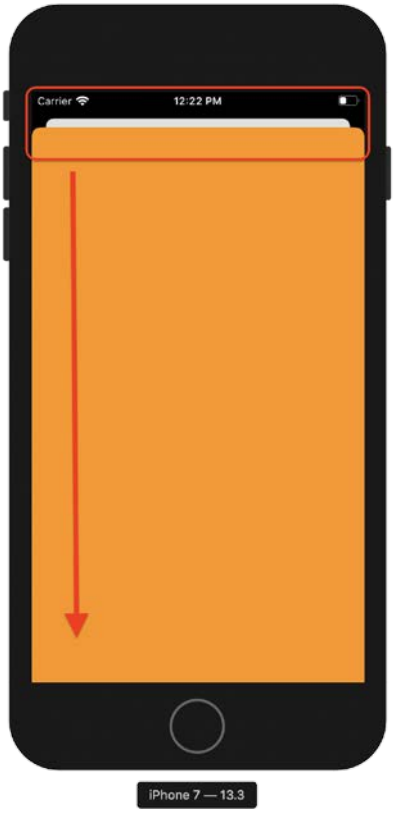

Рис. 37.30. Новая сцена на экране симулятора

ПРИМЕЧАНИЕ Возможно, что кнопка Show Next Screen на первой сцене будет располагаться не совсем там, где мы рассчитывали (это зависит от того, на каком именно симуляторе вы запускаете приложение). Для ее корректного размещения, независимо от выбранного устройства, потребовалось бы создать дополнительные ограничения (constraints) вроде тех, что были созданы для первой кнопки, однако рассмотрение данного материала выходит за рамки темы данной книги.

Такие переходы в Xcode называются **segue** (среди программистов — *сигвей*). С их помощью можно не просто осуществлять перемещение между сценами, но и передавать между ними данные. Предположим, что ваше приложение является книгой контактов и имеет два экрана: первый — список контактов, второй — информация о конкретном контакте.

При переходе с первого экрана на второй с помощью segue вы передаете данные о том, какой именно контакт необходимо отобразить, после чего загружаете требуемую информацию и выводите ее на сцену.

Класс `ViewController`, объявленный в файле ViewController.swift, связан с первой сценой приложения. С его помощью обрабатывается нажатие кнопки Hello World и выводится текст на консоль. В дальнейшем и новая сцена получит некоторые функциональные возможности. Для этого требуется связать ее с собственным классом, в котором будут обрабатываться события:

- В Project Navigator щелкните правой кнопкой мыши и выберите пункт New File.
- В появившемся окне выберите Cocoa Touch Class и нажмите Next (рис. 37.31).

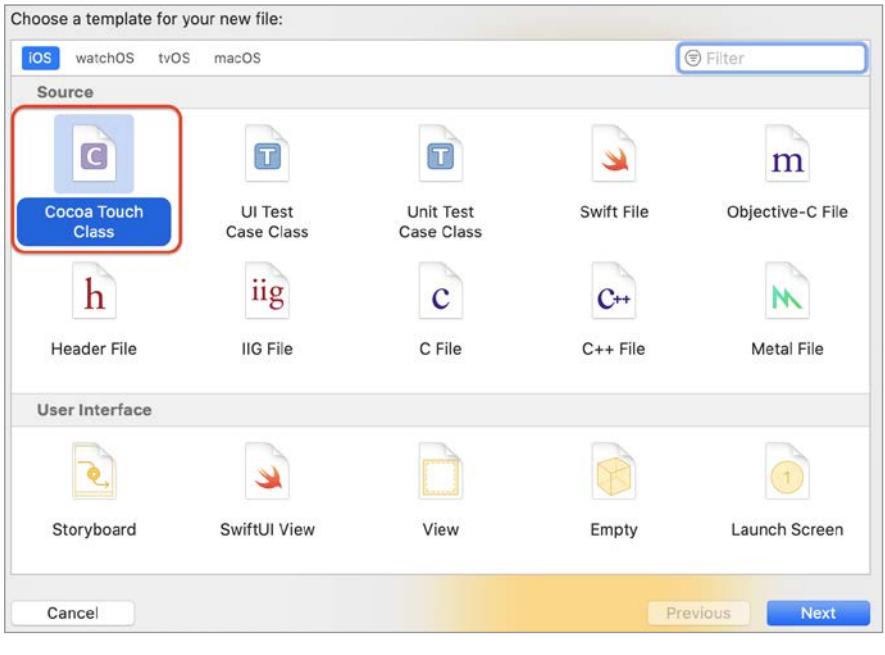

Рис. 37.31. Создание нового файла в составе проекта

- В новом окне в качестве имени класса укажите `SecondViewController`, а в качестве родительского класса (поле Subclass of) — `UIViewController` (рис. 37.32). Далее нажмите Next и подтвердите сохранение файла.

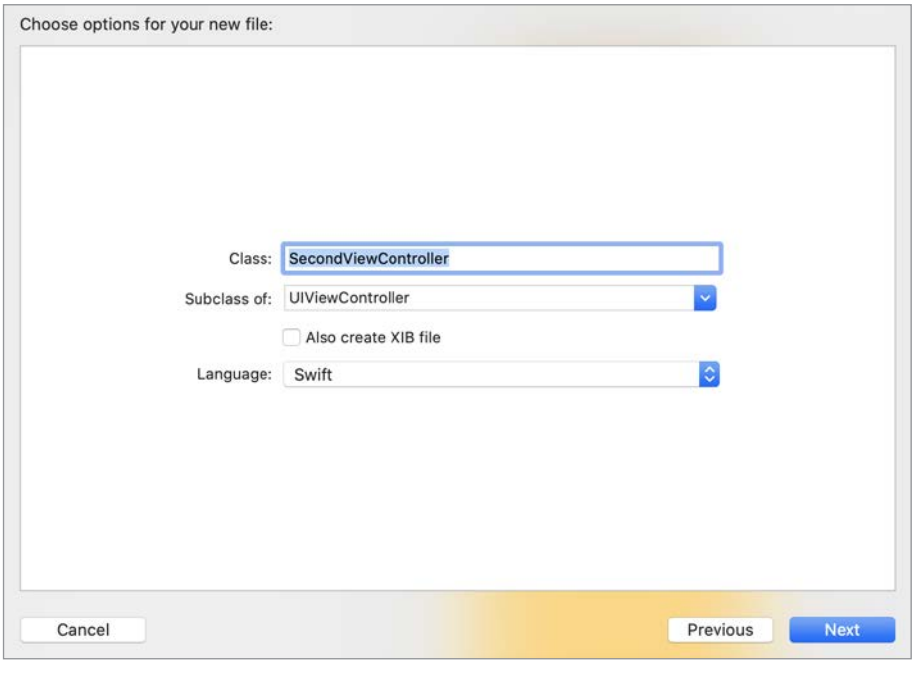

Рис. 37.32. Создание нового файла в составе проекта

В Project Navigator появился новый файл Project Navigator.swift. Если открыть его, то в нем вы найдете новый класс `SecondViewController`. Свяжем его с новой сценой:

- В Project Navigator щелкните по Main.storyboard.
- В Document Outline (или прямо в storyboard) выделите элемент View Controller новой сцены.

Откройте панель Identity Inspector и в поле Class введите `SecondViewController`, после чего нажмите Enter на клавиатуре.

> **ПРИМЕЧАНИЕ** Будьте внимательны: выбирайте View Controller именно новой (оранжевой) сцены.

Теперь новая сцена связана с новым классом и при необходимости мы можем обеспечить обработку события, например нажатия кнопок.

37.8. Отображение всплывающего окна. Класс UIAlertController

Следующим шагом станет реализация появления всплывающего окна при нажатии на кнопку на сцене. Для этого будем использовать вторую сцену и связанный с ней класс `SecondViewController`.

В состав `UIKit` входит класс `UIAlertController`, который позволяет организовать вывод всплывающих окон на экран устройства. Такой вид оповещения пользователя может быть отображен в классическом виде в центре экрана (тип `Alert`) (рис. 37.33) или в нижней части сцены (тип `Action Sheet`) (рис. 37.34).

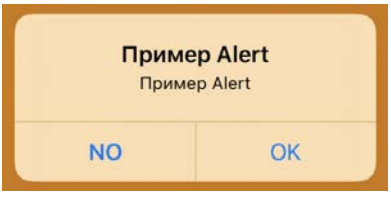

Рис. 37.33. UIAlertController в виде Alert

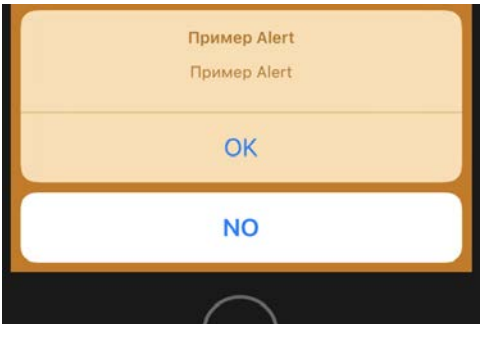

Рис. 37.34. UIAlertController в виде Action Sheet

Рассмотрим использование `UIAlertController` подробнее.

СИНТАКСИС

```
Класс UIAlertController: UIViewController
```

Предназначен для создания всплывающего окна. Является потомком класса `UIViewController`.

Инициализаторы:

```
init(
    title: String?,
    message: String?,
    preferedStyle: UIAlertController.Style)
```

Основные свойства и методы:

`var title: String?` — заголовок окна.

`var message: String?` — текст, выводимый ниже заголовка.

`var preferedStyle: UIAlertController.Style` — стиль всплывающего окна.

`func addAction( UIAlertAction )` — создание кнопки во всплывающем окне.

Пример

```
let alert = UIAlertController(
    title: "Пример Alert",
    message: "Пример Alert",
    preferredStyle: .alert)
```

Аргумент `preferedStyle` инициализатора класса, имеющий тип `UIAlertController.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
UIAlertController.Style.alert
```

Реализуем всплывающее окно в коде нашего приложения.

- На новой сцене (с оранжевым фоном) в центре разместите кнопку. При этом измените ее текст на Show Alert и не забудьте создать ограничения (constraints), обеспечивающие размещение кнопки в центре экрана любого устройства.
- В классе `SecondViewController` объявите новый action-метод `showAlert()`. (листинг 37.6).

Листинг 37.6

```
@IBAction func showAlert(){}
```

- Свяжите нажатие кнопки с вызовом метода `showAlert()`.

Теперь при нажатии кнопки на сцене с оранжевым фоном будет вызываться метод `showAlert()` класса `SecondViewController`.

- В методе `showAlert()` создайте экземпляр класса `UIAlertController` и проинициализируйте его константе `alertController` (листинг 37.7).

Листинг 37.7

```
@IBAction func showAlert(){
     let alertController = UIAlertController(
        title: "Welcome",
        message: "This is myName App",
        preferredStyle: .alert)
}
```

Данные, хранящиеся в `alertController`, являются не чем иным, как экземпляром класса `UIAlertController`. При выполнении метода `showAlert()` всплывающее окно при этом не отобразится. Для того чтобы графический элемент был показан поверх сцены, используется метод `present(_:animated:completion)`, входящий в состав класса `UIViewController`. Он должен быть применен не к всплывающему окну, а к самой сцене. При этом экземпляр класса `UIAlertController` передается ему в качестве аргумента.

СИНТАКСИС

`Метод UIViewController.present(_:animated:completion)`

Предназначен для показа элемента в виде модального окна. Данный метод должен быть вызван после внесения всех изменений в выводимый элемент.

Аргументы:

`_: UIViewController` — элемент, который будет показан на экране устройства.

`animated: Bool` — флаг, указывающий на необходимость использования анимации при отображении элемента.

`completion: (() -> Void)? = nil` — замыкание, исполняемое после завершения показа элемента.

Пример

```
self.present(alertController, animated: true, completion: nil)
```

- Дополните `showAlert()` выводом всплывающего окна с помощью метода `present(_:ani-mated:completion)` (листинг 37.8).

Листинг 37.8

```
@IBAction func showAlert(){
    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)
}
```

- Осуществите запуск приложения на симуляторе, перейдите к оранжевой сцене и нажмите на кнопку Show Alert.

После выполнения указанных действий будет произведен вывод оповещения (рис. 37.35).

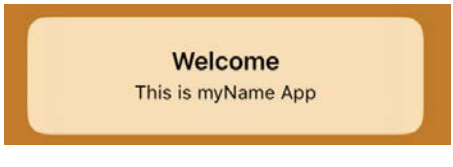

Рис. 37.35. UIAlertController типа Alert без функциональных кнопок

Проблема в том, что данное оповещение невозможно закрыть, так как у него отсутствуют функциональные кнопки. 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(buttonOK)
```

Аргумент типа `UIAlertAction` в методе `addAction(_:)` 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

```
let buttonOK = UIAlertAction(
  title: "OK",
  style: UIAlertAction.Style.default,
  handler: nil)
```

Параметр `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 без выделения.
- `cancel` — текст кнопки выделен жирным.
- `destructive` — текст кнопки выделен 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 37.9) 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 `present`.

Листинг 37.9

```
@IBAction func showAlert(){
    let alertController = UIAlertController(
        title: "Welcome",
        message: "This is myName App",
        preferredStyle:.alert)
    // создаем кнопку OK
    let actionOK = UIAlertAction(title: "OK", style: .default, handler: nil)
    // создаем кнопку Cancel
    let actionCancel = UIAlertAction(title: "Cancel", style: .cancel,
        handler: nil)
    // Добавляем обе кнопки в Alert Controller
    alertController.addAction(actionOK)
    alertController.addAction(actionCancel)
    self.present(alertController, animated: true, completion: nil)
}
```

Теперь вы можете запустить приложение в эмуляторе и насладиться его работой. Модальное окно будет отображаться при нажатии кнопки Show Alert и закрываться после нажатия любой из ее кнопок (рис. 37.36).

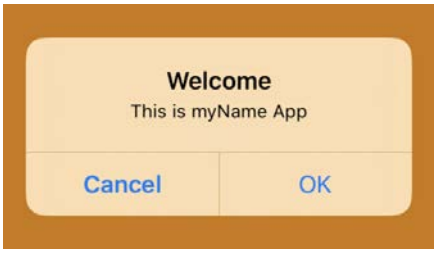

Рис. 37.36. Вывод модального окна с двумя кнопками

ПРИМЕЧАНИЕ Обратите внимание, что в редакторе кода все action-методы, имеющие связь с каким-либо графическим элементом, отмечаются серым кружком (рис. 37.37). Нажав на него, вы сможете осуществить переход в IB к элементу сцены.

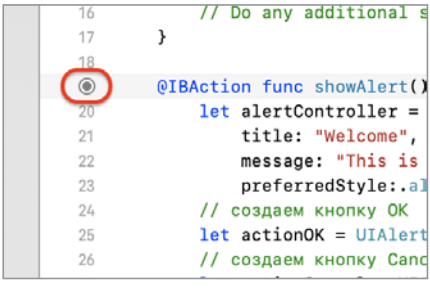

Рис. 37.37. Указатель на связь кодовой конструкции и графического элемента

37.9. Изменение атрибутов кнопки

Каждый графический элемент, который может быть расположен на сцене, имеет набор атрибутов, доступных для редактирования в Interface Builder. 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 уже знакомая вам панель Attribute Inspector (инспектор атрибутов), которую можно отобразить с помощью кнопки Show the Attribute Inspector ( ), расположенной в верхней части Utilities Area. Мы уже меняли в ней текст кнопок и цвет фона второй сцены приложения.

На оранжевой сцене изменим следующие свойства кнопки:

- цвет текста кнопки сделаем серым;
- текст — жирным;
- размер текста — 20.

Для этого в IB выделите кнопку на сцене и перейдите к Attribute Inspector. За цвет текста отвечает атрибут Text Color, находящийся в разделе Button. Вы можете изменить его значение на требуемое. Для этого щелкните на синем прямоугольнике и выберите Custom в появившемся списке, после чего отобразится всплывающее окно с палитрой доступных цветов. Выберите в палитре серый цвет, после чего закройте окно палитры.

Настройки шрифта определены в атрибуте Font. Если нажать на иконку с буквой T, то в появившемся окне можно изменить шрифт, стиль его начертания и размер (рис. 37.29). Для установки жирного шрифта выберите Bold в качестве значения параметра Style. Размер текста можно изменить в поле Size: установите его равным 20.

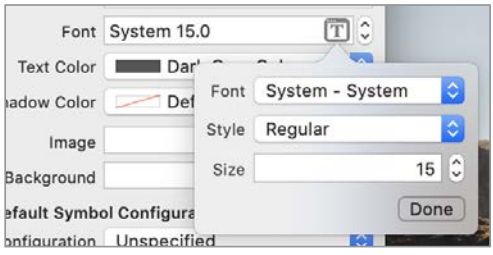

Рис. 37.38. Настройки шрифта

Теперь кнопка в вашей программе выглядит иначе (рис. 37.39), при этом ее функциональность совершенно не изменилась — она по-прежнему вызывает всплывающее окно.

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 образом, потянув за уголок или грань.

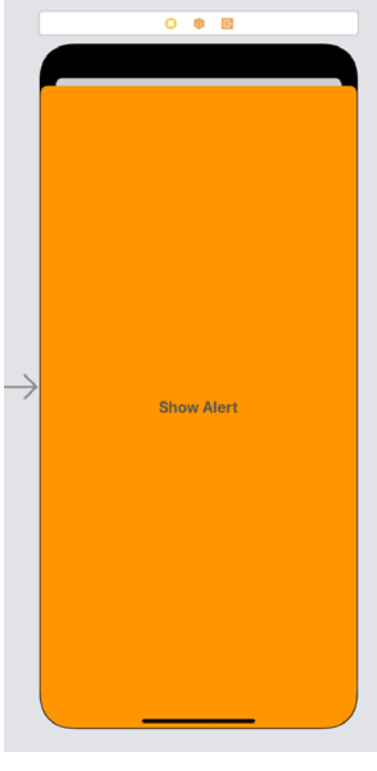

Рис. 37.39. Измененная кнопка в приложении

С помощью атрибутов вы можете изменять внешний вид своих программ. При этом вам доступны возможности значительно более широкие, чем просто смена шрифта и цвета. Советую самостоятельно протестировать смену значений различных атрибутов.

37.10. Доступ кода к UI. Определитель типа @IBOutlet

Вы уже знаете, как обеспечить доступ элементов сцены к методам класса, — для этого используется ключевое слово `@IBAction`. Для создания обратной связи (организовать доступ к графическому элементу из кода) служит ключевое слово `@IBOutlet` под названием *аутлет*.

Сейчас мы рассмотрим пример его использования: запрограммируем изменение текста метки, размещенной на сцене, по нажатии кнопок Left и Right.

Добавим на оранжевую сцену новые графические элементы:

- Найдите в библиотеке объектов и разместите на сцене в произвольном месте (рис. 37.40):
  - кнопку (Button) с текстом Left;
  - метку (Label);
  - кнопку (Button) с текстом Right.

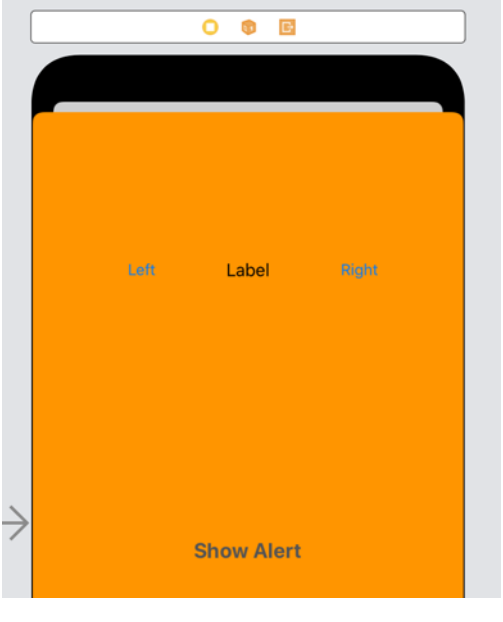

Рис. 37.40. Новые элементы на сцене

Xcode позволяет делить Project Editor на несколько отдельных областей, в каждой из которых отображается свой элемент проекта. Например, вы можете отобразить две области: в одной показать storyboard, а в другой файл SecondViewController.swift (рис. 37.41).

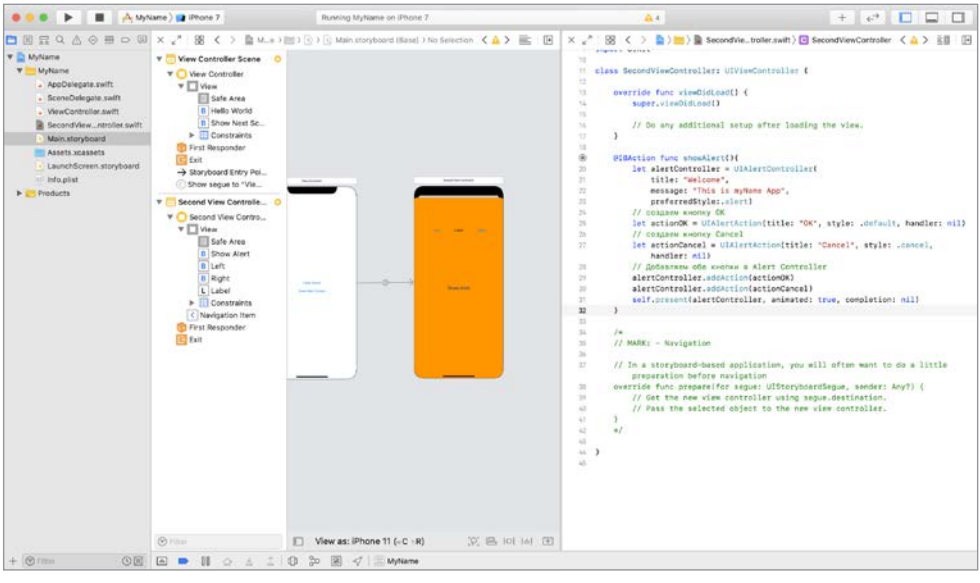

Рис. 37.41. Несколько областей в Project Editor

Такой способ отображения позволяет в некоторых случаях значительно упростить работу, когда требуется постоянное переключение между несколькими ресурсами. Так, если вы работаете на Мас с большой диагональю, то при необходимости можете разделить Project Editor даже больше чем на две области.

Кроме того, этот режим обеспечивает еще один способ создания аутлетов (`@IBOutlet`) и экшенов (`@IBAction`), который мы сейчас рассмотрим. Но сперва разделим Project Editor на две области:

- В правом верхнем углу Project Editor нажмите кнопку Add Editor on Right (⊞), после чего рабочая зона разделится на две области.

ПРИМЕЧАНИЕ Если ваш Мас имеет небольшую диагональ, то для удобства работы можно временно скрыть панели Inspectors и Navigator (с помощью кнопок, расположенных в правом верхнем углу Xcode).

- С помощью панели Jump Bar, расположенной выше Project Editor, организуйте отображение Main.storyboard в левой области, а SecondViewController.swift — в правой (см. рис. 37.41).

Теперь создадим аутлет для элемента Label:

- Удерживая нажатой клавишу Control, в IB выделите элемент Label, после чего перетащите его в соседнюю область, в которой открыт файл SecondViewController.swift. Расположите появившуюся синюю линию в верхней части класса `SecondViewController`. При этом должна отображаться надпись Insert Outlet or Outlet Collection (рис. 37.42).

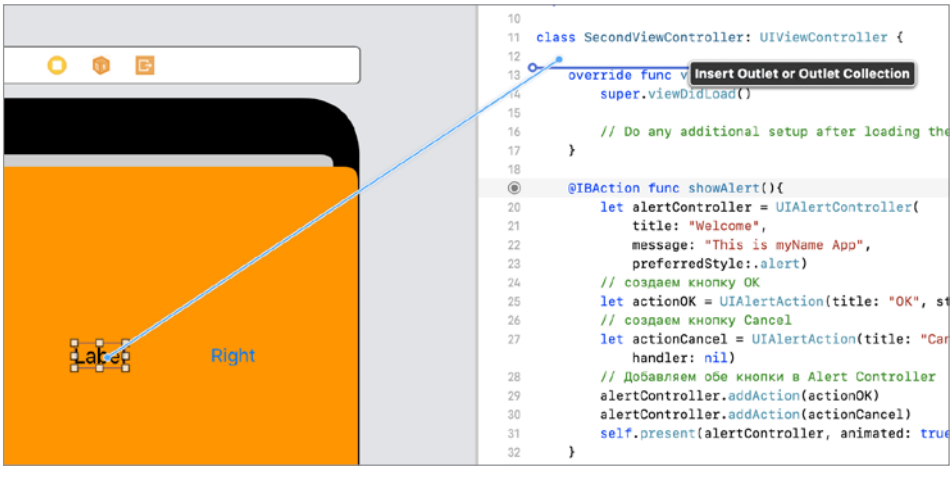

Рис. 37.42. Создание аутлета

После того как вы отпустите кнопку мыши, отобразится окно создания Outlet (рис. 37.43).

- Убедитесь, что поле Connection имеет значение Outlet.
- В поле Name введите значение myLabel.
- Не изменяя значения остальных полей, нажмите кнопку Connect.

После этого у класса `SecondViewController` появится новое свойство `myLabel` (листинг 37.10).

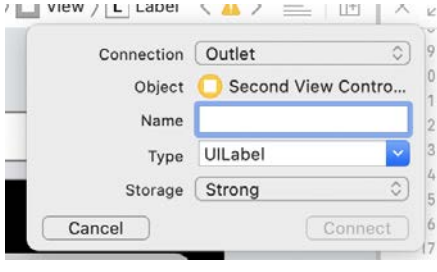

Рис. 37.43. Окно создания Outlet

ПРИМЕЧАНИЕ Поле Connection позволяет указать тип создаваемой связи. Таким же образом могут быть созданы не только аутлеты, но и action-методы. В данном случае поле Connection не содержит значение 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.

ПРИМЕЧАНИЕ В зависимости от версии Xcode и ваших предыдущих действий, поле Storage по умолчанию может иметь значение Weak. В этом случае автоматически будет создана слабая ссылка (свойство класса будет идти с ключевым словом weak).

Листинг 37.10

```
class SecondViewController: UIViewController {
    @IBOutlet var myLabel: UILabel!
    //...
}
```

Свойство `myLabel` содержит определитель типа `@IBOutlet`. Слева от него отображена пиктограмма в виде серого круга, указывающая на то, что свойство имеет настроенную связь с элементом в Interface Builder. Если навести на свойство указатель мыши, то связанный с ним элемент будет подсвечен прямо на сцене в соседней рабочей зоне (рис. 37.44).

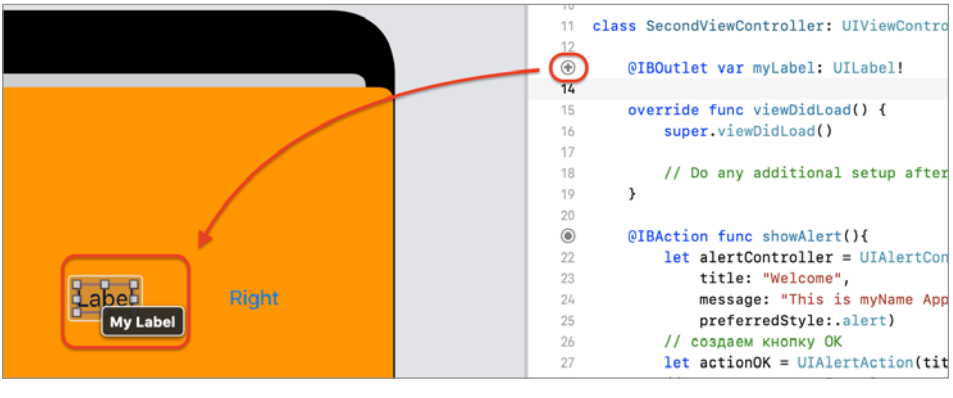

Рис. 37.44. Отображение связи между элементом и аутлетом

Следующим шагом в реализации функции смены текста метки станет создание action-метода `changeLabelText()` (метода с определителем `@IBAction`). Внутри своей реализации он будет использовать аутлет `myLabel` для доступа к элементу Label на сцене и изменения свойства, отвечающего за его текст. Изменять текст метки будут две кнопки: Left и Right. При этом не требуется создавать отдельный action-метод для каждой из них — Xcode позволяет создать связь одного action-метода сразу с несколькими элементами на сцене.

Возможно, у вас возник вопрос: как внутри метода будет определяться то, какая из кнопок нажата? Дело в том, что метод, помеченный определителем `@IBAction`,

позволяет указать входной параметр `sender`, содержащий ссылку на тот элемент, который вызвал данный метод.

Как уже говорилось, режим, при котором Project Editor отображает несколько областей, может использоваться не только для создания outlet-свойств, но и для создания action-методов.

- Удерживая клавишу Control, перетащите кнопку Right ниже метода `showAlert`, после чего отобразится окно создания связи (рис. 37.45).
- В появившемся окне в поле Connection выберите значение Action.
- 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, его оставьте без изменений. Обратите внимание на поле Arguments: если выбрать None, то создаваемый метод не будет иметь каких-либо параметров, и мы не сможем определить, какая кнопка нажата. Для того чтобы action-метод мог обратиться к вызвавшему его элементу, в этом поле необходимо указать Sender.

После нажатия кнопки Connect в редакторе кода появится новый метод `changeLabelText(_:)` с определителем типа `@IBAction` (листинг 37.11).

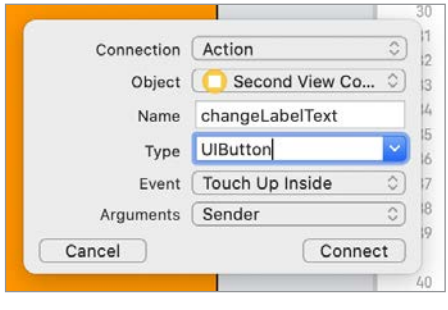

Рис. 37.45. Окно создания Action

Листинг 37.11

```
class SecondViewController: UIViewController {
   //...
   @IBAction func changeLabelText(_ sender: UIButton) {
   }
   //...
}
```

Внутри созданного метода вы можете обратиться к свойству `myLabel` для доступа к элементу Label на сцене с целью изменения его параметров. Реализуем в методе изменение текста метки, используя при этом текст самой кнопки (листинг 37.12).

Листинг 37.12

```
@IBAction func changeLabelText(_ sender: UIButton) {
    if let buttonText = sender.titleLabel!.text {
        self.myLabel.text = "\(buttonText) button was pressed"
    }
}
```

Аутлет-свойство `myLabel` хранит в себе экземпляр класса `UILabel`, соответствующий элементу Label, расположенному на сцене. Поэтому мы используем его для изменения текста метки (свойство text). Как только изменяется значение какого-либо свойства аутлета `myLabel`, эти изменения тут же отображаются на экране устройства.

Параметр `sender` позволяет обратиться к экземпляру класса `UIButton`, который соответствует кнопке на сцене, использующейся для вызова метода `changeLabelText(_:)`. Свойство `titleLabel` возвращает опциональный экземпляр класса `UILabel`, который как раз и содержит описание текста кнопки. Для доступа к самому тексту используется свойство `text`.

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 достаточно зажать кнопку мыши на сером круге левее метода `changeLabelText(_:)` и перетащить его на кнопку Left на сцене (рис. 37.46).

Теперь если запустить программу и нажать на любую из двух кнопок, то текст метки изменится.

Рис. 37.46. Создание дополнительной связи action-метода и Button

Возможно, что при нажатии кнопок текст в метке будет выводиться не полностью. В этом случае потребуется переместить Label и растянуть его на всю ширину сцены, после чего изменить свойство Alignment в инспекторе атрибутов на «выравнивание по центру».

Вы получили первые и самые важные навыки, которые будете применять при создании собственных приложений. Впереди еще много нового и интересного. На этом наше знакомство с UIKit в данной книге завершено, перейдем к разработке приложения с помощью SwiftUI.

Глава 38. Разработка приложения с использованием SwiftUI

Навыки работы с UIKit и понимание его структуры для будущего Swift-разработчика, безусловно, являются критически важными. Именно по этой причине бóльшая часть учебного материала моей следующей книги будет посвящена знакомству с его широкими возможностями.

Как неоднократно говорилось, одной из важнейших функций UIKit является создание графических интерфейсов мобильных приложений. Однако у этого фреймворка уже есть довольно серьезный конкурент — SwiftUI, знакомству с которым будет посвящена эта глава.

SwiftUI — новый фреймворк от Apple, презентация которого стала, на мой взгляд, одним из важнейших событий в жизни Swift за все время его существования. Подход к изучению данного фреймворка будет извилистым, и прежде чем подступиться к нему, вы познакомитесь с Property Wrappers, фреймворком Combine, реактивным программированием и многими другими темами. Но все это оставим для следующих книг. Сегодня перед нами стоит задача опробовать SwiftUI в действии, используя лишь малую толику тех возможностей, которые он предлагает разработчикам.

Отличия UIKit и SwiftUI

SwiftUI позволяет создавать интерфейсы приложений совершенно новым, **декларативным** способом. Если при использовании UIKit вы полностью контролируете каждый элемент интерфейса и полностью программируете его функциональность (императивный способ), то SwiftUI позволяет максимально упростить этот процесс. Все, что от вас требуется, — кратко написать, что вы хотите видеть на экране, а дальше всю работу SwiftUI берет на себя.

Самое важное, что делает этот фреймворк, — позволяет нам не углубляться в реализацию интерфейса, а заниматься в основном функциями, доступными пользователю.

> **ПРИМЕЧАНИЕ** Уверен, что пока вам сложно понять, что подразумевает под собой «декларативный способ» и чем он отличается от того, что предлагает UIKit. Но со временем, читая учебный материал и получая опыт разработки, вы хорошо поймете различия UIKit и SwiftUI, а также научитесь совместно их использовать.

Чтобы вы лучше почувствовали возможности SwiftUI, рассмотрим простой пример, где по шагам распишем создание табличного представления на UIKit и SwiftUI.

На UIKit:

1. Разместить на сцене элемент `UITableView`.
2. Указать `ViewController` в качестве делегата для `UITableView` на сцене.
3. Подписать `ViewController` на протокол `UITableViewDataSource`.
4. Реализовать методы `numberOfSections(in:)` и `tableView(_: numberOfRowsIn Section:)`, определяющие количество секций и строк в таблице.

На SwiftUI:

1. Создать экземпляр структуры `List`.

SwiftUI радикально отличается от UIKit.

А теперь приступим к разработке приложения.

38.1. Создание нового проекта

- Создайте новый Xcode-проект.
- В качестве платформы выберите iOS, а в качестве шаблона — App.

В окне создания проекта укажите следующие значения (рис. 38.1):

- В поле Product Name введите RGBApp.
- В качестве способа создания графического интерфейса укажите SwiftUI.
- В поле Life Cycle (жизненный цикл приложения) выберите пункт SwiftUI App.
- Сохраните проект в произвольном месте.

> **ПРИМЕЧАНИЕ** В зависимости от размера экрана вашего компьютера, Project Editor может быть разделен на область с кодом и область с серым фоном либо вертикально (рис. 38.2), либо горизонтально.

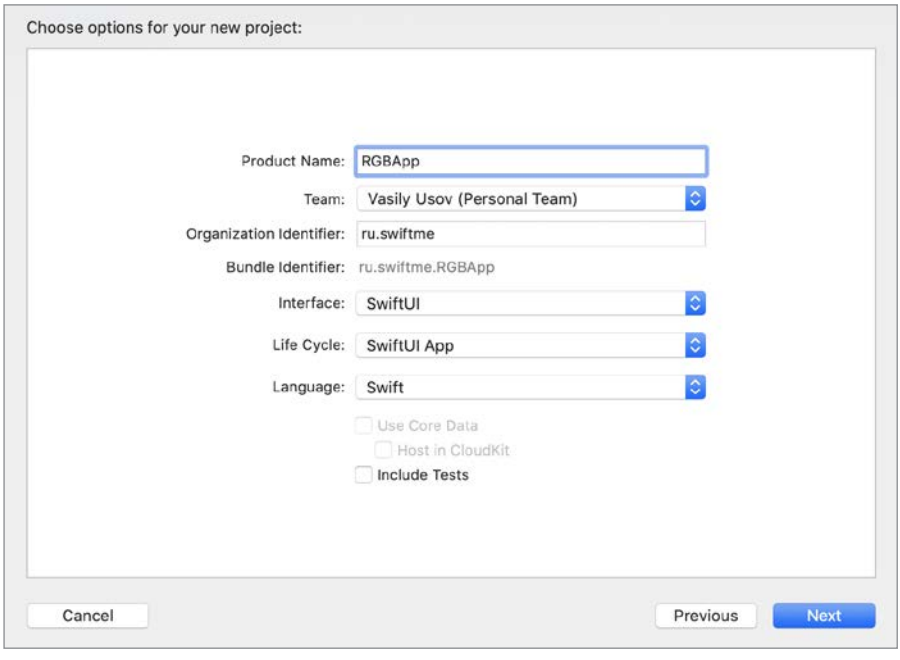

Рис. 38.1. Создание проекта на SwiftUI

После сохранения проекта перед вами откроется рабочая среда Xcode (рис. 38.2).

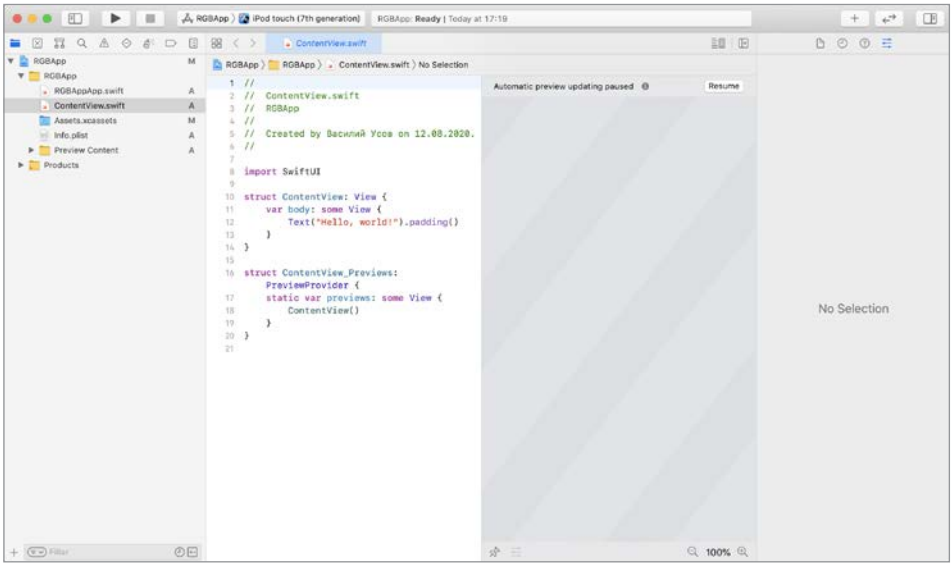

Рис. 38.2. Интерфейс нового Xcode-проекта

38.2. Структура проекта на SwiftUI

Если сравнивать этот проект с тем, что был реализован в предыдущей главе, то первым бросается в глаза изменившийся состав ресурсов в Project Navigator (рис. 38.3). Теперь в нем не найти файлы ViewController.swift и Main.storyboard. При использовании SwiftUI необходимости в них нет. Весь интерфейс описывается по-новому, очень кратко и лаконично, с помощью программного кода. Также отсутствует и файл AppDelegate.swift, который позволял управлять жизненным циклом приложения (о том, что это такое, мы поговорим в следующей книге).

ПРИМЕЧАНИЕ Файл LaunchScreen.storyboard, описывающий интерфейс экрана запуска (лаунч-скрина) приложения, все еще находится в списке ресурсов проекта. Он будет подробно рассмотрен в следующей книге.

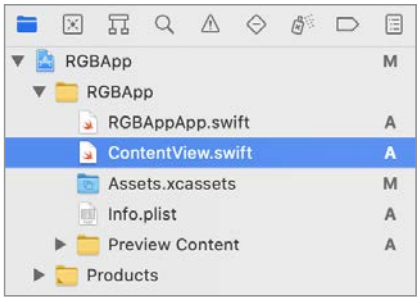

Рис. 38.3. Project Navigator

➤ В Project Navigator щелкните на файле ContentView.swift.

Файл ContentView.swift содержит пример реализации простейшего графического интерфейса с текстовой меткой в центре. Обратите внимание, что для его реализации написано всего несколько строчек кода.

Помимо Project Navigator, также изменился и Project Editor — теперь он разделен на две области (рис. 38.4): в левой части располагается редактор кода, а в правой — Canvas.

Canvas — это специальная панель предварительного просмотра, появившаяся в Xcode вместе со SwiftUI. Она в некотором роде заменяет уже знакомый вам storyboard, позволяя, без необходимости запускать приложение на симуляторе, видеть созданный с помощью программного кода графический интерфейс. Пока Canvas не отображает ничего, но в его верхней части расположена кнопка запуска Resume (рис. 38.5).

➤ Нажмите кнопку Resume.

После довольно быстрой компиляции на Canvas отобразится тот самый графический интерфейс с текстовой меткой в центре, о котором говорилось ранее (рис. 38.6).

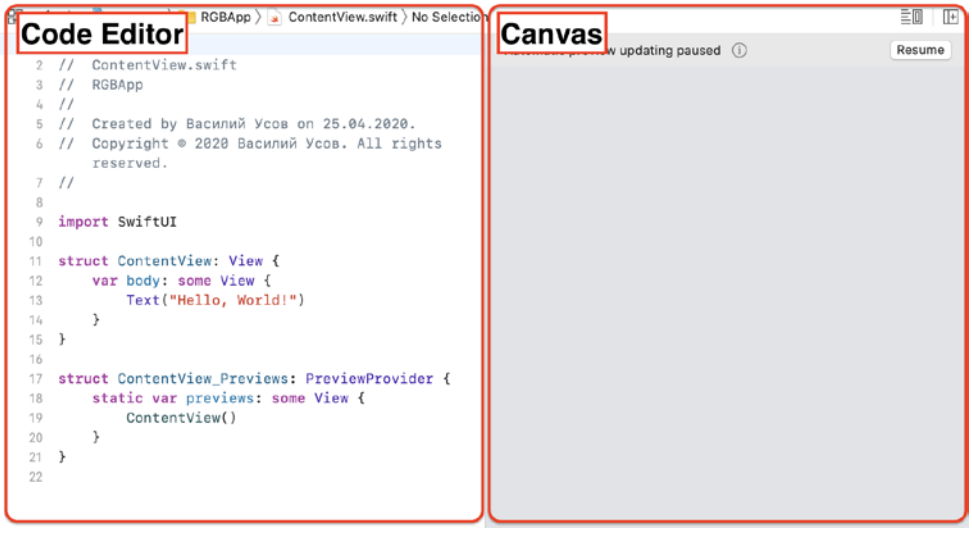

Рис. 38.4. Создание проекта на SwiftUI

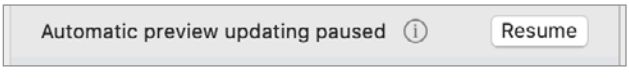

Рис. 38.5. Автоматическое обновление Canvas

ПРИМЕЧАНИЕ Canvas, как и SwiftUI, является очень свежим продуктом, и по этой причине он периодически может «подглючивать». Из-за этого запуск может сильно растянуться во времени или вовсе «зависнуть». В этом случае просто перезагрузите Xcode и попробуйте заново.

ПРИМЕЧАНИЕ Для быстрого обновления Canvas используйте сочетание клавиш Command + Option + P.

С помощью кнопки Editor Options, расположенной в верхнем правом углу Project Editor, вы можете изменить внешний вид этой области среды разработки (рис. 38.7). Если нажать на нее, то отобразится соответствующее меню, в котором можно отключить Canvas, изменить его расположение или использовать другие возможности.

- Активируйте Minimap в меню Editor Options.

Теперь Project Editor содержит мини-карту программного кода, которая позволит вам с легкостью производить навигацию в файлах, содержащих большое количество исходного кода (рис. 38.8).

Если на клавиатуре удерживать клавишу Command и навести курсор на Minimap, вы увидите активные ярлыки, соответствующие всем сущностям (структурам, классам, методам, свойствам и т. д.) файла ContentView.swift, что значительно упростит перемещение между ними (рис. 38.9).

Рис. 38.6. Canvas, отображающий интерфейс приложения

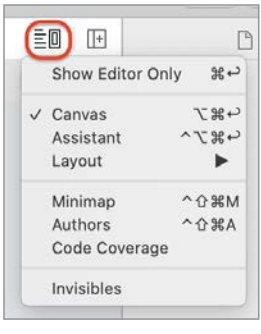

Рис. 38.7. Меню Editor Options

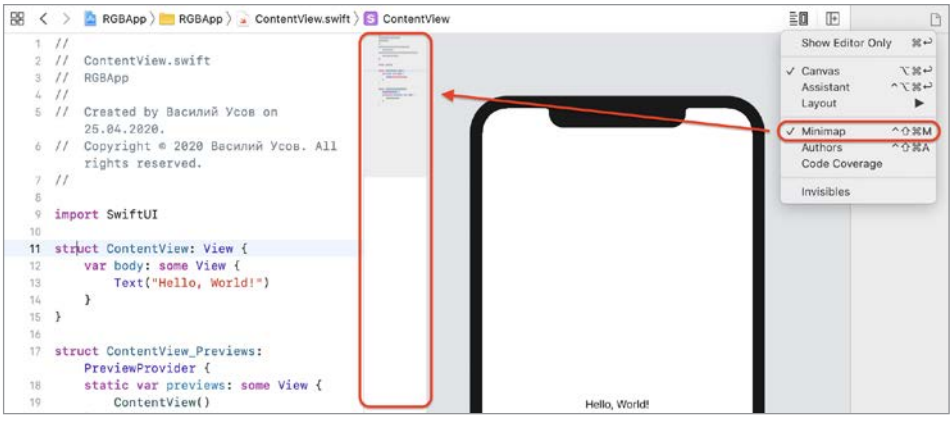

Рис. 38.8. Minimap в составе Code Editor

Внешний вид приложения, отображенный на Canvas, создан в результате обработки программного кода файла ContentView.swift, расположенного в левой части Project Editor (листинг 38.1).

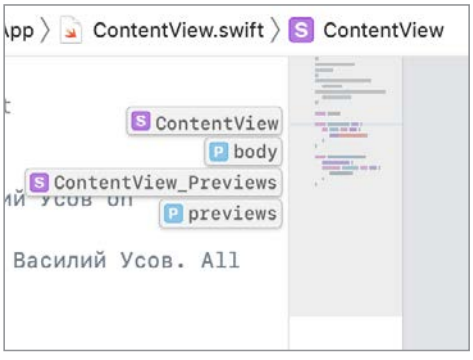

Рис. 38.9. Навигация по Minimap

Листинг 38.1

```
struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Структура `ContentView` определяет интерфейс приложения, а выводится он в Canvas с помощью структуры `ContentView_Previews`. Подробному изучению данных элементов будет посвящено большое количество материала в будущих книгах, сейчас же я лишь кратко поясню принцип их работы.

Протокол `View` — это основа любого графического элемента в SwiftUI. То, что возвращает свойство `body` структуры, которая принимает данный протокол, и будет выводиться на экран устройства.

Так, в данном случае `ContentView` — это базовое родительское представление, в состав которого входит другое представление (`Text` также соответствует протоколу `View`). В качестве аналогии вспомните проект с шариками, когда подложка выполняла роль родительского View, а шарики входили в его состав, или проект из предыдущей главы, когда родительское View соответствовало фону сцены, а в его состав входили все остальные View (кнопки, текстовые метки). `ContentView` в данном случае выполняет ту же самую роль — возвращает дочернее представление, которое также выводится на экран устройства.

В данном примере свойство `body` возвращает текстовую метку. В UIKit метка была представлена классом `UILabel`, а в SwiftUI — структурой `Text` (листинг 38.2).

Листинг 38.2

```
Text("Hello, World!").padding()
```

- Уберите вызов метода `padding()`.
- Измените текст метки на `Hello, Swift`.

Вносимые в код изменения сразу же отображаются на Canvas (рис. 38.10).

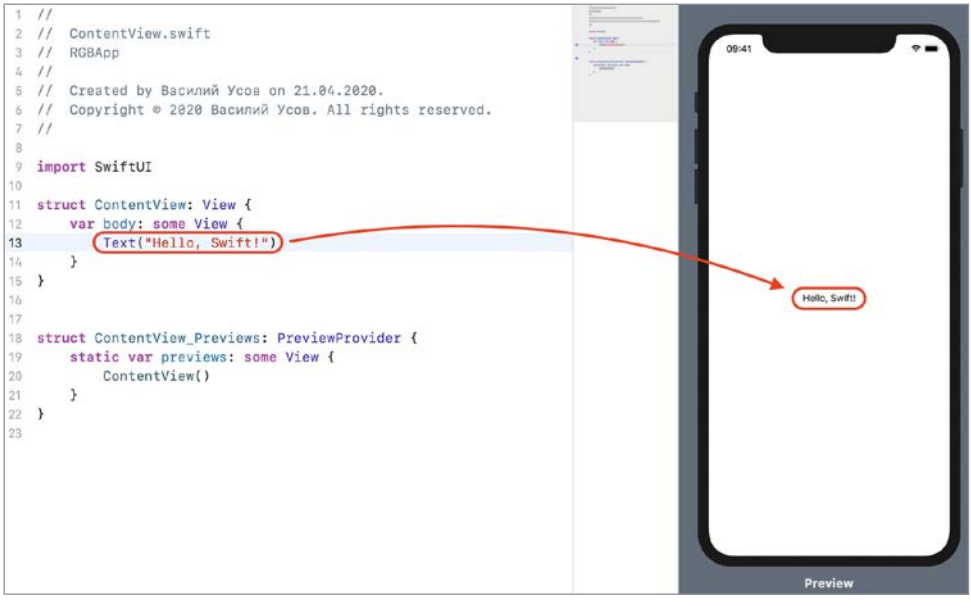

Рис. 38.10. Изменения Canvas вслед за кодом

Интерфейс, отображаемый на Canvas, полностью соответствует тому, что вы увидите при запуске приложения на симуляторе или реальном устройстве.

- Произведите запуск приложения на симуляторе (рис. 38.11).

Вернемся к коду проекта. Структура `ContentView_Previews` (листинг 38.3) соответствует протоколу `PreviewProvider`, вследствие чего обеспечивает отображение графического интерфейса на Canvas. Все, что находится в свойстве `previews` данной структуры, будет отображено. Так, в данном случае в ней создается экземпляр структуры `ContentView`, в которой описана текстовая метка.

Листинг 38.3

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Рис. 38.11. Приложение, запущенное на симуляторе

При необходимости вы можете создать дополнительные Preview на Canvas.

- По аналогии со структурой `ContentView_Previews` реализуйте в файле ContentView.swift новую структуру `SecondContentView_Previews`, но вместо `ContentView` в свойстве `previews` создайте метку с помощью уже известной вам структуры `Text` (листинг 38.4).

Листинг 38.4

```
struct SecondContentView_Previews: PreviewProvider {
    static var previews: some View {
        Text("Second Screen")
    }
}
```

После обновления на Canvas будет отображено два устройства, каждое из которых соответствует своей структуре, подписанной на протокол `PreviewProvider` (рис. 38.12). При обновлении Canvas среда разработки Xcode автоматически рас-

познает все структуры, соответствующие протоколу `PreviewProvider`, и создает для них отдельный Preview (предпросмотр).

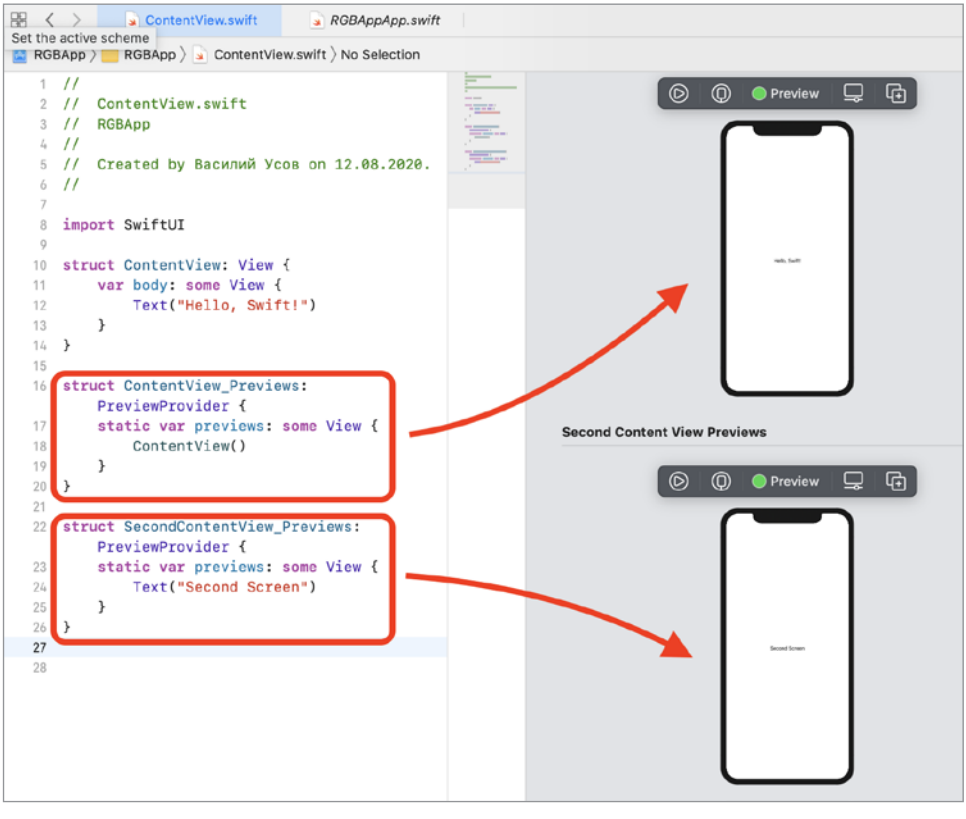

Рис. 38.12. Два Preview на Canvas

Чтобы осуществить предпросмотр интерфейса на другой модели iPhone, достаточно сменить ее в списке рядом с кнопкой запуска проекта, после чего Canvas автоматически обновится (рис. 38.13).

- Удалите структуры `ContentView_Previews` и `SecondContentView_Previews` из файла ContentView.swift.
- Запустите приложение на симуляторе.

Несмотря на то что из файла ContentView.swift были удалены все структуры, обеспечивающие предварительной просмотр интерфейса, в симуляторе все также отображается метка с текстом `Hello, Swift!`. Причина кроется в том, что протокол `PreviewProvider`, а также структуры, подписанные на него, определяют только то, что будет отображено в качестве предпросмотра на Canvas (для удобства разработчика при создании приложения). Они никак не влияют на интерфейс приложения

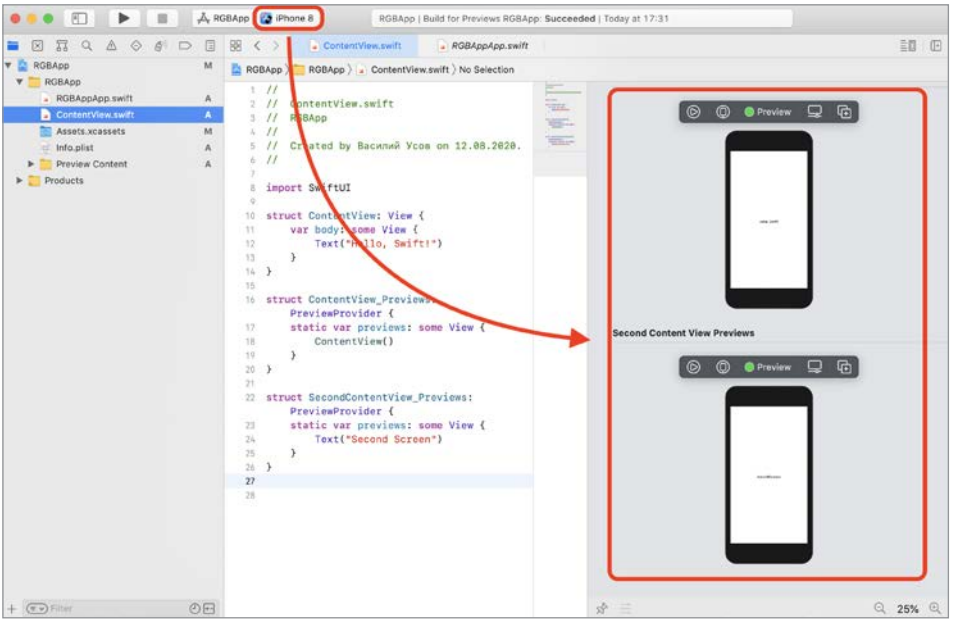

Рис. 38.13. Предпросмотр интерфейса на другом устройстве

на устройстве (или симуляторе). Первый рабочий экран устройства определяется в файле RGBAppApp.swift в одноименной структуре (листинг 38.5).

Листинг 38.5

```
@main
struct RGBAppApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

Рассмотрение этого файла выходит за рамки описываемого материала, ему будет посвящено несколько разделов в следующей книге.

38.3. Редактирование элементов интерфейса

Пришло время поработать с графическим интерфейсом.

- В Project Navigator выберите файл ContentView.swift.
- Вновь реализуйте структуру `ContentView_Previews`, обеспечивающую отображение интерфейса на Canvas (листинг 38.6).

Листинг 38.6

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Изменим некоторые характеристики метки с текстом.

- Щелкните на строчке кода, в которой объявляется экземпляр структуры `Text`.

Обратите внимание, что на панели Attributes Inspector отобразились параметры метки, которые могут быть изменены (рис. 38.14).

ПРИМЕЧАНИЕ Для выбора элемента и доступа к его настройкам можно просто щелкнуть на нем на Canvas.

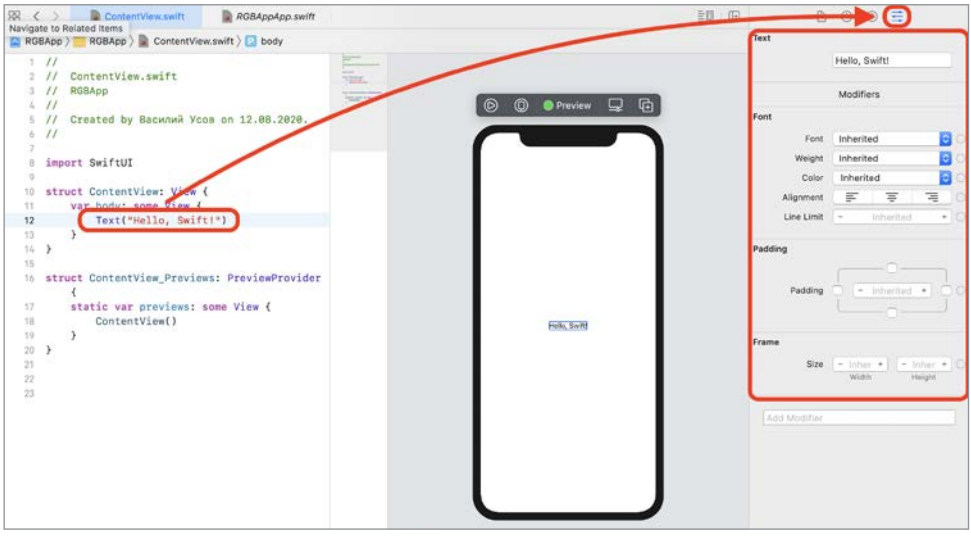

Рис. 38.14. Настройка параметров метки

- Измените следующие параметры метки:
  - Font на Title.
  - Weight на Bold.
  - Color на Red.

При изменении каждого параметра Canvas обновляет содержимое, а в Code Editor появляются соответствующие изменения (листинг 38.7).

Листинг 38.7

```
struct ContentView: View {
    var body: some View {
        Text("Hello, Swift!")
            .font(.title)
            .fontWeight(.bold)
            .foregroundColor(Color.red)
    }
}
```

Все доступные для настройки параметры — это методы структуры `Text`. Вы можете редактировать их код вручную или изменять с помощью Attributes Inspector. Все изменения всегда будут синхронизированы между кодом, панелью атрибутов и Canvas.

38.4. Создание приложения

Теперь приступим непосредственно к разработке приложения. С его помощью, управляя тремя ползунками, мы сможем изменять цвет прямоугольника (рис. 38.15). Каждый из ползунков будет соответствовать своему компоненту в составе цвета (при использовании цветовой модели RGB). Ничего сложного в такой программе нет, но работа над ней позволит почувствовать возможности SwiftUI. Самое удивительное в том, что для реализации подобного проекта вам потребуется написать не более 15 строчек кода!

- Удалите из свойства `body` структуры `ContentView` текстовую метку, она не потребуется (листинг 38.8).

Листинг 38.8

```
struct ContentView: View {
    var body: some View {

    }
}
```

ПРИМЕЧАНИЕ После удаления текстовой метки Xcode сообщит об ошибке, так как свойство body должно возвращать значение типа View, но сейчас оно пустое. Эта ошибка будет исправлена после добавления первого графического элемента.

В цветовой модели RGB любой цвет кодируется тремя параметрами: red (красный), green (зеленый) и blue (синий). Соотношение значений параметров определяет конечный цвет. Именно по этой причине в приложении будет три ползунка, каждый из которых отвечает за значение своего компонента в итоговом цвете. Для хранения значений ползунков нам потребуются три свойства.

- Добавьте свойства в структуру `ContentView` в соответствии с листингом 38.9.

Листинг 38.9

```
struct ContentView: View {

    // свойства для хранения компонентов цвета
    @State var redComponent: Double = 0.5
    @State var greenComponent: Double = 0.5
    @State var blueComponent: Double = 0.5

    var body: some View {

    }
}
```

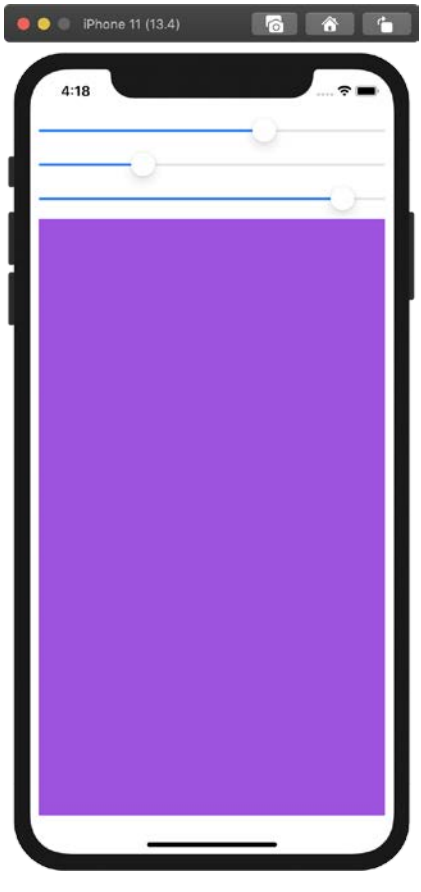

Рис. 38.15. Интерфейс приложения

В коде появился новый для вас элемент `@State`. С его помощью помечаются свойства, при изменении которых графический интерфейс должен перерисовываться. Как только значение одного из State-свойств будет изменено (в данном

случае с помощью ползунка), интерфейс приложения сразу же автоматически обновится, вследствие чего соответствующим образом изменится и цвет прямоугольника.

Все элементы будут сгруппированы с помощью структуры `VStack` (вертикальный стек), позволяющей размещать вложенные в него элементы столбиком. Как видно из окна автодополнения, `VStack` имеет два инициализатора (рис. 38.16).

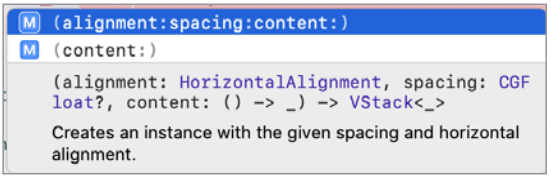

Рис. 38.16. Инициализаторы VStack

В данном случае нас интересует второй инициализатор, принимающий только замыкание в качестве значения аргумента `content`. Если бы вы выбрали его, то получили бы следующий код:

```
VStack(content: {
})
```

который может быть упрощен до:

```
VStack {
}
```

- В свойстве `body` структуры `ContentView` создайте экземпляр `VStack` (листинг 38.10).

Листинг 38.10

```
struct ContentView: View {

    @State var redComponent: Double = 0.5
    @State var greenComponent: Double = 0.5
    @State var blueComponent: Double = 0.5

    var body: some View {
        VStack {

        }
    }
}
```

SwiftUI содержит несколько компонентов, которые могут группировать множество других компонентов. К ним относятся объявленный выше `VStack`, а также `HStack` (горизонтальный стек), `ZStack` (стек в глубину) и `List` (таблица). Аргумент `content` в инициализаторе каждого из этих компонентов может принять

до 10 графических элементов (подписанных на протокол `View`) и вывести их в интерфейсе приложения, сгруппировав соответствующим образом. К примеру, `VStack` позволяет отображать элементы столбиком, один над другим (в виде вертикального стека).

Добавим в стек первый ползунок, который будет отвечать за красный цвет (рис. 38.17):

- (1) Откройте библиотеку объектов.
- (2) Откройте библиотеку отображений (Views library).
- (3) В списке элементов найдите Slider. При этом можете воспользоваться панелью поиска.
- (4) Перетяните его внутрь замыкания в инициализаторе `VStack`.

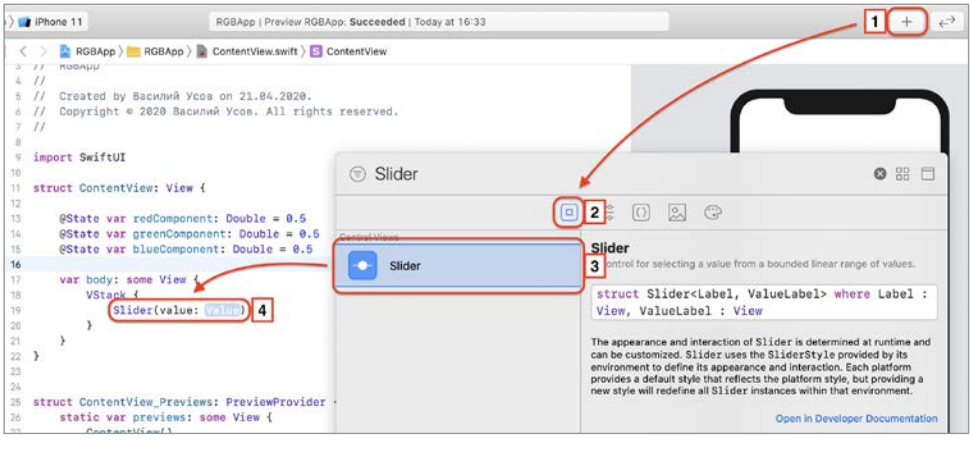

Рис. 38.17. Поиск элемента в библиотеке объектов

В качестве аргумента для `Slider` необходимо указать свойство `redComponent`, но так как оно помечено с помощью `@State` и слайдер должен изменять его, необходимо использовать символ `$` (листинг 38.11).

Листинг 38.11

```
VStack {
    Slider(value: $redComponent)
}
```

Таким образом, при перемещении слайдера на сцене будет автоматически изменяться значение State-свойства `redComponent`, что приведет к обновлению всего интерфейса, а значит, и к изменению цвета прямоугольника, который мы скоро разместим.

После обновления Canvas вы увидите на нем новый элемент — ползунок (рис. 38.18).

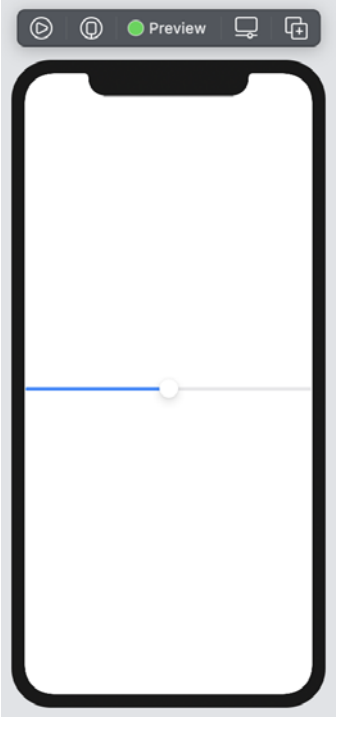

Рис. 38.18. Slider на сцене

Чтобы разместить несколько элементов внутри стека, в коде потребуется написать их каждый с новой строки. Они будут восприниматься как последовательность, автоматически обрабатываться и выводиться на экран.

- Создайте еще два слайдера, разместив их друг под другом в структуре `VStack`, и свяжите с соответствующими свойствами (листинг 38.12).

Листинг 38.12

```
VStack {
    Slider(value: $redComponent)
    Slider(value: $greenComponent)
    Slider(value: $blueComponent)
}
```

Теперь добавим в стек цветной прямоугольник:

- В библиотеке объектов найдите элемент Color и разместите его в самом низу стека. В качестве аргументов передайте свойства, описывающие

компоненты цвета. При этом нет необходимости использовать $, так как прямоугольник должен не менять значения этих свойств, а лишь получать их (листинг 38.13).

Листинг 38.13

```
VStack {
    Slider(value: $redComponent)
    Slider(value: $greenComponent)
    Slider(value: $blueComponent)
    Color(red: redComponent, green: greenComponent, blue: blueComponent)
}
```

Интерфейс приложения на Canvas меняется вслед за кодом, который вы пишете. Самое интересное, что для того, чтобы протестировать работу интерфейса, вам не нужно запускать приложение на симуляторе. Для этого достаточно нажать кнопку Live Preview, расположенную над Preview на Canvas (рис. 38.19), после чего все элементы интерфейса станут интерактивными прямо на Canvas.

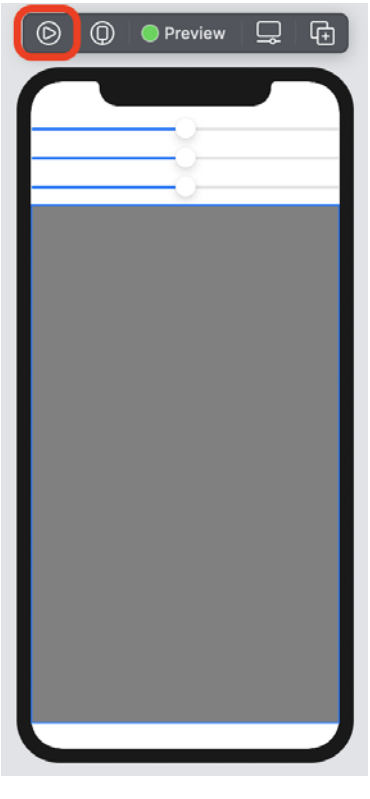

Рис. 38.19. Live Preview

Нам осталось лишь добавить отступы от краев экрана:

- Сделайте активным элемент `VStack` (щелкните по его имени в редакторе кода).
- На панели Attributes Inspector в разделе Padding отметьте все четыре галочки, а в центральном поле впишите `10` (рис. 38.20).

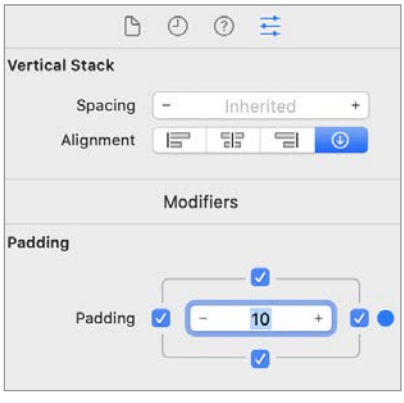

Рис. 38.20. Изменение отступа VStack

На этом мы закончили разработку приложения. Теперь вы можете запустить его на симуляторе или на реальном устройстве и проверить работу.

Глава 39. Паттерны проектирования

Вы прошли весь учебный материал курса, нерассмотренной осталась лишь одна ознакомительная тема, довольно важная для любого iOS-разработчика, чтобы не упомянуть о ней.

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, паттерны проектирования формализуют и описывают решение типовых задач программирования.

В настоящее время существует множество книг, описывающих всевозможные паттерны, 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 краткие описания нескольких базовых паттернов, с которыми вы встретитесь в ходе дальнейшего обучения разработке 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 большое количество проектов построены с учетом MVC (или других паттернов, взявших за основу этот прекрасный шаблон проектирования).

Как уже говорилось, данный шаблон подразумевает разделение всех составляющих проекта на три категории:

1. **Модель** — объекты, обеспечивающие хранение данных ваших приложений и логику их обработки.

2. **Отображение (Представление)** — объекты, описывающие различные графические элементы, которые видит пользователь при работе с приложением.
3. **Контроллер** — объекты, обеспечивающие совместную работу «отображения» и «модели».

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 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 других приложениях.

Все возможности по разработке iOS-приложений обеспечивает 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 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 будете создавать механизмы, обеспечивающие хранение и обработку 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 невоз-

можно 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 некоторый 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 свойство `Prefences.shared`. При попытке доступа к данному свойству из любого другого класса будет возвращен один и тот же экземпляр (листинг 39.1).

Листинг 39.1

```
class Prefences {
    // свойство для доступа к объекту
    static let shared = Prefences()

    // произвольные настройки проекта
    var backgroundColor: UIColor = .white
    var defaultUserLogin = "Guest"
}

// получаем класс с настройками с помощью Singleton-свойства
Prefences.shared.defaultUserLogin // Guest
Prefences.shared.backgroundColor  // UIColor.white

// или получаем ссылку на класс и записываем в произвольный параметр
var pref = Prefences.shared
pref.backgroundColor = .red
Prefences.shared.backgroundColor // UIColor.red
```

Примером использования паттерна Singleton при разработке под iOS может служить класс `UIApplication`, экземпляр которого является стартовой точкой каждого приложения. Любое создаваемое вами приложение содержит в себе и использует

только один экземпляр класса `UIApplication`, доступ к которому обеспечивается с помощью шаблона `Singleton`. Класс `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 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. Его знание 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.

Все описанные в данной главе паттерны и компоненты будут подробно рассмотрены в следующей книге.

Заключение

Поздравляю вас! Вы завершили вводный курс по основам разработки на языке программирования Swift. Вы освоили довольно большой объем учебного материала, попробовали самостоятельно создать несколько несложных приложений. Надеюсь, что сейчас вы вдохновлены тем, что узнали, и тем, что смогли попробовать. Но самое интересное ждет вас впереди. Мои следующие книги будут посвящены не возможностям Swift как языка программирования, а их применению при разработке реальных проектов.

Цели, которые мы ставили перед собой во время чтения книги, достигнуты, и вы можете двигаться дальше. А что же дальше?

Я предлагаю вам продолжить обучение по моим книгам, актуальную информацию о которых вы всегда сможете найти на сайте https://swiftme.ru. При этом наверняка у вас в голове зреет «план покорения мира», идея прекрасного приложения! Не уходите от нее и не откладывайте ее в долгий ящик. Параллельно с дальнейшим изучением возможностей Swift и Xcode попытайтесь реализовать ее. Это станет отличной практикой.

Создавайте и творите!

Вы сделали первый и поэтому самый важный шаг — дальше будет только интереснее.

И конечно же, присоединяйтесь к нам:

Сайт сообщества

https://swiftme.ru

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

Мы в Telegram

https://swiftme.ru/telegramchat или https://t.me/usovswift

Если в процессе чтения книги у вас появились вопросы, то вы можете задать их в нашем чате в Telegram.

Опечатки книги

https://swiftme.ru/typo16

Здесь вы можете посмотреть перечень всех опечаток, а также оставить информацию о найденных вами и еще не отмеченных. Документ создан в Google Docs, для доступа нужен Google-аккаунт.

Василий Усов

Swift. Основы разработки приложений под iOS, iPadOS и macOS

6-е издание, дополненное и переработанное

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*М. Петруненко*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
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: 10.2020. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

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

Подписано в печать 21.10.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 43,860. Тираж 1200. Заказ 0000.