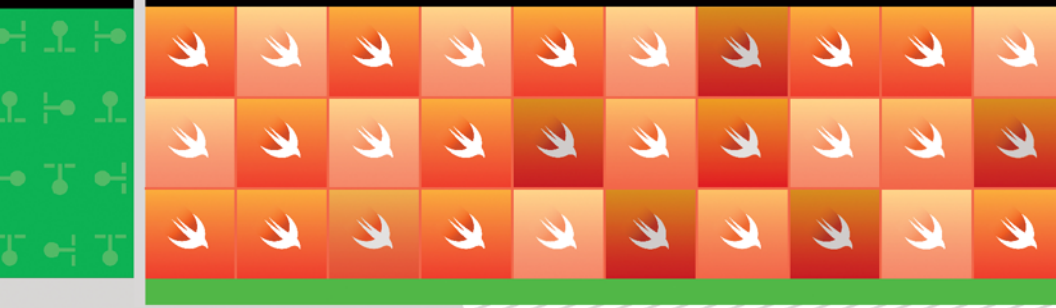
БИБЛИОТЕКА ПРОГРАММИСТА



Василий Усов

Swift

Основы разработки приложений под iOS и OS X

2-е издание

Swift 2.2

Василий Усов

Swift. Основы разработки приложений под iOS и OS X. 2-е изд.

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Римицан*
Литературный редактор	*Н. Суслова*
Художник	*С. Маликова*
Корректоры	*Н. Викторова, В. Сайко*
Верстка	*Л. Соловьева*

ББК 32.973.2-018.1
УДК 004.438

Усов В.

У76 Swift. Основы разработки приложений под iOS и OS X. 2-е изд. — СПб.: Питер, 2016. — 336 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-496-02451-8

Swift — быстрый, современный, безопасный и удобный язык программирования — появился совсем недавно и стал огромным сюрпризом для iOS-общественности. Спустя год Apple выпустила версию 2.0, а следом и 2.1, привнесшую в процесс разработки ряд значительных нововведений. И вот теперь выходит версия 2.2, готовящая нас к ожидаемому всеми iOS-программистами Swift 3.0.

Данная книга содержит исчерпывающую информацию для всех желающих научиться программировать на замечательном языке Swift и создавать собственные iOS-приложения. Вы найдете не только теоретический материал, но и большое количество практических примеров и заданий, которые позволят постичь все тонкости нового языка. Дерзайте, ведь, изучив Swift, вы сможете создавать приложения для любой платформы — iOS, OS X, tvOS или watchOS.

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

ISBN 978-5-496-02451-8

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.
Подписано в печать 20.04.16. Формат 60x90/16. Бумага писчая. Усл. п. л. 21,000. Тираж 1000. Заказ
Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1. Сайт: www.chpk.ru.
E-mail: marketing@chpk.ruФакс: 8(496) 726-54-10, телефон: (495) 988-63-87

Оглавление

Посвящается Ольге.
Девушке, которая вдохновила меня
на этот важный труд.

Введение

На ежегодной всемирной конференции разработчиков на платформе Apple (Worldwide Developers Conference, WWDC) 2 июня 2014 года «яблочная» компания приятно удивила iOS-общественность, представив новый язык программирования, получивший название Swift. Это стало большой неожиданностью: максимум, на что рассчитывали разработчики, привыкшие к теперь уже уходящему в прошлое языку Objective-C, — это обзор новых возможностей iOS 8 и новые прикладные программные интерфейсы для работы с ними. Оправившись от шока, разработчики подступились к Swift, изучая и, конечно же, критикуя его. Спустя год, выпустив несколько промежуточных обновленных версий языка, 8 июня 2015 года Apple анонсировала выход версии с индексом 2.0, которая стала доступна вместе с финальной сборкой мобильной операционной системы iOS 9. И вот теперь выходит версия 2.2, готовящая нас к ожидаемому всеми iOS-программистами Swift 3.0

Следующим значительным шагом в развитии языка стало открытие его исходного кода для разработчиков, другими словами, Swift получил статус open-source. Всю интересующую вас информацию, включая сам исходный код, вы можете найти на портале swift.org. В настоящее время Swift имеет поддержку во всей линейке продукции Apple, а также операционной системе Linux. В скором времени будет реализована поддержка и других платформ.

Если вы когда-либо писали приложения на языке Objective-C, то после изучения Swift с его многообразием возможностей вы, вероятно, захотите переписать свои приложения на новом языке программи-

рования[1]. После выхода Swift многие разработчики пошли именно по этому пути, понимая, что в будущем наибольшее внимание Apple будет уделять развитию нового языка. Более того, Swift стал первой разработкой Apple с открытым исходным кодом, что говорит о скором внедрении его поддержки и другими операционными системами (а не только iOS и OS X)[2].

О Swift

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

Изучая этот замечательный язык, вы удивитесь, насколько он в связке с Xcode (средой разработки, на мы ней остановимся позже) превосходит другие языки программирования, на которых вы писали программы ранее. Его простота, лаконичность и невероятные возможности просто поразительны!

Язык Swift создан полностью с нуля, поэтому обладает рядом особенностей:

- **Современность**

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

- **Объектно-ориентированность**

  Swift — объектно-ориентированный язык программирования, придерживающийся парадигмы «всё — это объект». Если в настоящий

---

[1] Swift в значительной мере 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] В настоящее время приложения на Swift можно разрабатывать не только для операционных систем iOS и OS X, но и для watchOS (операционная система «умных» часов Apple Watch) и tvOS (операционная система телевизионной приставки Apple TV 4-го поколения). Однако изучение приемов разработки приложений для различных операционных систем выходит за рамки темы данной книги.

момент данное утверждение показалось вам непонятным, не переживайте: чуть позже мы еще вернемся к нему.

- **Читабельность, экономичность и лаконичность кода**

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

- **Безопасность**

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

- **Производительность**

  Swift — очень молодой язык, тем не менее по производительности разрабатываемых программ он приближается (а в некоторых случаях уже и обгоняет) всем известного «старичка» — язык программирования C++[1].

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

О книге

Использование современных смартфонов для решения возникающих задач стало нормой. В связи с этим многие компании обращают все более пристальное внимание на обеспечение функционального доступа к предлагаемым ими услугам посредством мобильных приложений (будь то оптимизированный интернет-сайт, открываемый в браузере, или специальная программа). iOS является одной из популярнейших

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

мобильных операционных систем в мире, и в такой ситуации спрос на мобильное программирование растет небывалыми темпами.

Данная книга содержит исчерпывающую информацию для всех желающих научиться программировать на замечательном языке Swift с целью создания собственных iOS-приложений (а также OS X-, watchOS- и tvOS-приложений). 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. Несмотря на то что вам предстоит пройти большой путь, это будет полезный и очень важный опыт. Книга не показывает, как писать iOS-приложения, она предназначена для изучения самого языка программирования Swift. Я считаю, что она даст вам возможность освоить новый язык и в скором времени приступить к написанию собственных приложений для App Store или Mac App Store. Изучив язык, в дальнейшем вы сможете самостоятельно выбрать, для какой платформы создавать программы — для iOS, OS X, tvOS или watchOS.

Примеры кода в данной книге соответствуют Swift версии не ниже 2.2, iOS версии не ниже 9.3 и Xcode версии не ниже 7.3. Если у вас более свежие версии, не беспокойтесь, весь описанный материал с большой долей вероятности будет без каких-либо ошибок работать и у вас. Но небольшая возможность того, что Apple незначительно изменит синтаксис Swift, все же существует. Если вы встретитесь с такой ситуацией, прошу вас отнестись с пониманием и сообщить мне об этом на электронный адрес book@swiftme.ru.

Также хочу обратить ваше внимание на портал swiftme.ru. Это новое развивающееся сообщество программистов на Swift. Здесь вы найдете различные уроки и курсы, которые помогут вам более глубоко изучить тему разработки приложений.

Решения для всех заданий, приведенных в книге, вы сможете найти по адресу http://swiftme.ru/solutions.

Исправления во втором издании

Вы держите в руках второе издание книги «Swift. Основы разработки под iOS». В сравнении с предыдущим изданием данная книга содержит следующие изменения и дополнения:

- Весь материал актуализирован в соответствии со Swift версии 2.2 и Xcode 7.3.
- Добавлено большое количество нового учебного материала. В частности, переработана глава 26. Теперь книга не просто обучает Swift, но и готовит вас к полноценной разработке приложений.

- Учтены пожелания и замечания пользователей по оформлению и содержанию.
- Исправлены найденные опечатки.

Для кого написана книга

Если вы положительно ответите на следующие вопросы:

- Вы имеете хотя бы минимальные знания о программировании на любом языке высокого уровня?
- Вы хотите научиться создавать программы для операционной системы iOS (для вашего гаджета iPhone и iPad), OS X, watchOS или tvOS?
- Вы предпочитаете обучение в практической форме скучным и монотонным теоретическим лекциям?

тогда эта книга для вас.

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

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

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

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

Помните: для того чтобы стать Великим программистом, требуется время! Будьте терпеливы и внимательно изучайте материал.

Структура книги

Книга состоит из четырех больших частей и одного приложения:

- **Часть I. Подготовка к разработке Swift-приложений**. В первой части вы начнете ваше путешествие в мир Swift, выполните самые важные и обязательные шаги, предшествующие началу разработки собственных приложений. Вы узнаете, как завести собственную учетную

запись Apple ID, как подключиться к программе apple-разработчиков, где взять среду разработки Swift-приложений, как с ней работать.

- **Часть II. Базовые возможности Swift**. После знакомства со средой разработки Xcode, позволяющей приступить к изучению языка программирования, вы изучите базовые возможности Swift. Вы узнаете, какой синтаксис имеет Swift, что такое переменные и константы, какие типы данных существуют и как всем этим пользоваться при разработке программ.
- **Часть III. Основные средства Swift**. Третья часть фокусируется на рассмотрении и изучении наиболее простых, но очень интересных средств Swift. О некоторых из них (например, о кортежах) вы, возможно, никогда не слышали, другие (например, массивы) вы, вероятно, использовали и в других языках.
- **Часть IV. Нетривиальные возможности Swift**. В четвертой части подробно описываются приемы работы с наиболее мощными и функциональными средствами Swift. Материал этой главы вы будете использовать с завидной регулярностью при создании собственных приложений в будущем. Также отличительной чертой данной главы является большая практическая работа по созданию первого интерактивного приложения.
- **Приложение. Изменения и нововведения Swift 2.2.** Если вы изучали какую-либо из предыдущих версий Swift, то информация, приведенная в данном приложении, позволит вам оперативно ознакомиться со всеми нововведениями и изменениями, которые принесла новая версия языка программирования.

Условные обозначения

ПРИМЕЧАНИЕ В данном блоке приводятся примечания и замечания.

ВНИМАНИЕ Так выглядят важные замечания к материалу.

Листинг

```
А это примеры кода (листинги)
```

СИНТАКСИС

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

Задание

Задания для выполнения вы найдете в отдельных разделах.

Часть I
Подготовка к разработке Swift-приложений

В первой части вы узнаете о том, что необходимо для начала разработки приложений на языке Swift. В настоящее время существует возможность разработки Swift-приложений под операционными системами OS X и Linux. Так как на данном этапе нашей целью является лишь изучение данного языка программирования, то вы можете выбрать наиболее подходящую для вас платформу, но если в будущем вы ставите своей целью разработку собственных программ для продукции фирмы Apple, то работа в системе OS X является обязательным условием. Позже мы вернемся к данному вопросу.

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

- ✓ Глава 1. Подготовка к разработке в OS X
- ✓ Глава 2. Подготовка к разработке в Linux

1 Подготовка к разработке в OS X

1.1. Вам необходим компьютер Mac

Прежде чем приступить к разработке программ на языке Swift в OS X, вам потребуется несколько вещей. Для начала понадобится компьютер iMac, MacBook, Mac mini или Mac Pro с установленной операционной системой Mavericks (OS X 10.9), Yosemite (OS X 10.10) или El Capitan (OS X 10.11). Это первое и базовое требование связано с тем, что среда разработки приложений Xcode создана компанией Apple исключительно с ориентацией на собственную платформу.

Если вы ранее никогда не работали с Xcode, то будете поражены широтой возможностей данной среды разработки и необычным подходом к разработке приложений (в отличие от многих других сред программирования). Естественно, далеко не все эти возможности рассмотрены в книге, поэтому я советую вам в дальнейшем самостоятельно продолжить ее изучение.

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

1.2. Зарегистрируйтесь как Apple-разработчик

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

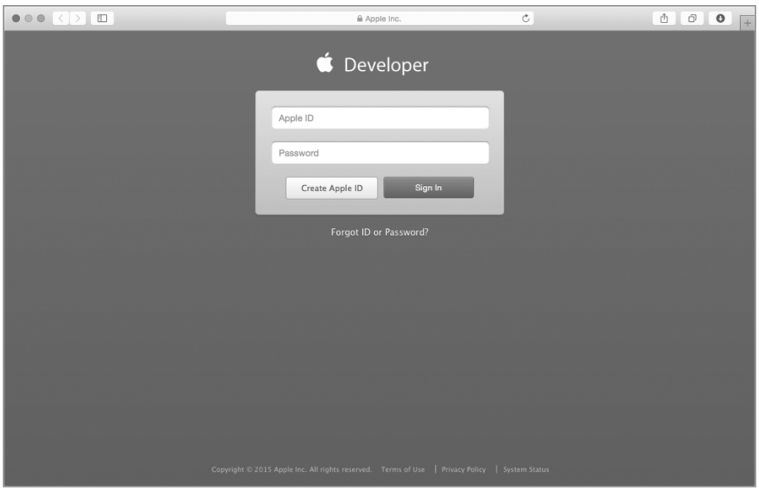


Рис. 1.1. Страница входа в Центр разработчиков

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

Чтобы создать новую учетную запись Apple ID, перейдите по ссылке Create Apple ID. Для регистрации вам потребуется ввести требуемые данные и щелкнуть на кнопке в нижней части формы.

Если у вас уже есть учетная запись Apple ID, то используйте данные своей учетной записи для входа в Центр разработчиков (кнопка Sign In).

На стартовой странице Центра разработчиков необходимо перейти по ссылке SDKs, после чего вам станет доступно все многообразие возможностей Центра Apple-разработчиков (рис. 1.2).

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

Регистрация в качестве разработчика бесплатна. То есть каждый может начать разрабатывать приложения, не заплатив за это ни копейки (если не учитывать стоимость компьютера). Тем не менее за 99 долларов в год вы можете участвовать в платной программе iOS-разработчиков (iOS Developer Program), которую вам предлагает Apple. Это не обязательно, но если вы хотите тестировать ваши

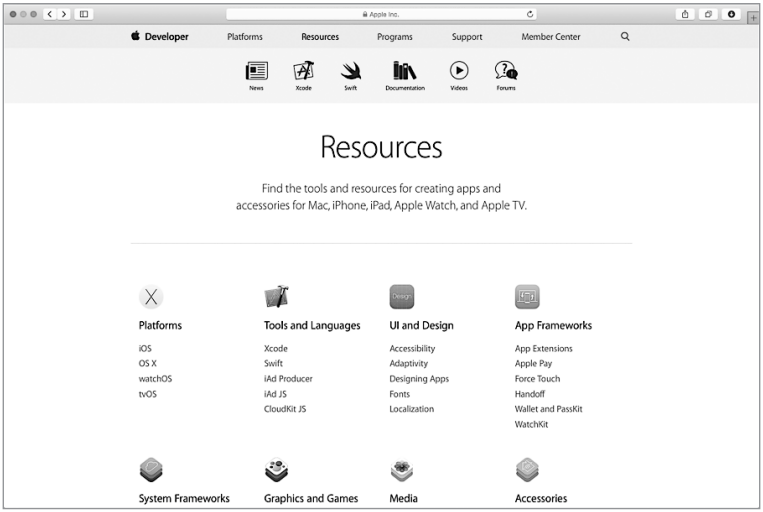


Рис. 1.2. Перечень ресурсов, доступных в Центре разработчиков

приложения на реальных устройствах (а не только в программном симуляторе) и использовать App Store (мобильный магазин приложений) для распространения своих программ, то участие в программе становится обязательным.

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

1.3. Установите Xcode

Теперь все, что вам необходимо, — скачать Xcode. Для этого перейдите по соответствующей ссылке в Центре разработчиков на страницу, посвященную этой замечательной среде разработки (рис. 1.3).

ПРИМЕЧАНИЕ Как вы можете видеть, Apple предоставляет вам доступ к бета-версиям Xcode. Настоятельно не рекомендую использовать их при изучении языка программирования, так как они предназначены для опытных Apple-разработчиков.

После щелчка по ссылке View in the Mac App Store произойдет автоматический переход в магазин приложений Mac App Store на страницу Xcode. Здесь вы можете увидеть краткое изложение всех основных

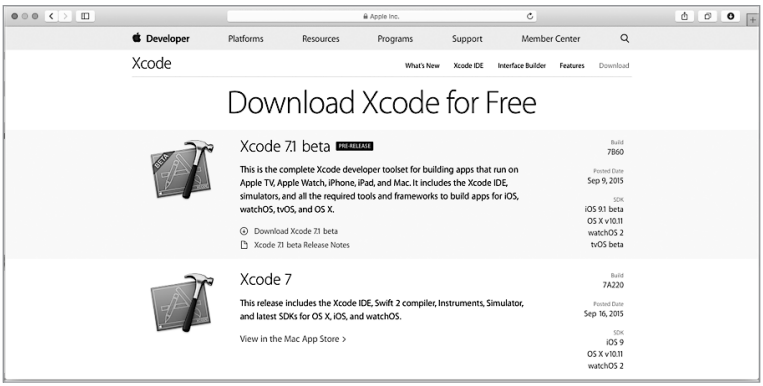


Рис. 1.3. Страница Xcode в Центре разработчиков

возможностей среды разработки, обзор последнего обновления и отзывы пользователей. Для скачивания Xcode просто щелкните на кнопке Загрузить и при необходимости введите данные своей учетной записи Apple ID.

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

1.4. Введение в Xcode

Изучение программирования на языке Swift мы начнем со среды разработки Xcode.

ПРИМЕЧАНИЕ Интегрированная среда разработки (Integrated Development Environment, IDE) — система программных средств, используемая 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 OS X. Xcode — это наиболее важный инструмент, который использует разработчик. Среда Xcode удивительна! Она предоставляет широкие возможности, и изучать их следует постепенно, исходя из поставленных и возникающих задач. Внешний вид рабочей среды приведен на рис. 1.4.

Именно с использованием этого интерфейса разрабатываются любые приложения для iOS и OS X. При изучении Swift вы будете взаимодействовать с иной рабочей областью — рабочим интерфейсом playground-проектов. О нем мы поговорим чуть позже.

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

Я надеюсь, что вы уже имеете на своем компьютере последнюю версию Xcode, а значит, мы можем перейти к изучению этой замечательной среды. А 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).

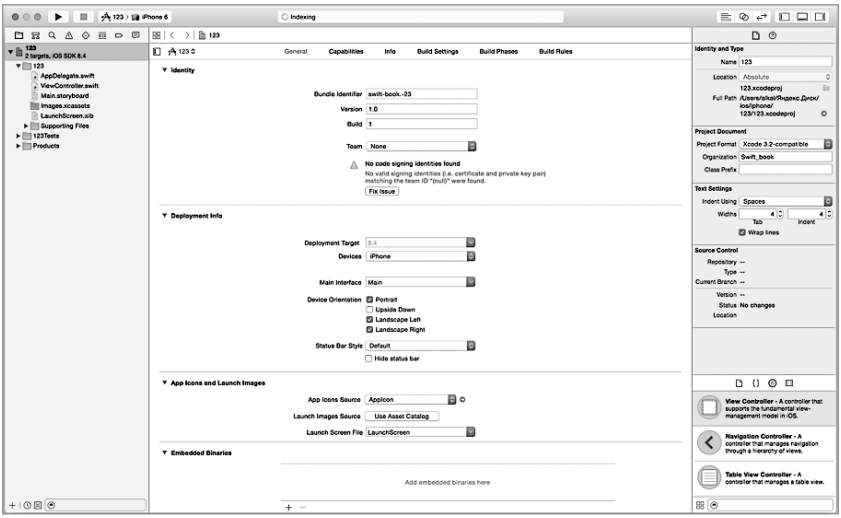


Рис. 1.4. Интерфейс Xcode

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

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

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

ПРИМЕЧАНИЕ Идея SCM (software configuration management) базируется на удаленном хранилище, хранящем файлы вашего проекта. При необходимости разработчики скачивают оригинальный файл из репозитория, изменяют его и загружают измененную версию назад. В состав SCM обычно входит CVS (система контроля версий), сохраняющая всю линейку изменений каждого отдельного файла. Наиболее популярным SCM-репозиторием является github.com.

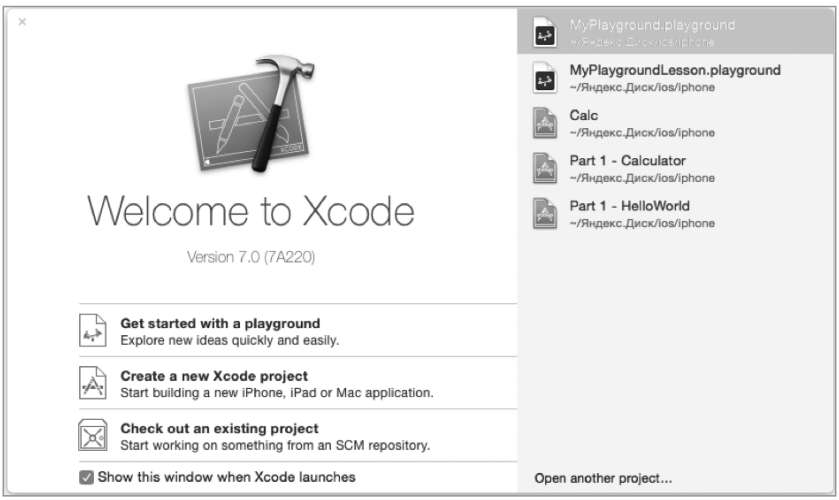


Рис. 1.5. Стартовое окно Xcode

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

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

Одним из потрясающих нововведений Xcode 7, помимо поддержки Swift, является появление playground-проектов. Playground-проект — это интерактивная среда разработки, своеобразная «песочница» или «игровая площадка», где вы можете комфортно тестировать создаваемый вами код и видеть результат его исполнения в режиме реального времени.

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

Как вы можете видеть, внешний вид интерфейса playground-проекта значительно отличается от рабочей области Xcode, которую вы видели ранее в книге. Повторю, что playground-проект позволяет писать код и незамедлительно видеть результат его исполнения, хотя и не служит

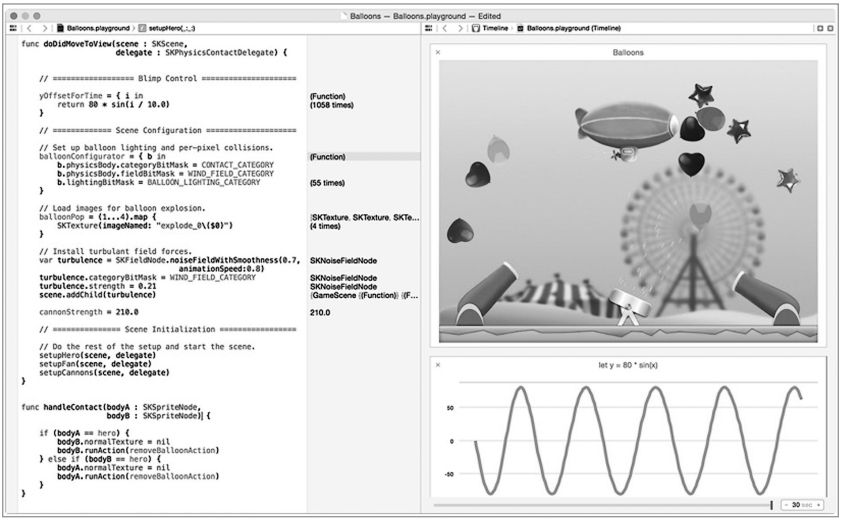


Рис. 1.6. Пример playground-проекта

для создания полноценных самостоятельных проектов. Каждый playground-проект хранится в файловой системе в виде особого файла с одноименным расширением. В то же время при разработке полноценного проекта вы можете встроить в него отдельный playground для решения возникающих проблем.

1.5. Интерфейс playground-проекта

Нет способа лучше для изучения языка программирования, чем написание кода. Playground-проект предназначен именно для этого. Выберите вариант Get started with a playground в стартовом окне для создания нового playground-проекта. Далее Xcode попросит вас ввести имя создаваемого playground-проекта, а также выбрать платформу, для которой вы будете писать код. На выбор предлагается две платформы: iOS и OS X. Разница лишь в доступных в playground фреймворках. Измените имя на «Part 1 Basics», выберите платформу iOS и щелкните на кнопке Next.

Далее требуется выбрать папку для сохранения создаваемого проекта; после выбора папки перед вами откроется рабочий интерфейс playground-проекта (рис. 1.7).

Рабочее окно состоит из двух частей:

- В левой части экрана расположен *редактор кода*, в котором вы можете писать и редактировать свой swift-код. В только что созданном нами файле имеется один комментарий и две строки кода.

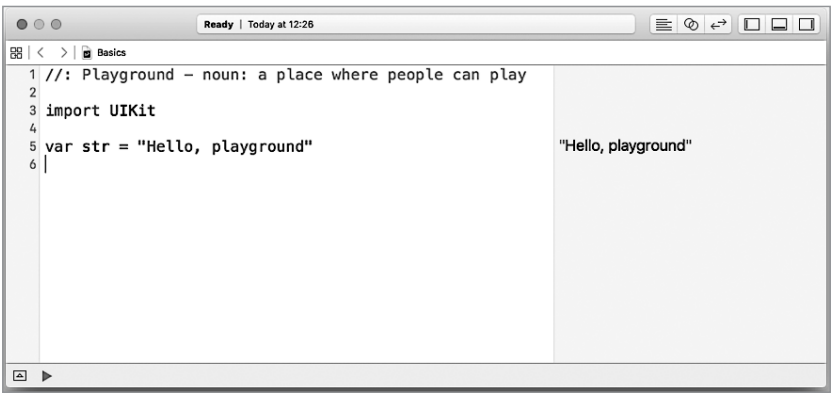


Рис. 1.7. Рабочее окно нового playground-проекта

- Как только код будет написан, Xcode моментально обработает его, отобразит ошибки и выведет результат в правой части экрана, в *области результатов*.

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

Если навести указатель мыши на строку `"Hello, playground"` в области результатов, то рядом появятся две кнопки, как показано на рис. 1.8.

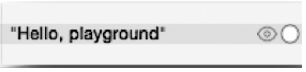


Рис. 1.8. Дополнительные кнопки в области результатов

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

1.6. Возможности playground-проекта

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

Начиная с версии 6.3 в Xcode появилась поддержка markdown-синтаксиса для комментариев. На рис. 1.9 приведен пример изменения внешнего вида комментариев после выбора в меню пункта Editor ▸ Show Rendered Markup. В верхней части изображен исходный код комментария, а в нижней — отформатированный.

В скором времени вы увидите, что в качестве результатов могут выводиться не только текстовые, но и графические данные (рис. 1.10).

Строки формата *N* `times` в области результатов, где *N* — целое число, говорят о том, что данная строка кода выводится *N* раз. Пример такой строки вы можете видеть на рис. 1.10. Подобные выводы результатов можно отобразить в виде графиков и таблиц. Со всеми возможными вариантами отображения результатов исполнения swift-кода вы познакомитесь в ходе работы с playground-проектами в Xcode.

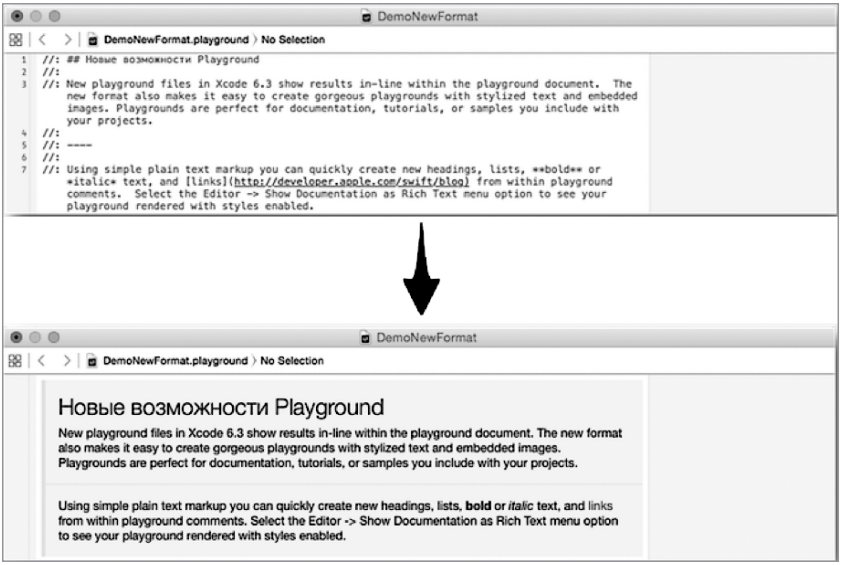


Рис. 1.9. Форматированный комментарий

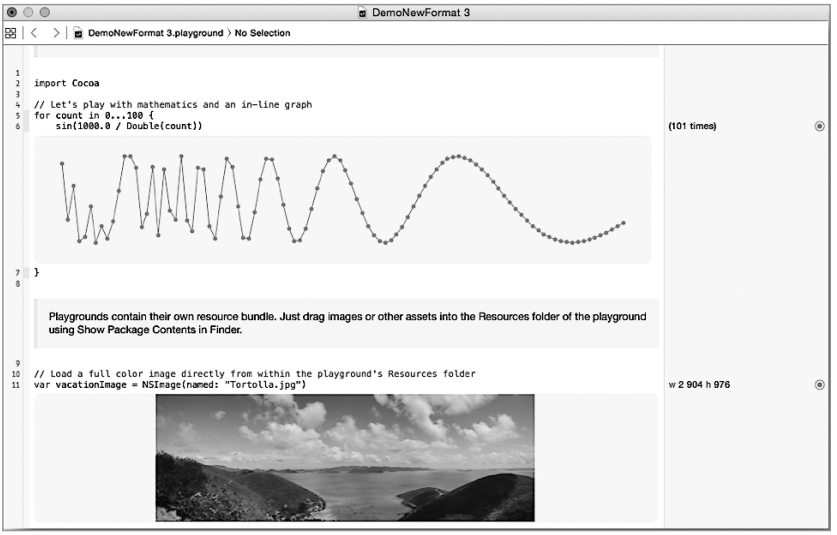


Рис. 1.10. Пример вывода результирующей информации

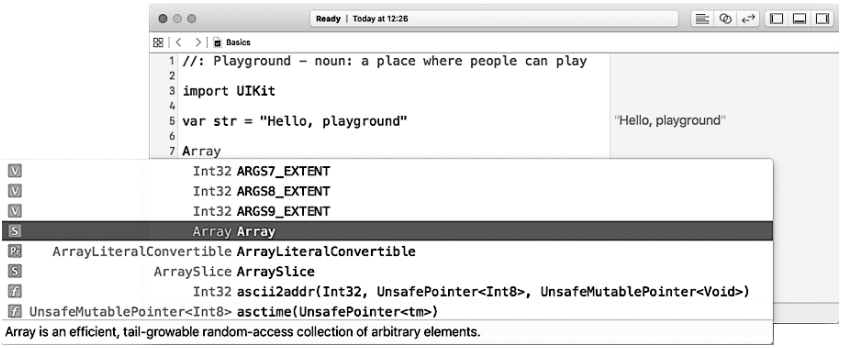


Рис. 1.11. Окно автодополнения в Xcode

Также Xcode имеет в своем арсенале такой полезный механизм, как автодополнение (в Xcode известное как автокомплит). Для примера в рабочей части только что созданного playground-проекта на новой строке напишите латинский символ «a» — вы увидите, что всплывет окно автодополнения (рис. 1.11).

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.

Одной из возможностей Xcode, которая значительно упрощает работу, является указание на ошибки в программном коде. Для каждой ошибки выводится подробная вспомогательная информация, позволяющая внести ясность и исправить недочет. Ошибка показывается с помощью красного значка в форме кружка слева от строки, где она обнаружена. При щелчке на этом значке появляется описание ошибки (рис. 1.12).

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

ПРИМЕЧАНИЕ В версиях Xcode ниже 7-й вся консольная информация выводилась в области Assistant Editor.

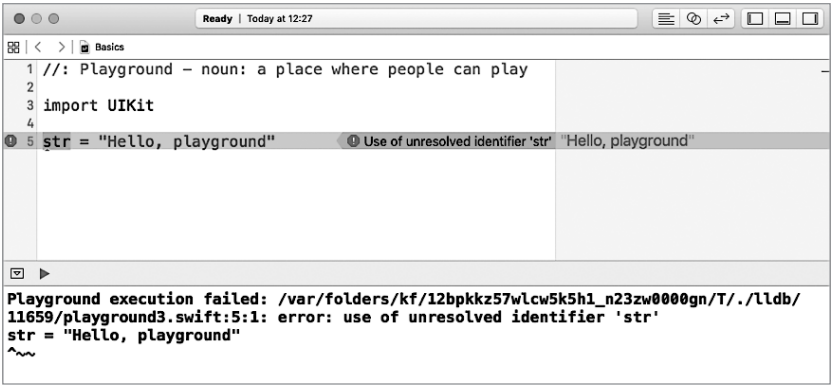


Рис. 1.12. Отображение ошибки в окне playground-проекта

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

Swift позволяет также получать исчерпывающую информацию об используемых в коде объектах. Если нажать клавишу Alt и щелкнуть на любом объекте в области кода (например, на `str`), то появится вспомогательное окно, позволяющее узнать тип объекта, а также имя файла, в котором он расположен (рис. 1.13).

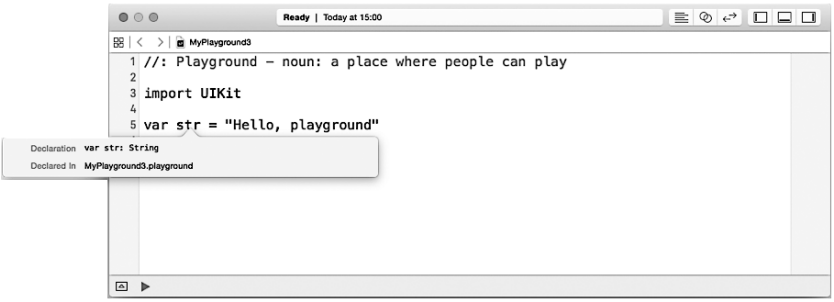


Рис. 1.13. Всплывающее окно с информацией об объекте

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

2 Подготовка к разработке в Linux

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

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

В качестве операционной системы мной была выбрана Ubuntu 14.04. Это проверенная временем очень стабильная сборка, имеющая максимально удобный интерфейс. В настоящее время на портале swift.org имеется версия и для Ubuntu 15.10.

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

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

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

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

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

```
parallels@ubuntu: ~
parallels@ubuntu:~$ sudo apt-get install clang libicu-dev
[sudo] password for parallels:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  binfmt-support clang-3.4 cpp-4.8 gcc-4.8 gcc-4.8-base icu-devtools libasan0
  libatomic1 libclang-common-3.4-dev libclang1-3.4 libffi-dev libffi6
  libgcc-4.8-dev libgomp1 libicu52 libitm1 libobjc-4.8-dev libobjc4
  libquadmath0 libstdc++-4.8-dev libstdc++6 libtinfo-dev libtsan0 llvm-3.4
  llvm-3.4-dev llvm-3.4-runtime
Suggested packages:
  gcc-4.8-locales gcc-4.8-multilib gcc-4.8-doc libgcc1-dbg libgomp1-dbg
  libitm1-dbg libatomic1-dbg libasan0-dbg libtsan0-dbg libquadmath0-dbg
  icu-doc libstdc++-4.8-doc llvm-3.4-doc
The following NEW packages will be installed:
  binfmt-support clang clang-3.4 icu-devtools libclang-common-3.4-dev
  libclang1-3.4 libffi-dev libicu-dev libobjc-4.8-dev libobjc4
  libstdc++-4.8-dev libtinfo-dev llvm-3.4 llvm-3.4-dev llvm-3.4-runtime
The following packages will be upgraded:
  cpp-4.8 gcc-4.8 gcc-4.8-base libasan0 libatomic1 libffi6 libgcc-4.8-dev
  libgomp1 libicu52 libitm1 libquadmath0 libstdc++6 libtsan0
13 upgraded, 15 newly installed, 0 to remove and 585 not upgraded.
Need to get 52.3 MB of archives.
After this operation, 178 MB of additional disk space will be used.
Do you want to continue? [Y/n]
```

Рис 2.1. Установка пакетов в Ubuntu

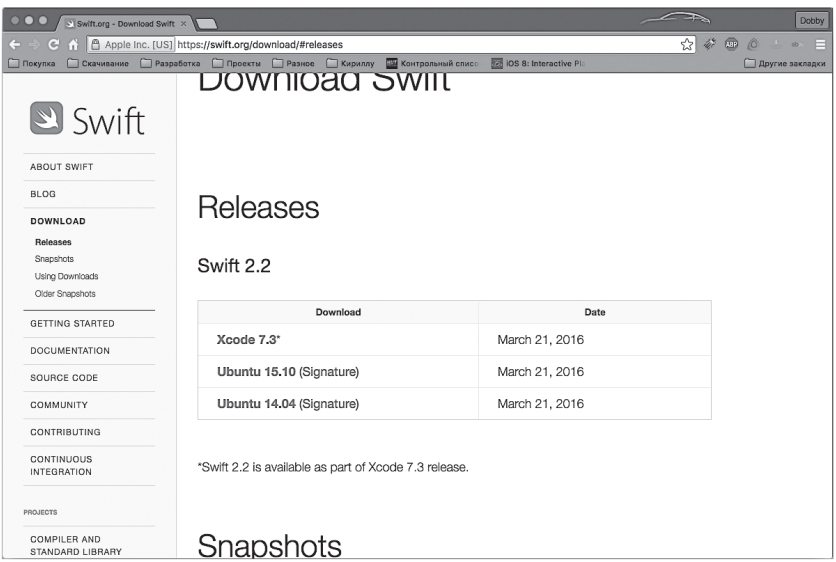


Рис 2.2. Доступные релизы Swift

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

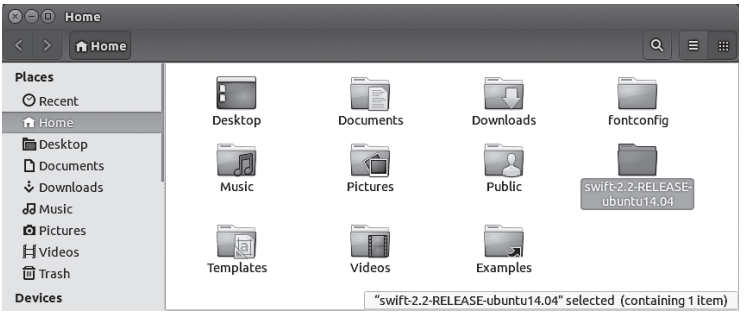


Рис 2.3. Распакованный архив в папке Home

Теперь введите в терминале следующую команду:

```
gedit .profile
```

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

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

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

```
home/parallels/swift-2.2-RELEASE-ubuntu14.04/usr/bin
```

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

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

```
swift –version
```

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

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

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

где

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

Это путь к созданному файлу.

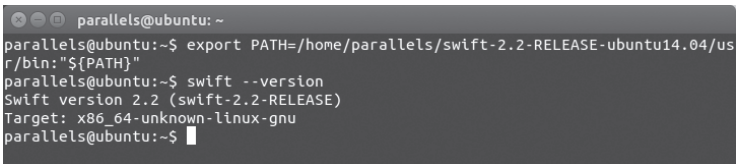


Рис 2.4. Вывод информации о версии Swift

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

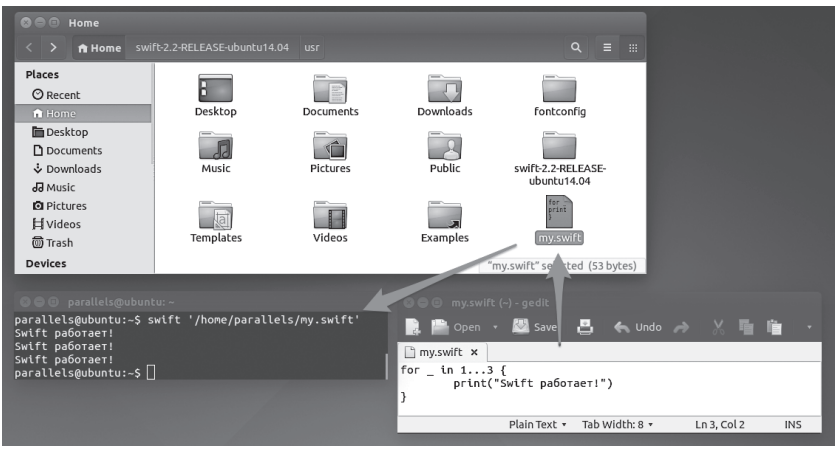


Рис 2.5. Пример выполненного swift-кода

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

В случае, если вы разрабатываете под Linux и у вас возникают какие-либо проблемы, прошу писать мне на book@swiftme.ru или искать информацию на портале swiftme.ru.

Часть II
Базовые возможности Swift

Пришло время перейти к изучению Swift. Мы будем писать код вместе, подробно разбирая все его возможности и механизмы.

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

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

- ✓ Глава 3. Отправная точка
- ✓ Глава 4. Типы данных и операции с ними

3 Отправная точка

Swift, как и любой другой язык программирования, выполняет свои функции с помощью команд, которые отдает (а точнее, пишет) разработчик. Завершенная команда языка Swift называется *выражением*. Файл с кодом обычно состоит из совокупности выражений, написанных на множестве строк. Примеры вы могли видеть на изображениях, демонстрирующих playground-проекты в Xcode.

Выражения в свою очередь состоят из операторов, модификаторов и других выражений. *Оператор* — это минимальная автономная единица (слово или группа слов), выполняющая определенную команду. *Модификаторы* — это функциональные единицы, расширяющие возможности операторов. Все зарезервированные языком программирования наборы символов называются *ключевыми словами*.

В Swift выделяют два важнейших понятия: объявление и инициализация.

- *Объявление* — это создание нового объекта, с которым планируется взаимодействие.
- *Инициализация* — это присвоение объявленному объекту определенного значения.

Рассмотрим следующий пример. Представьте, что существует дракон, у которого есть пустой сундук. Другими словами, существует (объявлено) хранилище с именем «Сундук дракона». Что будет храниться в данном хранилище, заранее неизвестно, но можно точно сказать, что к данному хранилищу можно обратиться по его имени, то есть среди множества различных хранилищ можно найти именно данный сундук (рис. 3.1).

Теперь дракон решил переложить в сундук все свое похищенное золото, или, другими словами, решил инициализировать «Сундук дракона»

определенным значением (рис. 3.2). Чтобы позже получить доступ к похищенному золоту, дракону нет необходимости искать и собирать его по всей пещере — достаточно будет лишь взять хранилище с именем «Сундук дракона» и обратиться к его содержимому.

Рис. 3.1. Пустое хранилище «Сундук дракона»

Рис. 3.2. Хранилище «Сундук дракона», содержащее в себе похищенное золото

Хранилище в данном примере является *объектом*, то есть некоторой сущностью, которая обладает набором свойств (свойством в данном случае является проинициализированное хранилищу значение). Значительное количество действий, которые вы будете программировать в будущем, будет либо определять некоторый объект, либо инициализировать его значение, либо запрашивать это значение, либо производить операции с этим значением. Например, при запросе того, что хранится в сундуке дракона, вам будет возвращено значение «Похищенное золото».

3.1. Инициализация и изменение значения

Базовые операторы

Как говорилось ранее, операторы — это минимальные автономные функциональные единицы, выполняющие некоторую команду. Операторы в Swift позволяют выполнять различные операции, например операции над значениями хранилищ. Значения, которые операторы затрагивают в своей работе, называются *операндами*. Другими словами, можно сказать, что операторы производят операции над операндами и с использованием операндов.

Рассмотрим простейший арифметический пример: 2 + 3. Данный пример является *выражением*, то есть законченной командой на языке математики. В нем можно выделить один *оператор* (сложение) и два *операнда* (2 и 3).

Вернемся к примеру с «Сундуком дракона». Чтобы увеличить количество золота в хранилище, выполняется команда «Прибавить к похищенному золоту три монеты». В этой команде один оператор: «Прибавить» и два операнда: «Похищенное золото» и «Три монеты». «Три монеты» — это тоже значение, но оно не назначено какому-либо хранилищу, а передается непосредственно.

Всего выделяют два основных вида операторов:

- **Простые операторы** выполняют операции с различными операндами. В их состав входят унарные и бинарные операторы. *Унарные операторы* выполняют операцию с одним операндом (например, `-a`). Они могут находиться перед операндом, то есть быть префиксными (например, `!b`), или после него, то есть быть постфиксными (например, `i++`). *Бинарные операторы* выполняют операцию с двумя операндами (например, `1+6`). Такой оператор всегда располагается между операндами и называется инфиксным.
- **Структурные операторы** влияют на ход выполнения программы. Отдельно можно выделить *тернарный оператор*, который выполняет операции с тремя операндами. В Swift имеется всего один тернарный оператор — оператор условия `a ? b : c`.

Оператор присваивания

Оператор присваивания (`=`) — это особый бинарный оператор. Он используется в типовом выражении `a = b`, инициализируя значение объекта `a` значением объекта `b`. Если обратиться к предыдущему примеру с «Сундуком дракона», то ему было присвоено значение «Похищенное золото» (он был инициализирован этим значением).

ПРИМЕЧАНИЕ В Swift оператор присваивания не возвращает присваиваемое значение, он лишь проводит инициализацию (установку значения).

Во многих языках программирования данный оператор возвращает присваиваемое значение, и вы можете, например, незамедлительно вывести его на консоль или использовать в качестве условия в операторе условия. В Swift подобный подход вызовет ошибку.

Запомните, что и левая и правая части оператора присваивания должны быть однотипными (то есть иметь одинаковый тип). Представьте, что помимо сундука дракона существует «Котел тролля», в котором находится желтая похлебка (рис. 3.3).

Исходя из содержимого обоих хранилищ, можно заключить, что сундук имеет тип «Твердые предметы», так как предназначен для хранения именно твердых предметов (например, золота), а котел, в свою очередь, имеет тип «Жидкости», так как предназначен исключительно для хранения жидкостей (например, похлебки).

ПРИМЕЧАНИЕ Тип определяет содержимое хранилища.

Так как типы хранилищ не совпадают, то и их содержимое нельзя поменять местами. Представьте, что будет, если, используя оператор присваивания, утверждать следующее:

```
Сундук дракона = Котел тролля
```

Таким образом, содержимое котла будет перелито (или присвоено) сундуку, откуда оно, конечно же, моментально вытечет и будет непригодно для приема.

По такому же принципу работают хранилища и в среде Swift, которая имеет ряд предустановленных типов этих самых хранилищ. Каждый из типов предназначен для хранения определенного вида значений (числа, текст, логические и другие значения).

Рис. 3.3. Хранилище «Котел тролля» с содержимым «Желтая похлебка»

3.2. Переменные и константы

Наиболее важное базовое понятие в любом языке программирования — переменная. *Переменная* — это некоторая именованная область (хранилище), в которой может храниться некоторое значение. Значение переменной может меняться с течением времени. Процесс изменения значения происходит не сам по себе, а инициируется извне при инициализации нового значения.

В ранее рассмотренном примере с «Сундуком дракона» и «Котлом тролля» сундук и котел — это переменные (или хранилища), содержащие значения «Похищенное золото» и «Желтая похлебка» соответственно. При необходимости можно изъять из сундука золото и положить туда другие твердые предметы, например мечи поверженных рыцарей. Таким образом, значение переменной изменится, но ее название останется прежним: все мечи поверженных рыцарей будут возвращены при обращении к хранилищу по имени «Сундук дракона».

Пришло время немного размять пальцы. Первое, чему вы научитесь при изучении Swift, — объявлять переменные. Переменные в Swift объявляются с помощью оператора `var`, за которым следует название и значение этой переменной.

СИНТАКСИС

```
var имя_переменной = значение_переменной
```

После оператора var указывается имя создаваемой переменной, с помощью которого будет производиться обращение к записанному в переменной значению (или с помощью которого это значение будет изменяться). Далее, после имени и оператора присваивания, следует инициализируемое переменной значение.

Пример объявления переменной и инициализации ее значения приведен в листинге 3.1.

Листинг 3.1

```
1  // переменные объявляются с помощью ключевого слова var
2  var dragonsBox = "Похищенное золото"
```

В данном примере создается переменная с именем `dragonsBox`, содержащая значение `"Похищенное золото"`. То есть код можно прочитать следующим образом:

> Объявить переменную с именем `dragonsBox` и присвоить ей значение `"Похищенное золото"`.

В ходе работы над программой можно объявлять произвольное количество переменных.

Для того чтобы изменить значение переменной, необходимо присвоить ей новое значение. Оператор `var` при этом повторно не используется (листинг 3.2).

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

Будьте внимательны: во время инициализации нового значения старое уничтожается.

Листинг 3.2

```
1  // переменные объявляются с помощью ключевого слова var
2  var dragonsBox = "Похищенное золото"
3  // изменяем значение объявленной ранее переменной
4  dragonsBox = "Мечи поверженных рыцарей"
```

В результате выполнения кода в переменной `dragonsBox` будет храниться значение `"Мечи поверженных рыцарей"`.

А что будет, если хранилище «Сундук дракона» запереть невскрываемым замком? В таком случае данный сундук будет навсегда связан с похищенным золотом и положить в него что-то иное (то есть изменить существующее значение) будет просто невозможно.

В Swift есть возможность единожды указать значение переменной (повесить на нее невскрываемый замок) и лишить возможности изменять это значение в будущем. Такие переменные имеют собственное название — *константы*. Они объявляются с помощью оператора `let`.

СИНТАКСИС

```
let имя_константы = значение_константы
```

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

Пример объявления константы и инициализации ее значения приведен в листинге 3.3.

Листинг 3.3

```
1  // константы объявляются с помощью ключевого слова let
2  let dragonsBox = "Похищенное золото"
```

В результате будет объявлена константа `dragonsBox`, содержащая значение `"Похищенное золото"`. Данное значение нельзя удалить ни при каких обстоятельствах.

Другими словами, константа — это некоторое именованное хранилище, значение которого можно задать лишь один раз. Указав значение единожды, его невозможно будет изменить на всем протяжении работы программы.

При попытке изменить значение константы Xcode сообщит об ошибке (листинг 3.4).

Листинг 3.4

```
1  let dragonsBox = "Похищенное золото"
2  dragonsBox = "Мечи рыцарей" // ОШИБКА: попытка изменить константу
```

Раньше, программируя на других языках, вы, вероятно, не очень активно использовали константы. В Swift для оптимизации работы программ вам следует прибегать к использованию констант во всех случаях, когда инициализированное значение не должно и не будет изменяться в ходе работы приложения. Более того, Xcode следит за тем, изменяется ли значение объявленных переменных в ходе работы программы, и при необходимости оповестит вас, что переменную желательно изменить на константу.

ПРИМЕЧАНИЕ Как говорилось ранее, объявлением переменной или константы называется операция задания имени и типа переменной или константы. Инициализацией называется операция назначения переменной или константы некоторого значения. Если вы одновременно определяете имя и значение переменной или константы, то объявление и инициализация совмещаются.

Запомните, что операторы `var` и `let` необходимо задействовать только при объявлении параметра. В дальнейшем при обращении к объявленным переменным и константам требуется использовать только их имена.

ПРИМЕЧАНИЕ Swift не ограничивает количество объявляемых переменных или констант после операторов var и let. Следующий код будет исполняться корректно:

```
var x = 0.1, y = 5
let a = 2, b = 7.1
```

Все объявляемые в одном выражении переменные и константы необходимо разделять символом «запятая».

Использование переменных и констант привносит удобство и логичность в Swift в сравнении, например, с языком Objective-C, в котором все было значительно сложнее.

3.3. Правила объявления переменных и констант

Swift дает широкий простор в создании (объявлении) переменных и констант. Вы можете называть их произвольными именами, использовать Unicode, эмодзи (даже так!), но при этом следует придерживаться некоторых правил:

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

  Хотя их имена и могут содержать любые Unicode-символы (листинг 3.5), их использование только мешает читабельности кода.

Листинг 3.5

```
1 // Попробуйте прочитать имя этой переменной
2 var dØw = "RǫÏ"
```

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

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

Если вам требуется дать переменной или константе имя, зарезервированное в Swift под какой-либо служебный оператор, то следует написать его в апострофах (`), но я настоятельно рекомендую вам избегать этого, чтобы не нарушать читабельность кода (листинг 3.6). В Swift существует оператор с именем `var`. Для того чтобы создать переменную с таким же именем, необходимо использовать апострофы.

Листинг 3.6

```
1 // создадим переменную с именем var
2 var `var` = "Пример переменной в апострофах"    "Пример переменной
                                                  в апострофах"
3 `var`                                           "Пример переменной
                                                  в апострофах"
```

Обратите внимание, что в предыдущем примере в области результатов дважды вывелось значение объявленной переменной `` `var` ``. Далее мы поговорим о том, как выводить информацию в эту область.

3.4. Вывод текстовой информации

Как говорилось ранее, в Xcode при работе с playground-проектами можно вывести любое значение в окне результатов. Для этого необходимо всего лишь написать имя объявленной и проинициализированной ранее переменной или константы (листинг 3.7).

Листинг 3.7

```
1  // объявим переменную и присвоим ей значение
2  var dragonsBox = "Мечи рыцарей"                    "Мечи рыцарей"
3  // отобразим значение в области результатов
4  dragonsBox                                         "Мечи рыцарей"
```

При разработке проектов (будь то playground-проект или полноценное приложение для iOS) любые данные можно отобразить на отладочной консоли (с ней мы знакомились раннее). Вывод на консоль осуществляется с помощью глобальной функции `print(_:)`.

ПРИМЕЧАНИЕ Функция — это именованный фрагмент программного кода, к которому можно обращаться многократно.

Функции предназначены для того, чтобы избежать дублирования кода. Они группируют часто используемый код и позволяют обращаться к нему с помощью уникального имени.

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

В некоторых случаях функция может получать входные параметры, которые будут использоваться внутри ее реализации для получения результата. Входные параметры, также известные как аргументы функции, указываются в скобках после имени самой функции.

При описании функций в тексте книги возможно указание на необходимость передачи входных параметров. При этом для каждого аргумента указывается его имя, после чего ставится двоеточие. Если входной параметр не имеет имени, то вместо него ставится нижнее подчеркивание (вы могли видеть это перед примечанием при указании на функцию print(_:)).

Таким образом, указание на использование функции someFunction(_:text:) говорит о том, что вы можете использовать функцию с именем someFunction, у которой есть два входных параметра: первый не имеет имени, а второй должен быть передан с именем text.

Пример вызова этой функции приведен ниже:

someFunction(21 text:"Hello!")

Пример работы с функцией `print(_:)` представлен в листинге 3.8.

Листинг 3.8

```
1  // вывод информации в дебаг-консоль
2  print("Привет, Дракон!")                    "Привет, Дракон!\n"
```

Консоль:

```
Привет, Дракон!
```

Приведенный код производит вывод информации, переданной функции `print(_:)`.

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

ПРИМЕЧАНИЕ В Xcode 6 существовало две функции для вывода информации в консоль: println(_:) и print(_:). Первая добавляла в конец выводимого выражения символ переноса строки, а вторая нет. В Xcode 7 функция print(_:) была упразднена, а println(_:) переименована в print(_:). Отныне любой вывод информации на консоль завершается символом переноса строки (\n).

Функция вывода на консоль может принимать на входе не только текст, но и произвольный параметр (переменную или константу), как показано в листинге 3.9.

Листинг 3.9

```
1  // объявим переменную и присвоим ей значение
2  var dragonsBox = "Мечи рыцарей"                "Мечи рыцарей"
3  // Выводим значение созданной ранее переменной
4  print(dragonsBox)                              "Мечи рыцарей\n"
```

Консоль:

```
Мечи рыцарей
```

Созданная переменная `dragonsBox` передается в функцию `print(_:)` в качестве входного аргумента (или входного параметра), и ее значение выводится также на консоли.

Помимо этого существует возможность объединить вывод текстовой информации со значением некоторого параметра (или параметров). Для этого в требуемом месте в тексте необходимо использовать символ обратной косой черты (слеша), после которого в круглых скобках нужно указать имя выводимого параметра. Пример приведен в листинге 3.10.

Листинг 3.10

```
1  // объявим переменную и присвоим ей значение
2  var dragonsBox = "Мечи рыцарей"
```

```
// Выводим значение созданной ранее переменной
print("В этом сундуке лежат \(dragonsBox)")
```

Консоль:

```
В этом сундуке лежат Мечи рыцарей
```

Вывод на консоль и текст в области результатов снова совпадают (за исключением символа переноса строки).

3.5. Комментарии

Стандартные комментарии

Правильно написанный код должен быть хорошо прокомментирован. Комментарии помогают нам не запутаться и не забыть о предназначении написанного. Если вы не будете их использовать, то рано или поздно попадете в ситуацию, когда навигация по собственному проекту станет невыносимо сложной.

Комментарии в Swift, как и в любом другом языке программирования, представляют собой блоки неисполняемого кода, например заметки или напоминания. В Swift присутствуют однострочные и многострочные комментарии.

Однострочные комментарии пишутся с помощью удвоенной косой черты (`// комментарий`) перед текстом комментария, в то время как многострочные обрамляются звездочками и косыми чертами с обеих сторон (`/* комментарий */`). Пример комментариев приведен в листинге 3.11.

Листинг 3.11

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

Весь текст, находящийся в комментарии, игнорируется компилятором и никоим образом не влияет на выполнение программы.

В предыдущей главе упоминалось, что playgound-проекты в Xcode поддерживают markdown-комментарии — особый форматированный вид комментариев. Они позволяют превратить playground-проект в настоящий обучающий материал. Такой вид комментариев должен начинаться с удвоенной косой черты и двоеточия (`//:`), после которых и следует текст комментария. Несколько примеров неформатированных комментариев приведены на рис. 3.4.

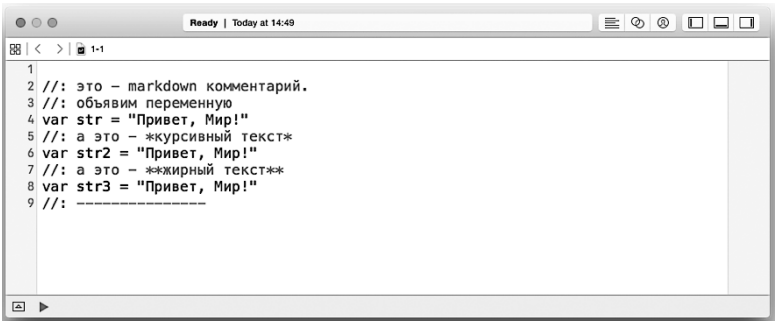


Рис. 3.4. Неформатированные markdown-комментарии

Включить форматирование комментариев, при котором все markdown-комментарии отобразятся в красивом и удобном для чтения стиле, можно, выбрав в меню Xcode пункт Editor ▸ Show Rendered Markup. Результат приведен на рис. 3.5.

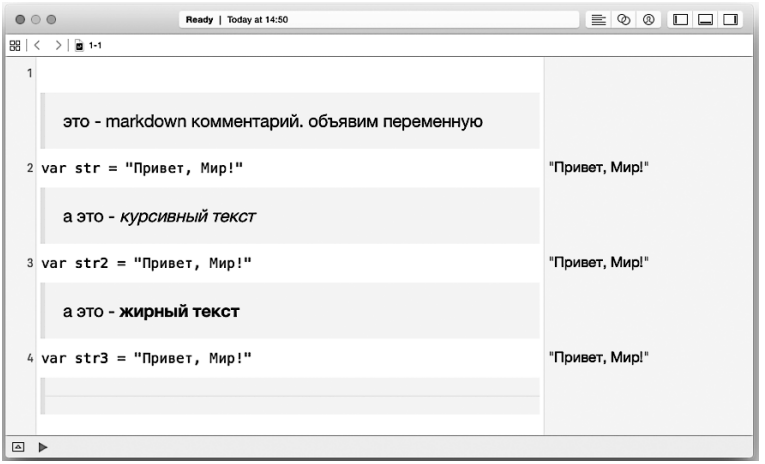


Рис. 3.5. Отформатированные markdown-комментарии

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

Обращу ваше внимание на то, что данный способ форматирования комментариев гарантированно возможен лишь в Xcode. Если вы раз-

рабатываете в ОС Linux, то данная функция определяется возможностями выбранного вами редактора кода.

Рассмотренные выше комментарии предназначены собственно для разработчика программного кода, т.е. он пишет их в большинстве случаев для себя, чтобы не запутаться, когда в код потребуется внести некоторые изменения.

Справочные комментарии

Помимо стандартных комментариев, которые вы могли видеть в любом языке программирования, Swift позволяет интегрировать комментарии в справочную систему. Ранее мы уже встречались со справочной системой, когда, удерживая клавишу Alt, щелкали по объекту в области кода.

Такой вид комментариев просто необходим, когда вы разрабатываете библиотеку функций (о них мы поговорим позже).

Справочные комментарии пишутся в виде многострочного комментария, но с использованием ключевых слов (листинг 3.12).

Листинг 3.12

```
/**
 This func say hello to user
 - parameter name:String Name of user
 - returns: Absolutely nothing
 - throws: Error when name is array.
 - authors: Bilbo Baggins
 - bug: This is very simple function
 */
func sayHello(name: String) {
    print("hello, \(name)!")
}
sayHello("Frodo")
```

В данном примере мы написали справочный комментарий для функции `sayHello(name:)`. При щелчке мышкой на имени этой функции при зажатой Alt отобразится справочное окно (рис. 3.6).

Далее описываются основные ключевые слова, которые вы можете использовать при написании справочного комментария:

`parameter <name>:<description>`

Указывает на параметр (аргумент), который необходимо передать в описываемый объект. После ключевого слова указывается имя параметра (без скобок), и через двоеточие приводится его описание.

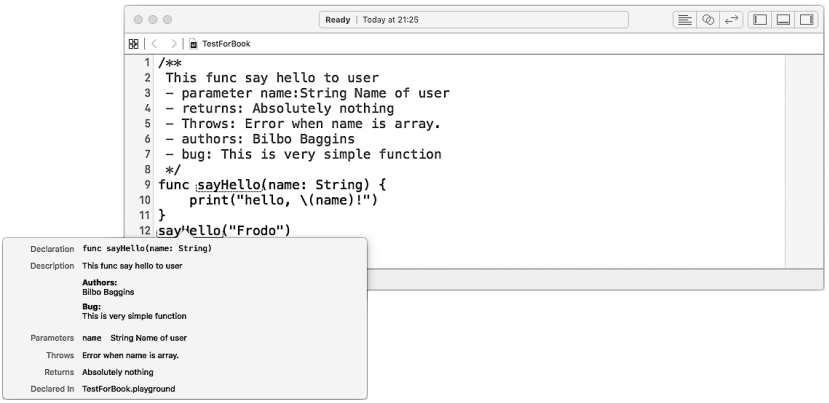


Рис. 3.6. Справочное окно

`returns: <description>`

Описывает возвращаемое объектом значение.

`throws: <description>`

Описывает возникающие в процессе использования объекта ошибки и их причины.

`author: <description>`

Описывает автора объекта.

`bug: <description>`

Описывает баги и недоработки объекта.

Обратите внимание, что перед каждым ключевым словом ставится символ «тире».

ПРИМЕЧАНИЕ С большой долей вероятности в настоящее время для вас остается много непонятного среди описанного материала. В скором времени вы подробно изучите функции, аргументы, объекты и много-многое другое, после чего прикладное применение справочных комментариев заиграет новыми красками.

Помимо описанных комментариев существует и слегка отличающийся синтаксис написания справочных комментариев, а также другие ключевые слова. Подробное изучение данного материала выходит за рамки книги, вы сможете самостоятельно найти интересующую вас

информацию в Интернете, в частности по адресу http://nshipster.com/swift-documentation/.

3.6. Точка с запятой

Если вы имеете за плечами какой-либо опыт программирования, то, возможно, привыкли завершать каждую строку кода символом точки с запятой (;). В таком случае вы, вероятно, заметили, что ни в одном из предыдущих листингов данный символ не используется. В отличие от многих других языков программирования, Swift не требует ставить точку с запятой в конце завершенного выражения. Единственным исключением является ситуация, когда в одну строку вы пишете сразу несколько команд. Пример приведен в листинге 3.13.

Листинг 3.13

```
1  // одно выражение в строке - точка с запятой не нужна
2  var number = 18
3  // несколько выражений в строке - точка с запятой нужна
4  number = 55; print(number)
```

Синтаксис Swift крайне дружелюбен. Этот замечательный язык программирования не требует от вас писать лишние символы, давая при это широкие возможности для того, чтобы код был понятным и прозрачным. В последующих главах вы увидите еще много необычного и интересного — того, что не встречали в других языках программирования.

4 Типы данных и операции с ними

Любая переменная или константа — это хранилище данных в памяти компьютера, в котором находится конкретное значение. Каждое хранилище может содержать данные определенного вида (типа). Например, знакомый нам «Сундук дракона» может содержать различные твердые предметы (значения), а в «Котле тролля» могут быть лишь жидкости. Такое разделение логично, поскольку позволяет понять, что именно мы можем делать с предметом в хранилище (его значением): предметы можно расплавить, воду вскипятить, числа сложить, строки объединить между собой.

Каждая переменная или константа может содержать в себе значение определенного типа, будь это целое число, дробное число, строка, отдельный символ, логическое значение или объект. В Swift, по аналогии с C, Objective-C, а также другими языками программирования, есть ряд предустановленных (как их называет Apple, фундаментальных) типов данных.

ПРИМЕЧАНИЕ Напомню, что тип данных переменной или константы определяет тип хранящихся в ней данных.

4.1. Виды определения типа данных

При инициализации переменной или константы необходимо указать ее тип данных. Swift — очень умный язык программирования, чаще всего он определяет тип инициализируемой переменной или константы автоматически. Рассмотрим пример из листинга 4.1.

Листинг 4.1

```
1  // объявим переменную и присвоим ей значение
2  var dragonsBox = "Мечи рыцарей"
```

Данный код уже встречался нам ранее. Напомню, что его можно прочитать вслух следующим образом:

> Объявить переменную с именем `dragonBox` и присвоить ей значение `"Мечи рыцарей"`.

При этом дополнительно не требуется указывать тип создаваемой переменной: Swift анализирует инициализируемое значение и принимает решение о типе переменной самостоятельно. Теперь, зная о том, что существуют типы данных, мы можем прочитать код листинга 4.1 следующим образом:

> Объявить переменную *строкового типа* с именем `dragonBox` и присвоить ей *строковое* значение `"Мечи рыцарей"`.

Или еще вариант:

> Объявить переменную *типа* `String` с именем `dragonBox` и присвоить ей *строковое* значение `"Мечи рыцарей"`.

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

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

При *явном* (непосредственном) *определении типа* переменной или константы после ее имени через двоеточие следует указать требуемый тип. При этом значение можно не передавать, тогда будет объявлена пустая переменная, то есть переменная, содержащая пустое значение. Примеры приведены в листинге 4.2.

Листинг 4.2

```
// объявляем пустую переменную dragonsBox
var dragonsBox: String
// создаем непустую переменную trollsPot с неявным определением типа
var trollsPot = "Желтая похлебка"
// создаем непустую переменную dragonsName с явным определением типа
var dragonsName: String = "Дракон Драконыч"
```

Все три переменные имеют строковый тип данных `String`, то есть могут хранить в себе строковые значения. Переменная `dragonsBox` — пустая, а переменные `trollsPot` и `dragonsName` хранят в себе текст.

У любой объявленной переменной или константы должен быть определен тип. Это значит, что вы должны либо инициализировать зна-

чение, либо явно указать тип данных. Игнорирование этого правила ведет к ошибке (листинг 4.3).

Листинг 4.3

```
1  var newDragonsBox // ОШИБКА: не указан тип переменной
```

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

Все фундаментальные типы данных — это так называемые *типы-значения*, или *значимые типы* (value type). Помимо типов-значений в Swift существуют *типы-ссылки*, или *ссылочные типы* (reference type). С примерами типов-ссылок мы познакомимся в последующих главах.

При передаче значения переменной или константы значимого типа в другую переменную или константу происходит копирование этого значения, а при передаче значения ссылочного типа передается ссылка на область в памяти, в которой хранится это значение.

Представьте себе, что «Котел тролля» хранит значение значимого типа, а «Сундук дракона» — ссылочного.

Чтобы присвоить значение хранилища «Котел тролля» (напомню, оно равно «Желтая похлебка») новому хранилищу «Чашка тролля», необходимо просто-напросто перелить часть содержимого из котла в чашку. В результате у нас получатся две совершенно одинаковые независимые переменные (если, конечно, смотреть не на количество, а только на тип и наименование содержимого), значения которых будут находиться отдельно друг от друга.

А вот с сундуком такой трюк не пройдет. При попытке скопировать значение в хранилище «Новый сундук дракона» мы не сможем взять половину золота и переложить его, ведь похищенное золото — это не просто безликая однородная золотая масса, как желтая похлебка, а монеты различного вида и достоинства. Единственный выход — положить в новый сундук записку, на которой будет написано, что похищенное золото хранится в старом сундуке.

В результате при попытке получить или изменить значение «Похищенное золото» через новый сундук мы по ссылке будем попадать в «Сундук дракона».

В будущем вы будете регулярно сталкиваться с обоими этими типами данных.

4.2. Числовые типы данных

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

Целочисленные типы данных

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

Числовой литерал — это фиксированная последовательность цифр, начинающаяся либо с цифры, либо с префиксного оператора минус/плюс.

Для объявления переменной или константы целочисленного типа необходимо использовать ключевые слова `Int` (для знаковых) и `UInt` (для беззнаковых). Пример инициализации целочисленных переменных приведен в листинге 4.4.

Листинг 4.4

```
1  // объявим переменную знакового целочисленного
      типа данных
2  var signedNum: Int
3  // и присвоим ей значение
4  signedNum = -32                                  -32
5  /* объявим переменную беззнакового
6   целочисленного типа данных
7   и сразу же проинициализируем
8   ее значением */
9  var unsignedNum: UInt = 128                      128
```

ПРИМЕЧАНИЕ Разница между знаковыми и беззнаковыми типами заключается в том, что значение знакового типа данных может быть в интервале от $-N$ до $+N$, а беззнакового — от 0 до $+2N$, где N определяется разрядностью операционной системы.

Также стоит отметить, что в Swift есть дополнительные целочисленные типы данных: `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`

и UInt64. Они определяют диапазон хранимых значений: 8-, 16-, 32- и 64-битные числа.

ПРИМЕЧАНИЕ Все приведенные целочисленные типы данных — это разные типы данных (как типы «Жидкость» и «Твердый предмет»), и значения в этих типах не могут взаимодействовать между собой напрямую. Все операции должны происходить между значениями одного и того же типа данных!

В самом начале книги говорилось об одной из особенностей Swift: всё в этом языке программирования является объектами. *Объект* — это некоторая абстрактная сущность, реализованная с помощью программы. Например, объектом является цифра 2 или продуктовый автомат. Каждая сущность может обладать набором характеристик (называемых свойствами) и запрограммированных действий (называемых методами). Каждое свойство и каждый метод имеют имя, позволяющее получить к нему доступ. Так, например, у объекта «продуктовый автомат» могут существовать свойство «вместимость» и метод «самоуничтожиться».

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

```
Продуктовый_автомат.вместимость = 490
Продуктовый_автомат.самоуничтожиться()
```

ПРИМЕЧАНИЕ О том, почему и для чего после имени метода добавляются скобки, мы поговорим позже.

Так как «всё — это объект», то и любой числовой тип данных также является объектом. Каждый из числовых типов данных (а их много) описывает сущность «целое число», но при этом свойства «максимально возможное число» и «минимально возможное число» каждого типа отличаются. В своей программе вы можете получить доступ к данным характеристикам через свойства min и max. Как говорилось ранее, эти значения зависят от разрядности системы, на которой исполняется приложение (листинг 4.5).

Листинг 4.5

```
1  // минимальное значение параметра типа Int8
2  var minInt8 = Int8.min                              -128
3  // максимальное значение параметра типа Int8
4  var maxInt8 = Int8.max                              127
5  // минимальное значение параметра типа UInt8
```

```
6  var minUInt8 = UInt8.min                          0
7  // максимальное значение параметра типа UInt8
8  var maxUInt8 = UInt8.max                          255
```

Рассматриваемые приемы относятся к объектно-ориентированному программированию (ООП), с которым вы, возможно, встречались в других языках. Подробнее объекты рассмотрены в последней части книги, в которой вы и научитесь использовать всю мощь ООП.

ПРИМЕЧАНИЕ Запомните, что Apple рекомендует использовать только типы данных Int и UInt, но для тренировки мы поработаем и с остальными типами.

Даже такой простой тип данных, как целые числа, в Swift наделен широкими возможностями. В дальнейшем вы узнаете о других механизмах, позволяющих обрабатывать различные числовые значения. А теперь пришло время выполнить самостоятельную работу.

ПРИМЕЧАНИЕ Для каждого задания создавайте новый playground-проект и сохраняйте его в специально отведенной папке на своем компьютере.

Задание

1. Объявите две пустые целочисленные переменные типов `Int8` и `Uint8`.
2. В одну из них запишите максимальное значение, которое может принять параметр типа `Uint8`, в другую — минимальное значение, которое может принять параметр типа `Int8`. Обратите внимание на то, какое значение в какую переменную может быть записано.
3. Выведите полученные значения на консоль.
4. Объявите две целочисленные однотипные переменные, при этом тип данных первой должен быть задан неявно, а второй — явно. Обеим переменным должны быть присвоены значения.
5. Поменяйте значения переменных местами.
6. Для этого вам придется использовать еще одну переменную, которая будет служить буфером.
7. Выведите получившиеся значения на консоль. При этом в каждом варианте выводимых данных текстом напишите, какую переменную вы выводите.

Числа с плавающей точкой

Вторым видом чисел, с которыми может работать Swift, являются числа с плавающей точкой, то есть числа, у которых присутствует дробная часть. Примерами могут служить числа 3.14 и -192.884022.

В данном случае разнообразие типов данных, способных хранить дробные числа, не такое большое, как в случае с целочисленными типами. Для объявления параметров, которые могут хранить числа с плавающей точкой, используются два ключевых слова: `Float` и `Double`. Оба типа данных являются знаковыми.

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

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

Пример объявления параметров, содержащих такие числа, приведен в листинге 4.6.

Листинг 4.6

```
// дробное число типа Float с явным указанием типа
var numFloat: Float = 104.3                          104,3
// пустая константа типа Double
let numDouble: Double
// дробное число типа Float с неявным указанием типа
var someNumber = 8.36                                8,36
```

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

ПРИМЕЧАНИЕ Значения типа дробных чисел не могут начинаться с десятичной точки. Я обращаю на это внимание, потому что вы могли видеть подобный подход в других языках программирования.

Задание

1. Объявите три параметра. Первый из них должен быть константой типа `Float` с произвольным числовым значением, второй — пустой константой типа `Float`, третий — пустой переменной типа `Double`.
2. Установите новое произвольное значение всем параметрам, для которых эта операция возможна.

Арифметические операторы

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

+ Бинарный оператор сложения складывает первый и второй операнды и возвращает результат операции (`a + b`). Тип результирующего значения соответствует типу операндов.
+ 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.
- Бинарный оператор вычитания вычитает второй операнд из первого и возвращает результат операции (`a - b`). Тип результирующего значения соответствует типу операндов.
- Унарный оператор «минус» используется в качестве префикса, то есть ставится перед операндом (`-a`). Инвертирует операнд и возвращает его новое значение.

ПРИМЕЧАНИЕ Символы «минус» и «плюс» используются как имена для двух операторов каждый. Данная практика должна быть знакома вам еще с занятий по математике.

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

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

% Бинарный оператор остатка от деления делит первый операнд на второй и возвращает остаток от деления. Или, другими словами, определяет, как много значений второго операнда поместится в первом, и возвращает значение, которое осталось, — оно называется остатком от деления (`a % b`). Тип результирующего значения соответствует типу операндов.

ПРИМЕЧАНИЕ В Swift 2.2 был изменен подход к использованию некоторых операторов. Например, довольно часто используемые в программировании операторы инкремента (++) и декремента (--) получили статус «устаревшие» (deprecated). Это говорит о том, что данные конструкции допустимы к использованию, но будут удалены в одной из следующих версий языка.

Тем не менее, так как данные операторы все еще не удалены, я опишу их, но их использование в дальнейшем коде будет исключено.

`++` Унарный оператор декремента увеличивает значение операнда на единицу, сохраняет его новое значение и возвращает результат операции (`++a` или `a++`). Возвращаемое значение зависит от того, расположен оператор перед операндом или после него.

`--` Унарный оператор инкремента уменьшает значение операнда на единицу, сохраняет новое значение и возвращает результат операции (`--a` или `a--`). Возвращаемое значение зависит от того, расположен оператор перед операндом или после него.

ПРИМЕЧАНИЕ О том, какие конструкции следует использовать вместо данных, будет сказано далее в книге.

Перечисленные операторы можно использовать для выполнения математических операций с любыми числовыми типами данных (целые или с плавающей точкой).

Чтобы продемонстрировать использование данных операторов, создадим две целочисленные переменные и две переменные типа `Double` (листинг 4.7).

Листинг 4.7

```
1  // целочисленные переменные
2  var numOne = 19                                  19
3  var numTwo = 4                                   4
4  // переменные типа числа с плавающей точкой
5  var numThree = 3.13                              3.13
6  var numFour = 1.1                                1.1
```

Для первых двух переменных неявно задан целочисленный тип данных `Int`, для вторых двух неявно задан тип `Double`.

Чтобы выполнить арифметические операции, достаточно использовать необходимые оператор и операнды. Рассмотрим пример в листинге 4.8.

Листинг 4.8

```
// целочисленные переменные
var numOne = 19
var numTwo = 4
// операция сложения
var sum = numOne + numTwo                    23
// операция вычитания
var diff = numOne - numTwo                   15
// операция умножения
var mult = numOne * numTwo                   76
// операция деления
var qo = numOne / numTwo                     4
```

Каждый из операторов производит назначенную ему операцию над переданными ему операндами. Вероятно, у вас возник вопрос относительно результата операции деления. Подумайте: каким образом при делении переменной `numOne`, равной 19, на переменную `numTwo`, равную 4, могло получиться 4? Ведь при умножении значения 4 на `numTwo` 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 Арифметические 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 4.9).

Листинг 4.9

```
// переменные типа числа с плавающей точкой
var numThree = 3.13
var numFour = 1.1
// операция сложения
var sumD = numThree + numFour                4,23
// операция вычитания
var diffD = numThree - numFour               2,03
// операция умножения
var multD = numThree * numFour               3,443
// операция деления
var qoD = numThree / numFour                 2,84545454545455
```

Результатом каждой операции является значение типа `Double`.

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

В предыдущем листинге не была представлена операция целочисленного деления. Рассмотрим ее отдельно (листинг 4.10).

Листинг 4.10

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

В данном примере рассматриваются три варианта операции целочисленного деления, отличающихся знаками операндов. Аналогом первой приведенной операции является следующее выражение:

```
numOne – (numTwo * 4) = 3
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`.

4				4				4				4				3		
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Таким образом, остаток от деления всегда меньше делителя.

В данном примере используется также оператор унарного минуса. Обратите внимание, что знак результата операции равен знаку делимого, то есть когда делимое меньше нуля, результат также будет меньше нуля.

Приведение числовых типов данных

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

Данная ситуация не осталась вне поля зрения разработчиков Swift, и они разработали специальный механизм, позволяющий преобразовывать одни числовые типы данных в другие. Данный механизм выполнен в виде глобальных функций.

ПРИМЕЧАНИЕ Справедливости ради стоит отметить, что на самом деле глобальные функции являются методами типов данных. Ранее мы говорили, что числовые типы данных — это объекты, и у них существуют запрограммированные действия, называемые методами.

У каждого объекта 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 инициализатор срабатывает.

Инициализатор имеет собственное фиксированное обозначение — init(), и для типа данных Double он может быть вызван следующим образом:

```
Double.init(_:)
```

В результате вызова данного метода будет создан новый объект, описывающий сущность «дробное число». Но Swift позволяет не писать имя инициализатора при создании объекта, а использовать сокращенный синтаксис:

```
Double(_:)
```

В результате выполнения данного кода также будет создан объект, описывающий «дробное число».

Далее в книге мы очень подробно разберем, что такое инициализаторы и для чего они нужны.

Названия вызываемых функций в Swift, с помощью которых можно преобразовать типы данных, соответствуют названиям типов данных:

`Int(_:)`
: Преобразование в тип данных `Int`.

`Double(_:)`
: Преобразование в тип данных `Double`.

`Float(_:)`
: Преобразование в тип данных `Float`.

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

Для применения данных функций в скобках после названия требуется передать преобразуемый элемент (переменную, константу, число). Рассмотрим пример, в котором требуется перемножить два числа: целое и дробное (листинг 4.11).

Листинг 4.11

```
1  // переменная типа Int
2  var numInt = 19                              19
3  //переменная типа Double
4  var numDouble = 3.13                         3,13
5  // операция перемножения чисел
6  var resD = Double(numInt) * numDouble        59,47
7  var resI = numInt * Int(numDouble)           57
```

Есть два способа перемножить данные числа:

- преобразовать число типа `Double` в `Int` и перемножить два целых числа;
- преобразовать число типа `Int` в `Double` и перемножить два дробных числа.

По выводу в области результатов видно, что переменная `resD` имеет более точное значение, чем переменная `resI`. Это говорит о том, что вариант, преобразующий целое число в дробное с помощью функции `Double(_:)`, точнее, чем использование функции `Int(_:)` для переменной типа `Double`.

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

Составной оператор присваивания

Swift позволяет максимально сократить объем кода. И чем глубже вы будете постигать этот замечательный язык, тем больше приемов, способных облегчить вам жизнь, узнаете. Одним из таких приемов является совмещение оператора арифметической операции (`+`, `-`, `*`, `/`, `%`) и оператора присваивания (`=`). Рассмотрим пример в листинге 4.12, в котором создадим целочисленную переменную и с помощью составного оператора присваивания будем изменять ее значение, используя минимум кода.

Листинг 4.12

```
1 // переменная типа Int
2 var someNumInt = 19                          19
3 // прибавим к ней произвольное число
4 someNumInt += 5                              24
5 /* эта операция аналогична выражению
6  someNumInt = someNumInt+5 */
7 // выведем результат операции
8 someNumInt                                   24
```

```
// умножим его на произвольное число
someNumInt *= 3                                    72
/* эта операция аналогична выражению
 someNumInt = someNumInt*3 */
// выведем результат операции
someNumInt                                         72
// вычтем из него произвольное число
someNumInt -= 3                                    69
/* эта операция аналогична выражению
someNumInt = someNumInt-3 */
// выведем результат операции
someNumInt                                         69
// найдем остаток от деления
someNumInt %= 8                                    5
/* эта операция аналогична выражению
someNumInt = someNumInt%8 */
// выведем результат операции
someNumInt                                         5
```

Для использования составного оператора присваивания необходимо всего-навсего после оператора арифметической операции без пробелов написать оператор присваивания. Результат операции моментально записывается в переменную, находящуюся слева от составного оператора, — это видно по дублирующимся записям в области результатов.

На данный момент мы уже знаем два различных способа увеличения значения переменной на единицу (листинг 4.13).

Листинг 4.13

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

Каждое последующее действие увеличивает переменную `someNumInt` ровно на единицу.

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

Пришло время попрактиковаться в работе с числовыми типами данных. Переходите к выполнению задания.

Задание

1. Объявите три пустые константы: одну типа `Int`, одну типа `Float` и одну типа `Double`. Сделайте это в одной строке.
2. Проинициализируйте для них следующие значения: `Int` — 18, `Float` — 16.4, `Double` — 5.7. Сделайте это в одной строке.
3. Найдите сумму всех трех констант и запишите ее в переменную типа `Float`.
4. Найдите произведение всех трех констант и запишите его в переменную типа `Int`. Помните, что вам необходимо получить результат с минимальной погрешностью.
5. Найдите остаток от деления константы типа `Float` на константу типа `Double` и запишите ее в переменную типа `Double`.
6. Выведите на консоль все три результата с использованием поясняющего текста.

Способы записи числовых значений

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

Swift при разработке программ позволяет задействовать самые популярные системы счисления:

- **Десятичная.** Числа записываются без использования префикса в привычном и понятном для нас виде.
- **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.

Целые числа могут быть записаны в любой из приведенных систем счисления. В листинге 4.14 показан пример записи числа 17 в различных видах.

Листинг 4.14

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

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

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

Помимо этого существует возможность записи экспоненты. Для этого используется символ `e` для десятичных чисел и символ `p` для шестнадцатеричных.

Для десятичных чисел экспонента указывает на степень десятки:

1.25e2 соответствует $1.25 * 10^2$, или 125.0.

Для шестнадцатеричных чисел экспонента указывает на степень двойки:

0xFp-2 соответствует $15 * 2^{-2}$, или 3.75.

В листинге 4.15 приведен пример записи десятичного числа 12.1875 в различных системах счисления и с использованием экспоненты.

Листинг 4.15

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

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

При записи числовых значений можно использовать символ нижнего подчеркивания (андерскор) для визуального отделения порядков числа (листинг 4.16).

Листинг 4.16

```
1  var number = 1_000_000                    1 000 000
```

В результате мы имеем все то же число, что и без символа нижнего подчеркивания, только записанное в более читабельной форме.

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

4.3. Текстовые типы данных

Было бы неправильно предположить, что Swift ограничивает разработчика лишь числовыми типами данных. Подумайте сами: о какой работе с пользователем можно было бы говорить, если бы программы не умели хранить текстовую информацию?

Инициализация строковых значений

В Swift существует два типа данных, предназначенных для хранения текстовой информации:

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

Оба типа обеспечивают быструю и корректную работу с текстом в вашем коде, при этом имеется полная поддержка символов Unicode. Синтаксис, как и при работе с другими типами данных, очень простой и читабельный. Для хранения информации в текстовых типах данных используются строковые литералы.

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

В первую очередь рассмотрим тип `Character`. Этот тип данных позволяет хранить в переменной или константе текстовый литерал длиной в один символ (листинг 4.17).

Листинг 4.17

```
1  var char: Character = "a"          "a"
2  char                               "a"
```

В переменной `char` хранится только один символ.

При попытке записать в параметр типа `Character` два и более символа Xcode сообщит об ошибке несоответствия типов записываемого значения и переменной. При попытке передать строковый литерал длиной более одного символа Swift рассматривает его в качестве значения типа данных `String` и не может записать его в переменную типа `Character`.

Получается, что тип данных `String` — это упорядоченный набор символов. Он позволяет хранить в переменной строку произвольной длины.

ПРИМЕЧАНИЕ В Swift версии 2 изменился принцип хранения информации в типе данных String. Раньше это было, по сути, множество значений типа Character.

Теперь же String — это совершенно новый тип данных, обладающий новыми свойствами и возможностями, позволяющими обращаться к хранящемуся в параметре строковому литералу как к множеству. Данное изменение позволило значительно расширить функционал типа String.

Рассмотрим пример работы с типом данных `String` (листинг 4.18).

Листинг 4.18

```
1  // переменная типа String
2  var stringOne = "Dragon"                    "Dragon"
```

При объявлении переменной мы передаем ей строковый литерал, тем самым неявно задавая тип данных `String`.

ПРИМЕЧАНИЕ Не забывайте, что строки, присвоенные переменной, могут быть изменены, а присвоенные константе — нет.

Пустые строковые литералы

Пустая строка также является строковым литералом. Вы можете передать ее в качестве значения параметру текстового типа данных (листинг 4.19).

Листинг 4.19

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

Обе строки в результате будут иметь идентичное (пустое) значение. Напомню, что инициализатор — это специальный метод, встроенный в тип данных `String`.

При явном указании типа `String` без инициализации значения в переменной или в константе оказывается пустое значение, а не пустой строковый литерал. Прежде чем использовать параметр, объявленный таким путем, требуется инициализировать его значение (листинг 4.20).

Листинг 4.20

```
1  // объявление переменной без указания значения
2  var str: String
3  // указание значения строковой переменной
4  str = "Hello, Troll!"                                    "Hello, Troll!"
5  str                                                      "Hello, Troll!"
```

Переменная `str` должна получить некоторое значение, прежде чем к ней можно будет обратиться. В данном примере вначале объявляется переменная, а уже потом инициализируется ее значение.

Приведение к строковому типу данных

В Swift методу инициализации `String(_:)` можно передать произвольное значение, чтобы присвоить его объявляемому параметру (листинг 4.21).

Листинг 4.21

```
1  // инициализация текстового значения
2  var notEmptyString = String("Hello, Troll!")   "Hello, Troll!"
```

В результате в переменной `notEmptyString` сохраняется текстовое значение `"Hello, Troll!"`.

Инициализатор `String(_:)` может получать на входе не только текстовое значение, но и переменную произвольного типа данных (листинг 4.22).

Листинг 4.22

```
1  // переменная типа Double
2  var numDouble = 74.22                                    74,22
3  // строка, созданная на основе переменной типа Double
4  var numString = String(numDouble)                        "74.22"
```

Функция `String(_:)` преобразует переданное в нее значение в тип `String`. С подобным механизмом мы уже встречались, когда рассматривали функции `Float(_:)`, `Double(_:)` и `Int(_:)`.

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 4.23).

Листинг 4.23

```
// переменная типа String
var name = "Дракон"                                  "Дракон"
// константа типа String с использованием
   интерполяции
let hello = "Привет, меня зовут \(name)!"            "Привет, меня зовут
                                                     Дракон!"
// интерполяция с использованием выражения
var meters: Double = 10                              10
let text = "Моя длина \(meters * 3.28) фута"         "Моя длина 32.8 фута"
```

При инициализации значения константы `hello` используется значение переменной `name` в рамках уже известной нам конструкции. Такой подход вполне допустим в Swift. Подобную конструкцию мы видели ранее при выводе информации на консоль с помощью функции `print(_:)`.

При интерполяции можно использовать выражения, в том числе арифметические операции, что и показано в примере.

При конкатенации происходит объединение различных строковых значений в одно с помощью символа арифметической операции сложения (листинг 4.24).

Листинг 4.24

```
// константа типа String
let firstText = "Мой вес "                           "Мой вес "
// переменная типа Double
var weight = 12.4                                    12,4
// константа типа String
let secondText = " тонны"                            " тонны"
// конкатенация строк при инициализации
   значения новой переменной
var text = firstText + String(weight) +              "Мой вес 12.4 тонны"
secondText
```

В данном примере используется оператор сложения для объединения различных строковых значений в одно. Тип данных переменной `weight` не строковый, поэтому ее значение преобразуется с помощью функции `String(_:)`.

ПРИМЕЧАНИЕ Значения типа Character при конкатенации также должны преобразовываться в тип String.

Коллекция символов в строке

Тип данных `String` предоставляет возможность обратиться к коллекции входящих в него символов с помощью свойства `characters`. Как и любое свойство в Swift, оно пишется через точку после имени строкового параметра (листинг 4.25).

Листинг 4.25

```
// переменная типа String
var str = "Hello, Troll!"                    "Hello, Troll!"
// получаем коллекцию символов
var collection = str.characters              String.CharacterView
```

В результате в переменной `collection` будет содержаться множество всех символов.

Множество элементов в контексте Swift носит название коллекции. *Коллекция* — это отдельный тип данных, все возможности которого мы рассмотрим позже.

У коллекции как у объекта существуют свои свойства. Для примера получим количество символов в строке, используя созданную коллекцию `collection` (листинг 4.26).

Листинг 4.26

```
// количество символов в строке
collection.count                             13
```

Полученный результат соответствует количеству символов в строке `str`.

ПРИМЕЧАНИЕ Не забывайте использовать функцию автодополнения в Xcode. С ее помощью можно получить полный список свойств и методов, доступных при работе с объектами.

Задание

1. Объявите переменную типа `String` и запишите в нее произвольный строковый литерал.
2. Объявите константу типа `Character`, содержащую произвольный символ латинского алфавита.
3. Объявите две переменные типа `Int` и запишите в них произвольные значения.
4. Одним выражением объедините в строковый литерал переменную типа `String`, константу типа `Character` и сумму переменных типа `Int`, а результат запишите в новую константу.
5. Выведите данную константу на консоль.

4.4. Логические значения

Логический тип данных

Изучение фундаментальных типов данных не завершается на числовых и строковых типах. В Swift существует специальный логический тип данных, называемый `Bool` и способный хранить одно из двух значений: «истина» или «ложь». Значение «истина» обозначается как `true`, а «ложь» — как `false`. Объявим переменную и константу логического типа данных (листинг 4.27).

Листинг 4.27

```
// логическая переменная с неявно заданным типом
var isDragon = true                                    true
// логическая константа с явно заданным типом
let isTroll: Bool = false                              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`, который в зависимости от значения переданного ему параметра позволяет пускать выполнение программы по различным ветвям (листинг 4.28).

Листинг 4.28

```
1  // логическая переменная
2  var isDragon = true                              true
3  // конструкция условия
4  if isDragon {
5      print("Привет, Дракон!")                     "Привет, Дракон!\n"
6  }else{
7      print("Привет, Тролль!")
8  }
```

Консоль:

```
Привет, Дракон!
```

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

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

Логические операторы

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

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

Унарный оператор *логического НЕ* является префиксным и записывается символом «восклицания». Он возвращает инвертированное логическое значение операнда, то есть если операнд имел значение `true`, то вернется `false`, и наоборот. Для выражения `!a` данный оператор может быть прочитан как «не `a`» (листинг 4.29).

Листинг 4.29

```
1  var someBool = true                          true
2  // инвертируем значение
3  !someBool                                    false
```

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

Бинарный оператор *логического И* записывается в виде удвоенного символа 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 4.30).

Листинг 4.30

```
1  let firstBool = true, secondBool= true, thirdBool = false
2  // группируем различные условия
3  var one = firstBool && secondBool                          true
4  var two = firstBool && thirdBool                           false
5  var three firstBool && secondBool && thirdBool             false
```

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

Вы можете группировать произвольное количество операндов, используя необходимое количество операторов.

Бинарный оператор *логического ИЛИ* выглядит как удвоенный символ прямой черты и является инфиксным. Он возвращает `true`, когда хотя бы один из операндов имеет значение `true`. Если значения обоих операндов равны `false`, то возвращается значение `false` (листинг 4.31).

Листинг 4.31

```
1  let firstBool = true, secondBool= false, thirdBool = false
2  // группируем различные условия
3  let one = firstBool || secondBool                          true
4  let two = firstBool || thirdBool                           true
5  let three secondBool || thirdBool                          false
```

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

Различные логические операторы можно группировать между собой, создавая сложные логические структуры (листинг 4.32).

Листинг 4.32

```
1  let firstBool = true, secondBool= false, thirdBool = false
2  var resultBool = firstBool && secondBool || thirdBool          false
3  var resultAnotherBool = thirdBool || firstBool && firstBool    true
```

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

В некоторых случаях необходимо указать, в каком порядке следует проводить отработку логических операторов. В Swift для этого используются круглые скобки. Выражения, помещенные в круглые скобки, выполняются в первую очередь (листинг 4.33).

Листинг 4.33

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

В данном примере используются указатели приоритета выполнения операций — круглые скобки. Данные указатели известны вам еще с занятий по математике. Те выражения, которые находятся в скобках, выполняются в первую очередь.

Задание

1. Объявите две логические переменные. Значение первой должно быть равно `true`, второй — `false`.
2. Запишите в две константы результат использования их в качестве операндов для операторов логического И и ИЛИ.
3. Выведите на консоль значения обеих получившихся констант.
4. Вычислите результат следующих логических выражений. При этом постарайтесь не использовать Xcode:

```
( ( true && false ) || true )
true && false && true || ( true || false )
false || ( false || true ) && ( true && false )
```

4.5. Псевдонимы типов

Swift предоставляет возможность создания псевдонима для любого типа данных. *Псевдонимом типа* называется дополнительное имя, по которому будет происходить обращение к данному типу. Для этого используется оператор `typealias`. Псевдоним необходимо применять тогда, когда существующее имя типа неудобно использовать в контексте программы (листинг 4.34).

Листинг 4.34

```
// 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`.

У типа может быть произвольное количество псевдонимов. И все псевдонимы вместе с оригинальным названием типа можно использовать в программе (листинг 4.35).

Листинг 4.35

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

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

Созданный псевдоним обладает теми же самыми возможностями, что и сам тип данных. Однажды объявив его, вы сможете использовать данный псевдоним для доступа к свойствам и методам типа (листинг 4.36).

Листинг 4.36

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

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

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 от двух и более подтипов. С такими составными типами также можно работать через псевдонимы.

Задание

1. Объявите строковую константу и запишите в нее ваше имя.
2. Объявите переменную типа `Double` и запишите в нее ваш вес в килограммах.
3. Объявите переменную типа `Int` и запишите в нее ваш рост в сантиметрах.
4. Вычислите ваш индекс массы тела и запишите его в переменную. Формула для расчета ИМТ:

   ```
   ИМТ = вес[кг] / рост[м]2
   ```

4.6. Операторы сравнения

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

`==`

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

`!=`

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

`>`

Бинарный оператор «больше чем» (`a > b`) возвращает `true`, когда значение первого операнда больше значения второго операнда.

`<`

Бинарный оператор «меньше чем» (`a < b`) возвращает `true`, когда значение первого операнда меньше значения второго операнда.

`>=`

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

`<=`

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

Каждый из приведенных операторов возвращает значение, указывающее на справедливость некоторого утверждения. Несколько примеров и значений, которые они возвращают, приведены в листинге 4.37.

Листинг 4.37

```
// Утверждение "1 больше 2"
1 > 2                                          false
// вернет false, так как оно ложно
// Утверждение "2 не равно 2"
2 != 2                                         false
// вернет false, так как оно ложно
// Утверждение "1 плюс 1 меньше трех"
(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
```

К вопросу сферы использования операторов сравнения мы обратимся в дальнейшем.

На этом изучение базовых возможностей Swift заканчивается! Весь изученный материал будет активно использоваться в дальнейшем, начиная уже со следующего листинга.

Часть III
Основные средства Swift

Конечно же, возможности языка Swift не заканчиваются рассмотренным в предыдущей части материалом. Это было лишь начало. Весь материал, который был изучен, подготовил вас к следующему шагу: изучению основных средств данного языка программирования.

В этой части вы вновь познакомитесь с уже известными вам по другим языкам программирования конструкциям, а также изучите совершенно новые, порой просто поразительные средства Swift, которые разбудят в вас еще больший интерес!

5 Кортежи

Возможно, вы никогда не встречали в программировании такого понятия, как кортежи (tuples), тем не менее это одно из очень интересных функциональных средств, доступное в Swift.

5.1. Основные сведения о кортежах

Объявление кортежа

Кортеж — это особый объект, который группирует значения различных типов в пределах одного составного значения. Более того, кортеж предлагает наиболее простой способ объединения значений различных типов в пределах одного значения. У каждого отдельного значения в составе кортежа может быть собственный тип данных, который никак не зависит от других.

СИНТАКСИС

```
(значение_1, значение_2, ...)
```

Кортеж состоит из набора независимых значений, записываемых в круглых скобках и отделенных друг от друга запятой. Количество элементов в кортеже может быть произвольным.

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

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

Для записи кортежа в переменную необходимо использовать оператор var, а для записи в константу — оператор let.

Для примера: объявим константу, в которую входят три значения различных типов данных (листинг 5.1).

Листинг 5.1

```
1  /* объявляем кортеж, состоящий
2   из трех различных параметров,
3   у каждого из которых свой тип
4   данных */
5  let myProgramStatus = (200, "In Work", true) (.0 200, .1 "In Work",
                                                 .2 true)
6  myProgramStatus                              (.0 200, .1 "In Work",
                                                 .2 true)
```

В данном примере `myProgramStatus` — это константа, содержащая в качестве значения кортеж, который описывает статус работы некой программы и содержит три различных параметра:

- 200 — целое число типа `Int`;
- `"In work"` — строковый литерал типа `String`;
- `true` — логическое значение типа `Bool`.

У каждого из этих значений свой тип данных: `Int`, `String` и `Bool`. В результате данный кортеж группирует вместе значения трех различных типов данных, позволяя хранить их в пределах одной переменной или константы.

Тип данных кортежа

У вас мог возникнуть вопрос: если кортеж группирует значения различных типов данных в одно, то какой же тогда тип данных у самого кортежа?

Тип данных кортежа — это фиксированный упорядоченный набор типов данных его значений, который записывается в скобках и элементы которого отделяются запятыми друг от друга. Для кортежа из предыдущего листинга тип данных — это (`Int`, `String`, `Bool`), и в данном примере он задан неявно. Порядок указания типов данных должен соответствовать порядку следования элементов в кортеже, поэтому, например, тип (`Bool`, `String`, `Int`) является совершенно другим типом данных кортежа.

При объявлении кортежа ему можно явно задать требуемый тип. Тип кортежа указывается точно так же, как тип переменных и констант, — через двоеточие (листинг 5.2).

Листинг 5.2

```
1  /* объявляем кортеж с явно
2   заданным типом */
3  let myProgramStatus: (Int, String,
   Bool) = (200, "In Work", true)       (.0 200, .1 "In Work", .2 true)
4  myProgramStatus                      (.0 200, .1 "In Work", .2 true)
```

Представленный здесь кортеж в точности соответствует объявленному в предыдущем примере, но на этот раз его тип данных задан явно.

Вы можете создать кортеж из произвольного количества значений различных типов данных, и он будет содержать столько различных значений, сколько вам захочется. Более того, вы можете использовать один и тот же тип данных (или даже его псевдонимы) в рамках одного кортежа произвольное количество раз (листинг 5.3).

Листинг 5.3

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

В кортеже `numberTuple` сгруппированы четыре целочисленных значения, тип двух из них определен через псевдоним.

5.2. Взаимодействие с элементами кортежа

Кортеж предназначен не только для установки и хранения некоторого набора значений, но и для взаимодействия с этими значениями. Swift позволяет разделить кортеж на отдельные значения и присвоить их набору переменных или констант. Это делается для обеспечения привычного способа доступа к значениям с использованием отдельных переменных.

Инициализация значений через параметры

Переменные, которым присваиваются значения кортежа, записываются в скобках после оператора `var` или `let`, при этом количество переменных (констант) должно соответствовать количеству значений в кортеже (листинг 5.4).

Листинг 5.4

```
// объявляем кортеж
let myProgramStatus = (200, "In Work",  (.0 200, .1 "In Work",
true)                                   .2 true)
// записываем значения кортежа
в переменные
```

```
var (statusCode, statusText,
statusConnect) = myProgramStatus
// выводим информацию
print("Код ответа - \(statusCode)")     "Код ответа - 200\n"
print("Текст ответа - \(statusText)")   "Текст ответа - In Work\n"
print("Связь с сервером - \            "Связь с сервером - true\n"
(statusConnect)")
```

Консоль:

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

Если в качестве правой части оператора присваивания передать не имя параметра, хранящего кортеж, а кортеж в виде набора значений, то присвоить набору переменных или констант можно произвольные значения (листинг 5.5).

Листинг 5.5

```
/* объявляем сразу несколько
 переменных и устанавливаем
 для них значения */
var (myName, myAge) = ("Тролль", 140)
// выводим их значения
print("Меня зовут \(myName), и мне          "Меня зовут Тролль, и мне
    \(myAge) лет")                          140 лет\n"
```

Консоль:

```
Меня зовут Тролль, и мне 140 лет
```

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

При установке значений переменных вы можете игнорировать произвольные элементы кортежа. Для этого в качестве имени переменной, соответствующей элементу, который необходимо проигнорировать, указывается символ нижнего подчеркивания. Пример приведен в листинге 5.6.

Листинг 5.6

```
// объявляем кортеж
let myProgramStatus: (Int, String,        (.0 404, .1 "Error",
Bool) = (404, "Error", true)               .2 true)
/* получаем только необходимые
 значения кортежа */
var (statusCode, _, _) = myProgramStatus
// выводим информацию
print(" Код ответа - \(statusCode)")      "Код ответа - 404\n"
```

Консоль:

```
Код ответа – 404
```

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

ПРИМЕЧАНИЕ Символ нижнего подчеркивания в Swift означает игнорирование параметра. Вы можете использовать его практически в любой ситуации, когда необходимо опустить создание нового параметра. С примерами работы данного механизма вы познакомитесь в дальнейшем.

Индексы для доступа к элементам

Для доступа к значениям отдельных элементов кортежа необходимо использовать числовой индекс, указываемый через точку после имени кортежа (листинг 5.7).

Листинг 5.7

```
// объявляем кортеж
let myProgramStatus: (Int, String, Bool) =
(200, "In Work", true)                          (.0 200, .1 "In Work",
                                                .2 true)
// выводим информацию с использованием
индексов
print(" Код ответа - \(myProgramStatus.0)")     "Код ответа - 200\n"
print(" Текст ответа -                          "Текст ответа - In
\(myProgramStatus.1)")                          Work\n"
print(" Связь с сервером -                      "Связь с сервером -
\(myProgramStatus.2)")                          true\n"
```

Консоль:

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

Индексы указываемых элементов соответствуют порядковым номерам элементов в кортеже.

ВНИМАНИЕ Индекс первого элемента в кортежах всегда нулевой. При этом элементы всегда идут по порядку и последовательно. В кортеже из *N* элементов индекс первого элемента будет равен 0, а индекс последнего — *N* – 1.

Имена для доступа к элементам

Для каждого элемента кортежа можно задать не только значение, но и имя. Имя элемента указывается отдельно перед каждым элементом

через двоеточие. При этом задать имена для отдельных элементов невозможно: вы должны либо указать имена для всех элементов, либо не использовать их вовсе (листинг 5.8).

Листинг 5.8

```
let myProgramStatus = (statusCode: 200,      (.0 200, .1 "In Work",
statusText: "In Work", statusConnect: true)  .2 true)
```

Указанные имена элементов кортежа можно использовать при получении значений этих элементов. При этом применяется тот же синтаксис, что и при доступе через индексы. Присвоение имен значениям не лишает вас возможности использовать индексы. Индексы в кортеже можно задействовать всегда. В листинге 5.9 используется созданный ранее кортеж `myProgramStatus`.

Листинг 5.9

```
// выводим информацию с использованием
индексов
print(" Код ответа - \(myProgramStatus.      "Код ответа - 200\n"
statusCode)")
print(" Текст ответа - \(myProgramStatus.    "Текст ответа - In
statusText)")                                Work\n"
print(" Связь с сервером - \                 "Связь с сервером -
(myProgramStatus.2)")                        true\n"
```

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.

В предыдущем примере, объявляя кортеж `myProgramStatus`, мы задавали имена элементов прямо в значении кортежа, но их можно указывать не только при инициализации значения, но и в самом типе кортежа (листинг 5.10).

Листинг 5.10

```
/* объявляем кортеж с
 указанием имен элементов
 в описании типа */
let myProgramStatus: (statusCode: Int,          (.0 200, .1
statusText: String, statusConnect: Bool) =      "In Work", .2 true)
(200, "In Work", true)
```

```
/* выводим значение элемента
 кортежа с помощью имени
 этого элемента*/
myProgramStatus.statusCode                    200
/* объявляем кортеж с
 указанием имен элементов
 при инициализации их значений */
let myNewProgramStatus = (statusCode: 404,    (.0 404, .1 "Error",
statusText:"Error", statusConnect:true)       .2 true)
/* выводим значение элемента
 кортежа с помощью имени
 этого элемента*/
myNewProgramStatus.statusText                 "Error"
```

Изменение значений кортежей

Для однотипных кортежей можно производить операцию инициализации значения одного кортежа в другом (листинг 5.11).

Листинг 5.11

```
// объявляем пустой кортеж
var myFirstTuple: (Int, String)
// создаем кортеж со значением
var mySecondTuple = (100, "Код")              (.0 100, .1 "Код")
// передаем значение кортежа
myFirstTuple = mySecondTuple                  (.0 100, .1 "Код")
myFirstTuple                                  (.0 100, .1 "Код")
```

Кортежи `myFirstTuple` и `mySecondTuple` имеют один и тот же тип данных, поэтому мы можем присвоить значение одного кортежа другому. У первого тип задан явно, а у второго — через инициализируемое значение.

Помимо изменения значения кортежа целиком вы можете, используя индексы или имена, изменять значения отдельных элементов (листинг 5.12).

Листинг 5.12

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

Кортежи позволяют делать то, чего вам, возможно, недоставало в других языках программирования, — возвращать набор значений с удобным интерфейсом доступа к ним. Такое применение кортежи могут найти, например, в функциях.

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

Сравнение кортежей

При необходимости вы можете сравнивать кортежи между собой. Сравнение кортежей производится последовательным сравнением элементов кортежей: вначале сравниваются первые элементы обоих кортежей, если они идентичны, то производится сравнение следующих элементов, и так далее до тех пор, пока не будет обнаружены неидентичные элементы (листинг 5.13).

Листинг 5.13

```
(1, "alpha") < (2, "beta")                    true
// истина т.к. 1 меньше 2.
// вторая пара элементов не учитывается
(4, "beta") < (4, "gamma")                    true
// истина т.к. "beta" меньше "gamma".
(3.14, "pi") == (3.14, "pi")                  true
//истина, т.к. 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 реализовывать собственные механизмы. Данное ограничение Apple ввел не от лени: если ваш кортеж имеет большое количество элементов, то есть повод задуматься о том, чтобы заменить его структурой или классом. О них мы поговорим далее.

Задание

1. Создайте кортеж с тремя параметрами: ваш любимый фильм, ваше любимое число и ваше любимое блюдо. Все элементы кортежа должны быть именованы.

2. Одним выражением запишите каждый элемент кортежа в три константы.
3. Создайте второй кортеж, аналогичный первому по параметрам, но описывающий другого человека (с другими значениями).
4. Обменяйте значения в кортежах между собой (с использованием дополнительного промежуточного кортежа).
5. Создайте новый кортеж, элементами которого будут любимое число из первого кортежа, любимое число из второго кортежа и разница любимых чисел первого и второго кортежей.

Напоминаю, что решения для всех заданий вы можете найти по адресу http://swiftme.ru/solutions.

6 Опциональные типы данных

Опциональные типы данных — это потрясающее нововведение языка Swift, которое вы, вероятно, никогда не встречали в других языках программирования и которое значительно расширяет возможности работы с типами данных.

6.1. Опционалы

Опциональные типы данных, также называемые *опционалами*, — это особый тип данных, который говорит о том, что некоторая переменная или константа либо имеет значение определенного типа, либо вообще не имеет никакого значения.

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 нуль — вполне конкретное значение числового типа данных. В случае же отсутствия значения в опциональном типе данных имеет место полное отсутствие значения как такового.

Рассмотрим абстрактный пример. Представьте, что у вас есть бесконечная плоскость. На ней устанавливают точку с определенными координатами (x, y). В любой момент времени мы можем говорить об этой точке и получать информацию о ней. Теперь уберем данную точку с плоскости. Несмотря на это вы все еще можете говорить о данной точке, но получить информацию о ней нельзя, поскольку точка уже не существует на плоскости. В данном примере точка — это некоторый объект (переменная, константа и т. д.), а ее координаты — опциональный тип данных, они могут иметь определенное значение, а могут отсутствовать в принципе.

Теперь рассмотрим опционал на практике. Вспомните метод-инициализатор класса `Int`, обозначающийся как `Int(_:)`. Данный метод

предназначен для создания целочисленного значения или конвертации некоторого числового значения в целочисленное. Не каждый передаваемый литерал может быть преобразован в целочисленный тип данных: например, строку `"1945"` можно конвертировать и вернуть в виде целого числа, а вот строку `"Привет, Дракон!"` вернуть в виде числа не получится (листинг 6.1).

Листинг 6.1

```
// переменная с числовым строковым литералом
let possibleString = "1945"                                    "1945"
/* при попытке конвертации
 преобразуется в целочисленный тип */
let convertPossibleString = Int(possibleString)                1 945
// переменная со строковым литералом
let unpossibleString = "Привет, Дракон!"                       "Привет,
                                                               Дракон!"
/* при попытке конвертации
 не преобразуется в целочисленный тип */
let convertUnpossibleString = Int(unpossibleString)            nil
```

В результате своей работы функция `Int(_:)` возвращает опциональный тип данных, то есть такой тип данных, который в данном случае может либо содержать целое число (`Int`), либо не содержать совершенно ничего.

Опциональный тип данных обозначается с помощью постфикса в виде знака вопроса после имени основного типа данных (то есть типа данных, на котором основан опционал). Для примера из предыдущего листинга опциональный тип `Int` обозначается как `Int?`. Опционалы могут быть основаны на любом типе данных, включая `Bool`, `String`, `Float` и `Double`.

Для того чтобы сообщить Swift о том, что значение в некотором объекте отсутствует, используется ключевое слово `nil`, указываемое в качестве значения этого объекта. Если для переменной указан опциональный тип данных, то она в любой момент времени может принять либо значение основного типа данных, либо `nil`. Значение опционала-константы задается единожды (листинг 6.2).

Листинг 6.2

```
/* переменная с опциональным типом Int
 и с установленным значением */
var dragonAge: Int? = 230                          230
// уничтожаем значение переменной
dragonAge = nil                                    nil
```

Переменная `dragonAge` является переменной опционального типа данных. Изначально ей присваивается значение, соответствующее основному для опционала типу данных (типу `Int` в данном случае). Так как `dragonAge` — это переменная, то мы можем изменить ее значение в любой момент. В результате мы присваиваем ей `nil`, после чего `dragonAge` не содержит никакого значения.

ПРИМЕЧАНИЕ В Swift ключевое слово nil можно использовать только с переменными опционального типа данных. При этом, если вы объявите такую переменную, но не инициализируете ее значение, Swift по умолчанию считает ее равной nil.

Для того чтобы объявить параметр опционального типа данных, можно использовать функцию `Optional(_:)`, как показано в листинге 6.3.

Листинг 6.3

```
// опциональная переменная с установленным значением
var optionalVar = Optional ("stringValue")            "stringValue"
optionalVar                                           "stringValue"
// уничтожаем значение опциональной переменной
optionalVar = nil                                     nil
optionalVar                                           nil
```

Так как функции `Optional(_:)` в качестве входного аргумента передано значение типа `String`, то возвращаемое ею значение имеет опциональный строковый тип данных.

В качестве значения данной функции необходимо передавать значение того типа данных, который должен стать основным для создаваемого опционала.

6.2. Извлечение опционального значения

Для того чтобы со значениями, содержащимися в опционалах, можно было работать, их необходимо специальным образом извлекать.

Принудительное извлечение значения

Запомните, что опциональный тип данных — это совершенно новый тип данных: `Int?` и `Int`, `String?` и `String`, `Bool?` и `Bool` — все это разные типы данных. Поэтому несмотря на то, что опционалы могут принимать значения основных типов данных, остальные свойства

этих типов к опционалам не относятся. Например, тип `Int?` не может использоваться в качестве операнда при выполнении арифметических операций (листинг 6.4).

Листинг 6.4

```
/* опциональная переменная
 с установленным значением */
var trollAge: Int? = 95                          95
trollAge = trollAge + 10 // ОШИБКА
```

Swift предлагает механизм решения данной проблемы, который называется *принудительным извлечением опционального значения* (forced unwrapping). При этом с помощью специального оператора значение опционального типа данных преобразуется в значение основного (для этого опционала) типа данных, например `Int?` преобразуется в `Int`. Для принудительного извлечения используется знак восклицания в качестве постфикса названия параметра (переменной или константы), содержащего значение опционального типа.

Исправим код, приведенный в предыдущем примере, для корректного подсчета суммы целых чисел (листинг 6.5).

Листинг 6.5

```
/* опциональная переменная
 с установленным значением */
var trollAge: Int? = 95                          95
// проведение арифметической операции
trollAge = trollAge! + 10                        105
trollAge                                         105
```

Для того чтобы преобразовать опциональное значение переменной `trollAge` в значение типа `Int`, к имени переменной добавим оператор принудительного извлечения опционального значения (!). В итоге выражение `trollAge! + 10` будет складывать два однотипных числа, и результат можно записать в качестве значения опционального типа `Int` в переменную `trollAge`.

При принудительном извлечении значения вы должны гарантировать, что параметр с опциональным типом данных содержит какое-либо значение, а не равен `nil`. В противном случае будет иметь место попытка преобразовать в основой тип данных несуществующее значение, поэтому Xcode сообщит об ошибке.

Косвенное извлечение значения

В противовес принудительному извлечению опционального значения Swift поддерживает *косвенное извлечение опционального значения* (implicitly unwrapping).

Если при инициализации значения опционала вы уверены, что данный параметр гарантированно будет иметь значение и никогда не будет равен `nil`, имеет смысл отказаться от принудительного извлечения значения с помощью знака восклицания всякий раз, когда это значение требуется. Для этой цели используется косвенное извлечение опционального значения.

При косвенном извлечении в качестве постфикса к типу данных (при указании типа данных) необходимо указывать не знак вопроса, а знак восклицания (например, `Int!` вместо `Int?`). В листинге 6.6 показана разница в использовании принудительного и косвенного извлечения опционального значения.

Листинг 6.6

```
var type: String
// принудительное извлечение опционального значения
let monsterOneType: String? = "Дракон"                    "Дракон"
type = monsterOneType!                                    "Дракон"
type                                                      "Дракон"
// косвенное извлечение опционального значения
let monsterTwoType: String! = "Тролль"                    "Тролль"
type = monsterTwoType                                     "Тролль"
type                                                      "Тролль"
```

При попытке косвенно извлечь несуществующее (то есть равное `nil`) опциональное значение Xcode сообщит об ошибке (листинг 6.7).

Листинг 6.7

```
let pointCoordinates: (Int, Int)! = nil
coordinates = pointCoordinates // ОШИБКА
```

В качестве основного типа для опционала можно использовать любой тип данных. Так как кортеж представляет собой отдельный тип данных, соответствующий типу входящих в него элементов, то и его тип можно брать за основу опционала, что и продемонстрировано в предыдущем примере.

7 Утверждения

Swift позволяет прервать выполнение программы в случае, когда некоторое условие не выполняется, — для этого служит специальный механизм *утверждений* (assertions). Утверждения используются на этапе отладки программы.

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

Утверждение представляет собой специальную функцию `assert()`, производящую проверку некоторого условия на предмет его истинности. Первый и второй параметры определяют проверяемое условие и отладочное сообщение соответственно. При этом обязательным является лишь первый.

Можно сказать, что функция `assert(_:_:file:line:)` «утверждает», что переданное ей условие истинно. Если функция возвращает `true`, то выполнение программы продолжается; если же функция возвращает `false`, то выполнение программы завершается. При этом если вы тестируете программу в среде разработки Xcode, отладочное сообщение выводится в консоли, а сама среда Xcode сообщает о возникшей ошибке. Другими словами, механизм действия утверждения может быть интерпретирован следующим образом:

> Если переданное условие выполняется, то продолжить выполнение программы, в ином случае прервать ее и вывести переданное отладочное сообщение.

Рассмотрим пример использования утверждения (листинг 7.1).

Листинг 7.1

```
1  /* переменная целочисленного типа
2   с установленным значением */
3  var dragonAge = 230                                    230
```

```
// проверка корректности условия
assert( dragonAge >= 225, "Ваш дракон
        слишком молод, чтобы продолжить" )
// выводим значение переменной
dragonAge                                    230
```

В утверждение передается условие `dragonAge >= 225`. Так как переменная `dragonAge` равна 230, то данное условие истинно и прерывания программы не произойдет.

Если изменить переданное в предыдущем листинге условие на `dragonAge >= 235`, то утверждение остановит выполнение программы и на консоли появится соответствующее сообщение с указанием строки, где произошла остановка выполнения (рис. 7.1).

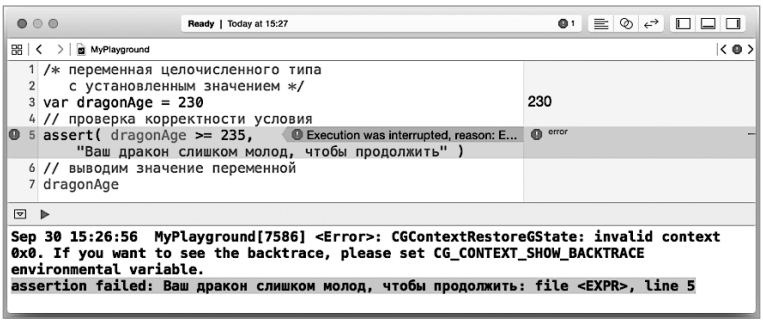


Рис. 7.1. Ошибка при невыполнении переданного в утверждение условия

Вы можете не передавать сообщение в качестве входного параметра в утверждение, ограничившись лишь условием (листинг 7.2).

Листинг 7.2

```
/* переменная целочисленного типа
 с установленным значением */
var dragonAge = 230                    230
// проверка корректности условия
assert( dragonAge >= 225 )
// выводим значение переменной
dragonAge                              230
```

Утверждения следует использовать, когда значение условия однозначно должно быть равно `true`, но есть вероятность, что оно вернет `false`. Таким образом, с помощью функции `assert(_:_:file:line)` вы

легко можете проверить, имеет ли некоторый опционал какое-либо значение. Для этого необходимо всего лишь сравнить (с помощью оператора сравнения) опционал и `nil` (листинг 7.3).

Листинг 7.3

```
1  // опционал с установленным значением
2  var isDragon: Bool? = true                          true
3  // проверка корректности условия
4  assert( isDragon != nil, "Персонаж отсутствует" )
```

Если бы переменная `isDragon` не имела значения (была равна `nil`), то на утверждении выполнение программы было бы прервано.

Недостатком данного подхода является то, что механизм утверждений позволяет лишь прервать выполнение программы и вывести отладочное сообщение. Выполнить произвольный код в зависимости от результата проверки он не позволяет.

8 Ветвления

Представьте себе реку: ее необузданный бурлящий поток, плавные повороты, резкие перепады и заводи. Человечество обладает технологиями, благодаря которым течением реки можно управлять: менять силу ее потока, создавать новые русла и т. д.

Поток в контексте программирования — это процесс выполнения программы. В Swift существуют специальные механизмы, позволяющие управлять этим процессом, например выполнять или, наоборот, игнорировать код в зависимости от установленных условий, а также многократно повторять определенные блоки кода.

Умение управлять ходом работы программы — очень важный аспект программирования на языке Swift, благодаря которому ваши программы смогут выполнять различные блоки кода в зависимости от возникающих условий. В данной главе рассмотрены механизмы, которые поддерживаются практически во всех языках программирования и без которых невозможно выполнение любой программы.

8.1. Оператор условия if

Синтаксис оператора if

Утверждения, с которыми вы познакомились в предыдущей главе, являются упрощенной формой оператора условия. Они анализируют переданное условие и позволяют либо продолжить выполнение программы, либо завершить его выводом отладочного сообщения. Условия, в отличие от утверждений, позволяют выполнять определенные блоки кода в зависимости от результата проверки некоторого условия. Порой данный механизм становится просто незаменимым.

СИНТАКСИС

```
if проверяемое_условие {
    // тело оператора
}
```

Оператор условия начинается с ключевого слова if, за которым следует проверяемое условие. Если проверяемое условие истинно, то выполняется код из тела оператора. В ином случае данный оператор игнорируется, и выполнение программы продолжается со следующего оператора.

Условие — это логическое выражение. Оно может принимать значение либо «истина», либо «ложь». Для хранения таких значений в Swift существует фундаментальный тип данных `Bool`, рассмотренный нами ранее. Получается, что, как и в утверждениях, условие, которое необходимо для работы оператора `if`, должно принимать значение типа `Bool`.

В простейшем виде пример использования оператора показан в листинге 8.1.

Листинг 8.1

```
1  // переменная типа Bool
2  var logicVar = true
3  // проверка значения переменной
4  if logicVar {
5      print("Переменная logicVar истинна")
6  }
```

Консоль:

```
Переменная logicVar истинна
```

В качестве условия для оператора `if` здесь используется логическая переменная `logicVar`. Так как ее значение истинно, код, находящийся в теле оператора, был выполнен, о чем свидетельствует сообщение на консоли.

Если изменить значение `logicVar` на `false`, то проверка не пройдет и функция `print(_:)` не сработает.

Не забывайте, что Swift — это язык со строгой типизацией. Любое условие должно возвращать либо `true`, либо `false`, и никак иначе. Поэтому если проверка возвращает значение другого типа, то Xcode сообщит об ошибке (листинг 8.2).

Листинг 8.2

```
1  var intVar = 1
2  if intVar { // ОШИБКА
3      // ...
4  }
```

Если вам необходимо выполнить блок кода при ложности переданного условия, то вы можете использовать оператор логического отрицания (листинг 8.3).

Листинг 8.3

```
// переменная типа Bool
var logicVar = false
// проверка значения переменной
if !logicVar {
    print("Переменная logicVar ложна")
}
```

Консоль:

```
Переменная logicVar ложна
```

С помощью логического отрицания значение переменной `logicVar` инвертируется, в результате оператор `if` обнаруживает, что проверяемое им условие истинно.

Используемый синтаксис записи оператора `if`, позволяющий указать блок выполняемого кода только для истинного результата проверки, называется *кратким*. Swift позволяет включить в рамки одного оператора условия блоки кода и для истинного, и для ложного результата проверки. Для этого необходимо использовать *стандартный синтаксис* оператора условия `if`.

СИНТАКСИС

```
if проверяемое_условие {
    // истинное тело оператора
} else {
    // ложное тело оператора
}
```

При истинности переданного условия выполняется код из «истинного» блока. Во всех остальных случаях выполняется «ложный» блок кода, который следует после ключевого слова else.

В листинге 8.4 приведен пример использования стандартного синтаксиса логического оператора `if`.

Листинг 8.4

```
// 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 результате мы предусмотрели оба результата проверяемого условия.

Ранее мы рассматривали логические операторы И и ИЛИ, позволяющие группировать значения. С их помощью вы можете объединять несколько условий (листинг 8.5).

Листинг 8.5

```
// переменные типа Bool
var firstLogicVar = true
var secondLogicVar = false
// проверка значения переменных
if firstLogicVar || secondLogicVar {
    print("Одна из переменных истинна")
} else {
    print("Обе переменные ложны")
}
```

Консоль:

```
Одна из переменных истинна
```

Так как оператор логического ИЛИ возвращает `true`, когда хотя бы один из операндов равен `true`, в данном случае оператор условия `if` выполняет первый блок кода.

Оператор условия `if` предлагает также расширенный синтаксис записи, который как бы объединяет несколько операторов `if`, позволяя проверять произвольное количество условий.

СИНТАКСИС

```
if проверяемое_условие1 {
    // код...
} else if проверяемое_условие2 {
    // код...
} else {
    // код...
}
```

Если первое проверяемое условие истинно, то выполняется блок кода первой ветви оператора условия. В противном случае происходит переход к проверке второго ус-

ловия. Если оно истинно, то выполняется код из второй ветви. В противном случае выполняется код блока else.

Количество операторов else if в рамках одного оператора условия может быть произвольным, а наличие оператора else не обязательно.

В листинге 8.6 продемонстрирован пример использования расширенного синтаксиса логического оператора условия.

Листинг 8.6

```
// переменные типа Bool
var firstLogicVar = true
var secondLogicVar = true
// 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
```

Обратите внимание, что оператор находит первое совпадение, после чего выполняется соответствующий блок кода и происходит выход из оператора. Если условие с логическим ИЛИ (`||`) расположить первым, а оно также возвращает `true`, то выведенный на консоль результат (одна из переменных истинна) будет лишь частично отражать реальную ситуацию.

Во всех примерах в качестве условий использовались переменные типа `Bool`. Но Swift не ограничивает нас этим. Ему важно только одно: чтобы условие в операторе `if` возвращало логическое значение. Давайте используем в качестве условия выражение, которое возвращает логическое значение.

Представьте, что у вас есть жилой дом, который вы сдаете в аренду. Стоимость аренды, которую платит каждый жилец, зависит от общего количества жильцов:

- Если количество жильцов меньше 5, то стоимость аренды жилья на месяц равна 1000 рублей с человека.
- Если количество жильцов от 5 до 7, то стоимость аренды равна 800 рублям с человека.

- Если количество жильцов более 7, то стоимость аренды равна 500 рублям с человека.

Напишем программу, реализующую подсчет общего количества денег, с помощью конструкции `if-else` (листинг 8.7). Программа получает количество жильцов в доме и в зависимости от стоимости аренды на одного жильца возвращает общую сумму собираемых средств.

Листинг 8.7

```
// количество жильцов в доме
var tenantCount = 6                                    6
// стоимость аренды на человека
var rentPrice = 0                                      0
/* определение цены на одного
 человека в соответствии с условием */
if tenantCount < 5 {
    rentPrice = 1000
} else if tenantCount >= 5 && tenantCount <= 7 {
    rentPrice = 800                                    800
} else {
    rentPrice = 500
}
// вычисление общей суммы
var allPrice = rentPrice * tenantCount                 4 800
```

В проверяемых условиях используются операторы сравнения и логические операторы. Результатом каждого из них является логическое значение.

Так как общее количество жильцов попадает во второй блок конструкции `if-else`, переменная `rentPrice` принимает значение 800. Итоговая сумма в результате равна 4800.

Тернарный оператор условия

Для вашего удобства Swift позволяет сократить стандартную форму записи оператора `if` всего до нескольких символов. Данная форма называется *тернарным оператором условия*.

СИНТАКСИС

```
проверяемое_условие ? код1 : код2
```

Данный оператор может выполнить один из двух блоков кода в зависимости от истинности проверяемого условия. Условие и выполняемый код отделяются друг от друга знаком вопроса. Сами же блоки кода отделяются друг от друга двоеточием.

При истинности проверяемого условия выполняется первый блок кода. Во всех остальных случаях выполняется второй блок.

Пример использования тернарного оператора условия приведен в листинге 8.8.

Листинг 8.8

```
// константы
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
```

Функциональность тернарного оператора точно такая же, как у стандартной формы записи оператора условия.

Самое интересное, что тернарные операторы не просто выполняют соответствующий блок кода, а возвращают его значение. По этой причине их можно использовать в составе других операторов (листинг 8.9).

Листинг 8.9

```
// переменная типа Int
var height = 180                                    180
// переменная типа Bool
var isHeader = true                                 true
// вычисление значения константы
let rowHeight = height + (isHeader ? 20 : 10 )      200
// вывод значения переменной
rowHeight                                           200
```

Использованный тернарный оператор условия возвращает целочисленное значение, зависящее от значения переменной `isHeader`. Данное возвращенное значение складывается с переменной `height` и записывается в константу `rowHeight`.

Оператор if для опционалов

Ранее мы выяснили, как определить факт существования значения в опционале с помощью утверждений. Основной недостаток утверждений в том, что они не позволяют выполнять произвольный код. Аналогию между утверждениями и оператором условия `if` мы проводили ранее, так что теперь можно попытаться использовать данный оператор в том числе и для обработки опционалов.

Конструкция `if-else` дает значительно более широкие возможности при разработке программ и, что самое важное, позволяет избежать ошибок, когда значение опционала получить необходимо, а оно отсутствует (листинг 8.10). Приведенный код позволяет определить факт присутствия троллей, о чем говорит значение переменной `trollsCount`. Если троллей нет, на консоль выводится соответствующее сообщение. В ином случае вычисляется и выводится количество котлов для троллей, которые необходимо приобрести.

Листинг 8.10

```
/* переменная опционального типа
с предустановленным значением */
var trollsCount: Int? = 8                                    8
// определение наличия значения
if trollsCount == nil {
    print("Тролли отсутствуют")
} else {
    // количество котлов для одного тролля
    var potCountForTroll = 2                                 2
    // общее количество требуемых котлов
    var allPotsCount = potCountForTroll * trollsCount!       10
}
```

Как и в случае использования функции `assert(_:_:file:line:)`, для определения факта наличия значения в опционале необходимо просто сравнить его с `nil`.

Обратите внимание, что при вычислении значения `allPotsCount` необходимо использовать принудительное извлечение опционального значения переменной `trollsCount`.

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.

В рамках использования конструкции `if-else` для проверки существования значения в опционале вы можете попробовать извлечь его и присвоить новому параметру. В этом случае если значение существует, то новый параметр будет создан и получит извлеченное опциональное значение. Если же значение равно `nil`, то попытка инициализации провалится.

Данный прием носит название *опционального связывания* (optional binding).

СИНТАКСИС

```
if var имя_создаваемой_переменной = имя_опционала {
    код1
} else {
    код2
}
```

Здесь вместо оператора var при необходимости может быть указан оператор let. Создаваемая переменная имеет ограниченную видимость (область действия). Использовать ее вы сможете только внутри данной конструкции if-else.

Если рассматриваемый опционал равен nil, то условие считается невыполненным и блок кода в операторе if опускается. Если в этом случае присутствует блок else, то выполняется код, размещенный в нем.

Если рассматриваемый опционал не равен nil, то опциональное значение автоматически извлекается, создается новый параметр (переменная или константа), ему присваивается только что извлеченное значение и блок кода в операторе условия if исполняется. При этом созданный параметр может быть использован только в данном блоке, то есть область видимости параметра ограничена данным блоком.

ПРИМЕЧАНИЕ Область видимости определяет, где в коде доступен некоторый объект. Если этот объект является глобальным, то он доступен в любой точке программы. Если объект является локальным, то он доступен только в том блоке кода (и во всех вложенных в него блоках), для которого он является локальным. Вне этого блока объект просто не виден.

Рассмотрим пример опционального связывания (листинг 8.11).

Листинг 8.11

```
1  /* переменная опционального типа
2   с предустановленным значением */
3  var monstersCount: Int? = 8                    8
4  // глобальная константа
5  var monsters = 0                               0
6  // определение наличия значения
7  if var monsters = monstersCount {
8      print("Всего \(monsters) монстров")        "Всего 8 монстров\n"
9  } else {
10     print("Тролли отсутствуют")
11 }
12 monsters                                        0
```

Консоль:

```
Всего 8 монстров
```

При опциональном связывании, используемом в операторе `if`, происходит автоматическое извлечение значения переменной `monstersCount`,

а так как оно существует, происходит инициализация этим значением объявляемой переменной `monsters`.

Обратите внимание, что переменная `monsters`, созданная вне оператора условия, и одноименная переменная `monsters`, созданная в процессе опционального связывания, — это разные переменные. Переменная, создаваемая при опциональном связывании, локальна для оператора условия, поэтому использовать ее можно только внутри данного оператора. При этом глобальная переменная `monsters` не затрагивается, о чем нам говорит вывод значения этой переменной в конце листинга.

Если бы в переменной `monstersCount` не существовало значения, то управление перешло бы к блоку `else`.

Хорошим примером опционального связывания является использование уже знакомой нам функции `Int(_:)`. Напомню, что данная функция при возможности преобразует строковый литерал в число и возвращает опциональный тип `Int`. Представьте, что у вас есть группа драконов. У большинства драконов есть свой сундук с золотом, и количество золотых монет в каждом из них разное. В любой момент времени вам необходимо знать общее количество монет во всей группе. Внезапно к вам поступает новый дракон. Вам необходимо написать код, который определит количество монет в сундуке нового дракона (если, конечно, у него есть сундук) и прибавит их к общему количеству золота (листинг 8.12).

Листинг 8.12

```
/* переменная типа String,
 содержащая количество золотых монет
 в сундуке нового дракона */
var coinsInNewChest = "140"                          "140"
/* переменная типа Int,
 в которой будет храниться общее
 количество монет у всех драконов */
var allCoinsCount = 1301                             1 301
// проверяем существование значения
if Int(coinsInNewChest) != nil{
    allCoinsCount += Int(coinsInNewChest)!           1 441
} else {
    print("У нового дракона отсутствует золото")
}
```

В данном случае функция `Int(_:)` используется дважды:

- в проверяемом условии оператора `if` при сравнении с `nil` преобразованного значения переменной `coinsInNewChest`;
- при сложении с переменной `allCoinsCount` в теле оператора `if`.

В результате происходит бесцельная трата процессорного времени, так как код выполняет одну и ту же функцию дважды. Можно избежать этого, заранее создав переменную, в которую будет извлечено значение опционала. Данную переменную необходимо использовать в обоих случаях вместо вызова функции `Int(_:)`. Пример показан в листинге 8.13.

Листинг 8.13

```
/* переменная типа String,
 содержащая количество золотых монет
 в сундуке нового дракона */
var coinsInNewChest = "140"                          "140"
/* переменная типа Int,
 в которой будет храниться общее
 количество монет у всех драконов */
var allCoinsCount = 1301                             1 301
/* извлекаем значение опционала
 в новую переменную */
var coins = Int(coinsInNewChest)                     140
/* проверяем существование значения
 с использованием созданной переменной */
if coins != nil{
    allCoinsCount += coins!                          1 441
} 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 тогда данная переменная уже не нужна.

При программировании вы должны всеми доступными способами избегать бесполезного расходования ресурсов компьютера, к которым относятся и процессорное время, и память.

Для того чтобы избежать расходования памяти, можно использовать опциональное связывание, так как после выполнения оператора условия созданная при связывании переменная автоматически удалится (листинг 8.14).

Листинг 8.14

```
/* переменная типа String,
 содержащая количество золотых монет
 в сундуке нового дракона */
var coinsInNewChest = "140"                          "140"
```

```
/* переменная типа Int,
 в которой будет храниться общее
 количество монет у всех драконов */
var allCoinsCount = 1301                                    1 301
/* проверяем существование значения
 с использованием опционального связывания */
if var coins = Int(coinsInNewChest){
    allCoinsCount += coins                                  1 441
} else {
    print("У нового дракона отсутствует золото")
}
```

Таким образом, мы избавились от повторного вызова функций и расходования оперативной памяти, получив красивый и оптимизированный код. В данном примере вы, вероятно, не ощутите увеличение скорости работы программы, но при разработке на языке Swift более сложных приложений для мобильных или стационарных устройств данный подход позволит вам получать вполне ощутимые результаты.

Задание

1. Создайте псевдоним типа `String` с именем Text.
2. Объявите три константы типа `Text`. Значения двух констант должны состоять только из цифр, третьей — из цифр и букв.
3. Из всех трех констант найдите те, которые состоят только из цифр, и выведите их на консоль. Для преобразования строки в число используйте функцию `Int(_:)`.
4. Создайте псевдоним для типа `(numberOne: Text?, numberTwo: Text?)?` с именем `TupleType`. Данный тип является опциональным и также содержит в себе опциональные значения.
5. Создайте три переменные типа `TupleType`, содержащие следующие значения: `("190", "67")`, `("100", nil)` , `("-65", "70")`.
6. Выведите значения элементов тех кортежей, в которых ни один из элементов не инициализирован как `nil`.

8.2. Оператор раннего выхода guard

Оператор `guard` называется *оператором раннего выхода*. Подобно оператору `if`, он проверяет истинность переданного ему условия. Отличие его в том, что он выполняет блок кода, расположенный в фигурных скобках, только в том случае, если условие вернуло значение `false`.

СИНТАКСИС

```
guard утверждение 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.

ВНИМАНИЕ Для данного оператора существует ограничение: его тело должно содержать один из следующих операторов — return, break, continue, throw (ни один из них до настоящего момента еще не рассматривался).

С примерами использования данного оператора мы познакомимся в следующих главах.

8.3. Операторы диапазона

Ранее мы уже познакомились с большим количеством различных операторов. В Swift существуют специальные операторы, с помощью которых можно объединить множество последовательных числовых значений (например, 1, 2, 3, 4), то есть создать диапазон (например, от 1 до 4). Такие операторы называются *операторами диапазона* (range operators).

Swift предлагает два оператора диапазона:

- Закрытый оператор: `a...b` определяет диапазон, который содержит все числа от `a` до `b` (включая `a` и `b`). Например, для целочисленного типа диапазон `1...4` включает в себя числа 1, 2, 3, 4.
- Полуоткрытый оператор: `a..<b` определяет диапазон чисел от `a` до `b` (включая только `a`). Например, для целочисленного типа диапазон `2..<5` включает в себя числа 2, 3, 4.

В скором времени мы рассмотрим примеры использованиях данных операторов.

8.4. Оператор ветвления switch

В некоторых ситуациях необходимо реализовать большое количество различных вариантов выполнения кода в зависимости от возможного значения параметра. Конечно же, для этого можно использовать расширенный синтаксис оператора `if`, многократно повторяя блоки `else-if`. Однако подобный подход вносит неразбериху в код и по-

этому является плохой практикой программирования. Для решения задачи в Swift существует оператор ветвления `switch`, позволяющий, как и `if`, в зависимости от значения переданного выражения выбрать соответствующий блок кода. Основное отличие в том, что `switch` дает возможность работать не только с логическим типом данных. Он может принимать для проверки параметр произвольного типа.

Синтаксис оператора switch

Рассмотрение оператора `switch` начнем с примера использования конструкции `if-else` для большого количества возможных вариантов (листинг 8.15). Данный код выводит на консоль определенный текст в зависимости от оценки, полученной учеником.

Листинг 8.15

```
/* переменная типа Int
 содержит оценку, полученную
 пользователем */
var userMark = 4
/* используем конструкцию if-else
 для определения значения
переменной userMark и вывода
 необходимого текста */
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!")
}
```

Консоль:

```
Неплохо, но могло быть и лучше
```

Для поиска необходимого сообщения оператор `if` последовательно проходит по каждому из указанных условий, пока не найдет то, которое возвращает `true`.

Несмотря на то что эта программа работает корректно, лучше использовать оператор ветвления `switch`, позволяющий выполнить ту же самую операцию, но код при этом получается более читабельным и прозрачным.

СИНТАКСИС

```
switch проверяемое_выражение {
    case значение1:
        код1
    case значение2, значение3:
        код2
        ...
    case значениеM:
        break
    default:
        кодN
}
```

Оператор switch, также называемый конструкцией switch-case, после вычисления переданного выражения производит поиск полученного значения среди указанных после ключевых слов case.

Количество блоков case может быть произвольным. После каждого ключевого слова case может быть указано любое количество значений, которые отделяются друг от друга запятыми. Указания на значения заканчиваются символом двоеточия, после которого следует блок исполняемого кода.

В зависимости от искомого значения выбирается соответствующий блок case и выполняется код, находящийся в этом блоке.

Предположим, что проверяемое выражение вернуло значение, соответствующее значению 1. В этом случае происходит выполнение первого блока кода. Если ни один из операторов case не имеет совпадения с искомым значением, то выполняется *N*-й блок кода, находящийся после ключевого слова default. Данное ключевое слово содержит код, выполняемый в том случае, когда не найдено ни одного совпадения в блоках case. Он подобен оператору else в конструкции if-else.

Блок default должен всегда располагаться последним в конструкции switch-case.

Тело конструкции switch-case обрамляется фигурными скобками точно так же, как в конструкции if-else обрамлялись отдельные блоки кода.

В конце каждого блока case нет необходимости ставить оператор break, как этого требуют другие языки программирования. Данный оператор ставится только в том случае, если блок case или default не содержит выполняемого кода.

Код, выполненный в любом блоке case, приводит к завершению работы оператора switch.

ПРИМЕЧАНИЕ Конструкция switch-case должна быть исчерпывающей, то есть содержать информацию обо всех возможных значениях проверяемого параметра. Это обеспечивается в том числе наличием блока default. Если данный блок не должен содержать никакого исполняемого кода, то просто расположите в нем оператор break.

Перепишем программу проверки оценки ученика с использованием конструкции `switch-case` (листинг 8.16).

Листинг 8.16

```
/* переменная типа Int
 содержит оценку, полученную
 пользователем */
var userMark = 4
/* используем конструкцию if-else
 для определения значения
 переменной userMark и вывода
 необходимого текста */
switch userMark {
    case 1:
        print("Единица на экзамене! Это ужасно!")
    case 2:
        print("С двойкой ты останешься на второй год!")
    case 3:
        print("Ты плохо учил материал в этом году!")
    case 4:
        print("Неплохо, но могло быть и лучше")
    case 5:
        print("Бесплатное место в университете тебе обеспечено!")
    default:
        break
}
```

Консоль:

```
Неплохо, но могло быть и лучше
```

Оператор `switch` получает значение переданной в него переменной `userMark` и последовательно проверяет каждый блок `case` в поисках совпадающего значения.

Как вы можете видеть, конструкция `switch-case` действительно сделала код более читабельным.

Оператор `switch` в Swift является одним из самых совершенных среди подобных операторов языков программирования. Его можно использовать для любых типов данных, объектов и даже кортежей. Он предоставляет поистине замечательные возможности для создания кода. Например, чтобы указать сразу на множество числовых значений, в качестве значения в блоках `case` можно использовать операторы диапазона или операторы условий (листинг 8.17).

Листинг 8.17

```
/* переменная типа Int
 содержит оценку, полученную
 пользователем */
var userMark = 4
```

```
// проверка значения с использованием диапазона
switch userMark {
    case 1..<3:
        print("Экзамен не сдан!")
    case 3...5:
        print("Экзамен сдан!")
    default:
        assert(false, "Оценка \(userMark) вне доступного
        диапазона")
}
```

Консоль:

```
Экзамен сдан!
```

Строка `"Экзамен не сдан!"` выводится на консоль при условии, что проверяемое значение в переменной `userMark` равно 1 или 2. В остальных случаях выводится строка `"Экзамен сдан!"`.

Блок `default` позволяет избежать ошибки, если вдруг оценка окажется вне диапазона `1...5`. Он прерывает выполнение программы, сообщая, что значение переменной `userMark` ошибочно.

Ключевое слово fallthrough

Как отмечалось ранее, после выполнения кода, расположенного в блоке `case`, происходит завершение работы конструкции `switch-case`. Однако в некоторых случаях требуется не завершать работу конструкции `switch-case`, а перейти к выполнению кода в следующем блоке `case`. Для этого в конце блока `case` указывается ключевое слово `fallthrough` (листинг 8.18). 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. В зависимости от переданного уровня необходимо вывести на консоль все предназначенные для выполнения мероприятия. При этом минимальный уровень готовности — это В, максимальный — А (включает в себя мероприятия уровней В и Б).

Листинг 8.18

```
/* переменная типа Character
 содержит уровень
 готовности */
var level: Character = "Б"
// определение уровня готовности
```

```
switch level {
    case "А":
        print("Выключить все электрические приборы ")
        fallthrough
    case "Б":
        print("Закрыть входные двери и окна ")
        fallthrough
    case "В":
        print("Соблюдать спокойствие")
    default:
        break
}
```

Консоль:

```
Закрыть входные двери и окна
Соблюдать спокойствие
```

При значении `"Б"` переменной `level` на консоль выводятся строки, соответствующие значениям `"Б"` и `"В"`, то есть когда программа встречает ключевое слово `fallthrough`, она переходит к выполнению кода, соответствующего значению `"В"`.

Ключевое слово where

С помощью блоков `case` вы определяете возможные значения, которые может принять выражение. В одном блоке `case` можно определить даже несколько возможных значений. Однако Swift позволяет в пределах одного блока `case` указать не только на значение рассматриваемого параметра, но и на зависимость от какого-либо условия. Данный функционал реализуется с помощью ключевого слова `where` в блоке `case`.

Рассмотрим пример. Пусть в вашем загоне с драконами есть три разные площадки. Вам необходимо распределять поступающих драконов между площадками в зависимости от их цвета и веса по следующим правилам:

- загон 1: зеленые драконы весом менее 2 тонн;
- загон 2: красные драконы весом менее 2 тонн;
- загон 3: зеленые и красные драконы весом от 2 тонн.

Здесь используются два различных параметра (цвет и вес) для определения номера загона. При этом `switch` может принять для проверки только один из них.

Для решения проблемы можно, конечно, использовать конструкцию `if-else`, проверяющую цвет, в каждой из ветвей которой создать

конструкции `switch-case` для проверки веса. Но данный подход — плохая практика программирования, так как есть более совершенный и удобный способ. Для решения возникшей проблемы создадим одну конструкцию `switch-case` и используем ключевое слово `where` (листинг 8.19).

Листинг 8.19

```
/* переменная типа Float
 содержит вес дракона */
var dragonWeight: Float = 2.4
/* переменная типа Float
 содержит цвет дракона */
var dragonColor = "зеленый"
// определение загона для поступившего дракона
switch dragonColor {
    case "зеленый" where dragonWeight < 2:
        print("Поместите дракона в загон 1")
    case "красный" where dragonWeight < 2:
        print("Поместите дракона в загон 2")
    case "зеленый", "красный" where dragonWeight >= 2:
        print("Поместите дракона в загон 3")
    default:
        break
}
```

Консоль:

```
Поместите дракона в загон 3
```

Ключевое слово `where` ставится в блоке `case` после перечисления значений, за ним следует логическое выражение, которое должно вернуть `true` или `false`.

Код в блоке `case` выполняется, когда найдено совпадающее значение и условие `where` вернуло `true`.

С помощью ключевого слова `where` вы можете также указывать на значение проверяемого в операторе `switch` параметра. В этом случае вместо значения блока `case` используйте уже знакомый вам символ нижнего подчеркивания.

Перепишем приведенный ранее пример для проверки оценки пользователя (листинг 8.20).

Листинг 8.20

```
var userMark = 4
switch userMark {
    case _ where userMark>1 && userMark<3:
```

```
4           print("Экзамен не сдан!")
5       case _ where userMark >= 3:
6           print("Экзамен сдан!")
7       default:
8           assert(false, "Оценка \(userMark) вне доступного
            диапазона")
9   }
```

Консоль:

```
Экзамен сдан!
```

Здесь мы не указываем значение после ключевого слова `case`, а вместо него ставим символ пропуска нижнего подчеркивания, поэтому проверяется лишь переданное после `where` условие.

Кортежи в операторе switch

Возможности оператора `switch` не заканчиваются использованием рассмотренных операторов и ключевых слов. Ранее вы уже познакомились с кортежами, благодаря которым значительно повышается эффективность работы со значениями различных типов. Swift умеет использовать кортежи в качестве передаваемого в конструкцию `switch-else` параметра. При этом искомое значение в блоке `case` необходимо указывать точно так же, как пишется значение самого кортежа, то есть в скобках, где элементы разделены запятыми. Каждый элемент кортежа может быть проверен с помощью произвольного значения или диапазона значений. Дополнительно для пропуска элемента можно использовать символ нижнего подчеркивания на месте любого элемента.

Вернемся к примеру с драконами. В нем значения переменных `dragonColor` и `dragonWeight` были разделены, поэтому приходилось использовать ключевое слово `where` для вывода необходимого текста. Однако эти переменные можно объединить в кортеж и проверить совместно (листинг 8.21).

Листинг 8.21

```
1  /* кортеж типа (String, Int)
2  содержит цвет и вес дракона */
3  var dragonCharacteristic = ("зеленый", 2.4)
4  // определение загона для поступившего дракона
5  switch dragonCharacteristic {
6      case ("зеленый", 0..<2):
7          print("Поместите дракона в загон 1")
```

```
8       case ("красный", 0..<2):
9           print("Поместите дракона в загон 2")
10      case ("красный", _), ("зеленый", _) where
        dragonCharacteristic.1 > 2:
11          print("Поместите дракона в загон 3")
12      default:
13          print("Дракон с неизвестными параметрами")
14  }
```

Консоль:

```
Поместите дракона в загон 1
```

Ключевое слово `case` тоже может содержать несколько значений кортежа, перечисляемых через запятую, как и значения отдельных параметров.

Для того чтобы найти всех драконов с весом более 2 тонн, нам пришлось игнорировать второй элемент кортежа (с помощью символа нижнего подчеркивания) и переносить проверку его значения в условие `where`.

Для того чтобы абстрагироваться в данном примере от имени переданного в оператор `switch` параметра, можно использовать замечательный прием, называемый *связыванием значений* (по аналогии с опциональным связыванием).

Суть его в том, чтобы объявить новую переменную или константу и в автоматическом режиме инициализировать ее значением проверяемого параметра (или элемента проверяемого параметра) для его проверки с помощью ключевого слова `where`. Пример приведен в листинге 8.22.

Листинг 8.22

```
1  /* кортеж типа (String, Int)
2  содержит цвет и вес дракона */
3  var dragonCharacteristic = ("зеленый", 2.4)
4  // определение загона для поступившего дракона
5  switch dragonCharacteristic {
6      case ("зеленый", 0..<2):
7          print("Поместите дракона в загон 1")
8      case ("красный", 0..<2):
9          print("Поместите дракона в загон 2")
10     case ("зеленый", let weight) where weight >= 2:
11         print("Поместите дракона в загон 3")
12     case ("красный", let weight) where weight >= 2:
13         print("Поместите дракона в загон 3")
```

```
    default:
        print("Дракон с неизвестными параметрами")
}
```

Консоль:

```
Поместите дракона в загон 3
```

Для того чтобы обработать оба варианта для больших драконов (массой более 2 тонн), необходимо создать два отдельных блока `case`: первый для зеленого цвета, второй — для красного. Это связано с тем, что Swift запрещает связывать один и тот же элемент при нескольких значениях в операторе `case`. Другими словами, следующий код вызовет ошибку:

```
case ("красный", let weight), ("зеленый", let weight) where weight >= 2:
```

Поэтому необходимо разделить оба варианта на два отдельных блока `case`.

Данный способ, при котором совпадающие условия разделяются на отдельные блоки `case`, доступен только в том случае, когда количество значений элемента, из-за которого происходит разделение, невелико. Однако представьте, что у вас 5–10 различных вариантов расцветки дракона. В таком случае количество блоков `case` станет слишком большим.

Для решения проблемы все возможные значения элемента можно объединить в переменную-множество, а при связывании значений использовать кортеж целиком (листинг 8.23).

Листинг 8.23

```
/* кортеж типа (String, Int)
содержит цвет и вес дракона */
var dragonCharacteristic = ("зеленый", 2.4)
// определение загона для поступившего дракона
switch dragonCharacteristic {
    case ("зеленый", 0..<2):
        print("Поместите дракона в загон 1")
    case ("красный", 0..<2):
        print("Поместите дракона в загон 2")
    case let(color, weight) where (color == "зеленый" || color ==
    "красный") && weight >= 2:
        print("Поместите дракона в загон 3")
    default:
        print("Дракон с неизвестными параметрами")
}
```

Консоль:

```
Поместите дракона в загон 3
```

Для того чтобы применить связывание значений для всех элементов массива, необходимо после оператора `let` (или `var`) в скобках указать имена объявляемых локальных параметров, которые будут инициализированы значениями кортежа. После этого, как и при связывании отдельных значений, после ключевого слова `where` указываются условия, при которых будет выполнен код из блока `case`.

Массивы подобны рассмотренному ранее типу данных `String`, который содержит в себе коллекцию элементов (символов). Массив же не просто содержит в себе коллекцию элементов как одну из своих характеристик, а непосредственно является этой коллекцией. Подробнее о массивах вы узнаете в следующей главе.

Описанные в этой главе приемы открывают просто фантастические возможности в разработке программ. О других очень полезных способах применения оператора `switch` вы узнаете при описании перечислений. А для завершения темы выполните самостоятельную работу.

Задание

1. Определите псевдоним `Operation` типу кортежа, содержащему три элемента со следующими именами: `operandOne`, `operandTwo`, `operation`.

   Первые два — это числа с плавающей точкой. Они будут содержать операнды для выполняемой операции.

   Третий элемент — строковое значение типа `Character`. Оно будет содержать указатель на проводимую операцию. Всего может быть четыре вида операций: `+`, `-`, `*`, `/`.

2. Создайте константу типа `Operation` и заполните значения ее элементов произвольным образом, например (`3.1`, `33`, `"+"`).
3. Используя конструкцию `switch-case`, вычислите значение операции, указанной в элементе `operation` созданного кортежа для операндов в его первых двух элементах. Оператор `switch` должен корректно обрабатывать любую из перечисленных операций.
4. В созданной константе замените символ операции другим произвольным символом и проверьте правильность работы конструкции `switch-case`.

9 Типы коллекций

Коллекция — это множество элементов. Swift предлагает три фундаментальных типа коллекций: *массив* (array), *набор* (set) и *словарь* (dictionary). Все три типа коллекций по аналогии с кортежами представляют собой отдельные типы данных.

Материал данной главы достаточно важен для того, чтобы потратить время для его подробного изучения.

9.1. Массивы

Объявление массива

Массив — это упорядоченная коллекция однотипных элементов, для доступа к которым используются индексы этих элементов. Упорядоченной называется коллекция, в которой элементы располагаются в порядке, заложенном разработчиком. Массивы являются очень важным функциональным типом, которым вы будете пользоваться практически в каждой программе.

Элементы массива представляют собой пары «индекс-значение».

Индекс элемента имеет тип данных `Int` и генерируется автоматически в зависимости от порядкового номера элемента. Индексы в массивах начинаются с нуля. У массива, содержащего 5 элементов, индекс первого элемента равен 0, а последнего — 4. Индексы всегда последовательны и неразрывны.

Значение элемента — это произвольное значение некоторого определенного типа данных. Как говорилось ранее, значения доступны по соответствующим им индексам. Очень важной чертой массивов в Swift является жесткая типизация значений его элементов, то есть тип данных всех элементов в пределах одного массива должен быть одним

и тем же. Даже если вами объявлен пустой массив, у него все равно должен быть определен тип пока еще не существующих элементов.

Значение массива задается с помощью *литерала массива*, в котором перечисляются входящие в состав элементов значения.

СИНТАКСИС

```
[значение_1, значение_2, ...,значение_N]
```

Литерал массива указывается в квадратных скобках, а значения отдельных элементов в нем разделяются запятыми. Массив может содержать произвольное количество элементов одного типа.

Индексы значений присваиваются автоматически в зависимости от порядка следования элементов.

Массивы хранятся в переменных и константах, поэтому для их объявления используются операторы `var` и `let`.

Для создания неизменяемого массива (состав и значения элементов такого массива невозможно изменить) используйте оператор `let`, в противном случае — оператор `var`.

Пример создания массива приведен в листинге 9.1.

Листинг 9.1

```
1  // неизменяемый массив
2  let alphabetArray = ["a", "b", "c"]
3  // изменяемый массив
4  var mutableArray = [2, 4, 8]
```

В приведенном примере создается два массива: `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.

Типом массива является тип-значение (value type), а не тип-ссылка (reference type). Это означает, что при попытке скопировать или передать значение массива создается его копия, с которой и происходит вся дальнейшая работа (листинг 9.2).

Листинг 9.2

```
1  // неизменяемый массив
2  let unmutableArray = ["one", "two", "three"]      ["one", "two",
                                                      "three"]
3  // копируем массив из константы в переменную
```

```
4  var newArray = unmutableArray                    ["one", "two",
                                                    "three"]
5  newArray                                         ["one", "two",
                                                    "three"]

6  // изменяем значение нового массива
7  newArray[1] = "four"                             "four"
8  // выводим значение исходного массива
9  unmutableArray                                   ["one", "two",
                                                    "three"]
```

При передаче значения исходного неизменяемого массива в новом массиве создается его полная копия. При изменении данной копии значение исходного массива остается прежним.

ПРИМЕЧАНИЕ Несмотря на логичность данного подхода, он иногда способен стать причиной определенных трудностей при разработке приложений, когда без необходимости создаются многочисленные копии массива, что ведет к росту бесцельно используемой памяти.

Подобно глобальным функциям (а точнее инициализатору типов данных), которые мы обсудили в разделе о приведении числовых типов данных (см. главу 4), вы можете создать массив с помощью функции `Array()`.

СИНТАКСИС

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

Таким образом, литерал массива передается в виде входных параметров функции `Array()`.

Пример создания массива приведен в листинге 9.3.

Листинг 9.3

```
1  // создание массива с помощью передачи списка значений
2  let alphabetArray = Array("a", "b", "c")
3  // создание массива с помощью оператора диапазона
4  let numArray = Array(0...9)
```

Как видите, для передачи параметров вы можете использовать оператор диапазона.

Сравнение массивов

Массивы, так же как значения фундаментальных типов данных, можно сравнивать между собой. Два массива являются эквивалентными, если:

- количество элементов в сравниваемых массивах одинаково;
- каждая соответствующая пара элементов эквивалентна.

Рассмотрим пример сравнения двух массивов (листинг 9.4).

Листинг 9.4

```
/* три константы, которые
 станут элементами массива */
let a1 = 1
let a2 = 2
let a3 = 3
if [1, 2, 3] == [a1, a2, a3] {
    print("Массивы эквивалентны")
} else {
    print("Массивы не эквивалентны")
}
```

Консоль:

```
Массивы эквивалентны
```

Несмотря на то что в массиве `[a1, a2, a3]` указаны не значения, а константы, содержащие эти значения, условия эквивалентности массивов все равно выполняются.

Доступ к элементам массива

Для доступа к отдельному элементу массива необходимо использовать индекс данного элемента, заключенный в квадратные скобки и указанный после имени массива (листинг 9.5).

Листинг 9.5

```
// неизменяемый массив
let alphabetArray = ["a", "b", "c"]          ["a", "b", "c"]
// изменяемый массив
var mutableArray = [2, 4, 8]                 [2, 4, 8]
// доступ к элементам массивов
alphabetArray[1]                             "b"
mutableArray[2]                              8
```

ПРИМЕЧАНИЕ Способ доступа к элементам с использованием ключевых наименований, в данном случае индексов, называется сабскриптом. В следующих разделах мы познакомимся со всеми возможностями сабскриптов.

С помощью индексов можно получать доступ к элементам массива не только для чтения, но и для изменения (листинг 9.6).

Листинг 9.6

```
// изменяемый массив
var mutableArray = [2, 4, 8]                    [2, 4, 8]
// изменение элемента массива
mutableArray[1] = 16                            16
// вывод нового массива
mutableArray                                    [2, 16, 8]
```

Попытка модификации массива, хранящегося в константе, вызовет ошибку.

При использовании оператора диапазона можно получить доступ сразу ко множеству элементов в составе массива, то есть к его подмассиву. Данный оператор должен указывать на индексы крайних элементов выделяемого множества (листинг 9.7).

Листинг 9.7

```
// изменяемый массив
var mutableArray = ["one", "two", "three", "four"]  ["one", "two",
                                                     "three", "four"]
// скопируем подмассив в отдельную переменную
var subArray = mutableArray[1...2]                  ["two", "three"]
/* заменим несколько элементов
 новым массивом */
mutableArray[1...2] = ["five"]                      ["five"]
mutableArray                                        ["one", "five",
                                                    "four"]
```

Выделенное множество элементов рассматривается в Swift как полноценный массив.

После замены элементов из диапазона `[1...2]` на `["five"]` индексы 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.

Явное указание типа элементов

Как при использовании фундаментальных типов данных, при объявлении переменных и констант Swift может автоматически определять тип элементов объявляемого массива по инициализируемым значениям. При необходимости можно явно указать тип данных элементов.

Существует две формы указания типа массива (и типов входящих в него элементов).

СИНТАКСИС

Полная форма записи:

```
var имяМассива: Array<ТипДанных>
```

Краткая форма записи:

```
var имяМассива: [ТипДанных]
```

В обоих случаях объявляется массив, элементы которого должны иметь указанный тип данных.

Тип массива в этом случае будет равен [ТипДанных] (с квадратными скобками) или Array<ТипДанных>. Оба обозначения типа массива эквивалентны. Типом каждого отдельного элемента является ТипДанных (без квадратных скобок).

При использовании данного синтаксиса массив объявляется, но его значение не инициализируется. Объявленный массив можно использовать только после того, как ему будет присвоено некоторое значение. Это можно сделать в том же выражении, что и объявление с указанием типа, либо написать новое выражение (листинг 9.8). Подобный подход вы встречали при создании переменных и констант фундаментальных типов.

Листинг 9.8

```
/* одним выражением объявляем массив,
 явно указываем тип его элементов
 и инициализируем его значение */
var firstArray: [String] = ["x", "y", "z"]     ["x", "y", "z"]
// объявляем массив без элементов
var secondArray: Array<Int>
// инициализируем его значение
secondArray = [1, 2, 3, 4, 5]                  [1, 2, 3, 4, 5]
// выводим значения обоих массивов
firstArray                                     ["x", "y", "z"]
secondArray                                    [1, 2, 3, 4, 5]
```

При попытке обратиться к неинициализированному массиву Xcode сообщит об ошибке.

Создание пустого массива

Если перед вами стоит цель получить пустой массив, то он должен инициализироваться значением, не содержащим каких-либо элементов. Для этого можно использовать один из следующих способов:

- явно указать тип создаваемого массива и передать ему значение `[]`;
- по аналогии с пустым значением одного из фундаментальных типов использовать специальную функцию (вроде `Int()` для целочисленного типа), но при этом наименование типа заключить в квадратные скобки, так как массив имеет тип данных `[ТипДанных]`.

Оба способа продемонстрированы в листинге 9.9.

Листинг 9.9

```
1  /* объявляем массив с пустым значением
2  с помощью переданного значения */
3  var emptyArray: [String] = []                [] 
4  /* объявляем массив с пустым значением
5  с помощью специальной функции */
6  var anotherEmptyArray = [String]()           []
```

В результате создаются два пустых массива, `emptyArray` и `anotherEmptyArray`, уже инициализированные значениями (хотя и не содержащими элементов), а значит, с ними можно взаимодействовать.

Функция `[ТипДанных]()` позволяет также создать массив, состоящий из определенного числа одинаковых значений. Для этого в качестве входных параметров необходимо передать параметр `count`, указывающий на количество элементов, и параметр `repeatedValue`, указывающий на значение элементов (листинг 9.10).

Листинг 9.10

```
1  /* объявляем массив с пятью
2  одинаковыми значениями */
3  var alphaArray = [String?]( count: 5,
   repeatedValue: nil)                    [nil, nil, nil, nil, nil]
```

В результате создается массив типа `[String?]`, содержащий пять элементов `nil`. Для получения такого массива можно было также передать литерал `[nil, nil, nil, nil, nil]` в качестве значения.

Тип значения параметра `repeatedValue` должен совпадать с типом элементов массива.

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 слива-

ются в одно, образуя новый массив. Обратите внимание на несколько моментов:

- Результирующий массив будет содержать значения из обоих массивов, но индексы этих значений могут не совпадать с родительскими.
- Значения элементов подлежащих слиянию массивов должны иметь один и тот же тип данных.

Операция слияния производится с помощью уже известного оператора сложения (+), как показано в листинге 9.11.

Листинг 9.11

```
1  // создаем два массива
2  let charsOne = ["a", "b", "c"]          ["a", "b", "c"]
3  let charsTwo = ["d", "e", "f"]          ["d", "e", "f"]
4  let charsThree = ["g", "h", "i"]        ["g", "h", "i"]
5  // создаем новый слиянием двух
6  var alphabet = charsOne + charsTwo      ["a", "b", "c", "d", "e",
                                           "f"]
7  // сливаем новый массив со старым
8  alphabet += charsThree                  ["a", "b", "c", "d", "e",
                                           "f", "g", "h", "i"]
10 alphabet[8]                             "i"
```

Полученное в результате значение массива `alphabet` собрано из трех других массивов.

Многомерные массивы

Элементами массива могут быть не только значения фундаментальных типов, но и другие массивы. Массивы, содержащие в себе другие массивы, называются многомерными. Необходимо обеспечить единство типа всех вложенных массивов.

Рассмотрим пример в листинге 9.12.

Листинг 9.12

```
1  var arrayOfArrays = [[1,2,3], [4,5,6], [7,8,9]]
```

В данном примере создается массив, содержащий массивы типа `[Int]` в качестве своих элементов. Типом основного массива `arrayOfArrays` является `[[Int]]` (с удвоенными квадратными скобками с каждой стороны).

Для доступа к элементу многомерного массива необходимо указывать несколько индексов (листинг 9.13).

Листинг 9.13

```
1  var arrayOfArrays = [[1,2,3], [4,5,6], [7,8,9]]
2  // получаем вложенный массив
3  arrayOfArrays[2]                                    [7, 8, 9]
4  // получаем элемент вложенного массива
5  arrayOfArrays[2][1]                                 7
```

Строка `arrayOfArrays[2]` возвращает третий вложенный элемент массива `arrayOfArrays`. Строка `arrayOfArrays[1][2]`, использующая два индекса, возвращает второй элемент подмассива, содержащегося в третьем элементе массива `arrayOfArrays`.

Базовые свойства и методы массивов

Массивы — очень функциональные элементы языка. Об этом позаботились разработчики Swift, предоставив нам набор свойств и методов, позволяющих значительно расширить их возможности в сравнении с другими языками.

Свойство `count` возвращает количество элементов в массиве (листинг 9.14).

Листинг 9.14

```
1  var someArray = [1, 2, 3, 4, 5]                     [1, 2, 3, 4, 5]
2  // количество элементов в массиве
3  someArray.count                                     5
```

Если значение свойства `count` равно нулю, то и свойство `isEmpty` возвращает `true` (листинг 9.15).

Листинг 9.15

```
1  var someArray: [Int] = []                           []
2  someArray.count                                     0
3  someArray.isEmpty                                   true
```

Вы можете использовать свойство `count` для того, чтобы получить требуемые элементы массива (листинг 9.16).

Листинг 9.16

```
1  var someArray = [1, 2, 3, 4, 5]                     [1, 2, 3, 4, 5]
2  // количество элементов в массиве
3  var newArray = someArray[someArray.count-3...       [3, 4, 5]
   someArray.count-1]
```

Другим средством получить множество элементов массива является метод `suffix(_:)` — в качестве входного параметра ему передается количество элементов, которые необходимо получить. Элементы отсчитываются начиная с последнего элемента массива (листинг 9.17).

Листинг 9.17

```
var someArray = [1, 2, 3, 4, 5]                [1, 2, 3, 4, 5]
// получаем три последних элемента массива
let subArray = someArray.suffix(3)             [3, 4, 5]
```

Свойства `first` и `last` возвращают первый и последний элементы массива (листинг 9.18).

Листинг 9.18

```
var someArray = [1, 2, 3, 4, 5]                [1, 2, 3, 4, 5]
// возвращает первый элемент массива
someArray.first                                1
// возвращает последний элемент массива
someArray.last                                 5
```

С помощью метода `append(_:)` можно добавить новый элемент в конец массива (листинг 9.19).

Листинг 9.19

```
var someArray = [1, 2, 3, 4, 5]                [1, 2, 3, 4, 5]
someArray.append(6)                            [1, 2, 3, 4, 5, 6]
```

Если массив хранится в переменной (то есть является изменяемым), то метод `insert(_:atIndex:)` вставляет в массив новый одиночный элемент с указанным индексом (листинг 9.20).

Листинг 9.20

```
var someArray = [1, 2, 3, 4, 5]                [1, 2, 3, 4, 5]
// вставляем новый элемент в середину массива
someArray.insert(100, atIndex: 2)              [1, 2, 100, 3,
                                               4, 5]
```

При этом индексы массива пересчитываются, чтобы обеспечить их последовательность.

Так же как в случае изменения массива, методы `removeAtIndex(_:)`, `removeFirst()` и `removeLast()` позволяют удалять требуемые эле-

менты. При этом они возвращают значение удаляемого элемента (листинг 9.21).

Листинг 9.21

```
var someArray = [1, 2, 3, 4, 5]                    [1, 2, 3, 4, 5]
// удаляем третий элемент массива (с индексом 2)
someArray.removeAtIndex(2)                         3
// удаляем первый элемент массива
someArray.removeFirst()                            1
// удаляем последний элемент массива
someArray.removeLast()                             5
/* итоговый массив содержит
всего два элемента */
someArray                                          [2, 4]
```

После удаления индексы оставшихся элементов массива перестраиваются. В данном случае в итоговом массиве `someArray` остается всего два элемента с индексами 0 и 1.

Если массив является неизменяемым (хранится в константе), то можно использовать методы `dropFirst(_:)` и `dropLast()`, возвращающие новый массив, в котором отсутствует несколько первых или последних элементов. При этом если в качестве параметра в функцию ничего не передавать, то из результата удаляется один элемент. В противном случае — столько элементов, сколько передано в метод (листинг 9.22).

Листинг 9.22

```
let someArray = [1, 2, 3, 4, 5]                    [1, 2, 3, 4, 5]
// удаляем последний элемент
someArray.dropLast()                               [1, 2, 3, 4]
// удаляем три первых элемента
var newArray = someArray.dropFirst(3)              [4, 5]
someArray                                          [1, 2, 3, 4, 5]
```

При использовании данных методов основной массив `someArray`, с которым выполняются операции, не меняется.

Метод `contains(_:)` определяет факт наличия некоторого элемента в массиве и возвращает `Bool` в зависимости от результата (листинг 9.23).

Листинг 9.23

```
let intArray = [1, 2, 3, 4, 5, 6]                  [1, 2, 3, 4, 5, 6]
// проверка существования элемента
let resultTrue = intArray.contains(4)              true
let resultFalse = intArray.contains(10)            false
```

Метод `indexOf(_:)` сообщает индекс первого вхождения искомого элемента в рассматриваемом массиве. Так как искомый элемент может отсутствовать, метод возвращает опциональное значение (листинг 9.24). Если элемент отсутствует, то возвращается `nil`.

Листинг 9.24

```
let intArray = [1, 2, 3, 4, 5, 6, 4, 5, 6]      [1, 2, 3, 4, 5, 6, 4, 5, 6]
// поиск первого вхождения элемента
let result = intArray.indexOf(4)                 3
let resultNIL = intArray.indexOf(99)             nil
```

Значение 4 первый раз встречается в массиве `intArray` в элементе с индексом 3. Значение 99 в массиве `intArray` отсутствует, в результате возвращается `nil`.

Для поиска минимального или максимального элемента в массиве применяются глобальные методы `minElement()` и `maxElement()`. Данные методы работают только в том случае, если элементы массива можно сравнить между собой (листинг 9.25).

Листинг 9.25

```
let intArray = [3, 2, 4, 5, 6, 4, 7, 5, 6]      [3, 2, 4, 5, 6, 4, 7, 5, 6]
// поиск минимального элемента
intArray.minElement()                            2
// поиск максимального элемента
intArray.maxElement()                            7
```

ПРИМЕЧАНИЕ Сопоставлять можно значения тех типов данных, которые являются хешируемыми, то есть в них должен присутствовать функционал вычисления хеша для значения. Большинство фундаментальных типов данных поддерживают хеширование. Хеш обеспечивает возможность сравнения (сопоставления) различных значений одного и того же типа. При его вычислении собственное значение параметра по специальному алгоритму преобразуется в числовое значение и помещается в свойство hashValue параметра. Для доступа к хешу параметра достаточно вызвать указанное свойство:

```
var a: Float = 3.5
a.hashValue // 1 080 033 280
```

Данный функционал обеспечивается стандартным для Swift протоколом Hashable. О том, что такое протоколы и как их применять для собственных разработок, вы узнаете позже.

Чтобы изменить порядок следования всех элементов массива на противоположный, используйте метод `reverse()`, как показано в листинге 9.26.

Листинг 9.26

```
1  var someArray = [1, 3, 5, 7, 9]          [1, 3, 5, 7, 9]
2  someArray.reverse()                      ReverseRandomAccess
                                            Collection<Array<Int>>
```

Удивительно, но метод `reverse()` возвращает вовсе не массив типа `[Int]`, а некое неизвестное нам значение типа `ReverseRandomAccessCollection`. Это одна из интереснейших особенностей языка Swift. О том, что это за тип данных и как из него получить привычный массив, мы поговорим в следующих разделах.

9.2. Наборы

Объявление набора

Набор — это неупорядоченная коллекция уникальных элементов. В отличие от массивов, у элементов набора нет какого-либо определенного порядка следования, важен лишь факт наличия некоторого значения в наборе. Определенное значение элемента может существовать в нем лишь единожды, то есть каждое значение в пределах одного набора должно быть уникальным. Возможно, в русскоязычной документации по языку Swift вы встречали другое название наборов — *множества*.

Исходя из определения набора ясно, что он позволяет собрать множество уникальных значений в пределах одного хранилища.

Представьте, что вы предложили друзьям совместный выезд на природу. Каждый из них должен взять с собой одно-два блюда. Вы получаете сообщение от первого товарища, что он возьмет хлеб и овощи. Второй друг готов привезти тушенку и воду. Все поступившие значения вы помещаете в отдельный набор, чтобы избежать дублирования блюд. В вашем наборе уже 4 элемента: `"Хлеб"`, `"Овощи"`, `"Тушенка"` и `"Вода"`. Третий друг чуть позже сообщает, что готов взять мясо и овощи. Однако при попытке поместить в набор элемент `"Овощи"` возникнет исключительная ситуация, поскольку данный элемент уже присутствует в наборе. И правильно, зачем вам нужен второй набор овощей!

Набор создается с помощью *литерала набора*. В плане синтаксиса он идентичен литералу массива, но при этом гарантирует отсутствие дубликатов значений.

СИНТАКСИС

```
[значение_1, значение_2, ...,значение_N]
```

Литерал набора указывается в квадратных скобках, а значения отдельных элементов в нем разделяются запятыми. Литерал может содержать произвольное количество уникальных элементов одного типа.

ПРИМЕЧАНИЕ Тип данных элементов набора должен быть хешируемым, то есть поддерживать протокол Hashable.

При создании набора необходимо явно указать, что создается именно набор. Если переменной передать литерал набора, то Swift распознает в нем литерал массива и вместо набора будет создан массив. В связи с этим необходимо:

- либо явно указать тип данных набора с использованием конструкции `Set<ТипДанных>`, где `ТипДанных` — тип элементов создаваемого набора;
- либо использовать функцию `Set<ТипДанных>`, которой в качестве входного передается параметр `arrayLiteral`, содержащий перечень элементов набора.

Тип данных набора — `Set<ТипДанных>`.

Если нет необходимости явно указывать тип данных значений создаваемого набора, можно использовать ключевое слово `Set` без конструкции `<ТипДанных>`.

ПРИМЕЧАНИЕ Для создания неизменяемого набора используйте оператор let, в ином случае — оператор var.

В листинге 9.27 продемонстрированы все возможные способы создания наборов.

Листинг 9.27

```
/* набор, созданный путем
явного указания типа */
var dishes: Set<String> = ["хлеб", "овощи",     {"овощи", "тушенка",
"тушенка", "вода"]                               "вода", "хлеб"}
/* набор, созданный без явного
указания типа данных */
var dishesTwo: Set = ["хлеб", "овощи",          {"овощи", "тушенка",
"тушенка", "вода"]                               "вода", "хлеб"}
/* набор, созданный с помощью
функции при явном
```

```
9  указании типа данных*/
10 var members = Set<String>(arrayLiteral:        {"Энекин", "Оби Ван",
   "Энекин", "Оби Ван", "Йода")                  "Йода"}
11 /* набор, созданный с помощью
12 функции без явного указания
13 типа данных */
14 var membersTwo = Set(arrayLiteral: "Энекин",   {"Энекин", "Оби Ван",
"Оби Ван", "Йода")                               "Йода"}
```

В переменных `members`, `membersTwo`, `dishes`, `dishesTwo` хранятся наборы уникальных значений.

Создание пустого набора

Пустой набор, то есть набор, значение которого не имеет элементов, создается с помощью пустого литерала набора `[]`. Вы также можете передать данный литерал для удаления всех существующих элементов набора (листинг 9.28).

Листинг 9.28

```
1  // набор со значениями
2  var dishes: Set<String> = ["хлеб", "овощи"]    {"овощи", "хлеб"}
3  // создание пустого набора
4  var members = Set<String>()                    []
5  // удаление всех элементов набора
6  dishes = []                                    []
```

Доступ к набору и модификация набора

Так как набор — это неупорядоченная коллекция элементов, не имеющая каких-либо индексов или ключей, обеспечивающих доступ к значениям, то использование синтаксиса сабскриптов невозможно.

Для создания нового значения в наборе применяется метод `insert(_:)`, которому передается создаваемое значение (листинг 9.29).

Листинг 9.29

```
1  // создаем пустой набор
2  var musicStyleSet: Set<String> = []            []
3  // добавляем к нему новый элемент
4  musicStyleSet.insert("Jazz")                   "Jazz"
```

Для удаления элемента из набора используется метод `remove(_:)`, который удаляет элемент с указанным значением и возвращает удаленное значение или `nil`, если удаляемого элемента не существует.

Также вы можете задействовать метод `removeAll()` для удаления всех элементов набора (листинг 9.30).

Листинг 9.30

```
// создание набора со значениями
var musicStyleSet: Set<String> =        {"Jazz", "Hip-Hop", "Rock"}
["Jazz", "Hip-Hop", "Rock"]
// удаляем один из элементов
musicStyleSet.remove("Hip-Hop")         "Hip-Hop"
musicStyleSet                           {"Jazz", "Rock"}
// удаляем несуществующий элемент
musicStyleSet.remove("Classic")         nil
// удаляем все элементы набора
musicStyleSet.removeAll()               []
```

Проверка факта наличия значения в наборе осуществляется методом `contains(_:)`. Данный метод возвращает значение типа `Bool` в зависимости от результата проверки (листинг 9.31).

Листинг 9.31

```
// создаем набор
var musicStyleSet: Set<String> = ["Jazz", "Hip-Hop", "Rock", "Funk"]
// проверка существования значения в наборе
if musicStyleSet.contains("Funk") {
    print("ты любишь хорошую музыку")
} else {
    print("послушай то, что я слушаю")
}
```

Консоль:

```
ты любишь хорошую музыку
```

Базовые свойства и методы наборов

Ранее мы сравнивали наборы с математическими множествами. Различные наборы, как и множества, могут содержать пересекающиеся и непересекающиеся между собой значения. Swift позволяет получать значения таких наборов в зависимости от потребностей разработчика.

В листинге 9.32 создаются три различных целочисленных набора (рис. 9.1). Один из наборов содержит четные числа, второй — нечетные, третий — те и другие.

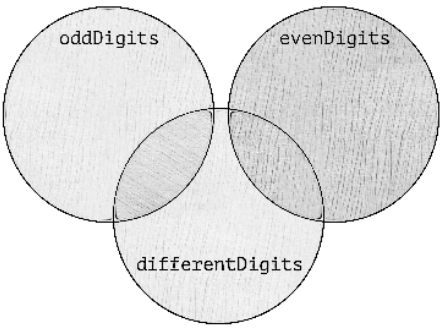


Рис. 9.1. Три набора целочисленных значений

Листинг 9.32

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

В наборах существуют как уникальные, так и общие элементы.

Для двух любых наборов можно произвести следующие операции (рис. 9.2):

- получить все общие для обоих наборов элементы (`intersect`);
- получить все непересекающиеся (не общие) для обоих наборов элементы (`exclusiveOr`);
- получить все элементы обоих наборов (`union`);
- получить разницу элементов, то есть элементы, которые входят в первый набор, но не входят во второй (`subtract`).

При использовании метода `intersect(_:)` создается новый набор, содержащий значения, общие для двух наборов (листинг 9.33).

Листинг 9.33

```
var inter = differentDigits.intersect(oddDigits).sort()    [3, 7]
```

Обратите внимание, что в данном примере впервые в одном выражении использована цепочка вызовов методов `intersect(_:)` и `sort()`.

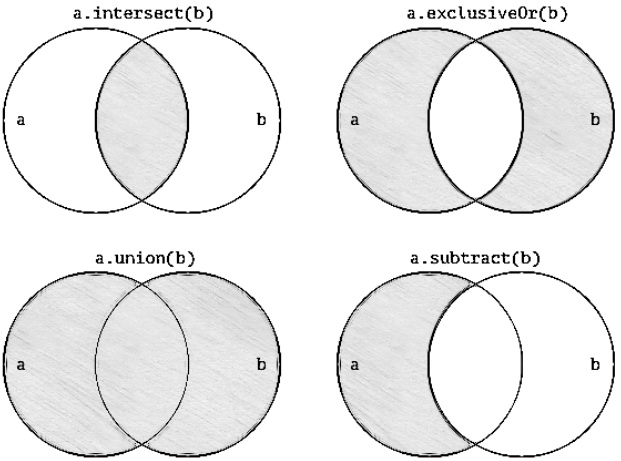


Рис. 9.2. Операции, проводимые с наборами

В результате вызова метода `sort()` возвращается отсортированный массив, в котором наименьшие значения располагаются первыми. Данный метод возвращает именно массив, так как набор — это неупорядоченная коллекция, где понятие порядка следования элементов отсутствует.

ПРИМЕЧАНИЕ 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) может быть произвольной.

Для получения всех непересекающихся значений служит метод `exclusiveOr(_:)`, представленный в листинге 9.34.

Листинг 9.34

```
1  var exclusive = differentDigits.exclusiveOr    [1, 4, 5, 8, 9]
   (oddDigits).sort()
```

Для получения всех элементов из обоих наборов применяется объединяющий метод `union(_:)`, как показано в листинге 9.35.

Листинг 9.35

```
1  var union = evenDigits.union(oddDigits).sort()      [0, 1, 2, 3, 4,
                                                       5, 6, 7, 8, 9]
```

Метод `subtract(_:)` возвращает все элементы первого множества, которые не входят во второе множество (листинг 9.36).

Листинг 9.36

```
1  var subtract = differentDigits.subtract(evenDigits).sort()   [3, 7]
```

Эквивалентность наборов

На рис. 9.3 изображены три набора: `aSet`, `bSet` и `cSet`. В наборах присутствуют как уникальные, так и общие элементы. Набор `aSet` — это супернабор для набора `bSet`, так как включает в себя все элементы из `bSet`. В то же время набор `bSet` — это субнабор (или поднабор) для `aSet`, так как все элементы `bSet` существуют в `aSet`. Наборы `cSet` и `bSet` являются непересекающимися, так как у них нет общих элементов, а наборы `aSet` и `cSet` — пересекающиеся, так как имеют общие элементы.

Два набора считаются эквивалентными, если у них один и тот же набор элементов. Эквивалентность наборов проверяется с помощью оператора эквивалентности (`==`), как показано в листинге 9.37.

Листинг 9.37

```
1  // создаем набор и его копию
2  var bSet: Set = [1, 3]
3  var copyOfBSet = bSet
4  /* в наборах bSet и copyOfBSet одинаковый состав
5  элементов. Проверим их эквивалентность */
6  if bSet == copyOfBSet {
7      print("Наборы эквивалентны")
8  }
```

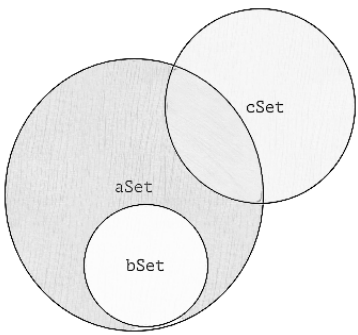


Рис. 9.3. Три набора значений с различными отношениями друг с другом

Консоль:
```
Наборы эквивалентны
```

Самое важное, чтобы при создании набора вы не забывали использовать ключевое слово `Set`, иначе в результате будет создан массив.

Метод `isSubsetOf(_:)` определяет, является ли один набор субнабором другого, как `bSet` для `aSet` (листинг 9.38).

Листинг 9.38
```
var aSet: Set = [1, 2, 3, 4, 5]
var bSet: Set = [1, 3]
if bSet.isSubsetOf(aSet) {
    print("bSet – это субнабор для aSet")
}
```

Консоль:
```
bSet – это субнабор для aSet
```

Метод `isSupersetOf(_:)` вычисляет, является ли набор супернабором для другого набора (листинг 9.39).

Листинг 9.39
```
var aSet: Set = [1, 2, 3, 4, 5]
var bSet: Set = [1, 3]
if aSet.isSupersetOf(bSet) {
    print("aSet – это супернабор для bSet")
}
```

Консоль:
```
aSet – это супернабор для bSet
```

Метод `isDisjointWith(_:)` определяет, существуют ли в двух наборах общие элементы, и в случае их отсутствия возвращает `true` (листинг 9.40).

Листинг 9.40
```
var bSet: Set = [1, 3]
var cSet: Set = [6, 7, 8, 9]
if bSet.isDisjointWith(cSet) {
    print("наборы bSet и cSet не пересекаются")
}
```

Консоль:
```
наборы bSet и cSet не пересекаются
```

Методы `isStrictSubsetOf(_:)` и `isStrictSupersetOf(_:)` определяют, является набор субнабором или супернабором, не равным указанному множеству (листинг 9.41).

Листинг 9.41

```
var aSet: Set = [1, 2, 3, 4, 5]
var bSet: Set = [1, 3]
if bSet.isStrictSubsetOf(aSet) {
    print("bSet - субнабор для aSet")
}
if aSet.isStrictSupersetOf(bSet) {
    print("aSet - супернабор для bSet")
}
```

Консоль:

```
bSet - субнабор для aSet
aSet - супернабор для bSet
```

9.3. Словари

Объявление словаря

Словарь — это неупорядоченная коллекция элементов одного и того же типа, для доступа к значениям которых используются ключи. Каждый элемент словаря состоит из уникального ключа, который указывает на данный элемент, и значения. В качестве ключа выступает не автоматически устанавливаемый индекс (как в массивах), а уникальный для словаря литерал произвольного типа, устанавливаемый разработчиком. Чаще всего ключи — это строковые литералы.

ПРИМЕЧАНИЕ Уникальные ключи словарей не обязаны иметь тип String. Основное требование, чтобы значение типа могло стать ключом, состоит в том, что данный тип должен быть хешируемым.

Другими словами, если каждый элемент массива — это пара «индекс-значение», то каждый элемент словаря — это пара «ключ-значение». Идея словарей в том, чтобы использовать уникальные произвольные ключи для доступа к значениям, при этом, как и в наборах, порядок следования элементов не важен.

Значение словаря устанавливается с помощью литерала словаря.

СИНТАКСИС

```
[ключ_1:значение_1, ключ_2:значение_2, ...,
ключ_N:значение_N]
```

Литерал словаря описывает элементы словаря. Он записывается в квадратных скобках, а указанные в нем элементы разделяются запятыми. Каждый элемент — это пара «ключ-значение», где ключ отделен от значения двоеточием.

ПРИМЕЧАНИЕ Для создания неизменяемого словаря используйте оператор let, в противном случае — оператор var.

Пример создания словаря приведен в листинге 9.42.

Листинг 9.42

```
1  var dictionary = ["one":"один", "two":"два", "three":"три"]
```

Словарь `dictionary` содержит три элемента. Здесь `one`, `two` и `three` — это ключи, которые служат для доступа к значениям словаря. Типом данных ключей, как и типом данных значений словаря, является `String`.

При попытке создания словаря с двумя одинаковыми ключами Xcode сообщит об ошибке.

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 9.43).

Листинг 9.43

```
1  var countryDict = ["RUS":"Россия", "BEL":      ["BEL": "Беларусь",
   "Беларусь", "UKR":"Украина"]                   "UKR": "Украина",
                                                  "RUS": "Россия"]

2  // получаем значение элемента
3  countryDict["BEL"]                             "Беларусь"
4  // изменяем значение элемента
5  countryDict["RUS"] = "Российская Федерация"    "Российская
                                                  Федерация"
6  countryDict                                    ["BEL": "Беларусь",
                                                  "UKR": "Украина",
                                                  "RUS": "Российская
                                                  Федерация"]
```

При изменении значения с использованием сабскрипта Swift возвращает устаналиваемое значение этого элемента.

В результате исполнения данного кода словарь `countryDict` получает измененное значение элемента с ключом `RUS`.

Для обновления значения элемента словаря можно также использовать метод `updateValue(value:forKey:)`. Как показано в листинге 9.44, при установке нового значения данный метод возвращает опциональное старое значение (или `nil`, если значения по изменяемому ключу не существует).

Листинг 9.44

```
1  var countryDict = ["RUS":"Россия", "BEL":      ["BEL": "Беларусь",
   "Беларусь", "UKR":"Украина"]                   "UKR": "Украина",
                                                  "RUS": "Россия"]
2  // изменяем значение элемента
3  countryDict.updateValue(value: "Российская     "Россия"
   Федерация", forKey: "RUS")
4  countryDict                                    ["BEL": "Беларусь",
                                                  "UKR": "Украина",
                                                  "RUS": "Российская
                                                  Федерация"]
```

Для изменения значения в метод `updateValue` передается новое значение элемента и параметр `forKey`, в значении которого указан ключ изменяемого элемента.

В отличие от варианта с использованием сабскрипта, данный метод возвращает не новое, а старое значение элемента словаря.

Для того чтобы создать новый элемент в словаре, достаточно задействовать сабскрипт с указанием на новый несуществующий ключ, передав ему требуемое значение (листинг 9.45).

Листинг 9.45

```
1  var countryDict = ["RUS":"Россия", "BEL":      ["BEL": "Беларусь",
   "Беларусь", "UKR":"Украина"]                   "UKR": "Украина",
                                                  "RUS": "Россия"]
2  // создание нового элемента словаря
3  countryDict["TUR"] = "Турция"                  "Турция"
4  countryDict                                    ["BEL": "Беларусь",
                                                  "UKR": "Украина",
                                                  "RUS": "Россия",
                                                  "TUR": "Турция"]
```

Для удаления некоторого элемента (пары «ключ-значение») достаточно присвоить удаляемому элементу `nil` или использовать метод `removeValueForKey`, указав ключ элемента (листинг 9.46).

Листинг 9.46

```
1  var countryDict = ["RUS":"Россия", "BEL":      ["BEL": "Беларусь",
   "Беларусь", "UKR":"Украина"]                    "UKR": "Украина",
                                                   "RUS": "Россия"]
2  // удаление элемента словаря
3  countryDict["UKR"] = nil                        nil
4  countryDict.removeValueForKey("BEL")            "Беларусь"
5  countryDict                                     ["RUS": "Россия"]
```

При использовании метода `removeValueForKey` возвращается значение удаляемого элемента.

Если вы попытаетесь получить доступ к несуществующему элементу словаря, это не приведет к ошибке — Swift просто вернет `nil`. Это говорит о том, что любое возвращаемое значение элемента словаря — опционал (листинг 9.47).

Листинг 9.47

```
1  var countryDict = ["RUS":"Российская            ["BEL": "Беларусь",
   Федерация", "BEL":"Беларусь", "UKR":"Украина"]  "UKR": "Украина",
                                                   "RUS": "Российская
                                                   Федерация"]
2  // получим значение элемента
3  let myCountry: String = countryDict["RUS"]!     "Российская
                                                   Федерация"
```

Для преобразования возвращаемого значения элемента `countryDict["RUS"]` из типа `String?` в тип `String` выполняется принудительное извлечение значения.

Явное указание типа данных словаря

Тип данных элементов словаря содержит в себе два типа данных: тип ключа и тип значения. Так же как и при работе с массивами и наборами, вы можете явно указать значение типа данных коллекции, но при этом необходимо задать типы данных и для ключа, и для значения.

СИНТАКСИС

```
var имя_словаря: Dictionary<ТипКлюча:ТипЗначения>
var имя_словаря: [ТипКлюча:ТипЗначения]
```

Тип словаря в этом случае равен [ТипКлюча:ТипЗначения] (с квадратными скобками) или Dictionary<ТипКлюча:ТипЗначения >. Оба варианта указания типа данных равнозначны.

Объявленный словарь, прежде чем с ним можно будет взаимодействовать, должен быть инициализирован.

Создание пустого словаря

Для того чтобы создать пустой словарь, он должен быть инициализирован значением без элементов. Это делается с помощью конструкции `[:]`, которая как раз и является литералом словаря, не имеющего элементов (листинг 9.48).

Листинг 9.48

```
var emptyDictionary: [String:Int] = [:]                    [:]
var AnotherEmptyDictionary: Dictionary<String,Int> = [:]   [:]
```

С помощью конструкции `[:]` можно также уничтожить все элементы словаря, если присвоить ее словарю в качестве значения (листинг 9.49).

Листинг 9.49

```
var countryDict = ["RUS":"Российская Федерация","BEL":"Беларусь", "UKR":"Украина"]
                                                   ["BEL": "Беларусь", "UKR": "Украина", "RUS": "Российская Федерация"]
countryDict = [:]                                  [:]
countryDict                                        [:]
```

ПРИМЕЧАНИЕ Значения в словаре хранятся вовсе не в том порядке, в каком вы их туда помещали. Словари — это не массивы, они не являются упорядоченными коллекциями. Вы не можете добавить элемент в конец словаря, вы просто добавляете новый элемент, а Swift самостоятельно решает, на какую позицию в данном словаре его поместить.

Базовые свойства и методы словарей

Словари, как и массивы с наборами, имеют большое количество свойств и методов. С некоторыми из них вы уже познакомились. Наиболее важные из оставшихся мы сейчас рассмотрим.

Свойство `count` возвращает количество элементов в словаре (листинг 9.50).

Листинг 9.50

```
var someDictionary = [1, 2, 3, 4, 5]          [1, 2, 3, 4, 5]
// количество элементов в словаре
someDictionary.count                          5
```

Если свойство `count` равно нулю, то свойство `isEmpty` возвращает `true` (листинг 9.51).

Листинг 9.51

```
var someDictionary: [String:Int] = [:]        [:]
someDictionary.count                          0
someDictionary.isEmpty                        true
```

При необходимости вы можете получить все ключи или все значения словаря с помощью свойств `keys` и `values` (листинг 9.52).

Листинг 9.52

```
var countryDict = ["RUS":"Российская        ["BEL": "Беларусь",
Федерация", "BEL":"Беларусь", "UKR":"Украина"] "UKR": "Украина",
                                               "RUS": "Российская
                                               Федерация"]

// все ключи словаря countryDict
var keys = countryDict.keys                   LazyMapCollection
                                              <Dictionary<String,
                                              String>, String>

// все значения словаря countryDict
var values = countryDict.values               LazyMapCollection
                                              <Dictionary<String,
                                              String>, String>
```

При вызове свойства `keys` или `values` Swift возвращает не массив, набор или словарь, а значение некоего типа `LazyMapCollection`. С подобным поведением мы встречались ранее, когда использовали функцию `reverse()` для изменения порядка следования элементов массива на обратный. О том, что это такое, мы поговорим в следующей главе.

Задание

1. Создайте псевдоним Chessman для типа словаря `[String:(alpha:Character,num:Int)?]`. Данный тип описывает шахматную фигуру на игровом поле. В ключе словаря должно храниться имя фигуры, например «Белый король», а в значении — кортеж, указывающий на координаты фигуры на игровом поле. Если вместо кортежа находится `nil`, это означает, что фигура убита (не имеет координат на игровом поле).
2. Создайте переменный словарь `Chessmans` типа `Chessman` и добавьте в него три произвольные фигуры, одна из которых не должна иметь координат.

3. Создайте конструкцию `if-else`, которая проверяет, убита ли переданная ей фигура (элемент словаря `Chessmans`), и выводит на консоль информацию либо о координатах фигуры, либо о ее отсутствии на игровом поле.
4. Для получения координат переданной фигуры используйте опциональное связывание.
5. Сохраните данную программу, так как мы вернемся к ней в последующем.

10 Циклы

Ранее мы начали изучать тему управления потоком, в которой рассмотрели операторы, позволяющие менять выполнение кода в зависимости от условий, возникших в ходе работы программы. В данной главе описываются механизмы, позволяющие циклично выполнять различные блоки кода и управлять данными циклами.

Цикличное выполнение кода обеспечивается *операторами повторения*. Swift предлагает всего три таких оператора: `for`, `while` и `repeat while`. Каждый из них имеет разные возможности.

Код может повторяться либо до выполнения какого-то условия, либо определенное количество раз. Каждое следующее выполнение блока кода называется *итерацией*.

10.1. Оператор повторения for

Оператор `for` предназначен для цикличного выполнения блоков кода. Swift предлагает одну форму использования данного оператора: `for-in`.

ПРИМЕЧАНИЕ В Swift 2.2 (а также в более ранних версиях языка) существует две формы оператора повторения: for и for-in.
Первая форма данного оператора имеет следующий синтаксис:

```
for стартовое_выражение; условие_окончания; действие {
            блок_кода
        }
```

Если вы занимались программированием ранее, то наверняка знакомы с ней. Тем не менее разработчики Apple присвоили данной форме оператора статус «устаревшая» (deprecated) и не советуют ее использовать при разработке программ, так как из Swift 3 она будет полностью исключена.

Цикл for-in

СИНТАКСИС

```
for переменная in последовательность {
    блок_кода
}
```

Цикл for-in выполняет блок_кода для каждого элемента в последовательности. Перед каждой итерацией очередной элемент из последовательности присваивается переменной, которая доступна в блоке кода. После перебора всех элементов последовательности цикл завершает свою работу.

Тело цикла, содержащее выполняемый блок_кода, заключается в фигурные скобки.

Цикл `for-in`, так же как и цикл `for`, позволяет выполнить определенное количество итераций кода. При этом он обладает куда более широкими возможностями. Пример использования цикла приведен в листинге 10.1. Данный код складывает все числа от 1 до 10 и выводит итоговый результат в области результатов.

Листинг 10.1

```
1  var totalSum = 0                          0
2  for i in 1...10 {
3      totalSum += i                         (10 times)
4  }
5  totalNum                                  55
```

После оператора `for` указывается имя объявляемой переменной, в данном случае это `i`. Далее ей присваивается первое значение из диапазона `1...10`, то есть 1, и выполняется код тела цикла. В следующей итерации переменной `i` присваивается второе значение из диапазона и повторно выполняется тело цикла. И так далее.

Несмотря на то что в цикле создается новая переменная `i`, Swift не требует писать оператор `var`.

Переменная, которая создается в цикле `for-in`, является локальной для данного цикла. То есть если существует внешняя одноименная переменная или константа, то ее значение не будет пересекаться с локальной переменной и все изменения локального параметра никак не повлияют на внешний (листинг 10.2).

Листинг 10.2

```
1  var totalNum = 0                          0
2  var i = 0                                 0
```

```
3  for var i in 1...10 {
4      totalNum += i                          (10 times)
5  }
6  totalNum                                   55
7  i                                          0
```

Несмотря на то что в цикле значение локальной переменной `i` изменяется, значение глобальной переменной `i` остается прежним.

Возможна ситуация, когда вам необходимо пройти от большего числа к меньшему. Для этого не следует указывать диапазон вроде (10...1): код будет откомпилирован, но во время выполнения возникнет ошибка. Правильным решением будет использование метода `reverse()` (листинг 10.3).

Листинг 10.3

```
1  var totalSum = 0                           0
2  for i in (10...1).reverse() {
3      totalSum += i                          (10 times)
4  }
5  totalNum                                   55
```

Если необходимо пройти определенный числовой интервал с некоторым шагом, можно использовать два одноименных метода: `stride(through:by:)` и `stride(to:by:)`. Рассмотрим каждый из них (листинг 10.4).

Листинг 10.4

```
1  for i in 1.stride(through:5, by:2) {
2  print(i)
3  }
4  for i in 1.stride(to:5, by:2) {
5  print(i)
6  }
```

Консоль:

```
1
3
5
1
3
```

Как вы можете видеть из вывода в консоли, в первом случае аргумент `through` содержит значение, включаемое в диапазон (от 1 до 5 с ша-

гом 2). Во втором случае аргумент `to` содержит не включаемое в диапазон число (от 1 до 4 с шагом 2).

С помощью цикла `for-in` очень удобно перебирать значения коллекций. Для этого требуется передать имя коллекции после ключевого слова `in` (листинг 10.5).

Листинг 10.5

```
1  var myMusicStyles = ["Rock", "Jazz", "Pop"]
2  for musicName in myMusicStyles {
3      print("Я люблю \(musicName)")
4  }
```

Консоль:

```
Я люблю Rock
Я люблю Jazz
Я люблю Pop
```

В результате переменная `musicName` получит по очереди каждое из значений, записанных в массив `myMusicStyles`.

Но что делать, если требуется получить все элементы не массива или набора, а словаря? Для этого необходимо использовать уже знакомые нам кортежи в качестве изменяемого параметра цикла (листинг 10.6).

Листинг 10.6

```
1  var countrysAndBlocks = ["Россия": "ЕАЭС", "США": "НАТО",
   "Франция":"ЕС"]
2  for (countryName, blockName) in countrysAndBlocks {
3      print("\(countryName) вступила в \(blockName)")
4  }
```

Консоль:

```
Россия вступила в ЕАЭС
Франция вступила в ЕС
США вступила в НАТО
```

Как видите, данный способ намного удобнее использования сложных конструкций в цикле `for`, и работа с коллекциями в цикле теперь доставляет одно удовольствие!

Возможна ситуация, когда требуется получить не пару «ключ-значение» из словаря, а только ключ или только значение. Для этого в кортеже на месте того элемента, который загружать не следует, необходимо вставить символ нижнего подчеркивания (листинг 10.7).

Листинг 10.7

```
var countrysAndBlocks = ["Россия": "ЕАЭС", "США": "НАТО",
"Франция":"ЕС"]
for (countryName, _) in countrysAndBlocks {
    print("страна – \(countryName)")
}
```

Консоль:

```
страна – Россия
страна – Франция
страна – США
```

С данным приемом вы познакомились еще во время изучения кортежей.

В Swift существует специальный метод `enumirate()`, который позволяет преобразовать массив таким образом, чтобы с помощью цикла `for-in` получить в виде кортежа каждую отдельную пару «индекс-значение» (листинг 10.8).

Листинг 10.8

```
var myMusicStyles = ["Rock", "Jazz", "Pop"]
for (index, musicName) in myMusicStyles.enumerate() {
    print("\(index+1). Я люблю \(musicName)")
}
```

Консоль:

```
1. Я люблю Rock
2. Я люблю Jazz
3. Я люблю Pop
```

Строка (то есть значение типа `String`) — это набор символов, который может быть представлен в виде коллекции. Для этого служит свойство `characters`. В таком виде коллекция символов может обрабатываться посимвольно с помощью цикла `for-in`. В листинге 10.9 значение типа `String` представляется в виде коллекции символов, и элементы этой коллекции по одному выводятся на консоль.

Листинг 10.9

```
let myName = "Troll"
for oneChar in myName.characters {
    print(oneChar)
}
```

Консоль:
```
T
r
o
l
l
```

Для обработки многомерных конструкций вроде вложенных коллекций вы можете вкладывать одни циклы в другие. В листинге 10.10 мы создадим словарь, который содержит результаты игр хоккейной команды в чемпионате. Ключ каждого элемента — это название команды соперника, а значение каждого элемента — массив результатов игр с командой, указанной в ключе. На консоль выводятся результаты всех игр с указанием команд и итогового счета.

Листинг 10.10
```
1  // словарь с результатами игр
2  var resultsOfGames = ["Red Wings":["2:1","2:3"],
   "Capitals":["3:6","5:5"],"Penguins":["3:3","1:2"]]
3  // обработка словаря
4  for (teamName, results) in resultsOfGames {
5  // обработка массива результатов
6      for oneResult in results {
7          print("Игра с \(teamName) - \(oneResult)")
8      }
9  }
```

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
```

Типом массива `resultsOfGames` является «словарь словарей» `Dictionary<String: Dictionary <String>>`. Переменная `teamName` локальна, но в ее область видимости попадает вложенный цикл `for-in`, поэтому в данном цикле можно использовать эту переменную для вывода ее значения.

10.2. Операторы повторения while и repeat while

Операторы `while` и `repeat while` позволяют выполнять блок кода до тех пор, пока проверяемое условие истинно. То есть в некотором смысле это объединенные операторы `for` и `if`.

Цикл while

СИНТАКСИС

```
while условие {
   блок_кода
}
```

Цикл начинается с оператора `while`, после которого указывается проверяемое условие.

Каждая итерация начинается с проверки условия. Если оно возвращает true, то выполняется блок_кода. Далее стартует следующая итерация. И так далее.

ВНИМАНИЕ Будьте осторожны при задании условия, поскольку по невнимательности можно указать такое условие, которое никогда не вернет false. В этом случае цикл будет выполняться бесконечно, что, вероятно, приведет к зависанию программы.

Пример реализации цикла `while` приведен в листинге 10.11, в котором складываются все числа от 1 до 10 и выводится результат.

Листинг 10.11

```
1  // начальное значение
2  var i = 1                                    1
3  // хранилище результата сложения
4  var resultSum = 0                            0
5  // цикл для подсчета суммы
6  while i <= 10 {
7      resultSum += i                           (10 times)
8      i++                                      (10 times)
9  }
10 resultSum                                    55
```

Переменная `i` является счетчиком в данном цикле. Именно по ее значению определяется необходимость выполнения тела 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 код.

Цикл repeat while

В противоположность оператору `while` оператор `repeat while` является циклом с последующей проверкой условия. В таком цикле сначала происходит выполнение кода, а уже потом проверяется условие.

ПРИМЕЧАНИЕ В первой версии Swift цикл repeat while назывался do while.

СИНТАКСИС

```
repeat {
    блок_кода
} while условие
```

Цикл начинается с оператора repeat, за которым следует тело цикла. В конце цикла пишется оператор while и условие выполнения цикла.

Пример реализации цикла `repeat while` приведен в листинге 10.12, в котором складываются все числа от 1 до 10 и выводится результат.

Листинг 10.12

```
// начальное значение
var i = 1                              1
// хранилище результата сложения
var resultSum = 0                      0
// цикл для подсчета суммы
repeat{
    resultSum += i                     (10 times)
    i++                                (10 times)
} while i <= 10
resultSum                              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.

10.3. Управление циклами

В Swift, по аналогии с другими языками программирования, есть два оператора, способных влиять на ход работы циклов, — это операторы `break` и `continue`.

Оператор continue

Оператор `continue` предназначен для перехода к очередной итерации, игнорируя следующий за ним код. В листинге 10.13 представлена программа, в которой переменная поочередно принимает значения от 1 до 10, причем когда значение нечетное, оно выводится на консоль.

Листинг 10.13

```
for i in 1...10 {
    if i%2 == 0 {
     continue
    } else {
     print(i)
    }
}
```

Консоль:

```
1
3
5
7
9
```

Проверка четности значения 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`.

Оператор break

Оператор `break` предназначен для досрочного завершения работы цикла. При этом весь последующий код в теле цикла игнорируется. В листинге 10.14 десять раз 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 10.14

```
import Foundation
for i in 1...10 {
    var randNum = Int(arc4random_uniform(10))
    if randNum == 5 {
        print("Итерация номер \(i)")
        break
    }
}
```

Консоль:

```
Итерация номер 7
```

Вывод в консоль в вашем случае может отличаться от того, что приведен в примере, так как используется генератор случайных чисел.

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.

В данном примере подгружается библиотека Foundation для обеспечения доступа к функции arc4random_uniform(). Данная функция предназначена для генерации случайного числа. Если убрать строку import Foundation, то Xcode сообщит о том, что функции arc4random_uniform() не существует.

Подробнее об использовании команды import и существующих библиотеках функции будет рассказано в главе 26.

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 типу `Int` с помощью соответствующей функции.

ПРИМЕЧАНИЕ Все создаваемые внутри цикла переменные и константы являются локальными для текущей итерации, то есть в следующей итерации данная переменная будет недоступна.

Может возникнуть ситуация, когда из внутреннего цикла необходимо прервать выполнение внешнего, — для этого в Swift существуют метки (листинг 10.15).

Листинг 10.15

```
1  mainLoop: for i in 1...5 {
2      for y in i...5 {
3          if y == 4 && i == 2{
4              break mainLoop
5          }
6          print("\(i) - \(y)")
7      }
8  }
```

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.

Для того чтобы изменить ход работы внешнего цикла, после оператора `break` или `continue` необходимо указать имя метки.

Задание 1

Представьте, что вы являетесь преподавателем курсов по шахматам. Ваши занятия посещают три ученика.

1. Создайте словарь, который будет содержать информацию о ваших студентах и об их успехах. Ключом словаря должна быть фамилия, а значением — другой словарь, содержащий дату занятия и полученную на этом занятии оценку.

   Тип данных словаря должен быть `[String:[String:UInt]]`.

   В вашем электронном журнале должно находиться по две оценки для каждого из трех учеников. Фамилии, даты занятий и оценки придумайте сами.

2. Посчитайте средний балл каждого студента и средний балл всей группы целиком и выведите всю полученную информацию на консоль.

Задание 2

Вернемся к заданию из главы 9, в котором вы описывали шахматную фигуру и создавали конструкцию `if-else`, проверяющую наличие фигуры на игровом поле.

Вам необходимо доработать данную программу таким образом, чтобы она автоматически анализировала не одну переданную ей фигуру, а все фигуры, хранящиеся в переменной `Chessmans`.

11 Функции

Ранее в процессе разбора программ из листингов мы неоднократно встречались с функциями. Все использованные функции были результатом трудов разработчиков языка Swift. В данной главе вы научитесь создавать функции самостоятельно в соответствии со своими потребностями.

Функция — это именованный фрагмент программного кода, к которому можно многократно обращаться. Функции позволяют избежать дублирования кода за счет его группировки. Это очень полезный инструмент Swift. Уверен, что вы уже знакомились с функциями в других языках программирования. Каких-либо кардинальных отличий в Swift нет, но есть ряд нюансов, о которых следует знать.

11.1. Объявление функций

Функции помогают структурировать код, объединяя его повторяющиеся блоки для их многократного использования. Функции позволяют следовать принципу «не писать один и тот же код дважды».

ПРИМЕЧАНИЕ У программистов существует шутка, что «любой код мечтает стать функцией».

СИНТАКСИС

```
func имяФункции (входные_параметры) ->
            ТипВозвращаемогоЗначения {
    тело_функции
}
```

Объявление функции начинается с ключевого слова func, за которым следует имя создаваемой функции. Далее в скобках указываются входные параметры, или аргументы, затем после стрелки — тип возвращаемого значения и, наконец, — в фигурных скобках тело функции, то есть блок кода, который содержит в себе всю логику ее работы.

Имя функции используется при каждом ее вызове в вашем коде. Имя функции должно быть записано в нижнем стиле камэлкейс. Например:

```
func myFirstFunc
```

Список входных параметров заключается в круглые скобки и состоит из разделенных запятыми элементов. Каждый отдельный элемент описывает один входной параметр и состоит из имени и типа этого параметра, разделенных двоеточием. Входные параметры, или аргументы, позволяют передать в функцию значения, которые ей требуются. Указанные параметры являются локальными для тела функции, то есть они доступны только в теле функции. Количество входных параметров может быть произвольным (также они могут отсутствовать). Например:

```
func myFirstFunc
 (someValue: Int, anotherValue: String)
```

После списка параметров следует стрелка (`->`), за которой указывается тип данных возвращаемого функцией значения. В качестве типа данных вы можете задать как фундаментальный тип, так и тип массива или кортежа. Например:

```
func myFirstFunc
 (someValue: Int, anotherValue: String)
-> String
```

Или:

```
func myFirstFunc
 (someValue: Int, anotherValue: String)
-> [(String,Int)?]
```

Тело функции заключается в фигурные скобки и содержит в себе всю логику ее 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` происходит завершение работы функции. Например:

```
func myFirstFunc
 (someValue: Int, anotherValue: String)
-> String {
    return String(someValue) + anotherValue
}
```

В представленных примерах объявление функции разнесено на разные строки для удобства восприятия кода. Вам не обязательно делать это, можете писать любые элементы объявления функции в одну строку. Например:

```
func myFirstFunc (someValue: Int) -> String {
    return String(someValue)
}
```

Объявим простейшую функцию, которая не имеет входных и выходных значений. Представьте, что при наступлении некоторого события вам необходимо выводить на консоль сообщение. Реализуем этот механизм с помощью функции (листинг 11.1).

Листинг 11.1

```
1  func printMessage() -> Void {
2      print("Сообщение принято")
3  }
4  // вызываем функцию по ее имени
5  printMessage()
```

Консоль:

```
Сообщение принято
```

Для вывода текста на консоль вы вызываете функцию `printMessage()`, просто написав ее имя с круглыми скобками. Функция `printMessage()` не имеет каких-либо входных параметров и возвращаемого значения. Она всего лишь выводит на консоль текстовое сообщение.

Если функция не возвращает никакого значения, то в качестве типа данных возвращаемого значения необходимо указать тип `Void`. Подобный С-образный подход используется во многих языках.

Существует альтернативная ключевому слову `Void` форма записи — пустые скобки, которые также сообщают о том, что функция не возвращает значение. Например:

```
func printMessage() -> () {
    print("Сообщение принято")
}
```

Кроме того, если функция не возвращает никакого значения, написание какой бы то ни было конструкции можно вовсе опустить. Например:

```
func printMessage() {
    print("Сообщение принято")
}
```

Процесс обращения к функции называется *вызовом функции*.

11.2. Входные параметры и возвращаемое значение

Функция может принимать некоторые аргументы в качестве входных значений и возвращать некоторый результат своей работы. При этом вывод информации на консоль не является возвращаемым значением.

Предположим, что вам необходимо написать функцию, которая складывает два значения, потом делит их на число π (3,14) и возвращает результат. В этой функции входными параметрами будут два неизвестных заранее числа. Не имеет никакого смысла передавать в качестве входного аргумента и число π, так как оно заранее известно и его можно жестко зафиксировать в теле функции. Результат вычислений как раз и будет возвращаемым значением функции.

Простые входные и выходные параметры

Напишем функцию, которая имеет один входной параметр, используемый в ее теле (листинг 11.2). В качестве аргумента функция принимает код ответа от сервера и выводит на консоль строку, содержащую вспомогательную информацию.

Листинг 11.2

```
1  func printCodeMessage(requestCode: Int) -> () {
2      print("Код ответа - \(requestCode)")
3  }
4  // вызываем функцию
5  printCodeMessage(200)
6  printCodeMessage(404)
```

Консоль:

```
Код ответа - 200
Код ответа - 404
```

Функция `printCodeMessage(_:)` имеет один входной параметр — `requestCode`. Имя данного параметра используется внутри тела функции для генерации выводимого строкового литерала.

Все находящиеся в списке входных аргументов параметры должны в обязательном порядке иметь значения в самом начале работы функции. Для этого значения этих параметров необходимо передавать в функцию при ее вызове. Они передаются в том же порядке, в каком описывались при объявлении функции. При этом только для первого параметра нет необходимости указывать имя, значения всех остальных параметров должны иметь имя входного аргумента, которому они присваиваются. В листинге 11.3 мы реализуем функцию, которая принимает на входе три значения, складывает их и выводит результат на консоль.

Листинг 11.3

```
1  func sum(a:Int, b: Int, c:Int) -> () {
2      print("Сумма - \(a+b+c)")
3  }
4  sum(10, b: 51, c: 92)
```

Консоль:

```
Сумма - 153
```

При вызове функции имя аргумента `a` не указывается, но указываются имена для всех оставшихся входных параметров. При этом все параметры передаются точно в том порядке, в каком они были описаны.

ПРИМЕЧАНИЕ Имена каких параметров необходимо указывать, вам подскажет Xcode, а точнее — механизм автозавершения кода. С ним мы знакомились в первой части книги.

Существует способ сообщить Swift, что для данной функции нет необходимости указывать имена входных аргументов при ее вызове. Для этого перед именем такого параметра необходимо поставить символ нижнего подчеркивания, отделив его от имени пробелом (листинг 11.4).

Листинг 11.4

```
1  func sum(a:Int, _ b: Int, c:Int) -> () {
2      print("Сумма - \(a+b+c)")
3  }
4  sum(15, 12, c: 9)
```

Консоль:

```
Сумма - 36
```

Как видите, в функции `sum(_:_:c:)` перед аргументом `b` стоит символ нижнего подчеркивания и при вызове функции указывается имя только для параметра `c`.

Переменные копии 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 11.5).

Листинг 11.5

```
1  func generateString(code: Int, _ text: String)
2   -> String {
3      var mutableText = text
   mutableText += String(code)
4      return mutableText
5  }
6  generateString(200, "Код:")                    "Код:200"
```

Функция `generateString(_:_:)` принимает на входе параметр `text`. В пределах функции создается переменная `mutabletext`, значение которой изменяется внутри самой функции.

ПРИМЕЧАНИЕ В Swift 2.1 и ранее для создания переменной копии параметра вы могли использовать ключевое слово var в перечне аргументов перед именем вводного параметра. Несмотря на то что в Swift 2.2 данный механизм будет работать, он имеет статус «устаревший» и будет полностью удален после выпуска Swift 3.

Сквозные параметры

Приведенный способ модификации значений аргументов позволяет получать доступ к изменяемому значению только в пределах тела самой функции. Для того чтобы входные аргументы сохранили свои значения даже после завершения вызова функции, необходимо использовать *сквозные параметры*.

Чтобы преобразовать входной параметр в сквозной, перед его описанием необходимо указать модификатор `inout`. Сквозной параметр передается в функцию, изменяется в ней и возвращается из функции, заменяя собой исходное значение входного аргумента. При вызове функции перед передаваемым значением аргумента необходимо

ставить символ амперсанда (&), указывающий на то, что параметр передается по ссылке. Функция в листинге 11.6 получает на входе два параметра и меняет их значения местами.

Листинг 11.6

```
1  func changeValues(inout a: Int, inout _ b: Int) -> () {
2      let tmp = a
3      a = b
4      b = tmp
5  }
6  var a = 150, b = 45
7  changeValues(&a, &b)
8  a                                                    45
9  b                                                    150
```

Консоль:

```
Код ответа – 200
Код ответа – 404
```

Функция принимает на входе две константы: a и b. Эти константы передаются в функцию как сквозные параметры, что позволяет изменять значения внешних переданных параметров внутри функции с сохранением их значений после завершения ее работы.

ПРИМЕЧАНИЕ Аргументом сквозного параметра может быть только переменная. Константы или литералы нельзя передавать, так как они являются неизменяемыми.

Функция в качестве значения аргумента

Вы можете использовать возвращаемое некоторой функцией значение в качестве значения входного аргумента другой функции. Таким образом, если какому-либо оператору или функции в качестве входного аргумента требуется значение определенного типа, то вы можете передать в качестве аргумента непосредственно саму функцию (листинг 11.7).

Листинг 11.7

```
1  func generateString(code: Int, message: String) -> String {
2      let returnMessage = "Получено сообщение "\(message)"
       с кодом \(code)"
3      return returnMessage
4  }
5  // используем функцию в качестве значения
6  print(generateString(200, message: "Сервер доступен"))
```

Консоль:
```
Получено сообщение "Сервер доступен" с кодом 200
```

Функция `generateString(_:message:)` возвращает значение типа `String`. В связи с этим в конце функции используется ключевое слово `return` с указанием возвращаемого параметра.

Уже известная нам функция `print(_:)` принимает на входе строковый литерал типа `String`, который выводится на консоль. Так как первая функция возвращает значение того же типа, что принимает на входе вторая, то можно указать в качестве входного аргумента функции `print(_:)` имя первой функции.

Входной параметр с переменным числом аргументов

В некоторых ситуациях необходимо, чтобы функция получала неизвестное заранее число однотипных аргументов. Для этого вы можете либо принимать массив значений, либо создать вариативный параметр, то есть параметр с переменным числом аргументов.

Вариативный параметр обозначается в списке входящих параметров указанием оператора закрытого диапазона (`...`) сразу после типа входного параметра. Значения для этого параметра при вызове функции задаются через запятую.

Рассмотрим пример из листинга 11.8. Представьте, что удаленный сервер на каждый запрос отправляет вам несколько ответов. Каждый ответ — это целое число, но количество ответов заранее неизвестно. Вам необходимо написать функцию, которая принимает на входе все полученные ответы и выводит их на консоль.

Листинг 11.8
```
func printRequestString(codes: Int...) -> () {
    var codesString = ""
    for oneCode in codes {
        codesString += String(oneCode) + " "
    }
    print("Получены ответы - \(codesString)")
}
printRequestString(600, 800, 301)
printRequestString(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 массив значений, поэтому его можно обработать в цикле `for-in`.

У одной функции может быть только один вариативный параметр. Он должен находиться в самом конце списка входных параметров.

Кортеж в качестве возвращаемого значения

При знакомстве с кортежами мы говорили о том, что их сильной стороной является возможность группировки нескольких значений в одном. Эту группу значений можно использовать в качестве возвращаемого значения функции. Представленная в листинге 11.9 функция принимает на входе код статуса ответа сервера и в зависимости от того, к какому диапазону относится переданный код, возвращает кортеж с его описанием.

Листинг 11.9

```
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)
}
print(getCodeDescription(150))
```

Консоль:

```
(150, "Correct")
```

В качестве типа возвращаемого значения функции `getCodeDescription(_:)` указан тип кортежа, содержащего два значения: код и его описание.

Функцию `getCodeDescription(_:)` можно улучшить, если указать не просто тип возвращаемого кортежа, а названия его элементов (листинг 11.10).

Листинг 11.10

```
func getCodeDescription(code: Int)
 -> (code: Int, description: String){
```

```
3       let description: String
4       switch code {
5           case 1...100:
6               description = "Error"              "Error"
7           case 101...200:
8               description = "Correct"
9           default:
10              description = "Unknown"
11      }
12      return (code, description)                 (.0 45, .1 "Error")
13  }
14  let request = getCodeDescription(48)           (.0 45, .1 "Error")
15  request.description                            "Error"
16  request.code                                   45
```

Полученное в ходе работы функции `getCodeDescription(_:)` значение записывается в константу `request`, у которой появляются свойства `description` и `code`, что соответствует именам элементов возвращаемого кортежа.

Рассмотрим реальный пример, для которого может потребоваться написание функции. Предположим, что у вас есть виртуальный кошелек. В любой момент времени в нем находятся купюры различного достоинства: от 50 до 5000 рублей. Вам необходимо написать функцию, которая будет подсчитывать общее количество денег в кошельке (листинг 11.11).

Листинг 11.11

```
1  func sumWallet( wallet: [Int] ) -> Int {     (2 times)
2      var sum = 0
3      for oneBanknote in wallet {
4          sum += oneBanknote                    (19 times)
5      }
6      return sum                                (2 times)
7  }
8  // кошелек с купюрами
9  var wallet = [50, 100, 100, 500, 50, 1000,   [50, 100, 100, 500, 50,
   5000, 50, 100]                                1 000, 5 000, 50, 100]
10 // сосчитаем сумму всех купюр
11 sumWallet(wallet)                             6 950
12 // добавим новую купюру
13 wallet.append(1000)                           [50, 100, 100, 500,
                                                 50, 1 000, 5 000, 50,
                                                 100, 1 000]
14 // снова сосчитаем сумму
15 sumWallet(wallet)                             7 950
```

В любой момент вы можете вызвать функцию `sumWallet(_:)` и передать в нее массив-кошелек. В результате будет возвращено значение типа `Int`, сообщающее о сумме денег в кошельке.

Значения по умолчанию для аргументов

Для любого входного параметра можно указать значение по умолчанию, то есть то значение, которое будет присвоено параметру, если для него не передано какое-либо входное значение. Давайте доработаем функцию `sumWallet(_:)` из предыдущего листинга для корректной отработки ситуации, когда в нее не передан какой-либо входной кошелек (листинг 11.12).

Листинг 11.12

```
func sumWallet( wallet: [Int]? = nil ) -> Int? {
    var sum = 0
    if wallet == nil {
        return nil
    }
    for oneBanknote in wallet! {
        sum += oneBanknote
    }
    return sum
}
// кошелек с купюрами
var wallet = [50, 100, 100, 500, 50, 1000,      [50, 100, 100,
5000, 50, 100]                                  500, 50, 1 000,
                                                5 000, 50, 100]
// сосчитаем сумму всех купюр
sumWallet(wallet)                               6 950
sumWallet()                                     nil
```

Как вы можете видеть, в рассматриваемой функции появилось сразу несколько нововведений. Разберем каждое из них по отдельности.

Тип данных входного параметра `wallet` изменился с `[Int]` на `[Int]?`. Это опциональный массив целочисленных значений. Обратите внимание, что знак опционала стоит именно после квадратной скобки массива, что говорит о том, что сам массив может отсутствовать, но если он присутствует, то в нем обязан существовать хотя бы один элемент типа `Int`.

Также у входного параметра появилось значение по умолчанию — `nil`. Оно присваивается переменной `wallet` в том случае, если при вызове функции не ей передано значение этого параметра.

Дополнительно изменился тип возвращаемого значения. В том случае, если входное значение параметра `wallet` не существует (то есть равно `nil`), функция возвратит в качестве ответа также `nil`.

Для корректного определения, существует ли у `wallet` значение, использован оператор `if`. В его теле находится оператор `return`, который завершает работу функции и возвращает `nil`.

ПРИМЕЧАНИЕ В функции может быть несколько операторов return. Каждый из них завершает выполнение функции и возвращает некоторое значение.

Внешние имена аргументов

Для любого входного параметра можно задать его внешнее имя, то есть имя, которое указывается при вызове функции. Оно пишется перед внутренним именем аргумента и отделяется от него пробелом (листинг 11.13).

Листинг 11.13

```
func sumWallet(banknotsArray wallet: [Int]? = nil ) ->
Int? {
    var sum = 0
    if wallet == nil {
        return nil
    }
    for oneBanknote in wallet! {
        sum += oneBanknote
    }
    return sum
}
// сосчитаем сумму всех купюр
sumWallet(banknotsArray: [50, 100, 100, 500, 50, 1000,    6 950
5000, 50, 100])
```

Входной параметр `wallet` теперь имеет внешнее имя `banknotsArray`, поэтому при вызове функции `sumWallet(_:)` необходимо указать не только значение параметра, но и его внешнее имя.

Внешние имена входных параметров служат для того, чтобы скрывать их внутренние имена. Например, в качестве внешних вы можете использовать логически понятные для разработчика длинные имена, а в качестве внутренних — сокращенные.

Функция в качестве аргумента

ПРИМЕЧАНИЕ Функция в Swift имеет свой функциональный тип данных! Удивлены? Тип данных функции обозначается с помощью конструкции, указывающей на тип входных и выходных значений.

Если функция ничего не принимает и не возвращает, то ее тип указывается так:

```
() -> ()
```

Если функция принимает на входе массив целочисленных значений, а возвращает опциональный кортеж из двух строковых элементов, то ее тип данных выглядит следующим образом:

```
([Int]) -> (String,String)?
```

В левой части данной конструкции указываются типы входных параметров, в правой — типы выходных значений.

Использование входных параметров не заканчивается на передаче им значений фундаментальных типов, массивов и кортежей. Ранее мы уже передавали значение, возвращаемое функцией, в качестве значения для аргумента. Теперь мы научимся передавать не значение, которое возвращает функция, а непосредственно саму функцию. Переданную функцию можно будет использовать в теле той функции, в которую происходит передача.

Для передачи функции в качестве значения необходимо указать функциональный тип принимаемой функции в качестве типа принимающего аргумента. Напишем новую функцию `generateWallet(_:)`, которая случайным образом генерирует массив банкнот. Она должна принимать на входе требуемое количество банкнот в кошельке. Также мы перепишем функцию `sumWallet(_:)` таким образом, чтобы она принимала на входе не массив значений, а функцию `generateWallet(_:)` и самостоятельно генерировала массив случайных банкнот (листинг 11.14).

Листинг 11.14

```
import Foundation
// 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 {
        let randomIndex = Int( arc4random_uniform(
        UInt32( typesOfBanknotes.count-1 ) ) )
        wallet.append( typesOfBanknotes[randomIndex] )
    }
    return wallet
}
// функция подсчета денег в кошельке
```

```
17 func sumWallet(banknotsFunction wallet: (Int)->([Int]) )
18  -> Int? {
19     // вызов переданной функции
20     let myWalletArray = wallet( Int( arc4random_uniform(10) ) )
21     var sum: Int = 0
22     for oneBanknote in myWalletArray {
23         sum += oneBanknote
24     }
25     return sum
26 }
27 // передача функции в функцию
28 sumWallet(banknotsFunction: generateWallet)                    2 700
```

Значение в области результатов, вероятно, будет отличаться от того, что отобразится у вас. Это связано с использованием глобальной функции `arc4random_uniform()`, возвращающей случайное число.

Функция `generateWallet(_:)` создает массив купюр такой длины, которая передана ей в качестве входного параметра. В массиве `typesOfBanknotes` содержатся все возможные типы купюр. Суть работы функции такова: случайным образом купюра изымается из массива `typesOfBanknotes`, после чего она помещается в массив-кошелек `wallet`, который и является возвращаемым значением данной функции.

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 под этот параметр не выделяется память, что благоприятно влияет на работу.

В качестве типа входного параметра `wallet` функции `sumWallet(_:)` указан тип функции `generateWallet(_:)`. При вызове `sumWallet(_:)` необходимо передать лишь имя требуемой функции.

ПРИМЕЧАНИЕ Как отмечалось ранее, символ нижнего подчеркивания может заменить объявляемый параметр практически в любом месте при условии, что данный параметр не будет использоваться. Им можно заменять в том числе и входные параметры функции, если важен только факт их передачи, а не значение:

```
func someFunction(_:Int) -> () {}
someFunction(100)
```

Функция в качестве возвращаемого значения

Так как функция имеет свой индивидуальный тип, его можно указывать не только для входных аргументов, но и для выходного значе-

ния. В листинге 11.15 мы снова перепишем функцию подсчета денег в кошельке.

Листинг 11.15

```
// функция вывода текста
func printText()
 -> (String) {
    return "Очень хорошая книга"
}
// функция, которая возвращает функцию
func returnPrintTextFunction()
 -> () -> (String) {
    return printText
}

let newFunctionInLet = returnPrintTextFunction()    () -> String
newFunctionInLet()                                  "Очень хорошая
                                                     книга"
```

В качестве типа возвращаемого функцией `returnPrintTextFunction()` значения указан тип `() -> (String)`. Он соответствует типу данных функции `printText()`.

В результате присвоения возвращенной функции константе `newFunctionInLet` ее тип данных становится `() -> (String)`, а сама она хранит в себе функцию, которую можно вызывать. В этом можно убедиться, если вывести справочное окно для этой константы (рис. 11.1).

Рис. 11.1. Справочное окно для константы, хранящей функцию в качестве значения

11.3. Тело функции как значение

Переменная или константа может хранить в себе функцию — об этом мы уже узнали. Но для того чтобы присвоить какой-либо константе

функцию, не обязательно возвращать ее из другой функции. Для этого можно создать *безымянную функцию* и передать ее в качестве значения в переменную или константу. Безымянные функции не имеют имен. Они состоят только из тела функции (листинг 11.16).

Листинг 11.16

```
// безымянная функция в качестве значения
let functionInLet = {return true}
functionInLet()                                    true
```

Созданная константа имеет функциональный тип `() -> Bool` и хранит в себе тело функции.

Безымянные функции называются *замыкающими выражениями*, или *замыканиями*. Замыкающие выражения могут не просто выполнять какие-либо действия, но и принимать входные параметры, а также возвращать произвольные значения. Об этом мы поговорим в следующей главе.

ПРИМЕЧАНИЕ Функция — это тип-ссылка, то есть она передается по ссылке. Например:

```
let trueFunc = {return true}
let anotherTrueFunc = trueFunc
```

Здесь константы trueFunc и anotherTrueFunc указывают на одну и ту же функцию.

11.4. Вложенные функции

Все функции, которые мы создавали ранее, являются глобальными. Помимо них можно создавать и локальные функции, вложенные друг в друга. Они обладают ограниченной областью видимости, то есть напрямую доступны только в пределах 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 возвращает новые координаты (листинг 11.17).

Листинг 11.17

```
func oneStep(inout coordinates: (Int, Int), stepType: String ) ->
(Int,Int) {
    func up(inout coords: (Int, Int)) -> (Int,Int) {
        return (coords.0+1, coords.1)
```

```
    }
    func right(inout coords: (Int, Int)) -> (Int,Int) {
        return (coords.0, coords.1+1)
    }
    func down(inout coords: (Int, Int)) -> (Int,Int) {
        return (coords.0-1, coords.1)
    }
    func left(inout coords: (Int, Int)) -> (Int,Int) {
        return (coords.0, coords.1-1)
    }

    switch stepType {
    case "up":
        return up(&coordinates)
    case "right":
        return right(&coordinates)
    case "down":
        return down(&coordinates)
    case "left":
        return left(&coordinates)
    default:
        return (0, 0)
    }
}
var coordinates = (10, -5)                          (.0 10, .1 -5)
oneStep(&coordinates, stepType: "up")               (.0 11, .1 -5)
oneStep(&coordinates, stepType: "right")            (.0 11, .1 -4)
coordinates
```

Функция `oneStep(_:stepType:)` осуществляет перемещение точки по плоскости. У нее существует несколько вложенных функций, которые вызываются в зависимости от значения параметра `stepType`. Данный набор функций доступен только внутри родительской функции `oneStep(_:stepType:)`.

11.5. Перезагрузка функций

В Swift возможна *перезагрузка* (overloading) *функций*. Это значит, что в одной и той же области видимости можно создавать функции с одинаковыми именами. Различия функций должны заключаться лишь в типах и именах входных параметров и типе возвращаемого значения. В листинге 11.18 представлены функции, которые могут сосуществовать одновременно.

Листинг 11.18

```
1  func say(what: String){}
2  func say(what: Int){}
```

У данных функций одно и то же имя `say(_:)`, но разный набор входных параметров. Хотя параметры имеют одно и то же имя, их тип данных различается.

Если вы имеете один и тот же список входных параметров (их имена и типы идентичны), то для перезагрузки функций необходимо указать разные типы возвращаемых значений. Рассмотрим пример из листинга 11.19. Представленные в нем функции также могут сосуществовать одновременно.

Листинг 11.19

```
1  func cry() -> String {
2      return "one"
3  }
4  func cry() -> Int {
5      return 1
6  }
```

Однако в данном случае вы не можете присвоить значение функции переменной или константе без явного указания типа (листинг 11.20).

Листинг 11.20

```
1  let resultOfFunc = say()
```

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 константы, согласуемый с типом возвращаемого значения одной из функций, то код отработает корректно (листинг 11.21).

Листинг 11.21

```
1  let resultString: String = say()
2  let resultInt = say() + 100
```

11.6. Рекурсивный вызов функций

Функция может вызывать саму себя. Этот механизм называется *рекурсией*. Вполне возможно, что вы встречались с рекурсиями в других

языках программирования. Вам необходимо быть крайне осторожными с этим механизмом, так как по невнимательности можно создать «бесконечную петлю», в которой функция будет бесконечно вызывать саму себя. При корректном использовании рекурсий функция всегда в конце концов будет завершать свою работу.

Пример рекурсии приведен в листинге 11.22.

Листинг 11.22

```
func countdown(firstNum num: Int) {
    print(num)
    if num > 0 {
        // рекурсивный вызов функции
        countdown(firstNum:num-1)
    }
}
countdown(firstNum: 20)
```

Функция `countdown(firstNum:)` отсчитывает цифры в сторону понижения, начиная от переданного параметра `firstNum` и заканчивая нулем. Этот алгоритм реализуется рекурсивным вызовом функции.

Задание 1

Вернемся к заданию 2 из предыдущей главы. Объедините написанный код анализа коллекции шахмат, хранящейся в переменной `Chessmans`, в функции с именем `chessAnalizer()`. В качестве входного параметра функция должна принимать словарь того же типа, что и переменная `Chessmans`.

Задание 2

Создайте функцию, которая предназначена для изменения состава и характеристик фигур в переменной `Chessmans`. В качестве входных параметров она должна принимать саму переменную `Chessmans` (как сквозной параметр), в которую будут вноситься изменения, имя фигуры (значение типа `String`) и опциональный кортеж координат фигуры (значение типа `(Character, Int)?`).

При этом должна проводиться проверка факта существования фигуры в словаре. Если фигура не существует, то информация о ней добавляется, в противном случае информация обновляется в соответствии с переданной информацией.

12 Замыкания

Мы уже встречались с понятием замыканий во время изучения функций. Там они были представлены в виде безымянных функций. Как объясняет Apple в документации к языку Swift, *замыкания* (closures) — это организованные блоки с определенным функционалом, которые могут быть переданы и использованы в вашем коде.

Согласитесь, не очень доступное объяснение. Попробуем иначе.

Замыкания — это сгруппированный в контейнер код, который может быть передан в виде аргумента и многократно использован.

ПРИМЕЧАНИЕ Если вы ранее программировали на других языках, то аналогом замыканий для вас могут быть блоки (в C и Objective-C), лямбда-выражения и анонимные функции.

12.1. Функции как замыкания

Функции — это частный случай замыканий, так как они обладают следующими свойствами:

- группируют код для многократного использования;
- могут быть многократно вызваны посредством назначенного им имени;
- могут быть переданы в качестве аргументов.

Рассмотрим работу с замыканиями на примерах. Вернемся к примеру с электронным кошельком и купюрами различного достоинства в нем и напишем функцию, которая будет принимать на входе массив-кошелек и возвращать массив всех сторублевых купюр (листинг 12.1).

Листинг 12.1

```
1  // функция отбора купюр
2  func handle100(wallet: [Int]) -> [Int] {
3      var returnWallet = [Int]()
```

```
    for banknot in wallet {
        if banknot==100{
            returnWallet.append(banknot)
        }
    }
    return returnWallet
}
// электронный кошелек
var wallet = [10,50,100,100,5000,100,50,50,500,100]
handle100(wallet)
```

При каждом вызове функция `handle100(_:)` будет возвращать массив сторублевых купюр. Здесь `handle100(_:)` — это замыкание, так как оно обладает описанными ранее свойствами:

- группирует код;
- может быть многократно использовано;
- может быть передано в виде аргумента (с этим свойством мы знакомились, когда передавали функции в виде входных параметров и возвращали в виде выходных значений).

Расширим функционал кода, написав дополнительную функцию для отбора купюр достоинством 1000 рублей и более (листинг 12.2).

Листинг 12.2

```
func handleMore1000(wallet: [Int]) -> [Int] {
    var returnWallet = [Int]()
    for banknot in wallet {
        if banknot>=1000{
            returnWallet.append(banknot)
        }
    }
    return returnWallet
}
var wallet = [10,50,100,100,5000,100,50,50,500,100]
handleMore1000(wallet)
```

В результате получается, что при написании двух функций в значительной мере происходит дублирование кода. Разница функций `handle100(_:)` и `handleMore1000(_:)` лишь в проверяемом условии. Остальной код в обеих функциях один и тот же.

Для решения этой проблемы можно пойти двумя путями.

1. Весь функционал реализовать в пределах одной функции. С этим подходом мы встречались, когда изучали тему вложенных функций.

2. Создать одну функцию с общим для всех функций отбора кодом и две функции, проверяющие условия. В качестве аргумента в основную функцию передавать требуемую функцию проверки условия, то есть передавать функцию в качестве аргумента.

Если мы пойдем по первому пути, то при увеличении количества различных условий отбора единая функция будет разрастаться и в конце концов станет нечитабельной и слишком сложной. Поэтому воспользуемся вторым вариантом (листинг 12.3).

Листинг 12.3

```
// единая функция формирования результирующего массива
func handle(wallet: [Int], closure: (Int) -> Bool) -> [Int] {
    var returnWallet = [Int]()
    for banknot in wallet {
        if closure(banknot) {
            returnWallet.append(banknot)
        }
    }
    return returnWallet
}
// функция сравнения с числом 100
func compare100(banknot: Int) ->Bool {
    return banknot==100
}
// функция сравнения с числом 1000
func compareMore1000(banknot:Int) -> Bool {
    return banknot>=1000
}
var wallet = [10,50,100,100,5000,100,50,50,500,100]
handle(wallet, closure: compare100)
handle(wallet, closure: compareMore1000)
```

Функция `handle(_:closure:)` получает в качестве входного параметра `closure` одну из функций проверки условия и в операторе `if` вызывает переданную функцию. Функции проверки принимают на входе анализируемую купюру и возвращают `Bool` в зависимости от результата сравнения.

Чтобы получить купюры определенного достоинства, необходимо вызвать функцию `handle(_:closure:)` и передать в нее имя одной из функций проверки.

В итоге мы получим очень качественный код, который достаточно легко расширять.

12.2. Замыкающие выражения

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 100, 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 куда более эффективным становится использование замыкающих выражений.

Замыкающие выражения — это безымянные замыкания, написанные в облегченном синтаксисе.

СИНТАКСИС

```
{ (входные_аргументы) -> ТипВозвращаемогоЗначения in
    тело_замыкающего_выражения
}
```

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 соответствии с третьим свойством замыканий замыкающие выражения можно передавать в качестве аргументов. Давайте вызовем написанную ранее функцию `handle(_:closure:)`, передавая ей замыкающее выражение в качестве входного параметра (листинг 12.4).

Листинг 12.4

```
1  // отбор купюр достоинством выше 1000 рублей
2  handle(wallet, closure: {(banknot: Int) -> Bool in
3      return banknot>=1000
4  })
5  // отбор купюр достоинством 100 рублей
6  handle(wallet, closure: {(banknot: Int) -> Bool in
7      return banknot==100
8  })
```

В результате необходимость в существовании функций `compare100(_:)` и `compareMore1000(_:)` отпадает, так как код проверяющей функции передается напрямую в качестве аргумента `closure`.

Налицо увеличение гибкости и уменьшение объема кода.

Замыкающие выражения, которые мы встречали при изучении функций, не имели каких-либо входных параметров, да и их функциональный тип не указывался.

ПРИМЕЧАНИЕ Облегченный синтаксис замыкающих выражений упрощает работу и позволяет не писать лишний код, а оптимизированный код экономит время и приносит вам дополнительную выгоду.

Теперь приступим к оптимизации уже используемых замыкающих выражений. При объявлении входного параметра `closure` в функции `handle(_:closure:)` указывается его функциональный тип (он принимает функцию типа `(Int) -> Bool`), поэтому при передаче замыкающего выражения нет необходимости дублировать данную информацию (листинг 12.5).

Листинг 12.5

```
// отбор купюр достоинством выше 1000 рублей
handle(wallet, closure: {banknot in
    return banknot>=1000
})
// отбор купюр достоинством 100 рублей
handle(wallet, closure: {banknot in
    return banknot==100
})
```

В замыкающем выражении перед ключевым словом `in` необходимо передать только имя, которое будет присвоено передаваемому в него значению очередного элемента массива `wallet`.

В коде функции `handle` при вызове функции `closure` ей передается параметр `banknot` — именно по этой причине мы указываем в качестве входного аргумента переменную с аналогичным названием.

12.3. Неявное возвращение значения

Замыкающие выражения позволяют в значительной мере оптимизировать программы. Это лишь одна из многих возможностей Swift, обеспечивающих красивый и понятный исходный код для ваших проектов.

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 12.6).

Листинг 12.6

```
// отбор купюр достоинством выше 1000 рублей
handle(wallet,
 closure: {banknot in banknot>=1000})
// отбор купюр достоинством 100 рублей
handle(wallet,
 closure: {banknot in banknot==100})
```

В результате мы добились того, что замыкающее выражение записывается всего в одну короткую строку и при этом код становится понятнее.

12.4. Сокращенные имена параметров

Продолжим оптимизацию используемых нами замыканий. Для однострочных замыкающих выражений Swift автоматически предоставляет доступ ко входным аргументам с помощью сокращенных имен доступа. Сокращенные имена доступа к входным аргументам пишутся 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(_:closure:)` с использованием сокращенных имен (листинг 12.7).

Листинг 12.7

```
// отбор купюр достоинством выше 1000 рублей
handle(wallet,
 closure: {$0>=1000})
// отбор купюр достоинством 100 рублей
handle(wallet,
 closure: {$0==100})
```

Здесь `$0` — это входной аргумент `banknot` входного аргумента-замыкания `closure` в функции `handle(_:closure:)`.

В тех случаях, когда входной параметр-функция состоит всего из одного выражения, использование замыкающих выражений делает код более понятным.

Если входной параметр-функция расположен последним в списке входных параметров функции (как в данном случае в функции `handle(_:closure:)`, где параметр `closure` является последним), Swift позволяет вынести его значение (замыкающее выражение) за круглые скобки (листинг 12.8).

Листинг 12.8

```
// отбор купюр достоинством выше 1000 рублей
handle(wallet)
 {$0>=1000}
// отбор купюр достоинством 100 рублей
handle(wallet)
 {$0==100}
```

Данный пример, возможно, не в полной мере демонстрирует необходимость выноса замыкания за круглые скобки, но в случае, когда замыкающее выражение многострочное, данный прием делает код значительно понятнее. В листинге 12.9 помимо массива `wallet`, который содержит купюры, находящиеся в кошельке, мы создадим массив `successbanknots`, содержащий в себе только разрешенные для использования купюры. Задача состоит в том, чтобы из `wallet` отобрать все купюры, которые представлены в `successbanknot`.

Листинг 12.9

```
let successbanknot = [100, 500]
handle(wallet)
 {banknot in
    for number in successbanknot {
        if number == banknot {
            return true
        }
    }
    return false
}
```

Внутри замыкающего выражения мы обращаемся к массиву `successbanknot` без его непосредственной передачи через входные аргументы, но работа внутри производится не с самим массивом `successbanknot`, а с его копией. В связи с этой особенностью Swift любые изменения массива внутри замыкающего выражения не привнесут никаких изменений в исходный массив.

12.5. Переменные-замыкания

Ранее, когда мы рассматривали работу с функциями, мы передавали в качестве значения переменной безымянную функцию. Так мы узнали, что переменные могут хранить в себе замыкающие выражения. Мы ограничились лишь присвоением значения, упустив при этом вопросы задания функционального типа и входных параметров.

Рассмотрим пример из листинга 12.10, в котором создадим константу `closure` и в качестве ее значения определим замыкающее выражение, ничего не принимающее и ничего не возвращающее.

Листинг 12.10

```
1  let closure : () -> () = {
2      print("Замыкающее выражение")
3  }
4  closure()
```

Консоль:

```
Замыкающее выражение
```

Так как данное замыкающее выражение не имеет входных параметров и возвращаемого значения, то его функциональный тип равен `() -> ()`.

Для вызова записанного в константу замыкающего выражения необходимо написать имя константы с круглыми скобками, то есть точно так же, как мы вызываем функции.

Для того чтобы передать в замыкание некоторые значения (в качестве входящих параметров), необходимо описать их в функциональном типе данной константы. При описании можно использовать все изученные ранее возможности: внутреннее и внешнее имя, тип данных, и т. д. Для доступа к значениям входных аргументов внутри замыкающего выражения необходимо использовать сокращенные имена доступа (`$0`, `$1` и т. д.), как показано в листинге 12.11.

Листинг 12.11

```
1  var sum: (numOne: Int, numTwo:Int) -> Int = {
2      return $0 + $1
3  }
4  sum(numOne: 10, numTwo: 34)
```

Здесь замыкающее выражение, хранящееся в константе `sum`, принимает два входных аргумента типа `Int` и возвращает их сумму.

ПРИМЕЧАНИЕ Замыкающие выражения могут храниться как в константах, так и в переменных. Выбирайте правильный тип параметра в зависимости от того, будет ли перезаписано его значение.

В большинстве случаев сохранять замыкание в переменной не имеет смысла.

12.6. Захват переменных

Мы не рассмотрели еще одну интересную возможность Swift, позволяющую зафиксировать значения параметров (переменных и констант), которые они имели на момент определения замыкания.

Обратимся к примеру (листинг 12.12). Существует набор параметров, которые не передаются в качестве входных аргументов в замыкание, но используются им в своих целях. При каждом вызове такого замыкания оно будет определять значения данных параметров, прежде чем приступить к выполнению операции с их участием.

Листинг 12.12

```
1  var a = 1
2  var b = 2
3  let closureSum : () -> Int = {
4      return a+b
5  }
6  closureSum()
7  a = 3
8  b = 4
9  closureSum()
```

Замыкание, хранящееся в константе `closureSum`, складывает значения перемененных `a` и `b`. При изменении их значений возвращаемое замыканием значение также меняется.

Для того чтобы зафиксировать значения некоторых параметров, необходимо написать их имена.

Существует способ «захватить» значения параметров, то есть зафиксировать те значения, которые имеют эти параметры на момент написания 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 12.13).

Листинг 12.13

```
1  var a = 1
2  var b = 2
3  let closureSum : () -> Int = {
4      [a,b] in
5      return a+b
6  }
7  closureSum()
8  a = 3
9  b = 4
10 closureSum()
```

Замыкание, хранящееся в константе `closureSum`, складывает значения переменных `a` и `b`. При изменении этих значений возвращаемое замыканием значение также меняется.

12.7. Метод сортировки массивов

Swift предлагает большое количество функций и методов, позволяющих в значительной степени упростить разработку приложений. Одним из таких методов является `sort()`, 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 `sort()`, принимает на входе два очередных элемента сортируемого массива и возвращает значение `Bool` в зависимости от результата их сравнения. Для того чтобы получить отсортированный массив, необходимо передать соответствующее замыкание. В листинге 12.14 мы отсортируем массив `myArray` таким образом, чтобы элементы были расположены по возрастанию. Для этого в функцию `sort()` передадим такое замыкающее выражение, которое возвращает `true`, когда второе сравниваемое число больше.

Листинг 12.14

```
1  var array = [1,44,81,4,277,50,101,51,8]
2  array.sort({ (first: Int, second: Int) -> Bool in
3   return first < second
4  })
```

Теперь применим все рассмотренные ранее способы оптимизации замыкающих выражений:

- уберем функциональный тип замыкания;
- заменим имена переменных именами в сокращенной форме.

В результате получится выражение, приведенное в листинге 12.15. Как и в предыдущем примере, здесь тоже необходимо отсортировать массив `myArray` таким образом, чтобы элементы были расположены по возрастанию. Для этого в метод `sort()` передается такое замыкающее выражение, которое возвращает `true`, когда второе сравниваемое число больше.

Листинг 12.15

```
1  var array = [1,44,81,4,277,50,101,51,8]
2  var sortedArray = array.sort({$0<$1})
```

В результате код получается более читабельным и красивым.

В Swift существует особая реализация замыкания сравнения переменных, которая принимает на входе два сравниваемых элемента, а возвращает значения типа `Bool`. Она называется бинарным оператором сравнения. С ним мы уже давно знакомы.

То есть в качестве замыкающего выражения можно написать бинарный оператор сравнения без указания имен сравниваемых параметров, так как данный оператор является бинарным — совершающим операцию с двумя операндами (листинг 12.16).

Листинг 12.16

```
1  var array = [1,44,81,4,277,50,101,51,8]
2  var sortedArray = array.sort(<)
```

Надеюсь, вы приятно удивлены потрясающими возможностями Swift!

Часть IV
Нетривиальные возможности Swift

В предыдущих главах книги происходило плавное погружение в философию Swift путем изучения основных возможностей этого языка. Очень важно, чтобы вы поняли и уяснили принципы программирования на данном замечательном языке. Именно по этой причине приводилось большое количество примеров, а каждый из них подробно описывался.

И хотя ранее мы неоднократно обсуждали различные объекты, их свойства и методы, до сих пор мы не описывали, как эти объекты создаются и работают.

Так как Swift придерживается парадигмы «все — это объект», то любой параметр (переменная или константа) с определенным типом данных — это объект. Для реализации новых объектов вы уже изучили множество различных типов данных, но, как отмечалось ранее, Swift обладает функционалом создания собственных объектных типов. Для этого существует три механизма: перечисления (enum), структуры (struct) и классы (class). В чем разница между ними? Как их создавать и использовать? Данная часть расскажет об этом. Мы обсудим, что такое объектные типы в общем и в чем разница между ними. Следующим шагом будет изучение механизмов, позволяющих расширить возможности объектных типов, включая протоколы, расширения, универсальные шаблоны и т. д.

При изучении нового материала будет активно использоваться уже изученный, поэтому если вы пропустили какие-либо из заданий в предыдущих главах, попрошу вас вернуться к ним.

- ✓ Глава 13. ООП как фундамент
- ✓ Глава 14. Перечисления
- ✓ Глава 15. Структуры
- ✓ Глава 16. Классы

- ✓ Глава 17. Свойства
- ✓ Глава 18. Сабскрипты
- ✓ Глава 19. Наследование
- ✓ Глава 20. Псевдонимы Any и AnyObject
- ✓ Глава 21. Инициализаторы и деинициализаторы
- ✓ Глава 22. Удаление экземпляров и ARC
- ✓ Глава 23. Опциональные цепочки
- ✓ Глава 24. Расширения
- ✓ Глава 25. Протоколы
- ✓ Глава 26. Разработка первого приложения
- ✓ Глава 27. Универсальные шаблоны
- ✓ Глава 28. Обработка ошибок
- ✓ Глава 29. Нетривиальное использование операторов

13 ООП как фундамент

Объектно-ориентированный стиль программирования является основой всей разработки программ на языке Swift. Мы уже неоднократно встречались с одним фундаментальным для данного языка правилом «всё — это объект». Пришло время приступить к изучению наиболее интересных и сложных механизмов, доступных в Swift. Данная глава расскажет вам о наиболее важных терминах и понятиях, которые помогут вам освоить весь последующий материал.

Возможно, вы изучали ООП в прошлом на уроках информатики в школе или в институте, а может, самостоятельно. В таком случае вы, конечно же, заучивали три постулата ООП: инкапсуляция, наследование и полиморфизм. В данной книге я не предлагаю их подробное изучение и заучивание, я предпочитаю практический подход. Да и принципы разработки на языке Swift во многом отличаются от таковых для языков Pascal, Visual Basic, Python, PHP и т. д.

13.1. Экземпляры

Перечисления, структуры и классы имеют одну очень важную особенность: для них могут быть созданы *экземпляры*. Когда вы определяете объектный тип, вы лишь задаете тип данных. Создание экземпляра объектного типа — это создание хранилища (переменной или константы) для данных определенного типа.

ПРИМЕЧАНИЕ Несмотря на то что мы много раз говорили об объектах, правильнее называть их экземплярами. Об этом Apple говорит в документации к Swift.

Объектами в других языках программирования назывались экземпляры классов, а экземпляры структур и перечислений — просто экземплярами. Так как функционал структур, перечислений и классов очень близок по своим возможностям, в Swift соответствующие объекты называются просто экземплярами.

Представьте, что определен некоторый класс `Automobile` (автомобиль). Этот класс является типом данных. Данный «Автомобиль» является не каким-то конкретным автомобилем, а лишь конструкцией, с помощью которой можно определить этот конкретный автомобиль. Если создать переменную типа `Automobile`, то в результате мы получим экземпляр этого класса (рис. 13.1).

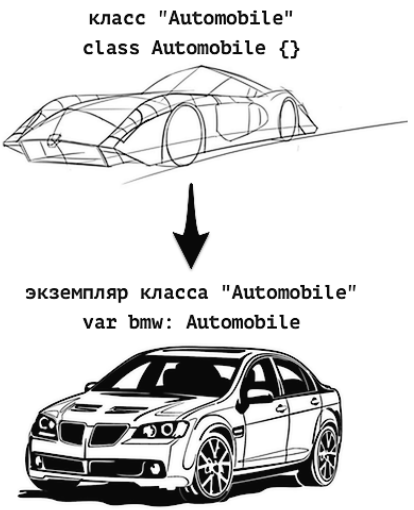


Рис. 13.1. Класс и его экземпляр

Сам класс на рисунке не имеет выраженных черт, так как еще неизвестно, какой же определенный объект реального (или нереального) мира он будет определять. Но когда создана переменная `bmw` типа `Automobile`, мы уже знаем, что с экземпляром в этой переменной мы будем работать словно с реальным авто марки BMW.

Объектный тип может наделять экземпляр некоторыми характеристиками. Для класса `Automobile` это могли бы быть: марка, модель, цвет, максимальная скорость, объем двигателя и т. д. Характеристики объектных типов называются *свойствами*. Для переменной `bmw` значения этих свойств могли бы быть следующими:

```
a.brand = "BMW"
a.type = "3"
a.maxSpeed = 210
a.engineCapacity = 1499
```

Свойства представляют собой хранилища данных, то есть это те же самые переменные и константы, но с ограниченным доступом: они доступны только через экземпляр.

Помимо свойств, у экземпляра могут быть определены методы. Методы, а с ними мы уже неоднократно встречались, — это функции, которые определены внутри объектных типов. Класс `Automobile` мог бы иметь следующие методы: издать сигнал, завестись, ускориться:

```
a.startEngine()
a.accelerate()
a.beep()
```

Таким образом, создавая экземпляр, мы можем наполнять его свойства информацией и использовать его методы. И свойства и методы определяются типом данных.

Способ разработки программ с использованием объектных типов называется объектно-ориентированным программированием (ООП). Этот стиль программирования позволяет достичь очень многого при разработке программ. Несмотря на то что вместо термина «объект» используется термин «экземпляр», аббревиатура ООП является устоявшейся в программировании, и в Swift она не трансформируется в ЭОП.

13.2. Пространства имен

Пространства имен (namespaces) — это именованные фрагменты программ. Пространства имен имеют одно очень важное свойство — они скрывают свою внутреннюю реализацию и не позволяют получить доступ к объектам внутри пространства имен без доступа к самому пространству имен. Это замечательная черта, благодаря которой вы можете иметь объекты с одним и тем же именем в различных пространствах имен.

Мы уже неоднократно говорили об областях видимости переменных и функций. Пространства имен как раз и реализуют в приложении различные области видимости.

Простейшим примером ограничения области видимости может служить функция. Все переменные, объявленные в ней, вне функции недоступны. Но при этом функция не является пространством имен, так как не позволяет получить доступ к объектам внутри себя извне.

К пространствам имен относятся *перечисления*, *структуры* и *классы*. Именно их изучением мы и займемся. Также к пространствам имен относятся модули, но их изучение не является темой данной книги. Вообще-то модули — это верхний уровень пространств имен. В простейшем варианте ваша программа — это модуль, а значит, это отдельное пространство имен, именем которого является название вашего приложения. Также модулями являются различные фреймворки. С одним из них, кстати, мы уже работали, когда выполняли импорт модуля: `import Foundation`.

Этот фреймворк называется Cocoa's Foundation Framework и содержит большое количество функциональных механизмов, позволяющих расширить возможности программы.

Одни пространства имен могут включать другие: так, модуль UIKit, ориентированный на разработку iOS-приложений, в своем коде выполняет импорт модуля Cocoa's Foundation Framework.

14 Перечисления

Перейдем к изучению механизмов создания объектных типов данных. Начнем с простейшего из них — перечисления. Это очень важная и интересная тема в Swift.

14.1. Синтаксис перечислений

Перечисление — это объектный тип данных, который предоставляет доступ к различным предопределенным значениям. Рассматривайте его как перечень возможных значений, то есть набор констант, значения которых являются альтернативами друг другу.

Рассмотрим хранилище, которое описывает произвольную денежную единицу (листинг 14.1). Для того чтобы решить поставленную задачу с помощью изученных ранее типов данных, можно использовать тип `String`. При таком подходе придется вести учет всех возможных значений для описания денежных единиц.

Листинг 14.1

```
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 массива (листинг 14.2). Массив содержит все возможные значения, которые доступны в программе. При необходимости происходит получение требуемого элемента массива.

Листинг 14.2

```
1  var currencyUnit: [String] = ["Rouble", "Dollar", "Euro"]
2  var russianCurrency = currencyUnit[0]
```

В действительности это очень хороший способ ведения списков возможных значений. И его положительные свойства заканчиваются ровно там, где они начинаются у перечислений.

Для того чтобы ввести дополнительную вспомогательную информацию для элементов массива (например, страну, доступные купюры и монеты определенного достоинства), потребуется создать множество дополнительных массивов и словарей. Идеальным было бы иметь отдельный тип данных, который позволил бы описать денежную единицу, но, к сожалению, специалисты Apple не предусмотрели его в Swift.

Значительно улучшить ситуацию позволяет использование перечислений вместо массивов или фундаментальных типов данных.

Перечисление — это набор значений определенного типа данных, позволяющий взаимодействовать с этими значениями. Так, с помощью перечислений можно создать набор доступных значений и одно из значений присвоить некоторому параметру.

СИНТАКСИС

```
enum ИмяПеречисления {
    case Значение1
    case Значение2
    ...
}
```

Перечисление объявляется с помощью ключевого слова enum, за которым следует имя перечисления. Имя должно определять предназначение создаваемого перечисления и, как название любого типа данных, соответствовать верхнему стилю камэлкейс.

Тело перечисления заключается в фигурные скобки и содержит перечень доступных значений. Эти значения называются членами перечисления. Каждый член определяется с использованием ключевого слова case, после которого без кавычек указывается само значение. Эти значения необходимо начинать с прописной буквы. Их количество в перечислении может быть произвольным.

ПРИМЕЧАНИЕ Объявляя перечисление, вы создаете новый тип данных.

Решим задачу указания типа денежной единицы с использованием перечислений (листинг 14.3). Перечисление подобно массиву. Оно содержит список значений, одно из которых мы можем присвоить некоторому параметру.

Листинг 14.3

```
enum CurrencyUnit {
    case Rouble
    case Dollar
    case Euro
}
```

Несколько членов перечисления можно писать в одну строку через запятую (листинг 14.4).

Листинг 14.4

```
enum CurrencyUnit {
    case Rouble, Dollar, Euro
}
```

Несмотря на то что перечисление `CurrencyUnit` создано и его члены определены, ни одно из значений не присвоено какому-либо параметру. Для того чтобы инициализировать некоторый параметр некоторым членом перечисления, используется специальный синтаксис.

СИНТАКСИС

Доступ к значениям перечисления происходит так же, как и к свойствам экземпляров, то есть через символ точки. Для инициализации значения существует два способа.

```
var имяПараметра = ИмяПеречисления.Значение
```

Инициализируемое значение пишется после имени перечисления, отделяясь от него точкой.

```
let имяПараметра: ИмяПеречисления
имяПараметра = .Значение
```

Имя перечисления выступает в качестве типа данных параметра. После этого доступ к значениям происходит уже без указания его имени.

Создадим несколько параметров, которые будут содержать значения созданного ранее перечисления (листинг 14.5).

Листинг 14.5

```
// способ 1
let roubleCurrency = CurrencyUnit.Rouble      Rouble
// способ 2
let dollarCurrency: CurrencyUnit
dollarCurrency = .Dollar                      Dollar
```

В результате создаются две константы типа `CurrencyUnit`, каждая из которых в качестве значения содержит определенный член перечисления `CurrencyUnit`.

ВНИМАНИЕ Члены перечисления не являются значениями какого-либо типа данных, например String или Int. Поэтому значения в следующих переменных currency1 и currency2 не эквивалентны:

```
var currency1 = CurrencyUnit.Rouble
var currency2 = "Rouble"
```

ПРИМЕЧАНИЕ В первых версиях Swift в области результатов вместо определенного значения перечисления выводилось сообщение Enum Value.

14.2. Ассоциированные параметры

У каждого из членов перечисления могут быть ассоциированные с ним значения, то есть его характеристики. Они указываются для каждого члена точно так же, как входящие аргументы функции, то есть в круглых скобках с заданием имен и типов, разделенных двоеточием. Набор ассоциированных параметров может быть произвольным для каждого отдельного члена.

Создадим ассоциированные параметры для перечисления `CurrencyUnit`, добавив параметры, позволяющие описывать страны, в которых используется валюта, с кратким наименованием данной валюты (листинг 14.6).

Листинг 14.6

```
1  enum CurrencyUnit {
2      case Rouble(countrys: [String], shortName: String)
3      case Dollar(countrys: [String], shortName: String)
4      case Euro(countrys: [String], shortName: String)
5  }
```

Параметр `countrys` является массивом, так как валюта может использоваться не в одной, а в нескольких странах: например, евро используется во всей Еврозоне.

Теперь для того, чтобы создать переменную или константу типа `CurrencyUnit`, необходимо указать значения для всех ассоциированных параметров (листинг 14.7).

Листинг 14.7

```
1  var roubleCurrency: CurrencyUnit
2  roubleCurrency = .Rouble(countrys:
   ["Russia"], shortName: "RUB")              Rouble(["Russia"], "RUB")
```

Теперь в переменной `roubleCurrency` хранится член `Rouble` со значениями двух ассоциированных параметров. При описании ассоциированных параметров в перечислении указывать их имена не обязательно. При необходимости можно указывать лишь их типы.

Для того чтобы расширить возможности перечисления, обратимся к его члену `Dollar`. Как известно, доллар является национальной валютой большого количества стран: доллар США, австралийский доллар, канадский доллар и т. д. Создадим новое перечисление, которое содержит этот список стран. Также создадим новый ассоциированный параметр для члена `Dollar`, который будет сообщать о том, валютой какой страны является данный доллар (листинг 14.8).

Листинг 14.8

```
// страны, использующие доллар
enum DollarCountrys {
    case USA
    case Canada
    case Australia
}
// тип данных "валюта"
enum CurrencyUnit {
    case Rouble(countrys: [String],
    shortName: String)
    case Dollar(countrys: [String], shortName:
    String, national: DollarCountrys)
    case Euro(countrys: [String], shortName:
    String)
}
var dollarCurrency: CurrencyUnit
dollarCurrency = .Dollar(countrys: ["USA"],  Dollar(["USA"], "USD",
shortName: "USD", national: .USA)            DollarCountrys.USA)
```

Для параметра `national` перечисления `CurrencyUnit` используется тип данных `DollarCountrys`. При указании этого параметра его тип уже известен, поэтому его название при задании значения можно опустить.

Так как перечисление `DollarCountrys` используется исключительно в перечислении `CurrencyUnit` и создано для него, его можно перенести внутрь этого перечисления (листинг 14.9).

Листинг 14.9

```
// тип данных "валюта"
enum CurrencyUnit {
    // страны, использующие доллар
    enum DollarCountrys {
        case USA
        case Canada
        case Australia
    }
```

```
9       case Rouble(countrys: [String], shortName: String)
10      case Dollar(countrys: [String], shortName: String, national:
        DollarCountrys)
11      case Euro(countrys: [String], shortName: String)
12  }
```

Теперь перечисление `DollarCountrys` 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 `CurrencyUnit` (листинг 14.10).

Листинг 14.10

```
1  var someVar: CurrencyUnit.DollarCountrys
2  someVar = .Australia                              Australia
```

Так как перечисление `DollarCountrys` находится в пределах перечисления `CurrencyUnit`, обращаться к нему необходимо как к свойству этого типа, то есть через точку.

В очередной раз отмечу, насколько язык Swift удобен в использовании. После перемещения перечисления `DollarCountrys` в `CurrencyUnit` код продолжает работать, а Xcode дает корректные подсказки в окне автодополнения.

14.3. Оператор switch для перечислений

Мы научились устанавливать в качестве значений переменных и констант члены перечисления и их ассоциированные параметры. Для их анализа и разбора предназначен оператор `switch`.

Рассмотрим пример из листинга 14.11, в котором нам необходимо проанализировать переменную, содержащую член перечисления в качестве значения, и вывести на консоль информацию из ассоциированных параметров.

Листинг 14.11

```
1  // создание переменной
2  var someCurrency = CurrencyUnit.Rouble(countrys: ["Russia",
   "Ukrain", "Belarus"], shortName: "RUB")
3  // анализ переменной
4  switch someCurrency {
5     case .Rouble(let countrys, let shortname):
6          print("Рубль. Страны: \(String(countrys)), краткое
           наименование: \(shortname)")
```

```
7      case let .Euro (countrys, shortname):
8           print("Евро. Страны: \(String(countrys)), краткое
            наименование: \(shortname)")
9      case .Dollar(let countrys, let shortname, let national):
10          print("Доллар \(national). Страны: \(String(countrys)),
            краткое наименование: \(shortname) ")
11 }
```

Консоль:

```
Рубль. Страны: ["Russia", "Ukrain", "Belarus"], краткое наименование: RUB
```

В операторе `switch` описан каждый элемент перечисления `CurrencyUnit`, поэтому использовать оператор `default` не обязательно. Доступ к ассоциированным параметрам реализуется связыванием значений. Так как для всех параметров создается константа со связываемым значением, оператор `let` можно ставить сразу после ключевого слова `case` (это продемонстрировано для члена `Euro`).

14.4. 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 привязанное к нему значение.

ПРИМЕЧАНИЕ Связанные значения также называют исходными, или сырыми. Но в данном случае термин «связанные» значительно лучше отражает их предназначение.

Указание связанных значений

Для задания исходных значений членов перечисления необходимо указать тип данных самого перечисления, соответствующий значениям членов (листинг 14.12).

Листинг 14.12

```
1  enum Smile: String {
2       case Joy = ":)"
3       case Laugh = ":D"
4       case Sorrow = ":("
5       case Suprise = "o_O"
6  }
```

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. Исходные значения 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 создании экземпляра этого перечисления.

ВНИМАНИЕ Одновременное определение исходных значений и ассоциированных параметров запрещено.

Если в качестве типа данных перечисления указать `Int`, то исходные значения создаются автоматически путем увеличения значения на единицу для каждого последующего члена (значение первого члена равно 0). Тем не менее, конечно же, можно указать эти значения самостоятельно. Например, в листинге 14.13 представлено перечисление, содержащее список планет Солнечной системы в порядке удаленности от Солнца.

Листинг 14.13

```
1  enum Planet: Int {
2      case Mercury = 1, Venus, Earth, Mars, Jupiter, Saturn, Uranus,
                    Neptune, Pluton = 999
3  }
```

Для первого члена перечисления в качестве исходного значения указано целое число 1. Для каждого следующего члена значение увеличивается на единицу, так как не указано иное: для `Venus` — это 2, для `Earth` — 3 и т. д.

Для члена `Pluton` исходное значение указано, 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 созданного ранее перечисления `Smile` и получим исходное значение установленного в этом экземпляре члена (листинг 14.14).

Листинг 14.14

```
1  var iAmHappy = Smile.Joy                    Joy
2  iAmHappy.rawValue                           ":)"
```

В результате использования свойства `rawValue` мы получаем исходное значение члена `Joy` типа `String`.

Инициализация через связанное значение

Свойство `rawValue` может быть использовано не только для получения связанного значения, но и для создания экземпляра перечисления. Для этого необходимо вызвать инициализатор перечисления и передать ему требуемое исходное значение (листинг 14.15). В данном примере создается экземпляр, содержащий в себе указатель на третью планету от Солнца. Для этого служит созданное ранее перечисление `Planet`.

Листинг 14.15

```
1  var myPlanet = Planet.init(rawValue: 3)          Earth
2  var anotherPlanet = Planet.init(rawValue: 11)    nil
```

Инициализатор — это метод `init(rawValue:)` перечисления `Planet`. Ему передается указатель на исходное значение, связанное с искомым членом этого перечисления. Не удивляйтесь, что метод `init(rawValue:)` не описан в пределах перечисления, — он существует там всегда по умолчанию.

Инициализатор `init(rawValue:)` возвращает опционал, поэтому если вы укажете несуществующее связанное значение, возвратится `nil`.

ПРИМЕЧАНИЕ Инициализаторы вызываются каждый раз при создании нового экземпляра какого-либо перечисления, структуры или класса. Для некоторых конструкций их можно и нужно создавать самостоятельно, а для некоторых, вроде перечислений, они существуют по умолчанию.

Инициализатор проводит процесс инициализации, то есть выполняет установку всех требуемых значений для параметров с непосредственным созданием экземпляра и помещением его в хранилище.

Инициализатор — это всегда метод с именем init(). Но даже если вы не вызываете его напрямую, он все равно срабатывает.

С инициализаторами мы познакомимся подробнее в следующих главах книги.

14.5. Свойства в перечислениях

Благодаря перечислениям можно смоделировать ситуацию, в которой существует ограниченное количество исходов. У таких ситуаций по-

мимо возможных результатов (членов перечисления) могут существовать и некоторые свойства.

Свойства позволяют хранить в перечислении вспомогательную информацию. Мы уже неоднократно встречались со свойствами в процессе изучения Swift.

Свойство в перечислении — это хранилище, аналогичное переменной или константе, объявленное в пределах перечисления и доступное через его экземпляр. В Swift существует определенное ограничение для свойств в перечислениях: в качестве их значений не могут выступать фиксированные значения-литералы, а лишь замыкания. Такие свойства называются *вычисляемыми*. При каждом обращении к ним происходит вычисление присвоенного замыкания с возвращением получившегося значения.

Для вычисляемого свойства после имени через двоеточие указывается тип возвращаемого значения и далее без оператора присваивания в фигурных скобках — тело замыкающего выражения, генерирующего возвращаемое значение.

Объявим вычисляемое свойство для разработанного ранее перечисления (листинг 14.16). За основу возьмем перечисление `Smile` и создадим вычисляемое перечисление, которое возвращает связанное с текущим членом перечисления значение.

Листинг 14.16

```
enum Smile: String {
    case Joy = ":)"
    case Laugh = ":D"
    case Sorrow = ":("
    case Suprise = "o_O"
    var description: String {return self.rawValue}
}
var 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.

14.6. Методы в перечислениях

Ранее мы уже встречались с методом-инициализатором `init()`. Он говорит о том, что перечисления могут группировать в себе не только

члены и другие перечисления, но и методы. *Методы* — это функции внутри перечислений, структур и классов, поэтому их синтаксис и возможности идентичны синтаксису и возможностям функций.

Вернемся к примеру с перечислением `Smile` и создадим метод, который выводит на консоль справочную информацию о предназначении перечисления (листинг 14.17).

Листинг 14.17

```
enum Smile: String {
    case joy = ":)"
    case laugh = ":D"
    case sorrow = ":("
    case suprise = "o_O"
    // метод для вывода описания
    func description(){
        print("Перечисление содержит список используемых смайликов:
        их названия и графические обозначения")
    }
}
var mySmile = Smile.joy
mySmile.description()
```

Консоль:

```
Перечисление содержит список используемых смайликов: их названия
и графические обозначения
```

В этом перечислении объявлен метод `description()`. После создания экземпляра метода и помещения его в переменную метод `description()` можно вызвать.

14.7. Оператор self

Методы могут не только совершать какие-либо не относящиеся к перечислению вещи, но и обрабатывать его члены и связанные значения. Для доступа к значению экземпляра внутри самого перечисления используется оператор `self`, который возвращает указатель на данный экземпляр перечисления.

Обратимся к примеру. Пусть нам требуется написать два метода, один из которых будет возвращать сам член перечисления, а второй — его связанное значение. Используем для этого перечисление `Smile` (листинг 14.18).

Листинг 14.18

```
enum Smile: String {
    case joy = ":)"
    case laugh = ":D"
    case sorrow = ":("
    case suprise = "o_O"
    func description(){
        print("Перечисление содержит список
               используемых смайлов: их названия
               и графические обозначения")
    }
    func descriptionValue() -> Smile{
        return self
    }
    func descriptionRawValue() -> String{
        return self.rawValue
    }
}
var mySmile = Smile.joy                                    joy
mySmile.descriptionValue()                                 joy
mySmile.descriptionRawValue()                              ":)"
```

При вызове метода `descriptionValue()` происходит возврат `self`, то есть самого экземпляра. Именно поэтому тип возвращаемого значения данного метода — `Smile`, он соответствует типу экземпляра перечисления.

Метод `descriptionRawValue()` возвращает связанное значение члена данного экземпляра также с использованием оператора `self`.

При необходимости вы даже можете выполнить анализ перечисления внутри самого перечисления с помощью конструкции `switch self {}`, где значениями являются члены перечисления.

Оператор `self` можно использовать не только для перечислений, но и для структур и классов. Об этом мы поговорим позже.

14.8. Рекурсивные перечисления

Перечисления отлично справляются с моделированием ситуаций, когда есть всего несколько вариантов развития ситуации. Но вы можете использовать их не только для того, чтобы хранить некоторые связанные и ассоциированные значения. Вы можете пойти дальше и наделить перечисление функционалом анализа собственного значения и вычисления на его основе некоторых 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 14.19). Для простоты рассмотрим только две операции: сложение и вычитание.

Листинг 14.19

```
enum ArithmeticExpression{
    // операция сложения
    case Addition(Int, Int)
    // операция вычитания
    case Substraction(Int, Int)
}
var expr = ArithmeticExpression.Addition(10, 14)   Addition(10, 14)
```

Данное перечисление имеет два члена в соответствии с арифметическими выражениями. Знак каждой из арифметических операций в Swift — это бинарный оператор, то есть оператор, проводящий операцию с двумя операндами. В связи с этим каждая из операций может принимать два значения для ассоциированных параметров.

В результате в переменной `expr` хранится 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 метода, который определяет наименование члена и возвращает результат данной операции (листинг 14.20).

Листинг 14.20

```
enum ArithmeticExpression{
    case Addition(Int, Int)
    case Substraction(Int, Int)
    // метод подсчета
    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)   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 данной арифметической операции.

Данный способ работает просто замечательно, но имеет серьезное ограничение: он способен моделировать только одноуровневые арифметические выражения: 1 + 5, 6 + 19 и т. д. В ситуации, когда выражение имеет вложенные выражения: 1 + (5 – 7), 6 – 5 + 4 и т. д., нам придется вычислять каждое отдельное действие с использованием собственного экземпляра.

Для решения этой проблемы необходимо доработать перечисление `ArithmeticExpression` таким образом, чтобы оно давало возможность складывать не только значения типа `Int`, но и другие выражения. Получается, что перечисление, описывающее выражение, должно давать возможность выполнять операции само с собой. Данный механизм реализуется в *рекурсивном перечислении*. Для создания такого типа перечисления 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.

Рассмотрим пример из листинга 14.21. Если в качестве ассоциированных параметров перечисления указывать значения типа самого перечисления, то возникает вопрос: а где же хранить числа, над которыми совершаются операции? Такие числа также необходимо хранить в самом перечислении, в его отдельном члене. В данном примере вычисляется значение выражения 20 + 10 – 34.

Листинг 14.21

```
enum ArithmeticExpression {
    case Number(Int)
    indirect case Addition(ArithmeticExpression,
    ArithmeticExpression)
    indirect case Subtraction(ArithmeticExpression,
    ArithmeticExpression)
    func evaluate(expression: ArithmeticExpression? =
    nil ) -> Int{
        let expression = (expression == nil ? self : expression)
        switch expression! {
        case .Number( let value ):
            return value
```

```
10              case .Addition( let valueLeft, let valueRight ):
11                  return self.evaluate( valueLeft ) +
                    self.evaluate( valueRight )
12              case .Subtraction( let valueLeft, let valueRight ):
13                  return self.evaluate( valueLeft ) -
                    self.evaluate( valueRight )
14          }
15      }
16  }
17  var expr = ArithmeticExpression.Addition( .Number(20),
    .Subtraction( .Number(10), .Number(34) ) )
18  expr.evaluate()                                                -4
```

Здесь у перечисления появился новый член `Number`. Он определяет целое число. Для членов арифметических операций использовано ключевое слово `indirect`, позволяющее передать значение типа `ArithmeticExpression` в качестве ассоциированного параметра.

Метод `evaluate(_:)` принимает на входе опциональное значение типа `ArithmeticExpression?`. Опционал в данном случае позволяет вызвать метод, не передавая ему экземпляр, из которого этот метод был вызван. В противном случае последняя строка листинга выглядела бы следующим образом:

```
expr.evaluate(expr)
```

Согласитесь, что существующий вариант значительно удобнее.

Оператор `switch`, используя принудительное извлечение, определяет, какой член перечисления передан, и возвращает соответствующее значение.

В результате данное перечисление позволяет смоделировать любую операцию, в которой присутствуют операторы сложения и вычитания.

Перечисления в Swift мощнее, чем аналогичные механизмы в других языках программирования. Вы можете создавать свойства и методы, применять к ним расширения и протоколы, а также делать многое другое. Обо всем этом мы вскоре поговорим.

Задание

Допишите перечисление `ArithmeticExpression` таким образом, чтобы оно могло реализовать любое выражение с использованием операций сложения, вычитания, умножения, деления и возведения в степень.

15 Структуры

Перечисления — это входной билет в разработку потрясающих приложений. Теперь пришло время познакомиться с еще более функциональными и интересными конструкциями — структурами.

Не удивляйтесь, но вы уже давно используете структуры при написании своего кода. Правда, все использованные структуры были реализованы другими разработчиками. Все фундаментальные типы данных — массивы, коллекции — являются структурами. Структуры в некоторой степени похожи на перечисления и во многом сходны с классами (с ними мы также познакомимся в скором времени).

15.1. Синтаксис объявления структур

Представим задачу, в которой необходимо описать игрока в шахматы. Требуется реализовать функционал, позволяющий для различных людей создавать различные хранилища (переменные) и работать с ними независимо друг от друга. При этом должен предоставляться удобный доступ к данным в этих хранилищах.

Для решения этой задачи можно использовать кортежи и хранить в переменной имя и количество побед игрока (листинг 15.1). Доступ к значениям будет осуществляться через элементы кортежа.

Листинг 15.1

```
1  var playerInChess = (name: "Василий", wins: 10)
```

Данный способ, конечно, решает поставленную задачу, но если требуется учитывать большое количество разнородных характеристик, то кортежи станут чересчур сложными. А уж как запомнить, какие характеристики и как называются, я просто не представляю.

Соответственно, нам нужен механизм, служащий своего рода договором, или набором правил, в котором будут описаны все возможные параметры (их имена и типы данных).

Структуры как раз и являются данным механизмом, позволяющим создать «скелет», описывающий некую сущность. Например, структура `Int` описывает сущность «целое число».

СИНТАКСИС

```
struct ИмяСтруктуры {
    // свойства и методы структуры
}
```

Структуры объявляются с помощью оператора struct, за которым следует имя создаваемой структуры. Требования к имени предъявляются точно такие же, как и к имени перечислений: оно должно писаться в верхнем стиле камэлкейс.

Тело структуры заключается в фигурные скобки и может содержать методы и свойства.

ПРИМЕЧАНИЕ Объявляя структуру, вы определяете новый тип данных.

Объявим структуру, которая будет описывать сущность «игрок в шахматы» (листинг 15.2).

Листинг 15.2

```
1  struct PlayerInChess {}
2  var oleg: PlayerInChess
```

Так как структура является типом данных, то с использованием данного типа можно объявить новое хранилище.

15.2. Свойства в структурах

Объявление свойств

В настоящее время созданная структура `PlayerInChess` пуста, то есть она не описывает какие-либо характеристики игрока. Добавим два свойства в соответствии с созданным ранее кортежем (листинг 15.3).

ПРИМЕЧАНИЕ Свойство — это характеристика сущности, которую описывает структура. Оно может быть переменной или константой и объявляется в теле структуры:

```
struct ИмяСтруктуры{
    var свойство1: ТипДанных
    let свойство2: ТипДанных
    // ...
}
```

Количество свойств в структуре неограниченно.

При изучении Swift мы уже неоднократно встречались со свойствами.

Листинг 15.3

```
1  struct PlayerInChess {
2      var name: String
3      var wins: UInt
4  }
```

Свойства `name` и `wins` характеризуют имя и количество побед игрока в шахматы.

Структуры, так же как и перечисления, имеют встроенный инициализатор, который не требуется объявлять. Данный инициализатор принимает на входе значения всех свойств структуры, производит их инициализацию и создает экземпляр структуры (листинг 15.4).

Листинг 15.4

```
1  var oleg = PlayerInChess(name: "Олег", wins: 32)
```

В результате создается новый экземпляр структуры `PlayerInChess`, обладающий значениями всех своих свойств.

ВНИМАНИЕ Значения свойств должны быть обязательно определены. Пропустить любое из них недопустимо! Если значение какого-либо из свойств не будет указано, Xcode сообщит об ошибке.

Значения свойств по умолчанию

Для свойств можно задавать значение по умолчанию. При этом Swift автоматически создает новый инициализатор, который позволяет создавать экземпляр без указания значений свойств. Вы сможете увидеть данный инициализатор в окне автодополнения во время создания экземпляра (рис. 15.1).

На рисунке изображены два разных инициализатора, доступных при создании экземпляра: один не требует указывать значения свойств, поскольку использует их значения по умолчанию, другой, наоборот,

требует указать эти значения. Инициализатор, который не требует указывать какие-либо значения, называется *пустым инициализатором*.

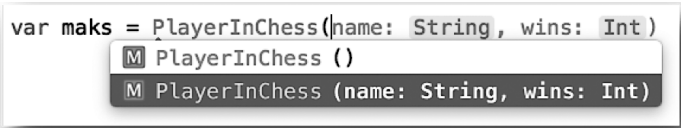


Рис. 15.1. Два инициализатора в окне автодополнения

Значения по умолчанию указываются вместе с объявлением свойств точно так же, как вы указываете значение любой переменной или константы. При этом если вы решили дать значение по умолчанию хотя бы одному свойству, то должны указывать его и для всех остальных свойств. Swift не позволяет определять значения по умолчанию лишь для некоторых свойств.

Объявим значения по умолчанию для структуры `PlayerInChess` (листинг 15.5).

Листинг 15.5

```
1  struct PlayerInChess {
2      var name = ""
3      var wins: UInt = 0
4  }
5  var oleg = PlayerInChess(name: "Олег", wins: 32)    PlayerInChess
6  var maks = PlayerInChess()                          PlayerInChess
```

В обоих случаях создается экземпляр структуры `PlayerInChess`. Если для создания экземпляра выбирается пустой инициализатор, параметрам `name` и `wins` присваиваются их значения по умолчанию.

15.3. Структура как пространство имен

Как отмечалось ранее, структура образует отдельное пространство имен. Поэтому для доступа к элементам этого пространства имен необходимо в первую очередь получить доступ к самому пространству.

В предыдущем примере была создана структура `PlayerInChess` с двумя свойствами. Каждое из свойств имеет некоторое значение, но от них не будет никакого толку, если не описать механизмы доступа к данным свойствам.

Доступ к элементам структур происходит с помощью экземпляров данной структуры (листинг 15.6).

Листинг 15.6

```
struct PlayerInChess {
    var name: String
    var wins: UInt = 0
}
var oleg = PlayerInChess(name: "Олег")
// доступ к свойству
oleg.name
oleg.wins = 20
```

Такой способ доступа обеспечивает не только чтение, но и изменение значения свойства экземпляра структуры.

15.4. Собственные инициализаторы

Инициализатор — это специальный метод, который носит имя `init()`. Если вас не устраивают инициализаторы, которые создаются для структур автоматически, вы имеете возможность разработать собственные.

ВНИМАНИЕ Автоматически созданные инициализаторы удаляются при объявлении первого собственного инициализатора.

Вам необходимо постоянно придерживаться правила: «все свойства структуры 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. Инициализаторы объявляются без использования ключевого слова `func`. При этом одна структура может содержать произвольное количество инициализаторов, каждый из которых должен иметь уникальный набор входных параметров.

Создадим инициализатор для структуры `PlayerInChess`, который принимает значение только для свойства `name` (листинг 15.7).

Листинг 15.7

```
struct PlayerInChess {
    var name: String
    var wins: UInt = 0
```

```
    init(name: String){
        self.name = name
    }
}
var oleg = PlayerInChess(name: "Олег")
```

При создании нового экземпляра вам будет доступен только разработанный вами инициализатор. В него передается значение параметра `name`, поэтому задавать значение по умолчанию для него не обязательно.

Доступ к свойствам экземпляра для их изменения осуществляется с помощью оператора `self`.

Свойство `wins` имеет значение по умолчанию, оно присваивается данному свойству автоматически.

Помните, что создавать собственные инициализаторы для структур не обязательно, так как структуры имеют встроенные инициализаторы.

ПРИМЕЧАНИЕ В том случае, если экземпляр структуры хранится в константе, модификация его свойств невозможна. Если же он хранится в переменной, то возможна модификация тех свойств, которые объявлены с помощью оператора var.

ВНИМАНИЕ Структуры — это типы-значения. При передаче экземпляра структуры от одного параметра в другой происходит его копирование. В следующем примере создаются два независимых экземпляра одной и той же структуры:

```
var olegMuhin = PlayerInChess(name: "asd")
var olegLapin = olegMuhin
```

15.5. Методы в структурах

Объявление методов

Помимо свойств структуры могут содержать и методы. Синтаксис объявления методов в структурах аналогичен объявлению методов в перечислениях. Они, как и обычные функции, могут принимать входные параметры. Для доступа к собственным свойствам структуры используется оператор `self`.

Напишем метод, который выводит справочную информацию об игроке в шахматы на консоль (листинг 15.8).

Листинг 15.8

```
struct PlayerInChess {
    var name: String
    var wins: UInt
```

```
4       func description(){
5           print("Игрок \(self.name) имеет \(self.wins) побед")
6       }
7   }
8   var oleg = PlayerInChess(name: "Олег", wins: 15)
9   oleg.description()
```

Консоль:

```
Игрок Олег имеет 15 побед
```

Изменяющие методы

По умолчанию методы структур, кроме инициализаторов, не могут изменять значения свойств, объявленные в тех же самых структурах. Для того чтобы обойти данное ограничение, перед именем объявляемого метода необходимо указать модификатор `mutating`.

Создадим метод, который не имеет ограничений на изменение свойств экземпляра структуры. Например, если возникнет необходимость изменения количества побед игрока в шахматы, можно создать метод, реализующий данный функционал (листинг 15.9).

Листинг 15.9

```
1   struct PlayerInChess {
2       var name: String
3       var wins: UInt
4   // метод, изменяющий значение свойства wins
5   mutating func addWins( countOfWins: Int ){
6   self.wins += countOfWins
7   }
8   }
9   var oleg = PlayerInChess(name: "Олег", wins: 15)
10  oleg.wins
11  oleg.addWins(3)
12  oleg.wins
```

ВНИМАНИЕ Структура может изменять значения свойств только в том случае, если экземпляр структуры хранится в переменной.

16 Классы

Классы являются наиболее функциональными конструкциями в разработке приложений. Данная глава призвана познакомить вас с этими замечательными конструкциями. Если вы ранее разрабатывали приложения на других языках программирования, то, возможно, вы уже осведомлены о классах. В этом случае данный опыт пригодится вам при их изучении в Swift.

Классы подобны структурам, но их отличают несколько ключевых моментов:

- **Тип**. Класс — это тип-ссылка. Экземпляры класса передаются по ссылке, а не копированием.
- **Изменяемость**. Экземпляр класса может изменять значения своих свойств, объявленных как переменная (`var`), даже если данный экземпляр хранится в константе (`let`). При этом использовать ключевое слово `mutating` для снятия ограничения на изменение методами значений свойств не требуется.
- **Наследование**. Два класса могут быть в отношении суперкласс—субкласс друг к другу. При этом, в отличие от наборов (`sets`), где супернабор включает все элементы субнабора, здесь ситуация обратная: субкласс наследует и включает в себя все характеристики (свойства и методы) суперкласса и может быть дополнительно расширен. Об ограничениях, связанных с наследованием, рассказано в соответствующей главе.
- **Инициализаторы**. Класс имеет только пустой встроенный инициализатор `init(){}`, который не требует передачи значения входных параметров для их установки в свойства.
- **Деинициализаторы**. Swift позволяет создать деинициализатор — специальный метод, который автоматически вызывается при удалении экземпляра класса.

- **Приведение типов**. В процессе выполнения программы вы можете проверить экземпляр класса на соответствие определенному типу данных.

Каждая из особенностей детально разбирается в книге.

При моделировании любой сущности вам необходимо научиться правильно выбирать требуемую для ее реализации конструкцию: перечисление или коллекцию, структуру, класс или просто кортеж. Чем больше времени вы будете тратить на реализацию различных идей, тем больший опыт накопите.

16.1. Синтаксис классов

Объявление класса очень похоже на объявление структуры и перечисления. Исключением является используемое ключевое слово.

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 стиле камэлкейс.

Тело класса заключается в фигурные скобки и может содержать методы, свойства и другие элементы, с которым мы еще не знакомы.

ПРИМЕЧАНИЕ В момент объявления нового класса в программе создается новый тип данных.

16.2. Свойства классов

Перейдем к практической стороне изучения классов. Класс, как и структура, может предоставлять механизмы для описания некоторой сущности. Эта сущность обычно обладает рядом характеристик, выраженных в классе в виде *свойств класса*. Свойства могут принимать значения в соответствии с определенным классом типов данных. Также для них могут быть указаны значения по умолчанию.

Как отмечалось ранее, класс имеет один встроенный инициализатор, который является пустым. Если у структуры инициализатор генерируется автоматически вместе с изменением состава ее свойств, то

у класса для установки значений свойств требуется разрабатывать инициализаторы самостоятельно. При создании экземпляра класса каждое свойство должно иметь определенное значение, а любое свойство, для которого не указано значение по умолчанию, должно получать значение с помощью разработанного вами инициализатора.

Создадим класс, предназначенный для описания сущности «шахматная фигура». Использование именно 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 тип-ссылка) значительно упростит работу.

У шахматной фигуры можно выделить следующий набор характеристик:

- тип фигуры;
- цвет фигуры;
- координаты на игровом поле.

Дополнительно мы будем сохранять символ, соответствующий шахматной фигуре. В Unicode существует набор символов, каждый из которых изображает отдельную фигуру определенного цвета. Вы можете самостоятельно найти их в Интернете.

Координаты послужат не только для того, чтобы определить местоположение фигуры на шахматной доске, но и вообще для определения факта ее присутствия. Если фигура убита или еще не выставлена, то значение координат должно отсутствовать (`nil`).

Не забывайте: так как существуют свойства, необходим инициализатор (листинг 16.1).

Листинг 16.1

```
class Chessman {
    let type: String
    let color: String
    var coordinates: (String, Int)? = nil
    let figureSymbol: Character
    init(type: String, color: String, figure: Character){
        self.type = type
        self.color = color
        self.figureSymbol = figure
    }
}
var kingWhite = Chessman(type: "king", color: "white", figure: "♔")
```

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. Координаты фигуры задаются с помощью строки и числа.

В разработанном инициализаторе указаны входные аргументы, значения которых используются в качестве значений свойств экземпляра.

В результате мы получили экземпляр класса, описывающий фигуру «Белый король». Фигура еще не имеет координат, а значит, не выставлена на игровое поле (ее координаты равны `nil`).

Для каждого из свойств `type` и `color` может быть создан список возможных значений. В связи с этим необходимо определить два перечисления: одно для описания типов шахматных фигур, другое для описания их цвета (листинг 16.2). Созданные перечисления должны найти место в качестве типов соответствующих свойств класса `Chessman`. Не забывайте, что и входные аргументы инициализатора должны измениться соответствующим образом.

Листинг 16.2

```
// типы фигур
enum ChessmanType {
    case King
    case Castle
    case Bishop
    case Pawn
    case Knight
    case Queen
}
// цвета фигур
enum ChessmanColor {
    case Black
    case White
}
class Chessman {
    let type: ChessmanType
    let color: ChessmanColor
    var coordinates: (String, Int)? = nil
    let figureSymbol: Character
    init(type: ChessmanType, color: ChessmanColor, figure:
    Character){
        self.type = type
        self.color = color
        self.figureSymbol = figure
    }
}
var kingWhite = Chessman(type: .King, color: .White, figure: "♔")
```

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.

16.3. Методы классов

Сущность «Шахматная фигура» является вполне рабочей. На ее основе можно описать любую фигуру. Тем не менее описанная фигура все еще является «мертвой» в том смысле, что она не может быть использована непосредственно для игры. Это связано с тем, что еще не разработаны механизмы, позволяющие установить фигуру на игровое поле и динамически ее перемещать.

Классы, как и структуры с перечислениями, могут содержать произвольные методы, обеспечивающие функциональную нагрузку класса. Не забывайте, что в классах нет необходимости использовать ключевое слово `mutating` для методов, меняющих значения свойств.

Немного оживим созданную модель шахматной фигуры, создав несколько методов (листинг 16.3). В классе `Chessman` необходимо реализовать следующие функции:

- установка координат фигуры (при выставлении на поле или при движении);
- снятие фигуры с игрового поля (окончание партии или гибель фигуры).

Листинг 16.3

```
class Chessman {
    let type: ChessmanType
    let color: ChessmanColor
    var coordinates: (String, Int)? = nil
    let figureSymbol: Character
    init(type: ChessmanType, color: ChessmanColor,
    figure: Character){
        self.type = type
        self.color = color
        self.figureSymbol = figure
    }
    // метод установки координат
    func setCoordinates(char c:String, num n: Int){
        self.coordinates = (c, n)
    }
```

```
15      // метод, убивающий фигуру
16      func kill(){
17          self.coordinates = nil
18      }
19  }
20  var kingWhite = Chessman(type: .King, color: .White, figure: "♔")
21  kingWhite.setCoordinates(char: "E", num: 1)
```

В результате фигура «Белый король» создается и выставляется в позицию с координатами E1.

На самом деле для действительного размещения фигуры на игровом поле необходимо смоделировать саму шахматную доску. Этим вопросом мы займемся в скором времени.

16.4. 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 экземпляров.

Рассмотрим процесс создания дополнительного инициализатора. Существующий класс `Chessman` не позволяет одним выражение 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 16.4).

Листинг 16.4

```
1  class Chessman {
2      let type: ChessmanType
3      let color: ChessmanColor
4      var coordinates: (String, Int)? = nil
5      let figureSymbol: Character
6      init(type: ChessmanType, color: ChessmanColor, figure:
       Character){
7          self.type = type
8          self.color = color
9          self.figureSymbol = figure
10     }
11     init(type: ChessmanType, color: ChessmanColor, figure:
       Character, coordinates: (String, Int)){
12         self.type = type
13         self.color = color
14         self.figureSymbol = figure
```

```
        self.setCoordinates(char: coordinates.0, num:
        coordinates.1)
    }
    func setCoordinates(char c:String, num n: Int){
        self.coordinates = (c, n)
    }
    func kill(){
        self.coordinates = nil
    }
}
var QueenBlack = Chessman(type: .Queen, color: .Black, figure:
"♕", coordinates: ("A", 6))
```

Так как код установки координат уже написан в методе `setCoordinates(char:num:)`, то во избежание дублирования в инициализаторе этот метод будет просто вызываться.

При объявлении нового экземпляра в окне автодополнения будут предлагаться на выбор два инициализатора, объявленных в классе `Chessman`.

Все тонкости работы с инициализаторами мы рассмотрим в отдельном разделе книги, а сейчас советую сохранить разработанные конструкции в отдельном файле, так как в будущем мы к ним еще вернемся.

16.5. Вложенные типы

Очень часто перечисления, структуры и классы создаются для того, чтобы поддержать функциональность 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 16.5).

Листинг 16.5

```
class Chessman {
    enum ChessmanType {
        case King
        case Castle
        case Bishop
        case Pawn
        case Knight
```

```
8           case Queen
9       }
10      enum ChessmanColor {
11          case Black
12          case White
13      }
14      let type: ChessmanType
15      let color: ChessmanColor
16      var coordinates: (String, Int)? = nil
17      let figureSymbol: Character
18      init(type: ChessmanType, color: ChessmanColor, figure:
            Character){
19          self.type = type
20          self.color = color
21          self.figureSymbol = figure
22      }
23      init(type: ChessmanType, color: ChessmanColor, figure:
            Character, coordinates: (String, Int)){
24          self.type = type
25          self.color = color
26          self.figureSymbol = figure
27          self.setCoordinates(char: coordinates.0, num:
                coordinates.1)
28      }
29      func setCoordinates(char c:String, num n: Int){
30          self.coordinates = (c, n)
31      }
32      func kill(){
33          self.coordinates = nil
34      }
35  }
```

Структуры `ChessmanColor` и `ChessmanType` теперь являются вложенными в класс `Chessman`.

Ссылки на вложенные типы

В некоторых ситуациях может возникнуть необходимость использовать вложенные типы вне определяющего их контекста. Для этого необходимо указать имя родительского типа, после которого должно следовать имя требуемого типа данных (листинг 16.6). В этом примере мы получаем доступ к одному из членов перечисления `ChessmanType`, объявленного в контексте класса `Chessman`.

Листинг 16.6

```
1  var linkToEnumType = Chessman.ChessmanType.Bishop
```

17 Свойства

В ходе изучения Swift мы уже неоднократно встречались со свойствами экземпляров различных типов данных. Однако чтобы изучить все возможности свойств, необходимо глубже погрузиться в эту тему.

17.1. Типы свойств

Свойства — это хранилища, объявленные в пределах объектного типа данных. Они позволяют хранить и вычислять значения, а также получать доступ к этим значениям.

По типу хранимого значения можно выделить два основных вида свойств:

- хранимые свойства могут использоваться в структурах и классах;
- вычисляемые свойства могут использоваться в перечислениях, структурах и классах.

Хранимые свойства

Хранимое свойство — это константа или переменная, объявленная в объектном типе и хранящая определенное значение. Хранимое свойство обладает следующими возможностями:

- получает значение по умолчанию в случае, если при создании экземпляра ему не передается никакого значения;
- получает значение в инициализаторе, передаваемое при создании экземпляра в качестве входного аргумента. Данное значение называется исходным;
- меняет значение в процессе использования экземпляра.

Мы уже неоднократно объявляли хранимые свойства.

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
lazy let имяСвойства2
```

Перед оператором объявления хранилища добавляется модификатор lazy, указывающий на «ленивость» определенного свойства.

Рассмотрим пример. Создадим класс, который позволяет получить некоторую информацию о человеке, а именно имя и фамилию (листинг 17.1).

Листинг 17.1

```
1  class AboutMan{
2      let firstName = "Егор"
3      let secondName = "Петров"
4      lazy var wholeName: String = self.generateWholeName()
5      func generateWholeName() -> String{
6          return self.firstName + " " + self.secondName
7      }
8  }
9  var Me = AboutMan()
10 Me.wholeName
```

Здесь экземпляр класса `AboutMan` описывает некоего человека. В свойстве `wholeName` должно храниться его полное имя, но при создании экземпляра значения этого свойства не существует. При этом оно не равно `nil`, оно просто не сгенерировано и не записано. Это связано с тем, что свойство является ленивым. Как только происходит обращение к данному свойству, его значение формируется.

Ленивые свойства позволяют экономить оперативную память и не расходовать ее до тех пор, пока значение какого-либо свойства не потребуется.

ПРИМЕЧАНИЕ Стоит отметить, что в качестве значений для хранимых свойств нельзя указывать методы того же объектного типа. Ленивые свойства не имеют этого ограничения, так как их значения формируются уже после создания экземпляров.

Вычисляемые свойства

С вычисляемыми свойствами мы встречались при изучении перечислений, для экземпляров которых это единственный доступный вид свойств.

Вычисляемые свойства фактически не хранят значение, а вычисляют его с помощью замыкающего выражения.

СИНТАКСИС

```
var имяСвойства1: ТипЗначения { 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.

Переработаем пример из предыдущего листинга с использованием ленивого вычисляемого свойства. Так как метод `wholeName()` в классе `AboutMan` служит исключительно для генерации значения свойства `wholeName`, имеет смысл перенести данный метод в виде замыкающего выражения в свойство. В результате получится вычисляемое свойство (листинг 17.2).

Листинг 17.2

```
1  class AboutMan{
2      let firstName = "Егор"
3      let secondName = "Петров"
4      var wholeName: String {return self.firstName + " " +
       self.secondName}
5  }
6  var Me = AboutMan()
7  Me.wholeName
```

Значение свойства `wholeName` идентично соответствующему значению из предыдущего примера.

17.2. Контроль получения и установки значений

Геттер и сеттер вычисляемого свойства

Для любого вычисляемого свойства существует возможность реализовать две специальные функции:

- **Геттер** (`get`) выполняет некоторый код при запросе значения вычисляемого свойства.
- **Сеттер** (`set`) выполняет некоторый код при попытке установки значения вычисляемого свойства.

Во всех объявленных ранее вычисляемых свойствах был реализован только геттер, поэтому они являлись свойствами «только для чтения», то есть попытка изменения вызвала бы ошибку. Таким образом, можно утверждать, что геттер является обязательным для существования вычисляемого свойства, а реализацию сеттера можно опустить.

СИНТАКСИС

```
var имяСвойства1: ТипЗначения {
    get {
        тело_геттера
    }
    set (передаваемый_параметр) {
        тело_сеттера
    }
}
```

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 параметр является локальным для тела функции `set()`.

Если в вычисляемом свойстве отсутствует сеттер, то есть реализуется только геттер, то можно использовать упрощенный синтаксис записи. В этом случае опускается ключевое слово `set` и указывается только тело замыкающего выражения. Данный формат мы встречали в предыдущих примерах.

Рассмотрим пример. Окружность на плоскости может иметь три основные характеристики: координаты центра, радиус и длину окружности. Радиус и длина имеют жесткую зависимость. Для хранения одной из них мы используем вычисляемое свойство с реализованными геттером и сеттером (листинг 17.3).

Листинг 17.3

```
struct Circle{
    var coordinates: (x: Int, y: Int)
    var radius: Float
    var perimeter: Float {
        get{
            return 2.0*3.14*self.radius
        }
        set(newPerimeter){
            self.radius = newPerimeter / (2*3.14)
        }
    }
}
var myNewCircle = Circle(coordinates: (0,0), radius: 10)
myNewCircle.perimeter // выводит 62.8
myNewCircle.perimeter = 31.4
myNewCircle.radius    // выводит 5
```

При запросе значения свойства `perimeter` происходит выполнение геттера, который генерирует возвращаемое значение с учетом свойства `radius`. При установке значения свойства `perimeter` срабатывает сеттер, который вычисляет и устанавливает значение свойства `radius`.

Сеттер также позволяет использовать сокращенный синтаксис записи, в котором не указывается имя входного параметра. При этом внутри сеттера для доступа к устанавливаемому значению необходимо задействовать автоматически объявляемый параметр с именем `newValue`. Таким образом, класс `Circle` может выглядеть как в листинге 17.4.

Листинг 17.4

```
struct Circle{
    var coordinates: (x: Int, y: Int)
    var radius: Float
    var perimeter: Float {
        get{
            return 2.0*3.14*self.radius
        }
        set{
            self.radius = newValue / (2*3.14)
        }
    }
}
```

Наблюдатели хранимых свойств

Геттер и сеттер позволяют выполнять код при установке и чтении значения вычисляемого свойства. Другими словами, у вас имеются

механизмы контроля процесса изменения и чтения значений. Наделив такими полезными механизмами вычисляемые свойства, разработчики Swift не могли обойти стороной и хранимые свойства. Специально для них были реализованы наблюдатели (observers).

Наблюдатели — это специальные функции, которые вызываются либо непосредственно перед, либо сразу после установки нового значения хранимого свойства.

Выделяют два вида наблюдателей:

- Наблюдатель `willSet` вызывается перед установкой нового значения.
- Наблюдатель `didSet` вызывается после установки нового значения.

СИНТАКСИС

```
var имяСвойства1: ТипЗначения {
    willSet (параметр){
        тело_геттера
    }
    didSet (параметр){
        тело_сеттера
    }
}
```

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 функционал, который при изменении радиуса окружности выводит соответствующую информацию на консоль (листинг 17.5).

Листинг 17.5

```
1  struct Circle{
2      var coordinates: (x: Int, y: Int)
3      var radius: Float {
4          willSet (newValueOfRadius) {
5              print("Вместо значения \(self.radius) устанавливается
               значение \(newValueOfRadius)")
```

```
        }
        didSet (oldValueOfRadius) {
            print("Вместо значения \(oldValueOfRadius) установлено
            значение \(self.radius)")
        }
    }
    var perimeter: Float {
        get{
            return 2.0*3.14*self.radius
        }
        set{
            self.radius = newValue / (2*3.14)
        }
    }
}
var myNewCircle = Circle(coordinates: (0,0), radius: 10)
myNewCircle.perimeter // выведет 62.8
myNewCircle.perimeter = 31.4
myNewCircle.radius // выведет 5
```

Консоль:

```
Вместо значения 10.0 устанавливается значение 5.0
Вместо значения 10.0 установлено значение 5.0
```

Наблюдатели вызываются не только при непосредственном изменении значения свойства вне экземпляра. Так как сеттер свойства `perimeter` также изменяет значение свойства `radius`, то наблюдатели выводят на консоль соответствующий результат.

17.3. 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 к типу данных. Значения этих свойств едины для всех экземпляров 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.

ВНИМАНИЕ Для хранимых свойств типа в обязательном порядке должны быть указаны значения по умолчанию. Это связано с тем, что сам по себе тип не имеет инициализатора, который бы мог сработать еще во время определения типа и установить требуемые значения для свойств.

Хранимые свойства типа всегда являются ленивыми, при этом они не нуждаются в использовании ключевого слова lazy.

Свойства типа могут быть созданы для перечислений, структур и классов.

СИНТАКСИС

```
struct SomeStructure {
    static var storedTypeProperty = "Some value"
    static var computedTypeProperty: Int {
        return 1
    }
}
enum SomeEnumiration{
    static var storedTypeProperty = "Some value"
    static var computedTypeProperty: Int {
        return 2
    }
}
class SomeClass{
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 3
    }
    class var overrideableComputedTypeProperty: Int {
        return 4
    }
}
```

Свойства типа объявляются с использованием ключевого слова static для всех трех объектных типов. Единственным исключением являются маркируемые словом class вычисляемые свойства класса, которые должны быть способны переопределяться в подклассе. О том, что такое подкласс, мы поговорим позже.

Создадим структуру для демонстрации работы свойств типа (листинг 17.6). Класс `AudioChannel` моделирует аудиоканал, у которого есть два параметра:

- максимально возможная громкость ограничена для всех каналов в целом;
- текущая громкость ограничена максимальной громкостью.

Листинг 17.6

```
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    // выводит 5
AudioChannel.maxVolume // выводит 5
AudioChannel.maxVolume = 10
AudioChannel.maxVolume // выводит 10
RightChannel.volume = 8
RightChannel.volume    // выводит 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.

18 Сабскрипты

Мы встречались с понятием сабскрипта, когда изучали тему доступа к элементам массива. Там сабскриптом назывался указываемый для доступа к определенному значению индекс. Однако сабскрипты позволяют также упростить работу со структурами и классами.

18.1. Назначение сабскриптов

С помощью сабскриптов структуры и классы можно превратить в некое подобие коллекций, то есть мы сможем обращаться к свойствам экземпляра с использованием ключей.

Предположим, что нами разработан класс `Chessboard`, моделирующий сущность «шахматная доска». Экземпляр класса `Chessboard` 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 ключу массива непосредственно экземпляру класса:

```
desk["A", 5]
```

ПРИМЕЧАНИЕ Сабскрипты доступны для структур и классов.

18.2. 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(новое_значение){
        // тело сеттера
    }
}
```

Сабскрипты объявляются в теле класса или структуры с помощью ключевого слова subscript. Далее указываются входные аргументы (в точности так же, как у методов) и тип значения. Входные аргументы — 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 параметра, которому будет присвоено новое значение. Если данный параметр не будет указан, то новое значение автоматически инициализируется локальной переменной `newValue`. При этом тип данных параметра будет соответствовать типу возвращаемого значения.

Сеттер является необязательным, и в случае его отсутствия может быть использован сокращенный синтаксис:

```
subscript(входные_параметры) -> возвращаемое_значение {
    // тело геттера
}
```

Сабскрипты поддерживают перегрузку, то есть в пределах одного объектного типа может быть определено произвольное количество сабскриптов, различающихся лишь набором входных аргументов.

ПРИМЕЧАНИЕ С перегрузками мы встречались, когда объявляли несколько функций с одним именем или несколько инициализаторов в пределах одного объектного типа. Каждый набор одинаковых по имени объектов отличался лишь набором входных параметров.

Для изучения сабскриптов вернемся к теме шахмат и создадим класс, описывающий сущность «шахматная доска» (листинг 18.1). 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 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.

Листинг 18.1

```
class gameDesk {
    var desk: [Int:[String:Chessman]] = [:]
    init(){
        for i in 1...8 {
            desk[i] = [:]
        }
    }
    func setChessman(chess: Chessman, coordinates: (String, Int)){
        if let oldCoordinates = chess.coordinates {
            desk[oldCoordinates.1]![oldCoordinates.0] = nil
        }
        self.desk[coordinates.1]![coordinates.0] = chess
        chess.setCoordinates(char: coordinates.0, num:
        coordinates.1)
    }
}
var game = gameDesk()
game.setChessman(QueenBlack, coordinates: ("B",2))
game.setChessman(QueenBlack, coordinates: ("A",3))
```

Класс `gameDesk` описывает игровое поле. Его основным и 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(_:coordinates:)` не просто устанавливает ссылку на фигуру в свойство `desk`, но и убирает старую ссылку. Это обеспечивается тем, что актуальные координаты всегда хранятся в свойстве `coordinates` экземпляра класса `Chessman`.

В написанном классе отсутствует возможность запроса информации о произвольной ячейке. Реализуем ее с использованием сабскрипта (листинг 18.2). 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 18.2

```
class gameDesk {
    var desk: [Int:[String:Chessman]] = [:]
    init(){
        for i in 1...8 {
            desk[i] = [:]
        }
    }
    subscript(alpha: String, number: Int) -> Chessman? {
        get{
            if let chessman = self.desk[number]![alpha] {
                return chessman
            }
            return nil
        }
    }
    func setChessman(chess: Chessman, coordinates: (String, Int)){
        if let oldCoordinates = chess.coordinates {
            desk[oldCoordinates.1]![oldCoordinates.0] = nil
        }
        self.desk[coordinates.1]![coordinates.0] = chess
        chess.setCoordinates(char: coordinates.0, num:
        coordinates.1)
    }
}
var game = gameDesk()
game.setChessman(QueenBlack, coordinates: ("A",3))
game["A",3]?.coordinates
```

Реализованный сабскрипт имеет только геттер, причем в данном случае можно было использовать краткий синтаксис записи (без ключевого слова `get`).

Так как сабскрипт возвращает опционал, перед доступом к свойству `coordinates` возвращенной шахматной фигуры необходимо выполнить извлечение опционального значения.

Теперь мы имеем возможность установки фигур на шахматную доску с помощью метода `setChessman(_:coordinates:)` и получения информации об отдельной ячейке с помощью сабскрипта.

Мы можем расширить функционал сабскрипта, добавив в него сеттер, позволяющий устанавливать фигуру на новые координаты (листинг 18.3).

Листинг 18.3

```
class gameDesk {
    var desk: [Int:[String:Chessman]] = [:]
    init(){
        for i in 1...8 {
            desk[i] = [:]
        }
    }
    subscript(alpha: String, number: Int) -> Chessman? {
        get{
            if let chessman = self.desk[number]![alpha] {
                return chessman
            }
            return nil
        }
        set{
            self.setChessman(newValue!, coordinates: (alpha,
            number))
        }
    }
    func setChessman(chess: Chessman, coordinates: (String, Int)){
        if let oldCoordinates = chess.coordinates {
            desk[oldCoordinates.1]![oldCoordinates.0] = nil
        }
         self.desk[coordinates.1]![coordinates.0] = chess
         chess.setCoordinates(char: coordinates.0, num:
         coordinates.1)
    }
}
var game = gameDesk()
game.setChessman(QueenBlack, coordinates: ("A",3))
game["C",5] = QueenBlack
QueenBlack.coordinates
```

Тип данных переменной `newValue` в сеттере сабскрипта соответствует типу данных возвращаемого сабскриптом значения. По этой причине необходимо принудительное извлечение значения 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.

Задание

Как вы, возможно, заметили, сеттер сабскрипта в объектном типе `gameDesk` не сможет корректно обработать ситуацию, когда ему в качестве значения передается `nil`. Хотя фигура благополучно удалится из соответствующего узла свойства `desk`, значение свойства `coordinates` этой фигуры останется неизменным.

1. Измените сабскрипт таким образом, чтобы он корректно обрабатывал удаление фигуры с шахматной доски. Не забывайте, что у класса `Chessman` существует метод `kill()`.
2. Реализуйте метод `printDesk()` в классе `gameDesk`, выводящий в консоль текстовое изображение шахматной доски примерно в следующем виде:

```
1 _ _ _ _ ♔ _ _ _
2 _ _ _ _ _ _ _ _
3 _ _ _ _ _ _ _ _
4 _ _ _ _ _ _ _ _
5 _ _ _ _ _ _ _ _
6 _ _ _ _ _ _ _ _
7 _ _ _ _ _ _ _ _
8 _ _ _ _ ♚ _ _ _
  A B C D E F G H
```

При этом должны выводиться изображения (свойство `symbol` класса `Chessman`), соответствующие каждой фигуре на шахматной доске.

19 Наследование

Одной из главных целей объектно-ориентированного подхода является многократное использование кода. Объединять код для его многократного использования позволяют замыкания и объектные типы данных. В методологии ООП, помимо создания экземпляров различных перечислений, структур и классов, существует возможность создания нового класса на основе уже существующего с автоматическим включением в него всех свойств, методов и сабскриптов класса-родителя. Данный подход называется *наследованием*.

В рамках наследования «старый» класс называется суперклассом (или базовым классом), а «новый» — подклассом (или субклассом, или производным классом).

19.1. Синтаксис наследования

Для наследования одного класса другим необходимо указать имя суперкласса через двоеточие после имени объявляемого класса.

СИНТАКСИС

```
class SomeSubClass: SomeSuperClass {
    // тело подкласса
}
```

Для создания производного класса SomeSubClass, для которого базовым является SomeSuperClass, необходимо указать имя суперкласса через двоеточие после имени подкласса.

В результате все свойства и методы, определенные в классе `SomeSuperClass`, становятся доступными в классе `SomeSubClass` 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 19.1). Данный класс описывает сущность «Четвероногое животное». Дополнительно объявляется субкласс `Dog`, описывающий сущность «Собака». Все характеристики класса `Quadruped` применимы и к классу `Dog`, поэтому их можно наследовать.

Листинг 19.1

```
// суперкласс
class Quadruped {
    var type = ""
    var name = ""
    func walk(){
        print("walk")
    }
}
// подкласс
class Dog: Quadruped {
    func bark(){
        print("woof")
    }
}
var dog = Dog()
dog.type = "dog"
dog.walk() // выводит walk
dog.bark() // выводит woof
```

Экземпляр `myDog` позволяет получить доступ к свойствам и методам родительского класса `Quadruped`. Кроме того, класс `Dog` расширяет собственные возможности, реализуя в своем теле дополнительный метод `bark()`.

ПРИМЕЧАНИЕ Класс может быть суперклассом для произвольного количества субклассов.

Доступ к наследуемым характеристикам

Доступ к наследуемым элементам родительского класса в производном классе реализуется так же, как к собственным элементам данного производного класса, то есть с использованием ключевого слова `self`. В качестве примера в класс `Dog` добавим метод, выводящий на консоль кличку собаки. Кличка хранится в свойстве `name`, которое наследуется от класса `Quadruped` (листинг 19.2).

Листинг 19.2

```
// подкласс
class Dog: Quadruped {
    func bark(){
        print("woof")
    }
    func printName(){
        print(self.name)
    }
}
var dog = Dog()
dog.name = "Dragon Wan Helsing"
dog.printName() // выведет 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 характеристик.

19.2. Переопределение наследуемых элементов

Субкласс может создавать собственные реализации свойств, методов и сабскриптов, наследуемых от суперкласса. Такие реализации называются *переопределением*. Для переопределения параметров суперкласса в Swift необходимо указать ключевое слово `override` перед определением элемента.

Переопределение методов

Довольно часто реализация метода, который «достался в наследство» от суперкласса, не соответствует требованиям разработчика. В таком случае в субклассе нужно переписать данный метод, обеспечив к нему доступ по прежнему имени. Объявим новый класс `NoisyDog`, который описывает сущность «Беспокойная собака». Класс `Dog` является суперклассом для `NoisyDog`. В описываемый класс необходимо внедрить собственную реализацию метода `bark()`, но так как одноименный метод уже существует в родительском классе `Dog`, мы воспользуемся механизмом переопределения (листинг 19.3).

Листинг 19.3

```
class NoisyDog: Dog{
    override func bark(){
        print ("woof")
```

```
        print ("woof")
        print ("woof")
    }
}

var badDog = NoisyDog()
badDog.bark() // выводит woof woof woof
```

С помощью ключевого слова `override` мы сообщаем Swift, что метод `bark()` в классе `NoisyDog` имеет собственную реализацию.

ПРИМЕЧАНИЕ Класс может быть суперклассом вне зависимости от того, является он субклассом или нет.

Переопределенный метод не знает деталей реализации метода родительского класса. Он знает лишь имя и перечень входных параметров родительского метода.

Доступ к переопределенным элементам суперкласса

Несмотря на то что переопределение изменяет реализацию свойств, методов и сабскриптов, Swift позволяет осуществлять доступ внутри производного класса к переопределенным элементам суперкласса. Для этого в качестве префикса имени элемента вместо `self` используется ключевое слово `super`.

В предыдущем примере в методе `bark()` класса `NoisyDog` происходит дублирование кода. В нем используется функция вывода на консоль литерала `"woof"`, хотя данный функционал уже реализован в одноименном родительском методе. Перепишем реализацию метода `bark()` таким образом, чтобы избежать дублирования кода (листинг 19.4).

Листинг 19.4

```
class NoisyDog: Dog{
    override func bark(){
        for _ in 1...3 {
            super.bark()
        }
    }
}
var badDog = NoisyDog()
badDog.bark() // выводит woof woof woof
```

Вывод на консоль соответствует выводу реализации класса из предыдущего примера.

Доступ к переопределенным элементам осуществляется по следующим правилам:

- Переопределенный метод с именем `someMethod()` может вызвать одноименный метод суперкласса, используя конструкцию `super.someMethod()` внутри своей реализации (в коде переопределенного метода).
- Переопределенное свойство `someProperty` может получить доступ к свойству суперкласса с таким же именем, используя конструкцию `super.someProperty` внутри реализации своего геттера или сеттера.
- Переопределенный сабскрипт `someIndex` может обратиться к сабскрипту суперкласса с таким же форматом индекса, используя конструкцию `super[someIndex]` внутри реализации сабскрипта.

Переопределение инициализаторов

Инициализаторы являются такими же наследуемыми элементами, как и методы. Если в подклассе набор свойств, требующих установки значений, не отличается, то наследуемый инициализатор может быть использован для создания экземпляра подкласса.

Тем не менее вы можете создать собственную реализацию наследуемого инициализатора. Запомните, что если вы определяете инициализатор с уникальным для суперкласса и подкласса набором входных аргументов, то вы не переопределяете инициализатор, а объявляете новый.

Если подкласс имеет хотя бы один собственный инициализатор, то инициализаторы родительского класса не наследуются.

ВНИМАНИЕ Для вызова инициализатора суперкласса внутри инициализатора субкласса необходимо использовать конструкцию super.init().

В качестве примера переопределим наследуемый от суперкласса `Quadruped` пустой инициализатор. В классе `Dog` значение наследуемого свойства `type` всегда должно быть равно `"dog"`. В связи с этим перепишем реализацию инициализатора таким образом, чтобы в нем устанавливалось значение данного свойства (листинг 19.5).

Листинг 19.5

```
1  class Dog: Quadruped {
2      override init(){
3          super.init()
4          self.type = "dog"
```

```
5       }
6       func bark(){
7           print("woof")
8       }
9       func printName(){
10          print(self.name)
11      }
12  }
```

Прежде чем получать доступ к наследуемым свойствам в переопределенном инициализаторе, необходимо вызвать инициализатор родителя. Он выполняет инициализацию всех наследуемых свойств.

Если в подклассе есть собственные свойства, которых нет в суперклассе, то их значения в инициализаторе необходимо указать до вызова инициализатора родительского класса.

Переопределение наследуемых свойств

Как отмечалось ранее, вы можете переопределять любые наследуемые элементы. Наследуемые свойства иногда ограничивают функциональные возможности субкласса. В таком случае можно переписать геттер или сеттер данного свойства или при необходимости добавить наблюдатель.

С помощью механизма переопределения вы можете расширить наследуемое «только для чтения» свойство до «чтение—запись», реализовав в нем и геттер и сеттер. Но обратное невозможно: если у наследуемого свойства реализованы и геттер и сеттер, вы не сможете сделать из него свойство «только для чтения».

ВНИМАНИЕ Хранимые свойства переопределять нельзя, так как вызываемый или наследуемый инициализатор родительского класса попытается установить их значения, но не найдет их.

Субкласс не знает деталей реализации наследуемого свойства в суперклассе, он знает лишь имя и тип наследуемого свойства. Поэтому необходимо всегда указывать и имя и тип переопределяемого свойства.

19.3. Превентивный модификатор final

Swift позволяет защитить реализацию класса целиком или его отдельных элементов. Для этого необходимо использовать превентивный

модификатор `final`, который указывается перед объявлением класса или его отдельных элементов:

- `final class` для классов;
- `final var` для свойств;
- `final func` для методов;
- `final subscript` для сабскриптов.

При защите реализации класса его наследование в другие классы становится невозможным. Для элементов класса их наследование происходит, но переопределение становится недоступным.

19.4. Подмена экземпляров классов

Наследование, помимо всех перечисленных возможностей, позволяет заменять требуемые экземпляры определенного класса экземплярами одного из подклассов.

Рассмотрим пример из листинга 19.6. В нем объявим массив элементов типа `Quadruped` и добавим в него несколько элементов.

Листинг 19.6

```
var animalsArray: [Quadruped] = []
var someAnimal = Quadruped()
var myDog = Dog()
var dabDog = NoisyDog()
animalsArray.append(someAnimal)
animalsArray.append(myDog)
animalsArray.append(badDog)
```

В результате в массив `animalsArray` добавляются элементы типов `Dog` и `NoisyDog`. Это происходит несмотря на то, что в качестве типа массива указан класс `Quadruped`.

19.5. 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 или наследует указанный после оператора класс. Для анализа возьмем определенный и заполненный ранее массив `animalsArray` (листинг 19.7).

Листинг 19.7

```
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 неоднократно отмечалось ранее, объявленный и наполненный 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;
- вариант `as! ИмяКласса` производит принудительное извлечение значения и возвращает экземпляр типа `ИмяКласса` или вызывает ошибку в случае неудачи.

ВНИМАНИЕ Тип данных может быть преобразован только в пределах собственной иерархии классов.

Снова приступим к перебору массива `animalsArray`. На этот раз будем вызывать метод `bark()`, который не существует в суперклассе `Quadruped`, но присутствует в подклассах `Dog` и `NoiseDog` (листинг 19.8).

Листинг 19.8

```
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` записывается в параметр `item`. Далее в теле цикла данный параметр с использованием оператора `as?` пытается преобразоваться в каждый из типов данных нашей структуры классов. Если `item` преобразуется в тип `NoisyDog` или `Dog`, то у него становится доступным метод `bark()`.

20 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.

20.1. Псевдоним Any

Благодаря псевдониму `Any` можно создавать хранилища неопределенного типа данных. Объявим массив, который может содержать элементы произвольных типов (листинг 20.1).

Листинг 20.1

```
1  var things = [Any]()
2  things.append(0)
3  things.append(0.0)
4  things.append(42)
5  things.append("hello")
6  things.append((3.0, 5.0))
7  things.append({ (name: String) -> String in "Hello, \(name)" })
```

Массив `things` содержит значения типов: `Int`, `Double`, `String`, `(Double, Double)` и `(String)->String`. То есть перед вами целый набор различных типов данных.

При запросе любого из элементов массива вы получите значение не того типа данных, который предполагался при установке конкретного значения, а типа `Any`.

ПРИМЕЧАНИЕ Псевдоним Any несовместим с протоколом Hashable, поэтому использование типа Any там, где необходимо сопоставление, невозможно. Это относится, например, к ключам словарей.

Приведение типа Any

Для анализа каждого элемента массива необходимо выполнить приведение типа. Так вы сможете получить каждый элемент, преобразованный в его действительный тип данных.

Рассмотрим пример, в котором разберем объявленный ранее массив поэлементно и определим тип данных каждого элемента (листинг 20.2).

Листинг 20.2

```
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).

20.2. Псевдоним AnyObject

Псевдоним `AnyObject` позволяет указать на то, что в данном месте должен или может находиться экземпляр любого класса. При этом вы будете довольно часто встречать массивы данного типа при разработке программ с использованием фреймворка Cocoa Foundation. Данный фреймворк написан на Objective-C, а этот язык не имеет массивов с явно указанными типами данных.

Объявим массив экземпляров с помощью псевдонима `AnyObject` (листинг 20.3).

Листинг 20.3

```
1  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 20.4).

Листинг 20.4

```
1  for object in someObjects {
2      let animal = object as! Dog
3      print("This is bad Dog")
4  }
```

Для сокращения записи вы можете выполнить приведение типа для преобразования всего массива целиком (листинг 20.5).

Листинг 20.5

```
1  for object in someObjects as! [Dog]{
2      print("This is bad Dog")
3  }
```

21 Инициализаторы и деинициализаторы

Инициализатор — это специальный метод, выполняющий подготовительные действия при создании экземпляра объектного типа данных. То есть инициализатор срабатывает при создании экземпляра, а при его удалении вызывается деинициализатор.

21.1. Инициализаторы

Инициализатор выполняет установку начальных значений хранимых свойств и различные настройки, которые нужны для использования экземпляра.

Назначенные инициализаторы

При реализации собственных типов данных во многих случаях не требуется создавать собственный инициализатор, так как классы и структуры имеют встроенные инициализаторы:

- классы имеют пустой встроенный инициализатор `init(){}`;
- структуры имеют встроенный инициализатор, принимающий в качестве входных аргументов значения всех свойств.

ПРИМЕЧАНИЕ Пустой инициализатор срабатывает без ошибок только в том случае, если у класса отсутствуют свойства или у каждого свойства указано значение по умолчанию.

Для опциональных типов данных значение по умолчанию указывать не требуется, оно соответствует `nil`.

Инициализаторы класса и структуры, производящие установку значений свойств, называются *назначенными* (designated). Вы можете разработать произвольное количество назначенных инициализаторов

с отличающимся набором параметров в пределах одного объектного типа. При этом должен существовать хотя бы один назначенный инициализатор, производящий установку значений всех свойств (если они существуют), и один из назначенных инициализаторов должен обязательно вызываться при создании экземпляра. Назначенный инициализатор не может вызывать другой назначенный инициализатор, то есть использование конструкции `self.init()` запрещено.

ПРИМЕЧАНИЕ Инициализаторы наследуются от суперкласса к подклассу.

Единственный инициализатор, который может вызывать назначенный инициализатор, — это инициализатор производного класса, вызывающий инициализатор родительского класса для установки значений наследуемых свойств. Об этом мы говорили довольно подробно, когда изучали наследование.

ПРИМЕЧАНИЕ Инициализатор может устанавливать значения констант.

Внутри инициализатора необходимо установить значения свойств класса или структуры, чтобы к концу его работы все свойства имели значения (опционалы могут соответствовать `nil`).

Вспомогательные инициализаторы

Помимо назначенных в Swift существуют *вспомогательные* (convenience) инициализаторы. Они являются вторичными и поддерживающими. Вы можете определить вспомогательный инициализатор для проведения настроек и обязательного вызова одного из назначенных инициализаторов. Вспомогательные инициализаторы не являются обязательными для их реализации в типе. Создавайте их, если это обеспечивает наиболее рациональный путь решения поставленной задачи.

Синтаксис объявления вспомогательных инициализаторов не слишком отличается от синтаксиса назначенных.

СИНТАКСИС

```
convenience init(параметры) {
    // тело инициализатора
}
```

Вспомогательный инициализатор объявляется с помощью модификатора convenience, за которым следует ключевое слово init. Данный тип инициализатора также может принимать входные аргументы и устанавливать значения для свойств.

В теле инициализатора обязательно должен находиться вызов одного из назначенных инициализаторов.

Вернемся к иерархии определенных ранее классов `Quadruped`, `Dog` и `NoisyDog`. Давайте перепишем класс `Dog` таким образом, чтобы при установке он давал возможность выводить на консоль произвольный текст. Для этого создадим вспомогательный инициализатор, принимающий на входе значение для наследуемого свойства `type` (листинг 21.1).

Листинг 21.1

```
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 myDog = 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 инциализатор должен вызвать назначенный инициализатор того же объектного типа.
- Вспомогательный инициализатор в конечном счете должен вызвать назначенный инициализатор.

Иллюстрация всех трех правил представлена на рис. 21.1.

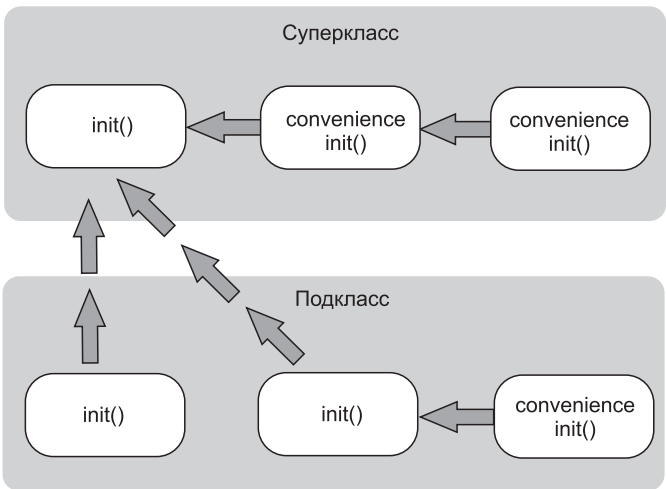


Рис. 21.1. Отношения между инициализаторами

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 конечном 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 21.2).

Листинг 21.2

```
1  class Rectangle{
2      var height: Int
3      var weight: Int
4      init?(height h: Int, weight w: Int){
5          self.height = h
6          self.weight = w
7          if !(h > 0 && w > 0) {
8              return nil
9          }
```

```
    }
}
var rectangle = Rectangle(height: 56, weight: -32) // возвращает nil
```

Инициализатор принимает и проверяет значения двух параметров. Если хотя бы одно из них не больше нуля, то возвращается `nil`. Обратите внимание, 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 21.3. 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 21.3

```
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
        }
```

```
14      }
15  }
16  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 21.4).

Листинг 21.4

```
1  enum TemperatureUnit: Character {
2      case Kelvin = "K", Celsius = "C", Fahrenheit = "F"
3  }
4  let fahrenheitUnit = TemperatureUnit(rawValue: "F")
5  fahrenheitUnit!.hasValue
```

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.

21.2. Деинициализаторы

Деинициализаторы являются отличительной особенностью классов.

Деинициализатор автоматически вызывается во время уничтожения экземпляра класса. Вы не можете вызвать деинициализатор самостоятельно. Один класс может иметь максимум один деинициализатор.

С помощью деинициализатора вы можете, например, освободить используемые экземпляром ресурсы, вывести на консоль журнал или выполнить любые другие действия.

СИНТАКСИС

```
deinit {
    // тело деинициализатора
}
```

Для объявления деинициализатора предназначено ключевое слово deinit, после которого в фигурных скобках указывается тело деинициализатора.

Деинициализатор суперкласса наследуется подклассом и вызывается автоматически в конце работы деинициализаторов подклассов. Деинициализатор суперкласса вызывается всегда, даже если деинициализатор подкласса отсутствует. Кроме того, экземпляр класса не удаляется, пока не закончит работу деинициализатор, поэтому все значения свойств экземпляра остаются доступными в теле деинициализатора.

Рассмотрим пример использования деинициализатора (листинг 21.5).

Листинг 21.5

```
class SuperClass {
    init?(isNil: Bool){
        if isNil == true {
            return nil
        }else{
            print("Экземпляр создан")
        }
    }
    deinit {
        print("Деинициализатор суперкласса")
    }
}
class SubClass:SuperClass{
```

```
14      deinit {
15          print("Деинициализатор подкласса")
16      }
17  }
18
19  var obj = SubClass(isNil: false)
20  obj = nil
```

Консоль:

```
Экземпляр создан
Деинициализатор подкласса
Деинициализатор суперкласса
```

При создании экземпляра класса `SubClass` на консоль выводится соответствующее сообщение, так как данный функционал находится в наследуемом от суперкласса проваливающемся инициализаторе. В конце программы мы удаляем созданный экземпляр, передав ему в качестве значения `nil`. При этом вывод на консоль показывает, что первым выполняется деинициализатор подкласса, потом — суперкласса.

22 Удаление экземпляров и ARC

Любой созданный экземпляр объектного типа данных, как и вообще любое хранилище вашей программы, занимает некоторый объем оперативной памяти. Если не производить своевременное уничтожение экземпляров и освобождение занимаемых ими ресурсов, то в программе может произойти утечка памяти.

ПРИМЕЧАНИЕ Утечка памяти — это программная ошибка, приводящая к излишнему расходованию оперативной памяти.

Одним из средств решения проблемы исключения утечек памяти в Swift является использование деинициализаторов, но возможности Swift не ограничиваются только этим.

22.1. Уничтожение экземпляров

Как мы отмечали в предыдущей главе, непосредственно перед уничтожением экземпляра класса вызывается деинициализатор, при этом остался нерассмотренным вопрос о том, как удаляется экземпляр. Любой экземпляр может быть удален одним из двух способов:

- его самостоятельно уничтожает разработчик;
- его уничтожает Swift.

Область видимости

Ранее мы самостоятельно уничтожали созданный экземпляр опционального типа `SubClass?`, передавая ему в качестве значения `nil`. Теперь обратимся к логике работы Swift. Для этого разработаем класс `myClass`, который содержит единственное свойство `description`. Дан-

ное свойство служит для того, чтобы отличать один экземпляр класса от другого.

Необходимо создать два экземпляра, один из которых должен иметь ограниченную область видимости (листинг 22.1).

Листинг 22.1

```
class myClass{
    var description: String
    init(description: String){
        print("Экземпляр \(description) создан")
        self.description = description
    }
    deinit{
        print("Экземпляр \(self.description) уничтожен")
    }
}
var myVar1 = myClass(description: "ОДИН")
if true {
    var myVar2 = myClass(description: "ДВА")
}
```

Консоль:

```
Экземпляр ОДИН создан
Экземпляр ДВА создан
Экземпляр ДВА уничтожен
```

Экземпляр `myVar2` имеет область видимости, ограниченную оператором `if`. Несмотря на то что мы не выполняли принудительное удаление экземпляра, для него был вызван деинициализатор, в результате он был автоматически удален.

Причина удаления второго экземпляра лежит в области видимости хранящей его переменной. Первый экземпляр инициализируется вне оператора `if`, а значит, является глобальным для всей программы. Второй экземпляр является локальным для условного оператора. Как только выполнение тела оператора завершается, область видимости объявленной в нем переменной заканчивается и она вместе с хранящимся в ней экземпляром автоматически уничтожается.

Количество ссылок на экземпляр

Рассмотрим пример, в котором на один экземпляр указывает несколько разных ссылок (листинг 22.2).

Листинг 22.2

```
class myClass{
    var description: String
    init(description: String){
        print("Экземпляр \(description) создан")
        self.description = description
    }
    deinit{
        print("Экземпляр \(self.description) уничтожен")
    }
}
var myVar1 = myClass(description: "ОДИН")
var myVar2 = myVar1
myVar1 = myClass(description: "ДВА")
myVar2 = myVar1
```

Консоль:

```
Экземпляр ОДИН создан
Экземпляр ДВА создан
Экземпляр ОДИН уничтожен
```

В переменной `myVar1` изначально хранится ссылка на экземпляр класса `myClass`. После записи данной ссылки в переменную `myVar2` на созданный экземпляр уже указывают две ссылки. В результате этот экземпляр удаляется лишь тогда, когда удаляется последняя ссылка на него.

Не забывайте, что экземпляры классов в Swift передаются не копированием, а по ссылке.

ПРИМЕЧАНИЕ Экземпляр существует до тех пор, пока на него указывает хотя бы одна ссылка.

22.2. Утечки памяти

Утечка памяти может привести к самым печальным результатам, одним из которых может быть отказ пользователей от вашей программы.

Пример утечки памяти

Рассмотрим ситуацию, при которой может возникнуть утечка памяти. Разработаем класс, который может описать человека и его родственные отношения с другими людьми. Для этого в качестве типа свойств класса будет указан сам тип (листинг 22.3).

Листинг 22.3

```
class Human {
    let name: String
    var child = [Human?]()
    var father: Human?
    var mother: Human?
    init(name: String){
        self.name = name
    }
    deinit {
        print(self.name+" - удален")
    }
}
if 1==1 {
    var Kirill = Human(name: "Кирилл")
    var Olga = Human(name: "Ольга")
    var Aleks = Human(name: "Алексей")
    Kirill.father = Aleks
    Kirill.mother = Olga
    Aleks.child.append(Kirill)
    Olga.child.append(Kirill)
}
```

На консоль ничего не выводится, хотя все операции выполняются в теле условного оператора, то есть в ограниченной области видимости. Нами создано три экземпляра, указаны перекрестные ссылки друг на друга, но эти экземпляры вовремя не удаляются.

Экземпляры остаются неудаленными, поскольку к моменту, когда их область видимости заканчивается, ссылки на объекты все еще существуют, и Swift не может понять, какие из ссылок можно удалить, а какие нельзя.

Это типичный пример утечки памяти в приложениях.

Сильные и слабые ссылки

Swift пытается не позволить программе создавать ситуации, приводящие к утечкам памяти. Представьте, что произойдет, если объекты, занимающие большую область памяти, не будут удаляться, занимая драгоценное свободное пространство. В конце концов приложение «упадет». Такие ситуации приведут к потере пользователей приложения.

Для решения описанной ситуации, когда Swift не знает, какие из ссылок можно удалять, а какие нет, существует специальный механизм,

называемый *сильными* и *слабыми ссылками*. Все создаваемые ссылки на экземпляр по умолчанию являются сильными. И когда два объекта указывают друг на друга сильными ссылками, то Swift не может принять решение о том, какую из ссылок можно удалить первой. Для решения проблемы некоторые ссылки можно преобразовать в слабые.

Слабые ссылки определяются с помощью ключевых слов `weak` и `onowned`. Модификатор `weak` указывает на то, что хранящаяся в параметре ссылка может быть в автоматическом режиме заменена на `nil`. Поэтому модификатор `weak` доступен только для опционалов. Но помимо опционалов бывают типы данных, которые обязывают переменную хранить значение (все неопциональные типы данных). Для создания слабых ссылок на неопционалы служит модификатор `unowned`.

Перепишем пример из предыдущего листинга, преобразовав в слабые ссылки в свойствах `father` и `mother` (листинг 22.4).

Листинг 22.4

```
class Human {
    let name: String
    var child = [Human?]()
    weak var father: Human?
    weak var mother: Human?
    init(name: String){
        self.name = name
    }
    deinit {
        print(self.name+" - удален")
    }
}
if 1==1 {
    var Kirill = Human(name: "Кирилл")
    var Olga = Human(name: "Ольга")
    var Aleks = Human(name: "Алексей")
    Kirill.father = Aleks
    Kirill.mother = Olga
    Aleks.child.append(Kirill)
    Olga.child.append(Kirill)
}
```

Консоль:

```
Алексей - удален
Ольга - удален
Кирилл - удален
```

В результате все три объекта будут удалены, так как после удаления слабых ссылок никаких перекрестных ссылок не остается.

Указанные ключевые слова можно использовать только для хранилища определенного экземпляра класса. Например, вы не можете указать слабую ссылку на массив экземпляров или на кортеж, состоящий из экземпляров класса.

22.3. Автоматический подсчет ссылок

Хотя в названии данной главы фигурирует аббревиатура ARC (Automatic Reference Counting — автоматический подсчет ссылок), но в ходе изучения мы еще ни разу к ней не обращались. На самом деле во всех наших действиях с экземплярами классов всегда участвовал механизм автоматического подсчета ссылок.

Понятие ARC

ARC в Swift автоматически управляет занимаемой памятью, удаляя неиспользуемые объекты. С помощью этого механизма вы можете «просто заниматься программированием», не переключаясь на задачи, которые система решает за вас в автоматическом режиме.

Как уже неоднократно повторялось, при создании нового хранилища, в качестве значения которому передается экземпляр класса, данный экземпляр помещается в оперативную память, а в хранилище записывается ссылка на него. На один и тот же экземпляр может указывать произвольное количество ссылок, и ARC в любой момент времени знает, сколько таких ссылок хранится в переменных, константах и свойствах. Как только последняя ссылка на экземпляр будет удалена или ее область видимости завершится, ARC незамедлительно вызовет деинициализатор и уничтожит объект.

Таким образом, ARC делает работу со Swift еще более удобной.

Сильные ссылки в замыканиях

Ранее мы выяснили, что использование сильных ссылок может привести к утечкам памяти. Также мы узнали, что для решения возникших проблем могут помочь слабые ссылки.

Сильные ссылки могут также стать источником проблем при их передаче в качестве входных параметров в замыкания. Захватываемые замыканиями экземпляры классов передаются по сильной ссылке и не освобождаются, когда замыкание уже не используется. Рассмотрим пример. В листинге 22.5 создается пустое опциональное замыкание, которому в зоне ограниченной области видимости передается значение (тело замыкания).

Листинг 22.5

```
class Human{
    var name = "Человек"
    deinit{
        print("Объект удален")
    }
}
var closure : (() -> ())?
if true{
    var human = Human()
    closure = {
        print(human.name)
    }
    closure!()
}
print("Программа завершена")
```

Консоль:

```
Человек
Программа завершена
```

Так как условный оператор ограничивает область видимости переменной human, содержащей экземпляр класса Human, то, казалось бы, данный объект должен быть удален вместе с окончанием условного оператора. Однако по выводу на консоль видно, что экземпляр создается, но перед завершением программы его деинициализатор не вызывается.

Созданное опциональное замыкание использует сильную ссылку на созданный внутри условного оператора экземпляр класса. Так как замыкание является внешним по отношению к условному оператору, а ссылка сильной, Swift самостоятельно не может принять решение о возможности удаления ссылки и последующего уничтожения экземпляра.

Для решения проблемы в замыкании необходимо выполнить захват переменной, указав при этом, что в переменной содержится слабая ссылка (листинг 22.6).

Листинг 22.6

```
class Human{
    var name = "Человек"
    deinit{
        print("Объект удален")
    }
}
var closure : (() -> ())?
if true{
    var human = Human()
    // измененное замыкание
    closure = {
        [unowned human] in
        print(human.name)
    }
    closure!()
}
```

Консоль:

```
Человек
Объект удален
Программа завершена
```

Захватываемый параметр `human` является локальным для замыкания и условного оператора, поэтому Swift без проблем может самостоятельно принять решение об уничтожении хранящейся в нем ссылки.

В данном примере используется модификатор `unowned`, поскольку объектный тип не является опционалом.

23 Опциональные цепочки

Как вы знаете, опционалы могут содержать некоторое значение, а могут не содержать его вовсе. Для доступа к опциональным значениям мы выполняем их принудительное извлечение, указывая знак восклицания, что в случае несуществующего значения вызывает ошибку.

23.1. Доступ к свойствам через опциональные цепочки

Но что делать, если в опциональном свойстве хранится экземпляр, к характеристикам которого требуется получить доступ?

Рассмотрим пример, в котором два класса описывают некую персону и ее место жительства (листинг 23.1).

Листинг 23.1

```
1  class Person {
2      var residence: Residence?
3  }
4  class Residence {
5      var numberOfRooms = 1
6  }
```

Экземпляры класса `Person` имеют единственное свойство со ссылкой на экземпляр класса `Residence`, который также имеет всего одно свойство.

Если вы создаете новый экземпляр класса `Person`, то свойство `residence` имеет значение `nil`, поскольку оно является опционалом. Если вы попытаетесь получить доступ к свойству `numberOfRooms`, используя принудительное извлечение, то получите ошибку, так как значения не

существует. Данный способ будет работать корректно только тогда, когда в свойстве `residence` хранится ссылка на экземпляр.

Для решения данной проблемы необходимо опциональное связывание (листинг 23.2).

Листинг 23.2

```
var man = Person()
if let manResidence = man.residence {
    print("Есть дом с \(manResidence.numberOfRooms) комнатами")
}else{
    print("Нет дома")
}
// вывод информации об отсутствии дома
```

Представленный подход позволяет решить проблему, но потребует писать лишний код, если вложенность классов в качестве опциональных свойств окажется многоуровневой.

Создадим новый класс, описывающий комнату, и добавим ссылку на экземпляр в класс `Residence` (листинг 23.3).

Листинг 23.3

```
class Person {
    var residence: Residence?
}
class Residence {
    var numberOfRooms = 1
    var room: Room?
}
class Room {
    var square: Int = 10
}
```

Как видите, для доступа к свойству `square` требуется строить вложенные конструкции опционального связывания.

Альтернативным способом доступа является использование *опциональных цепочек*. Они позволяют в одном выражении написать полный путь до требуемого элемента, при этом после каждого опционала необходимо ставить символ вопроса вместо восклицания.

Продемонстрируем это на примере (листинг 23.4). В нем создается трехуровневая структура вложенности, в которой требуется получить доступ к свойству `room`. При этом оба свойства (и `residence`, и `room`) — опционалы.

Листинг 23.4

```
var man = Person()
var residence = Residence()
var room = Room()
man.residence = residence
residence.room = room
if let square = man.residence?.room?.square {
    print("Площадь \(square) кв.м.")
}else{
    print("Комнаты отсутствуют")
}
// вывод общей площади
```

Если в каком-либо из узлов опциональной последовательности `man.residence?.room?.square` отсутствует значение (оно равно `nil`), то операция опционального связывания выполнена не будет и произойдет переход к альтернативной ветке условного оператора.

В случае, когда опциональная цепочка не может получить доступ к элементу, результатом выражения является `nil`. Для проверки доступности элемента вы можете просто сравнить ведущую к нему опциональную цепочку с `nil`.

ПРИМЕЧАНИЕ Опциональную последовательность можно было использовать и в первом примере с двухуровневой вложенностью. Это также обеспечило бы более удобный способ доступа к свойству экземпляра.

Вы можете использовать опциональные цепочки для вызова свойств, методов и сабскриптов для любого уровня вложенности типов друг в друга. Это позволяет «пробираться» через подсвойства внутри сложных моделей вложенных типов и проверять возможность доступа к свойствам, методам и сабскриптам этих подсвойств.

23.2. Установка значений через опциональные цепочки

Использование опциональных цепочек не ограничивается получением свойств и вызовом сабскриптов и методов. Они также могут быть использованы и для установки значений вложенных свойств.

Вернемся к примеру с жилищем человека (листинг 23.5). Пусть нам необходимо изменить значение свойства `square`.

Листинг 23.5

```
man.residence?.room?.square = 36
man.residence?.room?.square // выводит 36
```

Для решения поставленной задачи используется опциональная цепочка `man.residence?.room?.square`, указывающая на требуемый элемент. Если на каком-то из этапов экземпляр будет отсутствовать, программа не вызовет ошибку, а лишь не выполнит данное выражение.

23.3. Доступ к методам через опциональные цепочки

Как отмечалось ранее, опциональные цепочки могут быть использованы не только для доступа к свойствам, но и для вызова методов.

В класс `Residence` добавим новый метод, который должен обеспечивать вывод информации о количестве комнат (листинг 23.6).

Листинг 23.6

```
class Residence {
    var numberOfRooms = 1
    var room: Room?
    func returnNumberOfRooms() {
        return numberOfRooms
    }
}
```

Для вызова данного метода требуется использовать опциональную последовательность (листинг 23.7).

Листинг 23.7

```
man.residence?.printNumberOfRooms()
// вывод количества комнат
```

Принцип доступа к методу точно такой же, как к свойству.

24 Расширения

Расширения позволяют добавить новую функциональность к существующему объектному типу (классу, структуре или перечислению), а также к протоколу. Более того, вы можете расширять типы данных, доступа к исходным кодам которых у вас нет (например, типы, предоставляемые фреймворками, или фундаментальные для Swift типы данных).

ПРИМЕЧАНИЕ Подробно о назначении и работе с протоколами рассказывается далее.

Перечислим возможности расширений.

- добавление вычисляемых свойств экземпляра и вычисляемых свойств типа (`static`);
- определение методов экземпляра и методов типа;
- определение новых инициализаторов, сабскриптов и вложенных типов;
- обеспечение соответствия существующего типа протоколу.

Расширения могут добавлять новый функционал к типу, но не могут изменять уже существующий. Суть расширения состоит исключительно в наращивании возможностей, но не в их изменении.

СИНТАКСИС

```
extension ИмяРасширяемогоТипа {
    // описание новой функциональности для типа SomeType
}
```

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.

24.1. Вычисляемые свойства в расширениях

Расширения могут добавлять вычисляемые свойства экземпляра и вычисляемые свойства типа в определенный тип данных. Рассмотрим пример расширения функционала типа `Double`, создав в нем ряд новых вычисляемых свойств (листинг 24.1) и обеспечив тип `Double` возможностью работы с единицами длины.

Листинг 24.1

```
extension Double {
    var km: Double { return self * 1000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1000.0 }
    var ft: Double { return self / 3.28084 }
}
let oneInch = 25.4.mm
print("Один фут - это \(oneInch) метра")
// выводит "Один фут - это 0.0254 метра"
let threeFeet = 3.ft
print("Три фута - это \(threeFeet) метра")
// выводит "Три фута - это 0.914399970739201 метра"
```

Созданные вычисляемые свойства позволяют использовать дробные числа как конкретные единицы измерения длины. Добавленные новые свойства могут применяться для параметров и литералов типа `Double`.

В данном примере подразумевается, что значение 1.0 типа `Double` отражает величину один метр. Именно поэтому свойство `m` возвращает значение `self`.

Другие свойства требуют некоторых преобразований перед возвращением значений. Один километр — это то же самое, что 1000 метров, поэтому при запросе свойства `km` возвращается результат выражения `self * 1000`.

Чтобы после определения новых вычисляемых свойств использовать всю их мощь, требуется создавать и геттеры, и сеттеры.

ПРИМЕЧАНИЕ Расширения могут добавлять только новые вычисляемые свойства. При попытке добавить хранимые свойства или наблюдателей свойств происходит ошибка.

24.2. Инициализаторы в расширениях

Расширения могут добавлять инициализаторы к уже существующему типу. Таким образом вы можете расширить существующие 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 ведет к ошибке.

В качестве примера напишем инициализатор для типа `Double` (листинг 24.2). В этом примере создается структура, описывающая линию на плоскости. Необходимо реализовать инициализатор, принимающий в качестве входного аргумента экземпляр линии и устанавливающий значение, соответствующее длине линии.

Листинг 24.2

```
import UIKit
// сущность "линия"
struct Line{
    var pointOne: (Double, Double)
    var pointTwo: (Double, Double)
}
// расширения для Double
extension Double {
    init(line: Line){
        self = sqrt(pow((line.pointTwo.0 - line.pointOne.0),2) +
        pow((line.pointTwo.1 - line.pointOne.1),2))
    }
}
var myLine = Line(pointOne: (10,10), pointTwo: (14,10))
var lineLength = Double(line: myLine) // выводит 4
```

Библиотека UIKit обеспечивает доступ к математическим функциям `sqrt(_:)` и `pow(_:_:)` (соответственно квадратный корень и возведение в степень), которые требуются для вычисления длины линии на плоскости.

Структура `Line` описывает сущность «Линия», в свойствах которой указываются координаты точек ее начала и конца. Созданный в расширении инициализатор принимает на входе экземпляр класса `Line` и на основе значений его свойств вычисляет требуемое значение.

При разработке нового инициализатора в расширении будьте крайне внимательны к тому, чтобы к завершению инициализации каждое из свойств имело определенное значение.

24.3. Методы в расширениях

Следующей рассматриваемой функцией расширений является создание новых методов в расширяемом типе данных. Рассмотрим пример (листинг 24.3). В этом примере путем расширения типа `Int` мы добавляем метод `repetitions`, принимающий на входе замыкание типа `() -> ()`. Данный метод предназначен для того, чтобы выполнять переданное замыкание столько раз, сколько указывает собственное значение целого числа.

Листинг 24.3

```
extension Int {
    func repetitions(task: () -> ()) {
        for _ in 0..<self {
            task()
        }
    }
}
3.repetitions({
    print("Swift")
})
/* выводит
    Swift
    Swift
    Swift */
```

Для изменения свойств перечислений и структур реализуемыми расширением методами необходимо не забывать использовать модификатор `mutating`. В следующем примере реализуется метод `square()`, который возводит в квадрат собственное значение экземпляра. Так как тип `Int` является структурой, то для изменения собственного значения экземпляра необходимо использовать ключевое слово `mutating` (листинг 24.4).

Листинг 24.4

```
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
someInt.square()
```

24.4. Сабскрипты в расширениях

Помимо свойств, методов и инициализаторов, расширения позволяют создавать новые сабскрипты.

Создаваемое в листинге 24.5 расширение типа `Int` реализует новый сабскрипт, который позволяет получить определенную цифру собственного значения экземпляра. В сабскрипте указывается номер позиции числа, которое необходимо вернуть.

Листинг 24.5

```
extension Int {
    subscript(var digitIndex: Int) -> Int {
        var decimalBase = 1
        while digitIndex > 0 {
            decimalBase *= 10
            --digitIndex
        }
        return (self / decimalBase) % 10
    }
}
746381295[0]
// возвращает 5
746381295[1]
// возвращает 9
```

Если у числа отсутствует цифра с запрошенным индексом, возвращается 0, что не нарушает логику работы.

25 Протоколы

В процессе изучения Swift мы уже неоднократно встречались с протоколами, но каждый раз мы касались их косвенно, без подробного изучения механизмов взаимодействия с ними.

Протоколы содержат перечень свойств, методов и сабскриптов, которые должны быть реализованы в объектном типе. Протокол сам непосредственно не реализует какой-либо функционал, он лишь является своеобразным набором правил и требований к типу. Любой объектный тип данных может принимать протокол. Наиболее важной функцией протокола является обеспечение целостности объектных типов путем указания требований к их реализации.

Протоколы объявляются независимо от каких-либо элементов программы, так же как и объектные типы данных.

СИНТАКСИС

```
protocol ИмяПротокола {
    // тело протокола
}
```

Для объявления протокола используется ключевое слово protocol, после которого указывается имя создаваемого протокола.

Для того чтобы принять протокол к исполнению каким-либо объектным типом, необходимо написать его имя через двоеточие сразу после имени реализуемого типа:

```
struct ИмяПринимающейСтруктуры: ИмяПротокола{
    // тело структуры
}
```

После указания имени протокола при объявлении объектного типа данный тип обязан выполнить все требования протокола. Вы можете указать произвольное количество принимаемых протоколов.

Если класс не только принимает протоколы, но и наследует некоторый класс, то имя суперкласса необходимо указать первым, а за ним через запятую — список протоколов:

```
class ИмяПринимающегоКласса: ИмяСуперКласса, Протокол1,
      Протокол2{
    // тело класса
}
```

25.1. Требуемые свойства

Протокол может потребовать соответствующий ему тип реализовать свойство экземпляра или свойство типа (`static`), имеющее конкретные имя и тип данных. При этом протокол не указывает на вид свойства (хранимое или вычисляемое). Также могут быть указаны требования к доступности и изменяемости параметра.

Если у свойства присутствует требование доступности и изменяемости, то в качестве данного свойства не могут выступать константа или вычисляемое свойство «только для чтения». Требование доступности обозначается с помощью конструкции `{get}`, а требование доступности и изменяемости — с помощью конструкции `{get set}`.

Создадим протокол, содержащий ряд требований к принимающему его типу (листинг 25.1).

Листинг 25.1

```
protocol SomeProtocol {
    var mustBeSettable: Int { get set }
    var doesNotNeedToBeSettable: Int { get }
}
```

Протокол `SomeProtocol` требует, чтобы в принимающем типе были реализованы два изменяемых (`var`) свойства типа `Int`. При этом свойство `mustBeSettable` должно быть и доступным и изменяемым, а свойство `doesNotNeedToBeSettable` — как минимум изменяемым.

Протокол определяет минимальные требования к типу, то есть тип данных обязан реализовать все, что описано в протоколе, но он может не ограничиваться этим набором элементов. Так, для свойства `doesNotNeedToBeSettable` из предыдущего листинга может быть реализован не только геттер, но и сеттер.

Для указания в протоколе на свойство типа необходимо использовать модификатор `static` перед ключевым словом `var`. Данное требование выполняется даже в том случае, если протокол принимается структурой или перечислением (листинг 25.2).

Листинг 25.2

```
protocol AnotherProtocol {
    static var someTypeProperty: Int { get set }
}
```

В данном примере свойство типа `someTypeProperty` должно быть обязательно реализовано в принимающем типе данных.

В следующем примере мы создадим протокол и принимающий его требования класс (листинг 25.3).

Листинг 25.3

```
protocol FullyNamed {
    var fullName: String { get }
}
struct Person: FullyNamed {
    var fullName: String
}
let john = Person(fullName: "John Wick")
```

В данном примере определяется протокол `FullyNamed`, который обязывает структуру `Person` иметь доступное свойство `fullName` типа `String`.

25.2. Требуемые методы

Протокол может требовать реализации определенного метода экземпляра или метода типа. Форма записи для этого подобна указанию требований к реализации свойств.

Для требования реализации метода типа необходимо использовать модификатор `static`. Также протокол может описывать изменяющий метод. Для этого служит модификатор `mutating`.

ПРИМЕЧАНИЕ Если вы указали ключевое слово mutating перед требованием метода, то вам уже не нужно указывать его при реализации метода в классе. Данное ключевое слово требуется только в реализации структур.

При реализации метода в типе данных необходимо в точности соблюдать все требования протокола, в частности имя метода, наличие или отсутствие входных аргументов, тип возвращаемого значения, модификаторы (листинг 25.4).

Листинг 25.4

```
protocol RandomNumberGenerator {
    func random() -> Double
    static func getRandomString()
    mutating func changeValue(newValue: Double)
}
```

В данном примере реализован протокол `RandomNumberGenerator`, который содержит требования реализации трех методов. Метод экземпляра `random()` должен возвращать значение типа `Double`. Метод `getRandomString` должен быть методом типа, при этом требования к возвращаемому им значению не указаны. Метод `changeValue(_:)` должен быть изменяющим и принимать в качестве входного параметра значение типа `Double`.

Данный протокол не делает никаких предположений относительно того, как будет вычисляться случайное число, ему важен лишь факт выполнения требований.

25.3. Требуемые инициализаторы

Дополнительно протокол может предъявлять требования к реализации инициализаторов. Необходимо писать инициализаторы точно так же, как вы пишете их в объектном типе, опуская фигурные скобки и тело инициализатора.

Требования к инициализаторам могут быть выполнены в соответствующем классе в форме назначенного или вспомогательного инициализатора. В любом случае, перед объявлением инициализатора в протоколе необходимо указывать модификатор `required`. Это гарантирует, что вы реализуете указанный инициализатор во всех подклассах данного класса.

ПРИМЕЧАНИЕ Нет нужды обозначать реализацию инициализаторов протокола модификатором required в классах, которые имеют модификатор final.

Реализуем протокол, содержащий требования к реализации инициализатора, и класс, выполняющий требования протокола (листинг 25.5).

Листинг 25.5

```
protocol Named{
    init(name: String)
}
class Person : Named {
```

```
    var name: String
    required init(name: String){
        self.name = name
    }
}
```

25.4. Протокол в качестве типа данных

Протокол сам по себе не несет какой-либо функциональной нагрузки, он лишь содержит требования к реализации объектных типов. Тем не менее протокол является полноправным типом данных.

Используя протокол в качестве типа данных, вы указываете на то, что записываемое в данное хранилище значение должно иметь тип данных, который соответствует указанному протоколу.

Так как протокол является типом данных, вы можете организовать проверку соответствия протоколу с помощью оператора `is`, который мы обсуждали при изучении темы приведения типов. При проверке соответствия возвращается значение `true`, если проверяемый экземпляр соответствует протоколу, и `false` в противном случае.

25.5. Расширение и протоколы

Расширения могут взаимодействовать не только с объектными типами, но и с протоколами.

Добавление соответствия типа протоколу

Вы можете использовать расширения для добавления требований по соответствию некоторого объектного типа протоколу. Для этого в расширении после имени типа данных через двоеточие необходимо указать список новых протоколов (листинг 25.6).

Листинг 25.6

```
protocol TextRepresentable {
    func asText() -> String
}
extension Int: TextRepresentable {
    func asText() -> String {
        return String(self)
    }
}
5.asText()
```

В данном примере протокол `TextRepresentable` требует, чтобы в принимающем объектном типе был реализован метод `asText()`. С помощью расширения мы добавляем требование о соответствии типа `Int` к данному протоколу, при этом, поскольку где-то ранее был реализован сам тип данных, в обязательном порядке указывается реализация данного метода.

Расширение протоколов

С помощью расширений мы можем не только указывать на необходимость соответствия новым протоколам, но и расширять сами протоколы, поскольку протоколы являются полноценными типами данных. Данный функционал появился только в Swift 2.0.

При объявлении расширения необходимо использовать имя протокола, а в его теле указывать набор требований с их реализациями. После расширения протокола описанные в нем реализации становятся доступны в экземплярах всех классов, которые приняли данный протокол к исполнению.

Напишем расширение для реализованного ранее протокола `TextRepresentable` (листинг 25.7).

Листинг 25.7

```
extension TextRepresentable {
    func description() -> String {
        return "Данный тип поддерживает протокол TextRepresentable"
    }
}
5.description()
```

Расширение добавляет новый метод в протокол `TextRepresentable`. При этом ранее мы указали, что тип `Int` соответствует данному протоколу. В связи с этим появляется возможность обратиться к указанному методу для любого значения типа `Int`.

25.6. Наследование протоколов

Протокол может наследовать один или более других протоколов. При этом он может добавлять новые требования поверх наследуемых, — тогда тип, принявший протокол к реализации, будет вы-

нужден выполнить требования всех протоколов в структуре. При наследовании протоколов используется тот же синтаксис, что и при наследовании классов.

Работа с наследуемыми протоколами продемонстрирована в листинге 25.8.

Листинг 25.8

```
protocol SuperProtocol {
    var someValue: Int { get }
}
protocol SubProtocol: SuperProtocol {
    func someMethod()
}
struct SomeStruct: SubProtocol{
    let someValue: Int = 10
    func someMethod() {
        // тело метода
    }
}
```

Протокол `SuperProtocol` имеет требования к реализации свойства, при этом он наследуется протоколом `SubProtocol`, который имеет требования к реализации метода. Структура принимает к исполнению требования протокола `SubProtocol`, а значит, в ней должны быть реализованы и свойство, и метод.

25.7. Классовые протоколы

Вы можете ограничить протокол таким образом, чтобы его могли принимать к исполнению исключительно классы (а не структуры и перечисления). Для этого у протокола в списке наследуемых протоколов необходимо указать ключевое слово `class`. Данное слово всегда должно указываться на первом месте в списке наследования.

Пример создания протокола приведен в листинге 25.9. В нем мы изменим протокол `SubProtocol` таким образом, чтобы его мог принять исключительно класс.

Листинг 25.9

```
protocol SubProtocol: class, SuperProtocol {
    func someMethod()
}
```

25.8. Композиция протоколов

Иногда бывает удобно требовать, чтобы тип соответствовал не одному, а нескольким протоколам. В этом случае, конечно же, можно создать новый протокол, наследовать в него несколько необходимых протоколов и задействовать имя только что созданного протокола. Однако для решения данной задачи лучше воспользоваться *композицией протоколов*, то есть скомбинировать несколько протоколов.

СИНТАКСИС

```
protocol<Протокол1, Протокол2 ...>
```

Для композиции протоколов необходимо использовать ключевое слово protocol, после которого в угловых скобках указать список объединяемых протоколов.

В листинге 25.10 приведен пример, в котором два протокола комбинируются в единственное требование.

Листинг 25.10

```
protocol Named {
    var name: String { get }
}
protocol Aged {
    var age: Int { get }
}
struct Person: Named, Aged {
    var name: String
    var age: Int
}
func wishHappyBirthday(celebrator: protocol<Named, Aged>) {
    print("С Днем Рождения, \(celebrator.name)! Тебе уже \
    (celebrator.age)!")
}
let birthdayPerson = Person(name: "Джон Уик", age: 46)
wishHappyBirthday(birthdayPerson)
// выводит "С Днем Рождения, Джон Уик! Тебе уже 46!"
```

В данном примере объявляются два протокола: `Named` и `Aged`. Созданная структура принимает оба протокола и в полной мере выполняет их требования.

Входным аргументом функции `wishHappyBirthday()` должно быть значение, которое удовлетворяет обоим протоколам. Таким значением является экземпляр структуры `Person`, который мы и передаем.

ПРИМЕЧАНИЕ Ранее мы встречались с типом Any. Данный тип позволяет нам передавать значения произвольного типа данных. Any является псевдонимом пустой композиции protocol<>, поэтому поддерживает передачу любого значения.

26 Разработка первого приложения

Целью данной главы является более глубокое погружение в разработку приложений в Xcode. Как вы уже знаете, playground не предназначен для создания полноценных приложений, но набросать небольшой проект и протестировать его не выходя из среды playground-проекта, вполне возможно. Именно этим мы и займемся.

Изучение языка само по себе не дает вам навыков разработки приложений. Для этого требуется изучить большое количество дополнительного материала, связанного не столько с программированием на Swift, сколько с разработкой в Xcode.

Данная глава познакомит вас со структурой проекта, понятием API, разграничением доступа к различным объектам, а также покажет важность работы с документацией.

26.1. Важность работы с документацией

Думаю, что вы уже неоднократно убеждались, что Swift значительно интереснее, чем можно было бы ожидать при первом знакомстве. Порой при разработке программ мы попадаем в такие ситуации, для корректного разрешения которых приходится обращаться к документации, но эффект от решения возникшей проблемы доставляет программисту истинное удовольствие.

Swift имеет большое количество разнообразных типов, которые используются в зависимости от возникшей ситуации. Эта глава посвящена тому, чтобы понять, что это за типы, и как находить ответ на вопрос «Что делать?» при встрече значения совершенно неизвестного вам типа данных.

Ранее неоднократно отмечалось, что в Swift «все — это объект», и каждый раз этому приводились подтверждения, например, когда у обыч-

ного целого числа появлялись свойства и методы, позволяющие работать с ним уже не просто как с числом, а с довольно функциональным объектом.

В Swift реализован очень интересный подход к организации типов данных. Но что же такое тип данных по своей сути? Тип данных — это некая функциональная конструкция, определяющая способ хранения информации и операции, которые с этой информацией можно выполнять.

Помимо самих типов существуют конструкции, расширяющие их возможности. Например, упомянутый нами ранее протокол `Hashable` — это не что иное, как расширение возможностей типов данных. Если какой-либо тип данных реализует данный протокол, то он гарантирует то, что функция вычисления хеша и сопоставления элементов данного типа будет выполняться корректно.

В Swift существует протокол `CollectionType`. Он выставляет требования хранения множества элементов в виде одного значения, при этом каждый элемент должен иметь уникальный адрес. В результате требования данного протокола выполняют такие конструкции, как `Array`, `Set` и `Dictionary`.

Вернемся к среде разработки Xcode. Объявим переменную типа `String` (листинг 26.1).

Листинг 26.1

```
1  var str: String
```

Теперь, удерживая клавишу Option, наведите указатель мыши на имя типа данных `String`. Когда значок указателя сменится со стрелки на знак вопроса, щелкните левой кнопкой мыши. Перед вами появится окно со справочной информацией о типе данных.

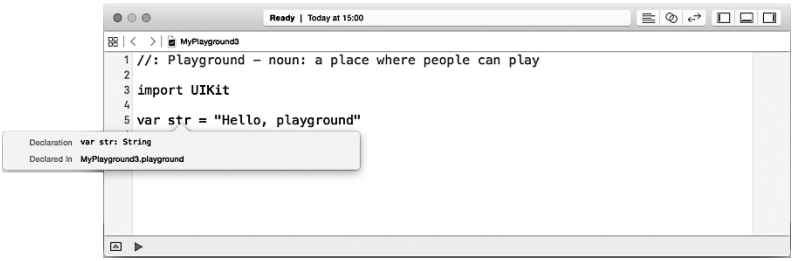


Рис 26.1. Окно со справочной информацией о переменной

Не отпуская клавишу Option, наведите указатель мыши на имя переменной `str` и снова щелкните левой кнопкой мыши. Перед вами появится уже знакомое окно со справочной информацией о данной переменной (рис. 26.1).

В строке Declaration щелкните на слове String, после чего Xcode откроет окно документации для типа данных `String` (рис. 26.2).

ПРИМЕЧАНИЕ Для загрузки документации необходим доступ в Интернет. Кроме того, в меню настроек Xcode вы можете загрузить документацию для автономного доступа к ней.

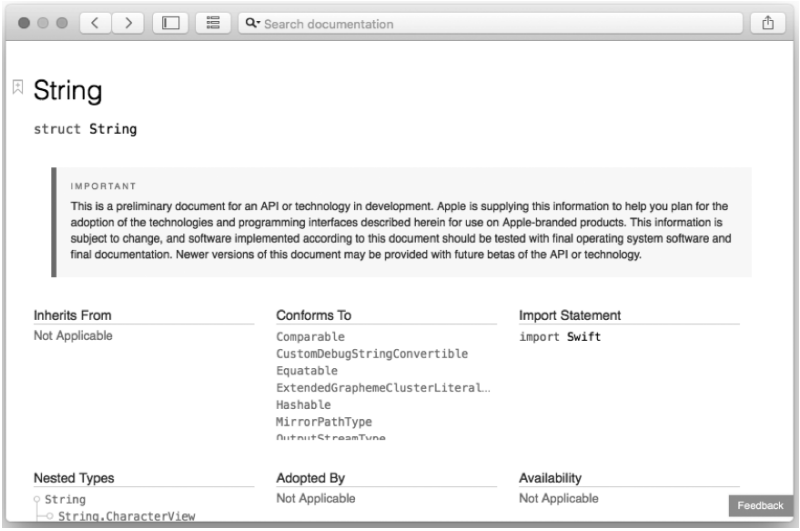


Рис. 26.2. Окно документации о типе данных String

Используя документацию от Apple, вы можете найти ответ практически на любой вопрос, связанный с разработкой на языке Swift или использованием Xcode.

В разделе Conforms To перечислены протоколы, которым соответствует тип данных `String`, среди них вы можете увидеть и упомянутый ранее `Hashable`. Это говорит о том, что значения данного типа хешируемы и могут быть сравнены между собой.

Перейдем к примеру, с которым мы встречались при изучении словарей (листинг 26.2).

Листинг 26.2

```
1  var countryDict = ["RUS":"Российская Федерация",
                      "BEL":"Беларусь",
                      "UKR":"Украина"]            ["BEL": "Беларусь",
                                                   "UKR": "Украина",
                                                   "RUS": "Российская
                                                    Федерация"]
2  // все ключи словаря countryDict
3  var keysOfDict = countryDict.keys              LazyMapCollection
                                                  <Dictionary<String,
                                                   String>, String>
4  // все значения словаря countryDict
5  var valuesOfDict = countryDict.values          LazyMapCollection
                                                  <Dictionary<String,
                                                   String>, String>
```

Свойства `keys` и `values` служат для того, чтобы получить коллекцию, состоящую только из ключей или только из значений данного словаря. При обращении к `keys` и `values` Swift возвращает не массив, набор или словарь, а некую конструкцию, которая имеет тип данных:

```
LazyMapCollection<Dictionary<String, String>, String>
```

Удерживая клавишу Option и щелкнув на имени переменной `keysOfDict`, можно открыть окно со справочной информацией и из этого окна перейти к документации, относящейся к конструкции `LazyMapCollection` (рис. 26.3).

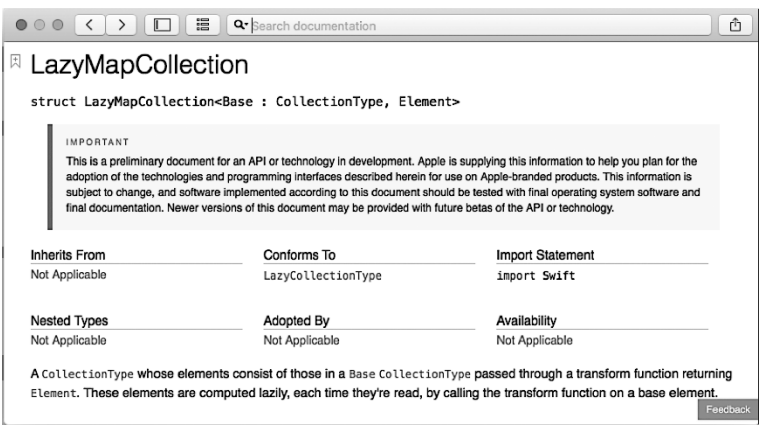


Рис. 26.3. Документация для типа LazyMapCollection

Слово `Collection` в названии типа говорит о том, что данный тип — это коллекция элементов.

Вверху страницы приведено полное наименование типа данных:
`struct LazyMapCollection<Base : CollectionType, Element>`
Если ознакомиться с описанием, которое расположено ниже на странице документации, то становится ясно, что значение данного типа — не что иное, как коллекция, которая состоит из элементов базовой коллекции, обозначенной как `Base`. Элементы данной коллекции преобразуются специальной функцией и возвращаются как некий элемент `Element`.

Обратимся к разделу Conforms To. В нем указан всего один протокол `LazyCollectionType`. Щелкните на этом имени, чтобы перейти к соответствующей странице документации (рис. 26.4).

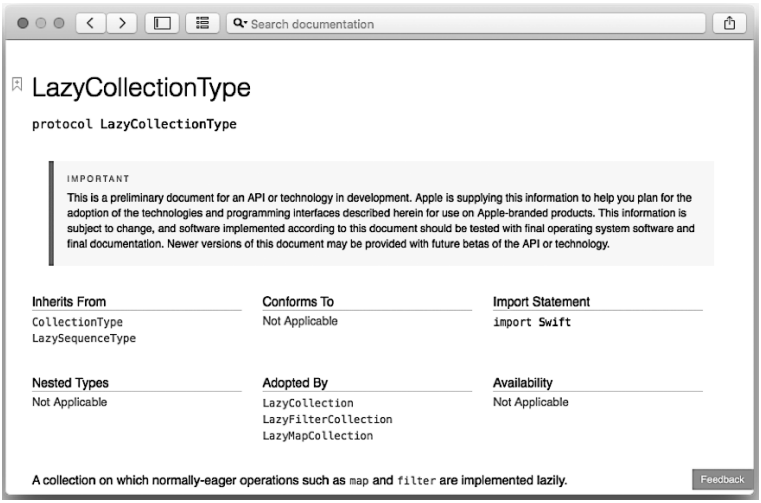


Рис. 26.4. Документация для протокола LazyCollectionType

Обратите внимание на раздел Inherits From: в нем указан протокол `CollectionType`, который наследуется в `LazyCollectionType`.

`CollectionType` — это не что иное, как протокол, который реализует требования уже известных нам коллекций элементов, например словаря, массива или набора. Убедимся в этом, перейдя на страницу документации протокола `CollectionType` (рис. 26.5).

В разделе Adopted By представлены конструкции, которые выполняют требования данного протокола. Среди них присутствуют и `Array`, и `Dictionary`, и `Set`.

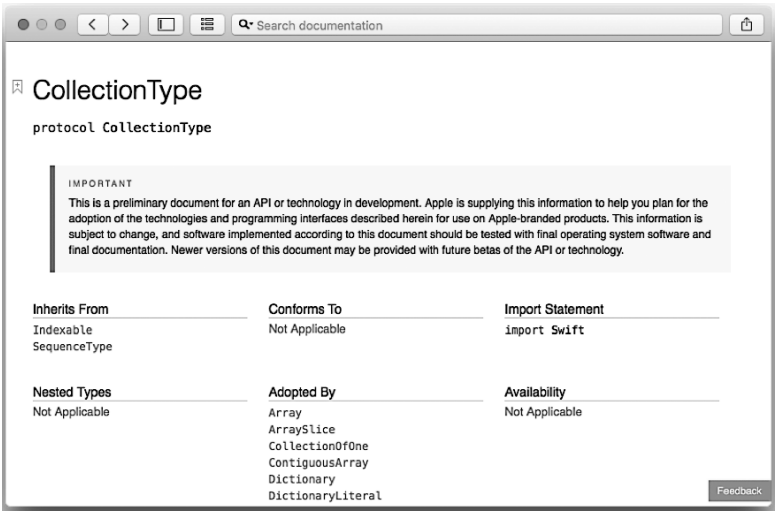


Рис. 26.5. Документация для протокола CollectionType

В результате получается незамысловатая схема зависимостей протоколов и типов данных, изображенная на рис. 26.6.

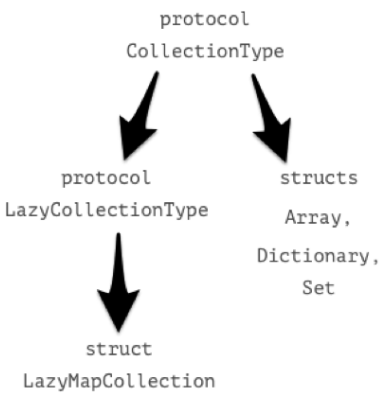


Рис. 26.6. Схема зависимостей протоколов и типов данных

Получается, что конструкции `Array`, `Dictionary` и `Set`, как и конструкция `LazyMapCollection`, реализуют один и тот же протокол `CollectionType`, а значит, являются коллекциями.

Вернемся к странице описания типа `LazyMapCollection` и некоему возвращаемому элементу `Element`. Сейчас необходимо найти, где указан тип данного элемента. Логично предположить, что он зависит от базовой коллекции, то есть в примере из листинга 26.2 — от словаря `countryDict`. Вернемся в окно кода Xcode и повторно вызовем справочную информацию о переменной `keysOfDict`. Вы увидите следующие данные:

```
var keys: LazyMapCollection<Dictionary<String, String>, String>
```

Получается, что тип данных этой коллекции выглядит так:

```
LazyMapCollection<Dictionary<String, String>, String>
```

Первый параметр (`Dictionary<String, String>`) описывает базовую коллекцию `countryDict`, а второй (`String`) — тип объектов в результирующей коллекции. Это и есть тот самый элемент `Element`.

Получается, что в результате вычисления свойств `keys` и `arrays` возвращается коллекция, элементы которой имеют тип данных `String`. Вспомните, какие еще коллекции вы знаете, которые могут хранить элементы этого типа? Конечно же, это массивы и наборы.

Значит, полученное в ходе обращения к свойствам значение вы можете преобразовать в массив или набор с помощью соответствующей функции (листинг 26.3).

Листинг 26.3

```
1  var countryDict = ["RUS":"Российская Федерация", "BEL":"Беларусь",
   "UKR":"Украина"]                                  ["BEL": "Беларусь",
                                                      "UKR": "Украина",
                                                      "RUS": "Российская
                                                             Федерация"]
2  var keys = Set(countryDict.keys)                  {"BEL", "UKR", "RUS"}
3  var values = Array(countryDict.values)            ["Беларусь",
                                                      "Украина",
                                                      "Российская
                                                       Федерация"]
```

Тот же самый принцип можно применить и к результату работы метода `reverse()`, который возвращает реверсивный массив. Если посмотреть справку для примера из листинга 26.4, то видно, что типом данных возвращаемой коллекции является `ReverseRandomAccessCollection <Array<Int>>`. Перейдя к описанию данного типа, можно заключить, что это — обыкновенная коллекция, которая возвращает те же самые элементы, что и базовая коллекция, а типом базовой и возвращаемой

коллекции является `Array<Int>` (или `<[Int]>`). Это не что иное, как массив целых чисел.

Листинг 26.4

```
1  var someArray = [1, 3, 5, 7, 9]                [1, 3, 5, 7, 9]
2  var reverseSomeArray = someArray.reverse()     ReverseRandomAccess
                                                  Collection<Array<Int>>
```

Таким образом, результат работы метода `reverse()` можно не преобразовывать в массив с помощью функции `Array()`, как в примере со свойствами `keys` и `values`, а присвоить напрямую массиву типа `Array<Int>` (листинг 26.5).

Листинг 26.5

```
1  var someArray = [1, 3, 5, 7, 9]                    [1, 3, 5, 7, 9]
2  let resArray: Array<Int> = someArray.reverse()     [9, 7, 5, 3, 1]
```

Основное назначение данной главы состоит в том, чтобы вы поняли, насколько важно пользоваться документацией, которой мы, к счастью, не обделены. В документации можно найти ответы на бóльшую часть возникающих у вас вопросов.

26.2. Модули

Структура проекта Playground

Playground-проект, создаваемый в Xcode, — это упрощенная версия проекта полноценного приложения. С разработкой приложений вы еще не знакомы, но после изучения материала, приведенного в книге, сможете приступить к реализации своих идей.

Сейчас вам необходимо создать новый playground-проект. Назовите его «Balls». После этого удалите весь код, размещенный изначально в проекте. Взгляните в верхний правый угол рабочего окна: там расположены шесть кнопок, позволяющих настраивать рабочую зону Xcode (рис. 26.7).

Кнопки разбиты на две панели, по три кнопки в каждой. Левая панель отвечает за внешний вид рабочей зоны вашего окна и содержит следующие кнопки:

- **Standart Editor**

  Активна по умолчанию. Позволяет отобразить рабочую область в привычном нам виде: с редактором кода и областью результатов.

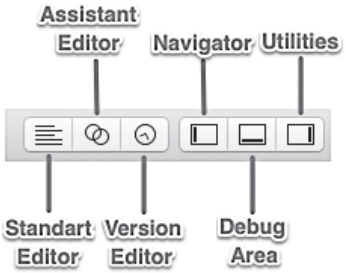


Рис. 26.7. Окно документации

- **Assistant Editor**

  Добавляет дополнительную область — «Ассистент разработки». По умолчанию она расположена правее редактора кода и содержит различную вспомогательную информацию, включая графическое отображение playground-страниц (рис. 26.8).

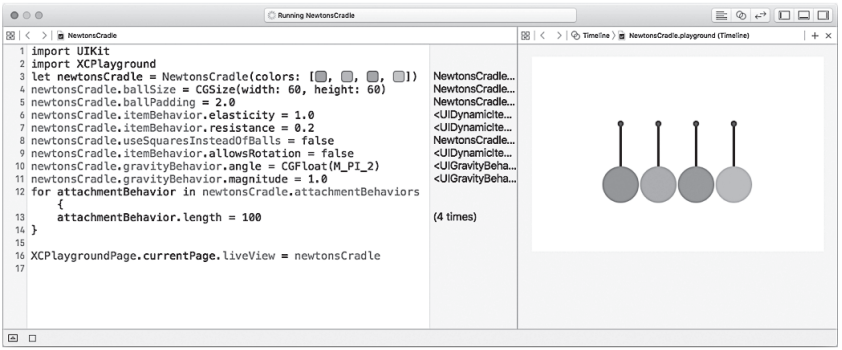


Рис. 26.8. Пример работы в режиме Assistant Editor

- **Version Editor**

  Позволяет работать с версионностью файлов. Данный режим работы доступен только при активной системе контроля версий.

Правая панель позволяет отобразить ряд вспомогательных областей:

- **Navigator**

  Отображает Project Navigator, содержащий структуру вашего проекта.

- **Debug Area**

  Открывает уже знакомую вам Debug Area.

- **Utilities**

  Открывает панель Utilities, отображающую некоторые характеристики проекта, а также справочную информацию по выделенному элементу редактора кода.

Откройте Project Navigator (рис. 26.9).

ПРИМЕЧАНИЕ Данная панель также вызывается сочетанием клавиш Cmd-1.

Project Navigator, как было сказано ранее, отображает структуру проекта. В настоящий момент проект «Balls» состоит из одного файла и двух пустых папок: Source и Resources.

Обычно в Source помещаются различные функциональные блоки, вроде файлов с исходным кодом. Resources в свою очередь предназначена для хранения ресурсов, используемых в проекте, таких как картинки, видео- и аудиофайлы и т. д.

Чуть ранее я упомянул про страницы playground-проекта. Весь код, который вы писали ранее, вы писали именно на странице проекта. Так как в каждом из ваших проектов такая страница была одна, она не отображалась отдельной записью в структуре проекта и доступ к ней происходил по нажатию на главный файл с именем проекта.

Вы можете создавать новые файлы, включая страницы, с помощью кнопки с изображением символа «плюс» в нижнем левом углу Project Navigator.

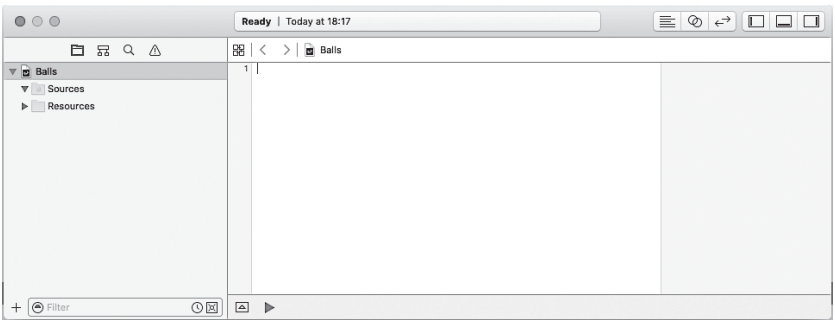


Рис. 26.9. Панель Project Navigator

Модули и API

Каждая программа или отдельный проект в Xcode — это модуль.

Модуль — это единый функциональный блок, выполняющий определенные задачи. Модули могут взаимодействовать между собой. Каждый внешний модуль, который вы используете в своей программе, для вас «черный ящик». Вам недоступна его внутренняя реализация — вы знаете лишь то, какие функции данный модуль позволяет выполнить (то есть что дать ему на вход и что получите на выходе). Модули состоят из исходных файлов и ресурсов.

Исходный файл — отдельный файл, содержащий программный код и разрабатываемый в пределах одного модуля.

Для того чтобы из набора файлов получить модуль, необходимо провести компиляцию, то есть из кода, понятного разработчику и среде программирования, получается файл (модуль), понятный компьютеру. Модуль может быть собран как из одного, так и из множества исходных файлов и ресурсов. В качестве модуля может выступать, например, целая программа, или фреймворк.

С фреймворками (или библиотеками) мы встречались ранее: вспомните про Foundation и UIKit, которые мы подгружали в программу с помощью директивы import. Данная директива служит для обеспечения доступа к функционалу внешней библиотеки. Для доступа к ее функциям чаще всего существует специальный набор правил, то есть интерфейс, называемый API.

API (application programming interface) — это набор механизмов (обычно функций и типов данных), включенных в состав некоторой библиотеки (модуля). API доступны при условии, что содержащая их библиотека подключена к проекту (импортирована в проект).

Если вернуться к фреймворку Foundation, то функция `arc4random_uniform()`, которую мы использовали ранее, является одной из большого перечня доступных API-функций. Пример подключения фреймворка к библиотеке приведен в листинге 26.6.

Листинг 26.6

```
1  import Foundation
```

При разработке приложений вы будете использовать большое количество различных библиотек, которые в том числе поставляются вместе с Xcode. Самое интересное, что Xcode содержит просто гигантское

количество возможностей: работа с 2D и 3D, различные визуальные элементы, физические законы и многое-многое другое. И все это реализуется благодаря дополнительным библиотекам. В этой главе вы познакомитесь с некоторыми их возможностями.

ПРИМЕЧАНИЕ Запомните, что одни библиотеки могут подключать другие. Так, например, UIKit подгружает в проект Foundation, и дополнительное подключение Foundation не требуется.

26.3. Разграничение доступа

Уровни доступа

База системы разграничения доступа при разработке приложений строится на основе понятия «модуль».

Всего Swift предлагает три различных уровня доступа для объектов вашего кода:

`public`

Открытый. Данный уровень открывает полную свободу использования объекта. Вы можете импортировать некоторый модуль и свободно использовать его public-объекты в своем коде.

`internal`

Внутренний. Данный уровень используется в случаях, когда необходимо ограничить использование объекта самим модулем. Таким образом объект будет доступен во всех исходных файлах модуля.

`private`

Частный. Данный уровень позволяет использовать объект только в пределах одного исходного файла.

ПРИМЕЧАНИЕ Частный уровень доступа в Swift отличается от большинства частных уровней в других языках программирования тем, что он действует на исходный файл целиком, а не только на функциональный блок, в котором он реализован. Таким образом, любой тип имеет доступ к private-объектам, если они определены в одном и том же файле.

По умолчанию все объекты вашего кода имеют уровень доступа internal. Для того чтобы изменить его, необходимо явно указать уровень. При этом, если вы разрабатываете фреймворк, то для того чтобы сделать некоторый объект частью API, вам необходимо изменить его уровень доступа на public.

Синтаксис определения

Чтобы определить уровень доступа к некоторому объекту, необходимо указать соответствующее ключевое слово (public, internal, private) перед определением объекта (`func`, `property`, `class`, `struct` и т. д.). Пример приведен в листинге 26.7.

Листинг 26.7

```
1  public class SomePublicClass {}
2  internal class SomeInternalClass {}
3  private class SomePrivateClass {}
4  public var somePublicVar = 0
5  private var somePrivatelet = 0
6  internal func someInternalFunc() {}
```

ПРИМЕЧАНИЕ Если уровень доступа к вашему объекту предполагается internal, то можно его не указывать, так как по умолчанию для любого объекта назначен именно этот уровень.

Уровень доступа к типам данных

Следует подробнее остановиться на определении уровня доступа различных типов объектов.

Как мы уже неоднократно говорили, Swift позволяет определять собственные типы данных. Если вам требуется указать уровень доступа к типу данных или его членам, то это необходимо сделать в момент определения типа. Новый тип данных может быть использован там, где это позволяет его уровень доступа.

Если ваш объект имеет вложенные объекты (например, класс со свойствами и методами), то уровень доступа родителя определяет уровни доступа к его членам. Таким образом, если вы укажете уровень доступа private, то все его члены по умолчанию будут иметь уровень доступа private. Для уровней доступа public и internal уровень доступа членов — internal.

Рассмотрим пример из листинга 26.8.

Листинг 26.8

```
1  public class SomePublicClass { //public класс
2     public var somePublicProperty = 0 // public свойство
3     var someInternalProperty = 0 // internal свойство
4     private func somePrivateMethod() {} // private метод
5  }
```

```
6
7  class SomeInternalClass { // internal класс
8     var someInternalProperty = 0 // internal свойство
9     private func somePrivatemethod() {} // private метод
10 }
11
12 private class SomeInternalClass { // private класс
13    var someInternalProperty = 0 // private свойство
14    func somePrivatemethod() {} // private метод
15 }
```

Всего было определено три класса с различными уровнями доступа. Из примера видно, как влияет определение уровня доступа к классу на доступ к его членам.

Обратите внимание, что при наследовании уровень доступа подкласса не может быть выше уровня родительского класса.

Уровень доступа к кортежу типа данных определяется наиболее строгим типом данных, включенным в кортеж. Так, например, если вы скомпонуете кортеж типа из двух разных типов, один из которых будет иметь уровень доступа internal, а другой — private, то результирующий уровень доступа будет private, то есть самый строгий.

ПРИМЕЧАНИЕ Запомните, что Swift не позволяет явно указать тип данных кортежа. Он вычисляется автоматически.

Уровень доступа к функции определяется самым строгим уровнем типов аргументов функции и типа возвращаемого значения. Рассмотрим пример из листинга 26.9.

Листинг 26.9

```
1  func someFunction() -> (SomeInternalClass, SomePrivateClass) {
2   // тело функции
3  }
```

Можно было ожидать, что уровень доступа функции будет равен internal, так как не указан явно. На самом деле эта функция вообще не будет скомпилирована. Это связано с тем, что тип возвращаемого значения — это кортеж с уровнем доступа private. При этом тип этого кортежа определяется автоматически на основе типов данных, входящих в него.

В связи с тем, что уровень доступа функции — private, его необходимо указать явно (листинг 26.10).

Листинг 26.10

```
1  private func someFunction() -> (SomeInternalClass,
SomePrivateClass) {
2    // тело функции
3  }
```

Что касается перечислений, стоит обратить внимание на то, что каждый член перечисления получает тот же уровень доступа, что установлен для самого перечисления.

26.4. Разработка интерактивного приложения

Постановка задачи

Что может быть интереснее, чем интерактивное приложение, в котором можно двигать, щелкать, толкать и т. д. Разработкой именно такого приложения мы и займемся.

Для реализации мы возьмем следующую задачу: в квадратном ящике расположены несколько цветных шариков. Пользователь может перемещать шарики и сталкивать их друг с другом.

Interactive Playgrounds, UIKit и XCPlayground

Начиная с версии 7.3 Xcode обладает потрясающим механизмом, называемым Interactive playgrounds. С его помощью вы можете нажимать, перемещать и совершать другие полезные действия с объектами прямо в окне playground-проекта. Interactive playgrounds помогут вам быстро создать прототип приложения, тем самым уменьшив вероятность неприятных ошибок в релизной версии программы.

Interactive playgrounds наделяют вас практически теми же возможностями по построению интерфейсов, что и при полноценной разработке приложений. Именно этот механизм мы будем использовать при разработке вашего первого приложения.

В Xcode присутствует модуль, предназначенный для работы с Interactive Playground. Он называется XCPlayground.

ПРИМЕЧАНИЕ XCPlayground — модуль, обеспечивающий интерактивное взаимодействие пользователя с Xcode при работе в playground.

Наиболее важным механизмом данной библиотеки является класс XCPlaygroundPage, предназначенный для создания интерактивного содержимого страниц в playground-проекте.

XCPlayground позволяет взаимодействовать с некоторыми объектами. Поэтому нам потребуется функционал, обеспечивающий графическое построение этих объектов. И таким функционалом обладает модуль UIKit.

UIKit — это библиотека, обеспечивающая ключевую инфраструктуру, необходимую для построения iOS-приложений. UIKit содержит огромное количество классов, позволяющих строить визуальные элементы, анимировать их, обрабатывать физические законы, управлять механизмами печати, обработки текста и многое-многое другое! Это важнейшая и совершенно незаменимая библиотека функций, которую вы будете использовать при разработке каждого приложения.

Импортируем в проект Ball библиотеки XCPlayground и UIKit (листинг 26.11).

Листинг 26.11

```
1  import XCPlayground
2  Import UIKit
```

Разделим функциональную нагрузку разрабатываемой программы между исходными файлами. В панели Project Navigator в папке Source создайте новый файл с именем Balls.swift. Для этого щелкните по названию папки правой кнопкой мыши и выберите пункт New File. В результате новый файл отобразится в составе папки Sources. Вам останется лишь указать его имя (рис. 26.10). Всю функциональную начинку мы разместим в этом файле, в то время как главный файл Balls будет использовать реализованный функционал.

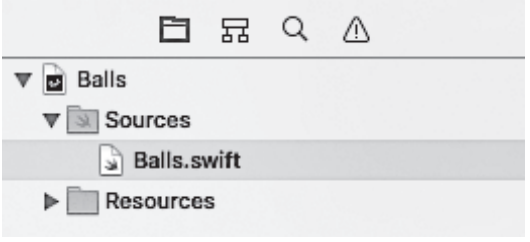


Рис. 26.10. Новый файл в Project Navigator

В только что созданном файле так же импортируйте библиотеку UIKit.

Класс Balls

Весь функционал будет заключен в одном-единственном классе, который мы назовем Balls и расположим в файле Balls.swift. Определите новый класс Balls, как это сделано в листинге 26.12.

Листинг 26.12

```
1  import UIKit
2  public class Balls: UIView  {
3  }
```

В качестве его уровня доступа класса указан public. Это связано с тем, что некоторые свойства и методы класса UIView, которые будут переопределены в Balls, также соответствуют public, а, как вы знаете, уровень доступа свойств и методов не может быть выше, чем уровень самого типа.

В состав подключенной библиотеки UIKit входит тип данных UIView, предназначенный для визуального отображения графических элементов и взаимодействия с ними. Например, мы можем отобразить прямоугольную рабочую область типа UIView, наполнив ее другими графическими элементами, каждый из которых также будет представлять собой отдельный экземпляр типа UIView. В этом и будет заключаться решение поставленной нами задачи с шариками (рис. 26.11).

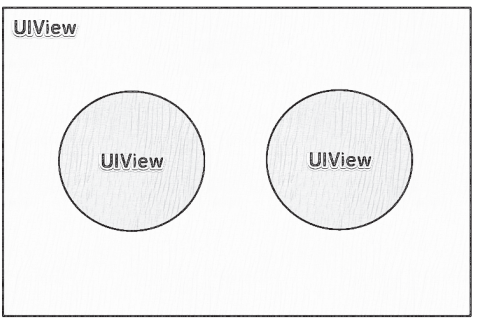


Рис. 26.11. Структура отображений

В связи с этим реализуемый класс Balls является подклассом для UIView.

У разрабатываемой системы можно выделить следующие свойства:

1. Существует прямоугольная область.
2. Внутри этой области расположены несколько шариков.
3. Каждый шарик имеет свой цвет.
4. Шарики могут перемещаться.
5. Шарики могут взаимодействовать друг с другом (ударяться).
6. Шарики могут взаимодействовать с границами прямоугольной области.

Благодаря пункту 3 вы получите очень важный опыт взаимодействия с типом данных UIColor, который позволяет работать с цветовой палитрой.

Мы будем определять количество шариков, передавая массив цветов. Каждый из переданных цветов будет указывать на один шарик. Объявим два свойства, содержащих информацию о цветах и шариках (листинг 26.13).

Листинг 26.13

```
// список цветов для шариков
private var colors: [UIColor]
// шарики
private var balls: [UIView] = []
```

Свойство `colors` — это массив значений типа `UIColor`. Сами шарики, как мы говорили выше, представляют собой экземпляры типа `UIView`, наложенные на отображение экземпляра типа Balls, который является наследником типа `UIView`. Мы не реализуем дополнительный тип данных для шариков, потому что функционала `UIView` вполне достаточно для решения поставленной задачи.

Для использования разрабатываемого класса нам потребуется реализовать инициализатор (листинг 26.14).

Листинг 26.14

```
public init(colors: [UIColor]){
    self.colors = colors
    super.init(frame: CGRect(x: 0, y: 0, width: 400, height: 400))
    backgroundColor = UIColor.grayColor()
}
```

Инициализатор `init(colors:)` в качестве входного параметра получает массив значений типа `UIColor`. В результате количество и порядок шариков будет зависеть именно от массива `colors`.

Класс `UIView` имеет встроенный инициализатор `init(frame:)`, который позволяет определить характеристики создаваемого графического элемента. При его вызове задаются значения для построения прямоугольной основы, на которую будут накладываться шарики. Для создания прямоугольников служит специальный класс `CGRect`, которому в качестве параметров передаются координаты левого верхнего угла и значения длин сторон.

Свойство `backgroundColor` определяет цвет создаваемого объекта, оно наследуется от класса `UIView`. Класс `UIColor` позволяет нам создать практически любой требуемый цвет. Для этого существует большое количество методов. В данном случае мы используем метод `grayColor()`, который определяет серый цвет. При необходимости создания произвольного цвета вы можете использовать соответствующие методы (в этом вам поможет окно автодополнения).

Обратите внимание, что Xcode отобразит ошибку, сообщающую об отсутствии инициализатора `init(coder:)` (рис. 26.12).

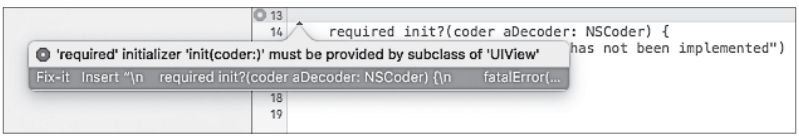


Рис. 26.12. Ошибка, сообщающая об отсутствии инициализатора init(coder:)

Вы можете написать код инициализатора самостоятельно (листинг 26.15) или позволить Xcode исправить ошибку.

Листинг 26.15

```
required public init?(coder aDecoder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
}
```

Теперь перейдем к файлу Balls в Project Navigator и создадим экземпляр класса `Balls` (листинг 26.16).

Листинг 26.16

```
let balls = Balls(colors: [UIColor.whiteColor()])
```

В качестве входного параметра инициализатора мы передаем массив объектов типа `UIColor`, содержащий всего один элемент, который определяет белый цвет. В результате в константу `balls` будет записан

экземпляр класса `Balls`, описывающий всю разрабатываемую систему целиком.

ПРИМЕЧАНИЕ Xcode позволяет преобразовать некоторые ресурсы из текстового вида в графический. Так, например, описание цвета может быть представлено в виде визуальной палитры цветов прямо в редакторе кода!

Для этого метод, описывающий белый цвет, необходимо заключить в символы «решетка» и «квадратные скобки».

[#UIColor.whiteColor()#]

В результате белый цвет из кода превратится в белый графический квадрат (рис. 26.13).

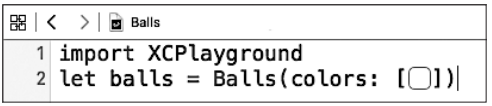


Рис. 26.13. Визуальное отображение ресурсов

Для выбора нового цвета вам необходимо щелкнуть по квадрату и выбрать подходящий из появившейся палитры. Это еще одна из поразительных возможностей среды разработки Xcode!

Обратите внимание, что такой подход возможен только в файлах, описывающих страницы playground. В Balls.swift вся информация отображается исключительно в текстовом виде.

Добавьте еще три произвольных цвета в массив `colors`, при этом не забудьте разделить их запятыми.

Теперь наш проект готов для предварительного отображения написанного кода. Для этого добавьте следующую строку в файл Balls (листинг 26.17).

Листинг 26.17

```
1  XCPlaygroundPage.currentPage.liveView = balls
```

В данном листинге используется функционал подключенной ранее библиотеки XCPlayground — класс `XCPlaygroundPage`, который позволяет вывести графические элементы на странице playground.

Теперь нажмите кнопку Assistant Editor и в правой части Xcode отобразится вывод экземпляра класса `Balls` (рис. 26.14).

В данный момент отображается только серый прямоугольник — он будет подложкой для отображения набора шариков.

Вернемся к файлу Balls.swift.

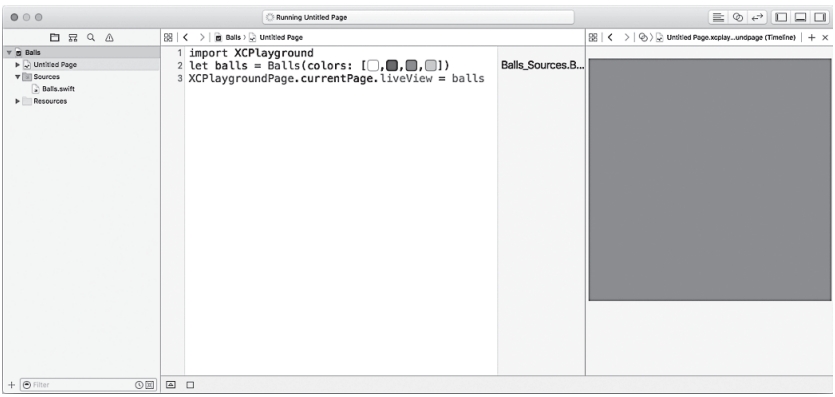


Рис. 26.14. Пример работы в режиме Assistant Editor

Xcode имеет довольно внушительное количество типов данных, как говорится, на все случаи жизни. Для указания размера шариков мы будем использовать тип данных `CGSize`. Создадим свойство класса `Balls`, описывающее высоту и ширину шарика (листинг 26.18).

Листинг 26.18

```
// размер шариков
private var ballSize: CGSize = CGSize(width: 40, height: 40)
```

В качестве аргументов инициализатор класса `CGSize` получает параметры `width` и `height`, определяющие ширину и высоту.

Теперь напишем метод, отвечающий за отображение шариков (листинг 26.19).

Листинг 26.19

```
func ballsView () {
    /* производим перебор переданных цветов
       именно они определяют количество шариков */
    for (index, color) in colors.enumerate() {
        /* шарик представляет из себя
           экземпляр класса UIView */
        let ball = UIView(frame: CGRect.zero)
        /* указываем цвет шарика
           он соответствует переданному цвету */
```

```
        ball.backgroundColor = color
        // накладываем отображение шарика на отображение подложки
        addSubview(ball)
        // добавляем экземпляр шарика в массив шариков
        balls.append(ball)
        /* вычисляем отступ шарика по осям X и Y, каждый
           последующий шарик должен быть правее и ниже предыдущего */
        let origin = 40*index + 100
        // отображение шарика в виде прямоугольника
        ball.frame = CGRect(x: origin, y: origin,
        width: Int(ballSize.width), height: Int(ballSize.height))
        // с закругленными углами
        ball.layer.cornerRadius = ball.bounds.width / 2.0
    }
}
```

Для обработки шариков мы используем свойство `colors`, которое хранит в себе массив переданных цветов. Метод `enumerate()` уже знаком нам — он позволяет перебрать все элементы массива, получая индекс и значение каждого элемента.

Как уже неоднократно говорилось, шарики, как и вся система целиком, это отображения. Но если подложка — это экземпляр потомка класса `UIView`, то каждый шарик — экземпляр самого `UIView`. Изначально в качестве отображения мы используем конструкцию `CGRect.zero`, которая соответствует `CGRect(x: 0, y: 0, width: 0, height:0)`, то есть прямоугольнику с нулевыми размерами.

Для того чтобы подложка и шарики выводились совместно, необходимо использовать функцию `addSubview(_:)`, которая накладывает одно отображение на другое.

Для того чтобы шарики обрисовывались так же, как и их подложка, необходимо добавить вызов метода `ballsVew()` в инициализатор `init(colors:)` (листинг 26.20).

Листинг 26.20

```
public init(colors: [UIColor]){
    self.colors = colors
    super.init(frame: CGRect(x: 0, y: 0, width: 400, height: 400))
    backgroundColor = UIColor.blueColor()
    // вызов функции отрисовки шариков
    ballsView()
}
```

Вернитесь к файлу Balls. В режиме Assistant Editor можно увидеть, что кроме серой подложки отрисовываются четыре разноцветных

шарика. В данный момент шарики статичны: вы не сможете с ними взаимодействовать.

Теперь займемся интерактивностью. Нам необходимо реализовать возможность перемещения шариков указателем мыши, их столкновения между собой и с бортами подложки.

Для анимации движения используется класс-аниматор `UIDynamicAnimator`. Он позволяет отображать обрабатываемые другими классами события, например перемещения.

Создадим новое свойство класса `Balls` (листинг 26.21).

Листинг 26.21

```
// аниматор графических объектов
private var animator: UIDynamicAnimator?
```

Обратите внимание, что аниматор — это опционал. Это связано с тем, что данное свойство не будет иметь какого-либо значения к моменту вызова родительского инициализатора `super.init(frame:)`.

Теперь необходимо подключить аниматор к отображению. Для этого добавим в инициализатор соответствующий код (листинг 26.22).

Листинг 26.22

```
public init(colors: [UIColor]){
    self.colors = colors
    super.init(frame: CGRect(x: 0, y: 0, width: 400, height: 400))
    backgroundColor = UIColor.blueColor()
    // подключаем аниматор с указанием на сам класс
    animator = UIDynamicAnimator(referenceView: self)
    ballsView()
}
```

Сам по себе аниматор не производит каких-либо действий. Для того чтобы он отображал некоторое изменение состояния объектов, необходимо использовать ряд дополнительных классов и связать каждый из этих классов с аниматором.

Для взаимодействия пользователя с графическими элементами UIKit предоставляет нам специальный класс `UISnapBehavior`.

ПРИМЕЧАНИЕ Каждый тип данных, в названии которого присутствует Behavior, предназначен для обработки некоторого поведения. Так, UISnapBehavior обрабатывает поведение при перемещении объектов от точки к точке.

Определим новое свойство типа `UISnapBehavior` (листинг 26.23).

Листинг 26.23

```
// обработчик перемещений объектов
private var snapBehavior: UISnapBehavior?
```

Класс `UISnapBehavior` позволяет обрабатывать касания экрана устройства (или щелчки мышки). Для этого в состав `UISnapBehavior` включены три метода:

1. `touchesBegan(_:withEvent:)`

   Метод вызывается в момент касания экрана.

2. `touchesMoved(_:withEvent:)`

   Метод срабатывает при каждом перемещении пальца, уже коснувшегося экрана.

3. `touchesEnded(_:withEvent:)`

   Метод вызывается по окончании взаимодействия с экраном (когда палец убран).

Все методы уже определены в `UISnapBehavior`, поэтому при создании собственной реализации этих методов необходимо их переопределять, то есть использовать ключевое слово `override`.

Реализуем метод `touchesBegan(_:withEvent:)` (листинг 26.24).

Листинг 26.24

```
override public func touchesBegan(touches: Set<UITouch>,
withEvent event: UIEvent?) {
    if let touch = touches.first {
        let touchLocation = touch.locationInView(self)
        for ball in balls {
            if (CGRectContainsPoint(ball.frame, touchLocation)) {
                snapBehavior = UISnapBehavior(item: ball,
                snapToPoint: touchLocation)
                snapBehavior?.damping = 0.5
                animator?.addBehavior(snapBehavior!)
            }
        }
    }
}
```

Коллекция `touches` содержит данные обо всех касаниях. Это связано с тем, что тач-панель современных устройств поддерживает мультитач, то есть одновременное касание несколькими пальцами. В начале метода извлекаются данные о первом элементе набора `touches` и помещаются в константу `touch`.

Константа `touchLocation` содержит координаты касания относительно всего отображения.

С помощью метода `CGRectContainsPoint` мы определяем, входят ли координаты касания в какой-либо из шариков. Если находится соответствие, то в свойство `snapBehavior` записываются данные об объекте, с которым происходит взаимодействие, и координатах, куда данный объект должен быть перемещен.

Свойство `damping` определяет плавность и затухание при движении шарика.

Далее, используя метод `addBehavior(_:)` аниматора, указываем, что обрабатываемое классом `UISnapBehavior` поведение объекта должно быть анимировано. Таким образом все изменения состояния объекта, производимые в свойстве `snapBehavior`, будут анимированы.

После обработки касания необходимо обработать перемещение пальца. Для этого реализуем метод `touchesMoved` (листинг 26.25).

Листинг 26.25

```
override public func touchesMoved(touches: Set<UITouch>,
withEvent event: UIEvent?) {
    if let touch = touches.first {
        let touchLocation = touch.locationInView(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(_:)` (листинг 26.26).

Листинг 26.26

```
public override func touchesEnded(touches: Set<UITouch>,
withEvent event: UIEvent?) {
    if let snapBehavior = snapBehavior {
        animator?.removeBehavior(snapBehavior)
    }
```

```
5       snapBehavior = nil
6   }
```

Данный метод служит одной очень важной задаче — очистке используемых ресурсов. После того как взаимодействие с шариком окончено, нет необходимости хранить информацию об обработчике поведения в `snapBehavior`.

ПРИМЕЧАНИЕ Возьмите за привычку удалять ресурсы, пользоваться которыми уже не будете. Это сэкономит вам изрядное количество памяти и уменьшит вероятность возникновения ошибок.

Перейдите в файл Balls и в окне Assistant Editor вы увидите изображение четырех шариков, но в отличие от предыдущего раза, сейчас вы можете указателем мыши перемещать любой из шариков (рис. 26.15)!

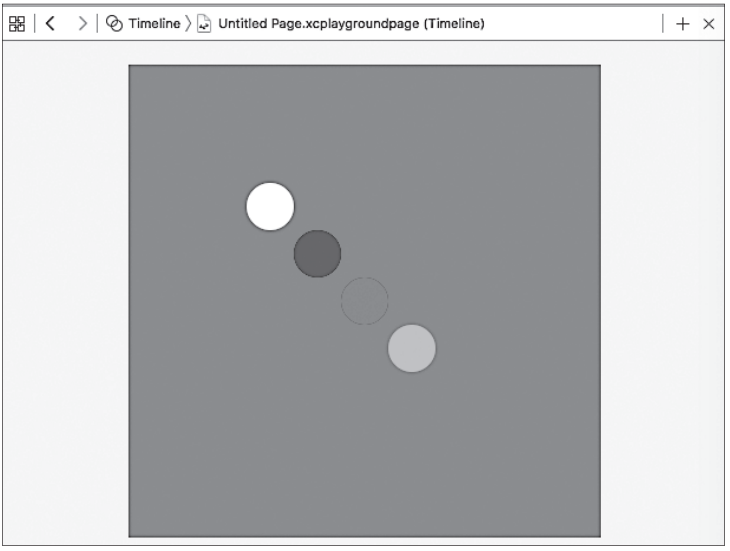


Рис. 26.15. Шарики, расположенные внутри подложки

Хотя шарики и могут перемещаться, они не взаимодействуют между собой и с границами подложки. Для обработки поведения при столкновениях служит класс `UICollisionBehavior`. Создадим новое свойство (листинг 26.27).

Листинг 26.27

```
// обработчик столкновений
private var collisionBehavior: UICollisionBehavior
```

Следующим шагом будет редактирование инициализатора (листинг 26.28).

Листинг 26.28

```
public init(colors: [UIColor]){
    self.colors = colors
    // создание значения свойства
    collisionBehavior = UICollisionBehavior(items: [])
    /* указание на то, что границы отображения
       также являются объектами взаимодействия */
       collisionBehavior.setTranslatesReferenceBoundsIntoBoundary
       WithInsets( UIEdgeInsets(top: 1, left: 1, bottom: 1,
       right: 1))
    super.init(frame: CGRect(x: 0, y: 0, width: 400, height: 400))
    backgroundColor = UIColor.grayColor()
    animator = UIDynamicAnimator(referenceView: self)
    /* добавляем обработчик поведения при столкновении
       к аниматору */
    animator?.addBehavior(collisionBehavior)
    ballsView()
}
```

Одним из свойств класса `UICollisionBehavior` является `items`. Оно указывает на набор объектов, которые могут взаимодействовать между собой. В момент работы инициализатора шарики еще не созданы, поэтому в качестве входного параметра при создании объекта типа `UICollisionBehavior` мы указываем пустой массив.

Для того чтобы обеспечить взаимодействие шариков не только друг с другом, но и с границами подложки, мы используем метод `setTranslatesReferenceBoundsIntoBoundaryWithInsets(_:)`. Он устанавливает границы коллизий с внутренним отступом в 1 пиксель с каждой стороны подложки.

В самом конце необходимо добавить обработчик поведения при коллизиях аниматору `animator`. Делается это с помощью уже известного нам метода `addBehavior(_:)`.

Теперь вам потребуется добавить каждый шарик в обработчик коллизий. Для этого существует метод `addItem(_:)`. Добавим соответствующую строку в метод `ballsView()` (листинг 26.29).

Листинг 26.29

```
1  func ballsView(){
2      for (index, color) in colors.enumerate(){
3          let ball = UIView(frame: CGRect.zero)
4          ball.backgroundColor = color
5          addSubview(ball)
6          balls.append(ball)
7          let origin = 40*index + 100
8          ball.frame = CGRect(x: origin, y: origin, width:
Int(ballSize.width), height: Int(ballSize.height))
9          ball.layer.cornerRadius = ball.bounds.width / 2.0
10         // добавим шарик в обработчик столкновений
11         collisionBehavior.addItem(ball)
12     }
13 }
```

Поздравляю вас! Это была последняя строчка кода, которую необходимо было написать для решения поставленной задачи. Сейчас вы можете перейти к файлу Balls и в Assistant Editor протестировать созданный прототип.

Как вы можете видеть, шарики невозможно переместить за границы, а при попытке соприкоснуть их они разлетаются.

UIKit и Foundation предоставляет еще огромное количество возможностей, помимо рассмотренных. Так, например, можно создать гравитационное, магнитное и любое другое взаимодействие элементов, смоделировать силу тяжести или турбулентности, установить параметры скорости и ускорения. Все это и многое-многое другое позволяет вам создавать поистине функциональные приложения, которые обязательно найдут отклик у пользователей.

27 Универсальные шаблоны

Универсальные шаблоны (generic) являются одним из мощнейших инструментов Swift. На их основе написано большинство библиотек. Даже если вы никогда специально не использовали универсальные шаблоны, на самом деле вы взаимодействовали с ними практически в каждой написанной программе.

Универсальные шаблоны позволяют создавать гибкие конструкции (функции, объектные типы) без привязки к конкретному типу данных. Вы лишь описываете требования и функциональные возможности, а Swift самостоятельно определяет, каким типам данных доступен разработанный функционал. Примером может служить тип данных `Array` (массив). Элементами массива могут выступать значения произвольных типов данных, и для этого разработчикам не требуется создавать отдельные типы массивов: `Array<Int>`, `Array<String>` и т. д. Для реализации коллекции использован универсальный шаблон, позволяющий при необходимости указать требования к типу данных. Так, например, в реализации типа `Dictonary` существует требование, чтобы тип данных ключа соответствовал протоколу `Hashable` (его предназначение мы обсуждали ранее).

27.1. Универсальные функции

Разработаем функцию, с помощью которой можно поменять значения двух целочисленных переменных (листинг 27.1).

Листинг 27.1

```
1  func swapTwoInts(inout a: Int, inout b: Int) {
2      let temporaryA = a
3      a = b
4      b = temporaryA
```

```
5  }
6  var firstInt = 4010
7  var secondInt = 13
8  swapTwoInts(&firstInt, &secondInt)
```

Функция `swapTwoInts(_:b:)` использует сквозные параметры, чтобы обеспечить доступ непосредственно к параметрам, передаваемым в функцию, а не к их копиям. В результате выполнения значения в переменных `firstInt` и `secondInt` меняются местами.

Данная функция является крайне полезной, но очень ограниченной в своих возможностях. Для того чтобы поменять значения двух переменных других типов, придется писать отдельную функцию: `swapTwoStrings()`, `swapTwoDoubles()` и т. д. Если обратить внимание на то, что тела всех функций должны быть практически одинаковыми, то мы просто-напросто займемся дублированием кода, хотя ранее в книге неоднократно рекомендовалось всеми способами этого избегать.

Для решения задачи было бы намного удобнее использовать универсальную функцию, позволяющую передать в качестве аргумента значения любого типа с одним лишь требованием: типы данных обоих аргументов должны быть одинаковыми.

Универсальные функции объявляются точно так же, как и стандартные, за одним исключением: после имени функции в угловых скобках указывается заполнитель имени типа, то есть литерал, который далее в функции будет указывать на тип данных переданного аргумента.

Преобразуем функцию `swapTwoInts(_:b:)` в универсальный вид (листинг 27.2).

Листинг 27.2

```
1  func swapTwoValues<T>(inout a: T, inout b: T) {
2      let temporaryA = a
3      a = b
4      b = temporaryA
5  }
6  var firstString = "one"
7  var secondString = "two"
8  swapTwoValues(&firstString, b: &secondString)
```

В универсальной функции заполнителем типа является литерал `T`, который и позволяет задать тип данных в списке входных аргументов вместо конкретного типа (`Int`, `String` и т. д.). При этом определяется, что `a` и `b` должны быть одного и того же типа данных.

Функция `swapTwoValues(_:b:)` может вызываться точно так же, как и определенная ранее функция `swapTwoInts(_:b:)`.

Используемый заполнитель называется параметром типа. Как только вы его определили, можете применять его для указания типа любого параметра или значения, включая возвращаемое функцией значение. При необходимости можно задать несколько параметров типа, вписав их в угловых скобках через запятую.

27.2. Универсальные типы

В дополнение к универсальным функциям универсальные шаблоны позволяют создать универсальные типы данных. К универсальным типам относятся, например, упомянутые ранее массивы и словари.

Создадим универсальную коллекцию `Stack` (стек). Разрабатываемый стек — это упорядоченная коллекция элементов, подобная массиву, но со строгим набором доступных операций:

- метод `push(_:)` служит для добавления элемента в конец коллекции;
- метод `pop()` служит для возвращения элемента из конца коллекции с удалением его оттуда.

Никаких иных доступных операций для взаимодействия со своими элементами стек не поддерживает.

В первую очередь создадим неуниверсальную версию типа (листинг 27.3).

Листинг 27.3

```
struct IntStack {
    var items = [Int]()
    mutating func push(item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

Данный тип обеспечивает работу исключительно со значениями типа `Int`. В качестве хранилища элементов используется массив `[Int]`. Сам тип для взаимодействия с элементами коллекции предоставляет нам оба описанных ранее метода.

Недостатком созданного типа является его ограниченность в отношении типа используемого значения. Реализуем универсальную версию типа, позволяющую работать с любыми однотипными элементами (листинг 27.4).

Листинг 27.4

```
struct Stack<T> {
    var items = [T]()
    mutating func push(item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
}
```

Универсальная версия отличается от неуниверсальной только тем, что вместо указания конкретного типа данных задается заполнитель имени типа.

Создавая новую коллекцию типа `Stack`, в угловых скобках необходимо указать тип данных, после чего можно использовать описанные методы для модификации хранилища (листинг 27.5).

Листинг 27.5

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
let fromTheTop = stackOfStrings.pop()
```

В коллекцию типа `Stack<String>` были добавлены два элемента и удален один.

Мы можем доработать описанный тип данных таким образом, чтобы при создании хранилища не было необходимости указывать тип элементов стека (листинг 27.6). Для реализации этой задачи опишем инициализатор, принимающий в качестве входного аргумента массив значений.

Листинг 27.6

```
struct Stack<T> {
    var items = [T]()
    init(){}
    init(_ elements: T...){
        self.items = elements
    }
```

```
7       mutating func push(item: T) {
8           items.append(item)
9       }
10      mutating func pop() -> T {
11          return items.removeLast()
12      }
13  }
14  var stackOfInt = Stack(1, 2, 4, 5)
15  var stackOfStrings = Stack<String>()
```

Так как мы объявили собственный инициализатор, принимающий входной параметр, для сохранения функциональности пришлось описать также пустой инициализатор.

Теперь мы можем не создавать стек без указания типа элементов, а просто передать значения в качестве входного аргумента в инициализатор типа.

27.3. Ограничения типа

Иногда бывает полезно указать определенные ограничения, накладываемые на типы данных универсального шаблона. В качестве примера мы уже рассматривали тип данных `Dictionary`, где для ключа существует требование: тип данных должен соответствовать протоколу `Hashable`.

Универсальные шаблоны позволяют накладывать определенные требования и ограничения на тип данных значения. Вы можете указать список типов, которым должен соответствовать тип значения. Если элементом этого списка является протокол (который также является типом данных), то проверяется соответствие типа значения данному протоколу; если типом является класс, структура или перечисления, то проверяется, соответствует ли тип значения данному типу.

Для определения ограничений необходимо передать перечень имен типов через двоеточие после заполнителя имени типа.

Реализуем функцию, производящую поиск элемента в массиве и возвращающую его индекс (листинг 27.7).

ПРИМЕЧАНИЕ Для обеспечения функционала сравнения двух значений в Swift существует специальный протокол Equatable. Он обязывает поддерживающий его тип данных реализовать функционал сравнения двух значений с помощью операторов равенства (==) и неравенства (!=). Другими словами, если тип данных поддерживает данный протокол, то его значения можно сравнивать между собой.

Листинг 27.7

```
func findIndex<T: Equatable>(array: [T], valueToFind: T) -> Int? {
    for (index, value) in array.enumerate() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
var myArray = [3.14159, 0.1, 0.25]
let firstIndex = findIndex(myArray, valueToFind: 0.1) // 1
let secondIndex = findIndex(myArray, valueToFind: 31) // nil
```

Параметр типа записывается как `<T: Equatable>`. Это означает «любой тип, поддерживающий протокол `Equatable`». В результате поиск в переданном массиве выполняется без ошибок, поскольку тип данных `Int` поддерживает протокол `Equatable`, а значит, значения данного типа могут быть приняты к обработке.

27.4. Расширения универсального типа

Swift позволяет расширять описанные универсальные типы. При этом имена заполнителей, использованные в описании типа, могут указываться и в расширении.

Расширим описанный ранее универсальный тип `Stack`, добавив в него вычисляемое свойство, возвращающее верхний элемент стека без его удаления (листинг 27.8).

Листинг 27.8

```
extension Stack {
    var topItem: T? {
        return items.isEmpty ? nil : items[items.count - 1]
    }
}
```

Свойство `topItem` задействует заполнитель имени типа `T` для указания типа свойства. Данное свойство является опционалом, так как значение в стеке может отсутствовать. В этом случае возвращается `nil`.

27.5. Связанные типы

При определении протокола бывает удобно использовать связанные типы, указывающие на некоторый, пока неизвестный тип данных.

Связанный тип позволяет задать заполнитель типа данных, который будет использоваться при заполнении протокола. Фактически тип данных не указывается до тех пор, пока протокол не будет принят каким-либо объектным типом.

Определим протокол `Container`, использующий связанный тип `ItemType` (листинг 27.9).

Листинг 27.9

```
protocol Container {
    typealias ItemType
    mutating func append(item: ItemType)
    var count: Int { get }
    subscript(i: Int) -> ItemType { get }
}
```

Протокол `Container` (контейнер) может быть задействован в различных коллекциях, например в описанном ранее типе коллекции `Stack`. В этом случае тип данных, используемый в свойствах и методах протокола, заранее неизвестен.

Для решения проблемы используется связанный тип `ItemType`, который определяется лишь при принятии протокола типом данных.

Пример принятия протокола к исполнению типом данных `Stack` представлен в листинге 27.10.

Листинг 27.10

```
struct Stack<T>: Container {
    associatedtype ItemType = T
    var items = [T]()
    var count: Int {
        return items.count
    }
    init(){}
    init(_ elements: T...){
        self.items = elements
    }
    subscript(i: Int) -> T {
        return items[i]
    }
    mutating func push(item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
```

```
20      mutating func append(item: T) {
21          items.append(item)
22      }
23  }
```

Так как тип `Stack` теперь поддерживает протокол `Container`, в нем появилось три новых элемента: свойство, метод и сабскрипт. Ключевое слово `associatedtype` указывает на то, какой тип данных является связанным в данном объектном типе.

ПРИМЕЧАНИЕ В более ранних версиях языка вместо ключевого слова associatedtype использовалось уже знакомое нам typealias. Так как этот подход в некоторой степени путал разработчиков, было принято решение изменить используемое ключевое слово.

Так как заполнитель имени использован в качестве типа аргумента `item` свойства `append` и возвращаемого значения сабскрипта, Swift может самостоятельно определить, что заполнитель `T` указывает на тип `ItemType`, соответствующий типу данных в протоколе `Container`. При этом указывать ключевое слово `associatedtype` не обязательно, если вы его удалите, то тип продолжит работать без ошибок.

28 Обработка ошибок

Обработка ошибок подразумевает реагирование на возникающие в процессе выполнения программы ошибки. Некоторые операции не могут гарантировать корректное выполнение вследствие возникающих обстоятельств. В этом случае очень важно определить причину возникновения ошибки и правильно обработать ее, чтобы не вызвать внезапного завершения всей программы.

В качестве примера можно привести запись информации в файл. При попытке доступа файл может не существовать или у пользователя могут отсутствовать права доступа для записи в него.

Отличительные особенности ситуаций могут помочь программе самостоятельно решать возникающие проблемы.

28.1. Выбрасывание ошибок

В Swift для создания перечня возможных ошибок служат перечисления, где каждый член перечисления соответствует отдельной ошибке. Само перечисление при этом должно поддерживать протокол `ErrorType`, который, хотя и является пустым, сообщает о том, что данный объектный тип содержит варианты ошибок.

Не стоит создавать одно перечисление на все случаи жизни. Группируйте возможные ошибки по их смыслу в различных перечислениях.

В следующем примере объявляется тип данных, который описывает ошибки в работе торгового автомата по продаже еды (листинг 28.1).

Листинг 28.1

```
enum VendingMachineError: ErrorType {
    case InvalidSelection
    case InsufficientFunds(coinsNeeded: Int)
    case OutOfStock
}
```

Каждый из членов перечисления указывает на отдельный тип ошибки:

- неправильный выбор;
- нехватка средств;
- отсутствие выбранного товара.

Ошибка позволяет показать, что произошла какая-то нестандартная ситуация и обычное выполнение программы не может продолжаться. Процесс появления ошибки называется *выбрасыванием ошибки*. Для того чтобы выбросить ошибку, необходимо воспользоваться оператором `throw`. Так, следующий код при попытке совершить покупку выбрасывает ошибку о недостатке пяти монет (листинг 28.2).

Листинг 28.2

```
1  throw VendingMachineError.InsufficientFunds(coinsNeeded: 5)
```

28.2. Обработка ошибок

Сам по себе выброс ошибки не приносит каких-либо результатов. Выброшенную ошибку необходимо перехватить и корректно обработать. В Swift существует четыре способа обработки ошибок:

- передача ошибки;
- обработка ошибки оператором `do-catch`;
- преобразование ошибки в опционал;
- запрет на выброс ошибки.

Если при вызове какой-либо функции или метода вы знаете, что он может выбросить ошибку, то необходимо перед вызовом указывать ключевое слово `try`.

Теперь разберем каждый из способов обработки ошибок.

Передача ошибки

При передаче ошибки блок кода (функция, метод или инициализатор), ставший источником ошибки, самостоятельно не обрабатывает ее, а передает выше в код, который вызвал данный блок кода.

Для того чтобы указать блоку кода, что он должен передавать возникающие в нем ошибки, в реализации данного блока после списка параметров указывается ключевое слово `throws`.

В листинге 28.3 приведен пример объявления двух функций, которые передают возникающие в них ошибки выше.

Листинг 28.3

```
func anotherFunc() throws {
    // тело функции
    var value = try someFunc()
    // ...
}
func someFunc() throws -> String{
    // тело функции
    try anotherFunc()
    // ...
}
try someFunc()
```

Функция `someFunc()` возвращает значение типа `String`, поэтому ключевое слово `throws` указывается перед типом возвращаемого значения.

Функция `anotherFunc()` в своем теле самостоятельно не выбрасывает ошибки, она может лишь перехватить ошибку, выброшенную функцией `anotherFunc()`. Для того чтобы перехватить ошибку, выброшенную внутри блока кода, необходимо осуществлять вызов с помощью упомянутого ранее оператора `try`. Благодаря ему функция `anotherFunc()` сможет отреагировать на возникшую ошибку так, будто она сама является ее источником. А так как эта функция также помечена ключевым словом `throws`, она просто передаст ошибку в вызвавший ее код.

Если функция не помечена ключевым словом `throw`, то все возникающие внутри нее ошибки она должна обрабатывать самостоятельно.

Рассмотрим пример из листинга 28.4.

Листинг 28.4

```
struct Item {
    var price: Int
    var count: Int
}
class VendingMachine {
    var inventory = [
        "Candy Bar": Item(price: 12, count: 7),
        "Chips": Item(price: 10, count: 4),
        "Pretzels": Item(price: 7, count: 11)
    ]
    var coinsDeposited = 0
    func dispenseSnack(snack: String) {
        print("Dispensing \(snack)")
    }
    func vend(itemNamed name: String) throws {
        guard var item = inventory[name] else {
            throw VendingMachineError.InvalidSelection
```

```
        }
        guard item.count > 0 else {
            throw VendingMachineError.OutOfStock
        }
        guard item.price <= coinsDeposited else {
            throw VendingMachineError.InsufficientFunds(coinsNeeded:
            item.price - coinsDeposited)
        }
        coinsDeposited -= item.price
        --item.count
        inventory[name] = item
        dispenseSnack(name)
    }
}
```

Структура `Item` описывает одно наименование продукта из аппарата 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(_:)` сообщает о том, что аппарат выдает некий товар. Метод `vend(itemNamed:)` непосредственно обслуживает покупку товара через аппарат.

При определенных условиях (запрошенный товар недоступен, его нет в наличии или количества внесенных монет недостаточно для покупки) метод `vend(itemNamed:)` может выбросить ошибку, соответствующую перечислению `VendingMachineError`. Сама реализация метода использует оператор `guard` для реализации преждевременного выхода с помощью оператора `throw`. Оператор `throw` мгновенно изменяет ход работы программы, в результе выбранный продукт может быть куплен только в том случае, если все условия покупки выполняются.

Так как метод `vend(itemNamed:)` передает все возникающие в нем ошибки вызывающему его коду, то необходимо выполнить дальнейшую обработку ошибок с помощью оператора `try` или `do-catch`.

Реализуем функцию, которая в автоматическом режиме пытается приобрести какой-либо товар (листинг 28.5). В данном примере словарь `favoriteSnacks` содержит указатель на любимое блюдо каждого из трех человек.

Листинг 28.5

```
let favoriteSnacks = [
    "Alice": "Chips",
    "Bob": "Licorice",
    "Eve": "Pretzels",
```

```
5  ]
6  func buyFavoriteSnack(person: String, vendingMachine:
   VendingMachine) throws {
7      let snackName = favoriteSnacks[person] ?? "Candy Bar"
8      try vendingMachine.vend(itemNamed: snackName)
9  }
```

Сама функция `buyFavoriteSnack(_:vendingMachine:)` не может выбросить ошибку, но так как она вызывает метод `vend(itemNamed:)`, для передачи ошибки выше необходимо использовать операторы `throw` и `try`.

Оператор do-catch

Выброс и передача ошибок вверх в конце концов должна вести к их обработке таким образом, чтобы это принесло определенную пользу пользователю и разработчику. Для этого вы можете задействовать оператор `do-catch`.

СИНТАКСИС

```
do {
    try имяВызываемогоБлока
} catch шаблон1 {
    // код...
} catch шаблон2 {
    // код...
}
```

Оператор содержит блок do и произвольное количество блоков catch. В блоке do должен содержаться вызов функции или метода, которые могут выбросить ошибку. Вызов осуществляется с помощью оператора try.

Если в результате вызова была выброшена ошибка, то данная ошибка сравнивается с шаблонами в блоках catch. Если в одном из них найдено совпадение, то выполняется код из данного блока.

Вы можете использовать ключевое слово where в шаблонах условий.

Блок catch можно задействовать без указания шаблона. В этом случае данный блок соответствует любой ошибке, а сама ошибка будет находиться в локальной переменной error.

Используем оператор `do-catch`, чтобы перехватить и обработать возможные ошибки (листинг 28.6).

Листинг 28.6

```
1  var vendingMachine = VendingMachine()
2  vendingMachine.coinsDeposited = 8
```

```
3  do {
4      try buyFavoriteSnack("Alice", vendingMachine: vendingMachine)
5  } catch VendingMachineError.InvalidSelection {
6      print("Invalid Selection.")
7  } catch VendingMachineError.OutOfStock {
8      print("Out of Stock.")
9  } catch VendingMachineError.InsufficientFunds(let coinsNeeded) {
10     print("Недостаточно средств. Пожалуйста, внесите еще \
       (coinsNeeded) монет(ы).")
11 }
12 // выводит "Недостаточно средств. Пожалуйста, внесите еще
   2 монет(ы)."
```

В приведенном примере функция `buyFavoriteSnack(_:vendingMachine:)` вызывается в блоке `do`. Поскольку внесенной суммы монет не хватает для покупки любимой сладости покупателя `Alice`, возвращается ошибка и выводится соответствующее этой ошибке сообщение.

Преобразование ошибки в опционал

Для преобразования выброшенной ошибки в опциональное значение используется оператор `try`, а точнее, его форма `try?`. Если в этом случае выбрасывается ошибка, то значение выражения вычисляется как `nil`.

Рассмотрим пример из листинга 28.7.

Листинг 28.7

```
1  func someThrowingFunction() throws -> Int {
2      // ...
3  }
4  let x = try? someThrowingFunction()
```

Если функция `someThrowingFunction()` выбросит ошибку, то в константе `x` окажется значение `nil`.

Запрет на передачу ошибки

В некоторых ситуациях можно быть уверенным, что блок кода во время исполнения не выбросит ошибку. В этом случае необходимо использовать оператор `try!`, который сообщает о том, что данный блок гарантированно не выбросит ошибку, — это запрещает передачу ошибки в целом.

Рассмотрим пример из листинга 28.8.

Листинг 28.8

```
let photo = try! loadImage("./Resources/John Appleseed.jpg")
```

Функция `loadImage(_:)` производит загрузку локального изображения, а в случае его отсутствия выбрасывает ошибку. Так как указанное в ней изображение является частью разрабатываемой вами программы и гарантированно находится по указанному адресу, с помощью оператора `try!` целесообразно отключить режим передачи ошибки.

Будьте внимательны: если при запрете передачи ошибки блок кода все же выбросит ее, то ваша программа экстренно завершится.

28.3. Отложенные действия по очистке

Swift позволяет определить блок кода, который будет выполнен лишь по завершении выполнения текущей части программы. Для этого служит оператор `defer`, который содержит набор отложенных выражений. С его помощью вы можете выполнить необходимую очистку независимо от того, как произойдет выход из данной части программы.

Отложенные действия выполняются в обратном порядке, то есть вначале выполняется блок последнего оператора `defer`, затем предпоследнего и т. д.

Рассмотрим пример использования блока отложенных действий (листинг 28.9).

Листинг 28.9

```
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }
        while let line = try file.readline() {
            // работа с файлом.
        }
    }
}
```

В данном примере оператор `defer` просто обеспечивает закрытие открытого ранее файла.

29 Нетривиальное использование операторов

Ранее вы уже познакомились с большим количеством операторов, которые предоставляет Swift. Однако возможна ситуация, в которой для ваших собственных объектных типов данных эти операторы окажутся бесполезными. В таком случае вам потребуется самостоятельно создать свои реализации стандартных операторов или полностью новые операторы.

29.1. 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 29.1).

Листинг 29.1

```
1  struct Vector2D {
2      var x = 0.0, y = 0.0
3  }
```

Свойства `x` и `y` показывают координаты конечной точки вектора. Начальная точка находится либо в точке с координатами (0,0), либо в конечной точке предыдущего вектора.

Если перед вами возникнет задача сложить два вектора, то проще всего воспользоваться операторной функцией и создать собственную реализацию оператора сложения (+), как показано в листинге 29.2.

Листинг 29.2

```
func + (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x + right.x, y: left.y + right.y)
}
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
```

Здесь операторная функция определена с именем, соответствующим оператору сложения. Так как оператор сложения является бинарным, он должен принимать два заданных значения и возвращать результат сложения.

Ситуация, когда несколько объектов имеют одно и то же имя, в Swift носит имя *перезагрузки*. С данным понятием мы уже неоднократно знакомились в ходе чтения книги.

Префиксные и постфиксные операторы

Оператор сложения является бинарным инфиксным, то есть он ставится между двумя операндами. Помимо инфиксных операторов в Swift существуют префиксные (предшествуют операнду) и постфиксные (следуют за операндом) операторы.

Для перезагрузки префиксного или постфиксного оператора перед объявлением операторной функции необходимо указать модификатор `prefix` или `postfix` соответственно.

Реализуем префиксный оператор унарного минуса для структуры `Vector2D` (листинг 29.3).

Листинг 29.3

```
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 перезагрузки 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 29.4 приведен пример реализации составного оператора присваивания-сложения для экземпляров типа `Vector2D`.

Листинг 29.4

```
func += (inout left: Vector2D, right: Vector2D) {
    left = left + right
}
var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd
// original теперь имеет значения (4.0, 6.0)
```

Так как оператор сложения был объявлен ранее, вам нет нужды реализовывать его в теле данной функции. Вы можете просто сложить два значения типа `Vector2D`.

Обратите внимание, что первый входной аргумент функции является сквозным.

Оператор эквивалентности

Пользовательские объектные типы не содержат встроенной реализации оператора эквивалентности, поэтому чтобы сравнить два экземпляра, необходимо перезагрузить данный оператор с помощью операторной функции.

В следующем примере приведена реализация оператора эквивалентности и оператора неэквивалентности (листинг 29.5).

Листинг 29.5

```
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.

29.2. 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 29.6).

Листинг 29.6

```
prefix operator +++ {}
prefix func +++ (inout vector: 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)
```

Заключение

Вы прошли полный курс подготовки к разработке приложений на Swift. Я уверен, что ваши знания стали намного глубже и вы прониклись этим необыкновенным духом разработки, который подарила нам Apple. Следующим шагом для вас должен стать поиск дополнительного материала, способствующего дальнейшему развитию навыков программирования и непосредственно обучению разработки в Xcode под iOS и OS X. Большую помощь в этом вам окажет интернет-портал http://swiftme.ru.

Вы сделали первый и самый важный шаг — дальше все будет еще интереснее.

Приложение. Изменения и нововведения Swift 2.2

Первое издание книги появилось в тот момент, когда правил бал Swift 2.1. С тех пор вышло уже несколько промежуточных версий, каждая из которых привнесла ряд изменений в процесс разработки. В данном приложении приведены подробные описания изменений для каждой из последующих, после Swift 2.1, версии языка.

Обратите внимание, что все механизмы, помеченные в Swift 2.2 как «устаревшие» (deprecated), будут удалены в Swift 3. Поэтому избавляйтесь от них как можно скорее, если планируете развивать ваши программы и в будущем.

Операторы инкремента и декремента

Операторы инкремента (`++`) и декремента (`--`) получили статус «устаревшие». В настоящей версии языка вы все еще можете использовать данные операторы, но в одной из будущих версий они будут удалены.

Вместо этих операторов следует использовать выражения `i += 1` и `i -= 1`.

Сравнение кортежей

В предыдущих версиях языка для сравнения кортежей требовалось самостоятельно проверять каждую соответствующую пару значений. Такой подход не удобен, потому что порождает большое количество лишнего кода. Swift 2.2 больше не требует сравнивать кортежи вручную — данный функционал встроен в ядро.

Листинг П.1

```
let singer = ("Alex", "Baskov")
let alien = ("Justin", "Bieber")   let alien = ("Justin", "Bieber")
if singer == alien {
    print("Matching tuples!")
} else {
    print("Non-matching tuples!")
}
```

Swift автоматически сравнивает каждую соответствующую пару значений и выдает результат в зависимости от результата операции.

ПРИМЕЧАНИЕ В настоящий момент Swift не позволяет сравнивать кортежи с количеством элементов больше 6. Запомните это, чтобы не совершить ошибок.

Кортежи в качестве группы аргументов

Если вы программировали на Swift ранее, то наверняка использовали возможность передачи нескольких аргументов в функцию в виде кортежа (листинг П.2).

Листинг П.2

```
func describePerson(name: String, age: Int) {
    print("\(name) is \(age) years old")  print("\(name) is \(age)
    years old")
}
let person = ("Taylor Swift", age: 26)
describePerson(person)
```

Подобный подход отныне является «устаревшим», поэтому требуется передавать значение для каждого аргумента отдельно (листинг П.3).

Листинг П.3

```
describePerson(person.0, age: person.1)
```

Циклы

Изменения коснулись в том числе и циклов.

Листинг П.4

```
for var i = 1; i <= 10; i += 1 {
    print("\(i) green bottles")
}
```

Уверен, что вы не единожды использовали данный синтаксис. Тем не менее в новой версии языка данный способ использования цикла `for` получил статус «устаревший» и будет полностью исключен из Swift 3.

В будущем вам потребуется использовать синтаксис `for-in`.

Листинг П.5

```
for i in 1...10{
    print("\(i) green bottles")
}
```

Запомните, что при необходимости прохода от большего числа к меньшему не нужно указывать диапазон 10...1. Для этого следует использовать метод `reverse()`.

Листинг П.6

```
for i in (1...10).reverse(){
    print("\(i) green bottles")
}
```

Ключевые слова в качестве имен аргументов

Рассмотрим следующий пример:

Листинг П.7

```
func printGreeting(name: String, repeat repeatCount: Int) {
    for _ in 0..<repeatCount {
        print(name)
    }
}
func printGreeting(name: String, repeat repeatCount: Int) {
```

Консоль:

```
Tiger
Tiger
Tiger
```

В функции `printGreeting()` в качестве внешнего имени аргумента `repeatCount` используется ключевое слово `repeat`. Несмотря на это функция будет выполнена корректно.

В Swift более ранних версий Xcode сообщал вам об ошибке и вынуждал вас заключать внешнее имя аргумента в обратные кавычки, то есть вам пришлось бы использовать конструкцию `` `repeat` ``, несмотря на то

что использование ключевого слова `repeat` в данном месте в принципе невозможно.

В Swift 2.2 мы можем использовать любые ключевые слова без обратных кавычек, за исключением `inout` и `let`, то есть ключевых слов, влияющих на входные аргументы.

Убрано ключевое слово var в списке аргументов

Любой аргумент функции — константа. При необходимости модификации значения аргумента в теле функции ранее требовалось указывать ключевые слова `var` или `inout`. Такой подход привносил некоторую путаницу, так как оба ключевых слова выполняли одну и ту же задачу, а различались лишь областью действия измененного параметра.

Для внесения большей ясности в процесс разработки ключевое слово `var` получило статус «устаревшее» и будет полностью удалено в Swift 3. Для модификации значения аргументов внутри тела функции без необходимости передачи сохранения значения теперь следует создавать новый параметр и производить работу с ним.

Листинг П.8

```
1  func printGreeting(name: String, repeat repeatCount: Int) {
2      let upperName = name.uppercaseString
3      for _ in 0..<repeatCount {
4          print(upperName)
5      }
6  }
7  printGreeting("Tiger", repeat: 2)
```

Консоль:

```
TIGER
TIGER
```

Переименованы отладочные идентификаторы

В Swift 2.1 и более ранних версиях для отладки кода использовались идентификаторы `__FILE__`, `__LINE__`, `__COLUMN__`, `__FUNCTION__`, которые при необходимости сообщают имя файла, номер строки, номер колонки и имя функции, в которых были вызваны.

В Swift 2.2 подобный синтаксис помечен как «устаревший» и заменен на `#file`, `#line`, `#column` и `#function`. Пример использования приведен ниже.

Листинг П.9

```
1  func printGreeting(name: String, repeat repeatCount: Int) {
2      print("This is on line \(__LINE__) of \(__FUNCTION__)")
3      print("This is on line \(#line) of \(#function)")
4      let upperName = name.uppercaseString
5      for _ in 0..<repeatCount {
6          print(upperName)
7      }
8  }
9  printGreeting("Tiger", repeat: 2)
10 func printGreeting(name: String, repeat repeatCount: Int) {
```

Консоль:

```
This is one line 2 of printGreeting
This is one line 3 of printGreeting
TIGER
TIGER
```

Как вы можете видеть, в настоящей версии корректно работают обе формы идентификаторов, но в Swift 3 устаревший вариант будет исключен из языка.

Проверка версии Swift

Swift 2.2 привнес новую опцию, которая позволит комбинировать код, написанный для различных версий языка, в одном файле. Рассмотрим приведенный ниже пример.

Листинг П.10

```
1  #if swift(>=2.2)
2      print("Running Swift 2.2 ot later")
3  #else
4      print("Running Swift 2.1 or earlier")
5  #endif
```

Подобный код обрабатывается компилятором, поэтому не вызывает ошибок при работе в любой версии Xcode.

Новые ключевые слова для документирования

Swift поддерживает механизм markdown-комментирования и добавления метаданных вашего кода. Вы можете написать код, подобный следующему.

Листинг П.11

```
/**
 Say hello to specific person
 - parameters:
 - name: the name of the person to greet
 - returns: Absolutely nothing
 - authors: Bilbo Baggins
 - bug: This is a deeply dull function
 */
func sayHello(name: String) {
    print("hello, \(name)!")
}
sayHello("Frodo")
```

Приведенные метаданные будут отображаться в панели помощи (при нажатии на Alt).

В Swift 2.2 были добавленные новые ключевые слова:

`keyword`
: Определяет ключевые слова для описываемого функционального элемента.

`recomended`
: Указывает на имя элемента, использование которого рекомендовано взамен описываемого.

`recomendedover`
: Указывает имя элемента, взамен которого рекомендуется использовать описываемый метод.

ПРИМЕЧАНИЕ Очень странно, что данные ключевые слова пока еще не поддерживаются в актуальной версии Xcode 7.3. Когда они станут доступны? — спросите вы. Надеюсь, что скоро! — отвечу я.

Замена ключевого слова typealias в описании протокола

В ранних версиях Swift при описании протоколов для указания на связанные типы использовалось ключевое слово `typealias`. Данный

подход мог привести к некоторой путанице, так как данное ключевое слово также используется при определении псевдонима для произвольного типа данных. Получается, что на одну и ту же команду назначены две совершенно разные функции.

В связи с этим было предложено использовать ключевое слово `associatedtype`, которое само по себе говорит о своем предназначении. Пример приведен в листинге П.12.

Листинг П.12

```
protocol Prot {
    associatedtype TypeItem
}
```

Ссылки на функции с именами аргументов

Как вам известно, любой объект может иметь набор методов с одним и тем же названием, но разным набором аргументов. Данный подход очень хорош, но в некоторых ситуациях мог бы привести к неразрешимым ситуациям.

Предположим, что у нас есть расширение UIView, имеющее набор одноименных методов (листинг П.13).

Листинг П.13

```
extension UIView {
    func insertSubview(view: UIView, at index: Int)
    func insertSubview(view: UIView, aboveSubview
    siblingSubview: UIView)
    func insertSubview(view: UIView, belowSubview
    siblingSubview: UIView)
}
```

Если необходимо вызвать некоторые из этих методов, нужно лишь передать требуемый набор аргументов (листинг П.14).

Листинг П.14

```
someView.insertSubview(view, at: 3)
someView.insertSubview(view, aboveSubview: otherView)
someView.insertSubview(view, belowSubview: otherView)
```

Но в ситуации, когда необходимо создать ссылку на один из методов, возникнет неразрешимая ситуация (листинг П.15).

Листинг П.15

```
1  let fn1 = someView.insertSubview //неразрешимая ситуация
```

В данном случае метод `insertSubview()` может быть одним из трех, и Swift не сможет определить это самостоятельно. Один из способов, которым можно однозначно определить метод, — указать тип данных параметра (листинг П.16).

Листинг П.16

```
1  let fn2: (UIView, Int) = someView.insertSubview //выбор однозначен
2  let fn3: (UIView, UIView) = someView.insertSubview
   //неразрешимая ситуация
```

Константа `fn2` будет содержать ссылку на четко определенный метод, но при этом под тип данных константы `fn3` подпадают два метода, что снова приведет к неразрешимой ситуации. Конечно, вы можете реализовать указатель в виде замыкания (листинг П.17).

Листинг П.17

```
1  let fn4: (UIView, UIView) = { view, otherView in
2      button.insertSubview(view, aboveSubview: otherView)
3  }
```

Данный подход хотя и решает поставленную перед нами задачу, но довольно труден для реализации.

Swift 2.2 предлагает нам указывать внешние имена аргументов при создании ссылки на метод (листинг П.18).

Листинг П.18

```
1  let fn5: (UIView, UIView) = someView.insertSubview(_:at:)
2  let fn6: (UIView, UIView) = someView.insertSubview(_:aboveSubview:)
```

Имена разделяются двоеточиями, а в случае, если они отсутствуют, ставится подчеркивание. Таким образом мы можем однозначно определить указатель.