



БИБЛИОТЕКА ПРОГРАММИСТА



Василий Усов

Swift

Основы разработки приложений под iOS и macOS

3-е издание
дополненное
и переработанное

 ПИТЕР®

Василий Усов

Swift. Основы разработки приложений под iOS и macOS

3-е издание, дополненное и переработанное

Серия «Библиотека программиста»

Заведующая редакцией

Ю. Сергиенко

Ведущий редактор

Н. Римицан

Литературный редактор

Н. Рощина

Художественный редактор

С. Маликова

Корректор

Н. Викторова

Верстка

Л. Егорова

ББК 32.973.2-018.1

УДК 004.438

Усов В.

У76 Swift. Основы разработки приложений под iOS и macOS. 3-е изд., доп. и перераб. — СПб.: Питер, 2017. — 368 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-496-03036-6

Swift — быстрый, современный, безопасный и удобный язык программирования. Если вы когда-либо писали приложения на Objective-C, то после изучения Swift с его многообразием возможностей вы, вероятно, захотите переписать приложения на новом языке программирования.

Мир Swift очень и очень молод, людей со значительным багажом знаний и опыта за плечами еще просто не существует в силу возраста языка, поэтому вы можете стать одним из них.

Третье издание книги было значительно обновлено и доработано. Примеры кода соответствуют Swift версии не ниже 3.0, iOS версии не ниже 10 и Xcode версии не ниже 8. Переработана глава, посвященная коллекциям, обновлены главы, посвященные функциям и замыканиям, добавлена информация о ленивых вычислениях и нетривиальных возможностях Swift.

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

ISBN 978-5-496-03036-6

© ООО Издательство «Питер», 2017

© Серия «Библиотека программиста», 2017

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Изготовлено в России. Изготовитель: ООО «Питер Пресс».

Место нахождения и фактический адрес: 192102, Россия, город Санкт-Петербург, улица Андреевская, дом 3, литер А, помещение 7Н. Тел.: +78127037373.

Дата изготовления: 03.2017. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —

Книги печатные профессиональные, технические и научные.

Подписано в печать 03.03.17. Формат 60×90/16. Бумага офсетная. Усл. п. л. 23,000.

Тираж 1000. Заказ 0000.

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.ru Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

Оглавление

Введение	9
О Swift	10
О книге	12
Исправления в третьем издании	13
Для кого написана книга	14
Что нужно знать, прежде чем начать читать	15
Структура книги	15
Условные обозначения	16
 Часть I. Подготовка к разработке Swift-приложений.	 17
Глава 1. Подготовка к разработке в macOS	18
1.1. Вам необходим компьютер Mac	18
1.2. Зарегистрируйтесь как Apple-разработчик	18
1.3. Установите Xcode	20
1.4. Введение в Xcode	21
1.5. Интерфейс playground-проекта	25
1.6. Возможности playground-проекта	26
Глава 2. Подготовка к разработке в Linux	30
 Часть II. Базовые возможности Swift	 34
Глава 3. Отправная точка	35
3.1. Инициализация и изменение значения	36
3.2. Переменные и константы	39
3.3. Правила объявления переменных и констант	42
3.4. Глобальные и локальные объекты	43
3.5. Вывод текстовой информации	44
3.6. Комментарии	47
3.7. Точка с запятой	51

Глава 4. Типы данных и операции с ними	52
4.1. Виды определения типа данных	52
4.2. Числовые типы данных	55
4.3. Текстовые типы данных	69
4.4. Логические значения	75
4.5. Псевдонимы типов	78
4.6. Операторы сравнения	80
4.7. Операторы диапазона	81
 Часть III. Основные средства Swift	 82
Глава 5. Кортежи	83
5.1. Основные сведения о кортежах	83
5.2. Взаимодействие с элементами кортежа	85
Глава 6. Опциональные типы данных	91
6.1. Опционалы	91
6.2. Извлечение опционального значения	93
Глава 7. Операторы условий	96
7.1. Утверждения	96
7.2. Оператор условия if	98
7.3. Оператор раннего выхода guard	110
7.4. Оператор ветвления switch	110
Глава 8. Коллекции	121
8.1. Массивы	121
8.2. Наборы	134
8.3. Словари	143
Глава 9. Циклы	149
9.1. Оператор повторения for	149
9.2. Операторы повторения while и repeat while	155
9.3. Управление циклами	156
Глава 10. Функции	160
10.1. Объявление функций	160
10.2. Аргументы функции и возвращаемое значение	163
10.3. Возможности функций	171
10.4. Безымянные функции	178

Глава 11. Замыкания	180
11.1. Функции как замыкания	180
11.2. Замыкающие выражения	182
11.3. Неявное возвращение значения	184
11.4. Сокращенные имена параметров	185
11.5. Переменные-замыкания	186
11.6. Метод сортировки массивов	187
11.7. Каррирование функций	189
11.8. Захват переменных	190
11.9. Замыкания — это тип-ссылка	193
11.10. Автозамыкания	194
11.11. Выходящие замыкания	196
Глава 12. Вспомогательные функциональные элементы	199
12.1 Метод <code>map(_)</code>	199
12.2 Метод <code>filter(_)</code>	202
12.3 Метод <code>reduce(_:_)</code>	203
12.4 Метод <code>flatMap(_)</code>	204
Глава 13. Ленивые вычисления	205
13.1. Понятие ленивых вычислений	205
13.2. Замыкания в ленивых вычислениях	206
13.3. Свойство <code>lazy</code>	207
Часть IV. Нетривиальные возможности Swift	208
Глава 14. ООП как фундамент	210
14.1. Экземпляры	210
14.2. Пространства имен	212
14.3. API Design Guidelines	213
Глава 15. Перечисления	215
15.1. Синтаксис перечислений	215
15.2. Ассоциированные параметры	218
15.3. Вложенные перечисления	220
15.4. Оператор <code>switch</code> для перечислений	221
15.5. Связанные значения членов перечисления	222
15.6. Свойства в перечислениях	225
15.7. Методы в перечислениях	226

15.8. Оператор self	227
15.9. Рекурсивные перечисления	228
Глава 16. Структуры	232
16.1. Синтаксис объявления структур	232
16.2. Свойства в структурах	233
16.3. Структура как пространство имен	235
16.4. Собственные инициализаторы	236
16.5. Методы в структурах	238
Глава 17. Классы	240
17.1. Синтаксис классов	241
17.2. Свойства классов	241
17.3. Методы классов	244
17.4. Инициализаторы классов	245
17.5. Вложенные типы	246
Глава 18. Свойства	248
18.1. Типы свойств	248
18.2. Контроль получения и установки значений	251
18.3. Свойства типа	254
Глава 19. Сабскрипты	257
19.1. Назначение сабскриптов	257
19.2. Синтаксис сабскриптов	258
Глава 20. Наследование	263
20.1. Синтаксис наследования	263
20.2. Переопределение наследуемых элементов	265
20.3. Превентивный модификатор final	268
20.4. Подмена экземпляров классов	269
20.5. Приведение типов	269
Глава 21. Псевдонимы Any и AnyObject	272
21.1. Псевдоним Any	272
21.2. Псевдоним AnyObject	274
Глава 22. Инициализаторы и деинициализаторы	275
22.1. Инициализаторы	275
22.2. Деинициализаторы	281

Глава 23. Удаление экземпляров и ARC	283
23.1. Уничтожение экземпляров	283
23.2. Утечки памяти	285
23.3. Автоматический подсчет ссылок	287
Глава 24. Опциональные цепочки	290
24.1. Доступ к свойствам через опциональные цепочки	290
24.2. Установка значений через опциональные цепочки	292
24.3. Доступ к методам через опциональные цепочки	293
Глава 25. Расширения	294
25.1. Вычисляемые свойства в расширениях	295
25.2. Инициализаторы в расширениях	296
25.3. Методы в расширениях	297
25.4. Сабскрипты в расширениях	298
Глава 26. Протоколы	299
26.1. Требуемые свойства	300
26.2. Требуемые методы	301
26.3. Требуемые инициализаторы	302
26.4. Протокол в качестве типа данных	303
26.5. Расширение и протоколы	303
26.6. Наследование протоколов	304
26.7. Классовые протоколы	305
26.8. Композиция протоколов	306
Глава 27. Разработка первого приложения	307
27.1. Важность работы с документацией	307
27.2. Модули	314
27.3. Разграничение доступа	318
27.4. Разработка интерактивного приложения	321
Глава 28. Универсальные шаблоны	335
28.1. Универсальные функции	335
28.2. Универсальные типы	337
28.3. Ограничения типа	339
28.4. Расширения универсального типа	340
28.5. Связанные типы	340

Глава 29. Обработка ошибок	343
29.1. Выбрасывание ошибок	343
29.2. Обработка ошибок	344
29.3. Отложенные действия по очистке	349
Глава 30. Нетривиальное использование операторов	350
30.1. Операторные функции	350
30.2. Пользовательские операторы	353
Заключение	354
Приложение. Изменения и нововведения Swift 3.0	355
Ассистент миграции	355
API Design Guidelines	355
Изменения в функциях	359
Удалены операторы инкремента и декремента	363
Удалена классическая форма цикла for	363
Доступ к типу	363
Добавлена ленивая реализация flatMap(_) для последовательности из опционалов	364
Тип Unmanaged использует UnsafePointer	365
Тестирование	365
Изменены модификаторы области видимости	365
Добавлен метод first(where:) для типа Sequence	366
Импорт констант Objective-C в виде типов данных	366
Точка перед членом перечисления	366
Уточнено взаимодействие комментариев и операторов	367
Параметры атрибутов через точку	367
Объявление параметров в конструкции switch-case	367
Уведомления о неиспользовании результатов работы функции	368
Универсальные псевдонимы типов	368
Изменения в композиции протоколов	368

*Посвящается девушке,
которая стала моей судьбой,
которая подарила мне смысл жизни*

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 вместе с iOS 10 выходит версия Swift 3.0, приносящая в этот замечательный язык программирования много нового.

ПРИМЕЧАНИЕ Совместно со Swift 3.0 была выпущена версия 2.3. Это сделано для того, чтобы вы не переживали, если не успели портировать свои приложения на Swift 3. Версия 2.3 — это все та же 2.2, за одним исключением — в ней поддерживаются новые возможности Xcode 8 и новых SDK, предназначенных для разработки под macOS 10.12, iOS 10, watchOS 3 и tvOS 10. 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 поддерживается при разработке программ для всей продукции фирмы Apple (операционные системы iOS, macOS, tvOS и watchOS), а также в 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 программирования¹. После выхода Swift многие разработчики пошли именно по этому пути, понимая, что в будущем наибольшее внимание Apple будет уделять развитию нового языка. Более того, Swift стал первой разработкой Apple с открытым исходным кодом, что говорит о скором внедрении его поддержки и другими операционными системами (ждем поддержки Windows)².

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 превосходит он другие языки программирования, на которых вы писали программы ранее. Его простота, лаконичность и невероятные возможности просто поразительны!

Язык Swift создан полностью с нуля и обладает рядом особенностей.

Современность

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

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.

Объектоориентированность

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

Читабельность, экономичность и лаконичность кода

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

Безопасность

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

Производительность

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

Актуальность

Swift — современный язык программирования и поэтому должен поддерживать свою репутацию на протяжении всего жизненного цикла. Это стало возможным благодаря активно растущему сообще-

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

ству swift-разработчиков, штаб-квартирой которых стал портал swift.org, на котором размещены все необходимые данные о том, как стать членом этой быстро растущей семьи. Для русскоязычного сообщества уже созданы несколько профильных сайтов, одним из которых является swiftme.ru.

Эти особенности делают Swift по-настоящему удивительным языком программирования. А сейчас для вас самое время погрузиться в мир Swift: он еще очень и очень молод, людей со значительным багажом знаний и опыта за плечами 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 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 (в том числе и для macOS, tvOS и watchOS) или программ для операционной системы Linux. В ходе чтения книги вы встретите не только теоретические сведения, но и большое количество практических примеров и заданий, выполняя которые вы углубите свои знания изучаемого материала. Несмотря на то что вам предстоит пройти большой путь, это будет полезный и очень важный опыт. Хотя книга предназначена в первую очередь непосредственно для изучения языка Swift, вы получите некоторые начальные сведения о принципах разработки полноценных приложений. Можете быть уверены, эта информация будет очень полезной. Я считаю, что книга 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. Несмотря на то что в книге приводится информация о разработке и под операционную систему Linux, основной упор делается именно на разработку на платформе от Apple. Примеры кода в данной

книге соответствуют Swift версии не ниже 3.0, iOS версии не ниже 10 и Xcode версии не ниже 8. 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 вновь 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. Так же на сайте вы сможете найти инструкцию для доступа к сообществу Swiftme в мессенджере Telegram, а это круглосуточный доступ к знаниям и общение с десятками, а может уже и сотнями начинающих программистов на Swift со всей России и ближнего зарубежья.

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

Исправления в третьем издании

Вы держите в руках третье издание книги «Swift. Основы разработки под iOS, macOS». Каждый уважающий себя писатель хочет создать действительно ценный продукт, и игнорировать мысли и предложения читателей было бы большой глупостью и проявлением эгоизма. В ходе долгого и плодотворного общения со многими из вас была выработана масса идей, благодаря которым новое издание стало по-настоящему полезным. В сравнении с предыдущим изданием данная книга содержит следующие изменения и дополнения:

- ❑ Весь материал актуализирован в соответствии со Swift версии 3.0 и Xcode 8.
- ❑ Добавлено большое количество нового учебного материала, в частности, связанного с обучением основам программирования.
- ❑ Переработана глава, посвященная коллекциям. Теперь ее изучение станет для вас такой же приятной задачей, как и изучение остального материала.
- ❑ Значительно доработаны главы, посвященные функциям и замыканиям. В них вы найдете много нового и интересного, в том числе связанного с изменениями, пришедшими вместе со Swift 3.0.

- ❑ Добавлены вспомогательные главы, позволяющие вам еще глубже погрузиться в изучаемый материал. В частности, теперь не будет проблем с изучением ленивых вычислений.
- ❑ Учтены пожелания и замечания пользователей по оформлению и содержанию.
- ❑ Исправлены найденные опечатки.
- ❑ В ходе написания книги я пытался выделить материал, предназначенный для начинающих программистов, в отдельные блоки, чтобы позволить читателям с опытом разработки на других языках не отвлекаться на ненужный им материал.

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), macOS, watchOS или tvOS?
- ❑ Вы предпочитаете обучение в практической форме скучным и монотонным теоретическим лекциям?

Тогда эта книга для вас.

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

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

Очень важно, чтобы вы не давали 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 помогут 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. Желаю увлекательного путешествия!

Что нужно знать, прежде чем начать читать

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

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

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 части вы будете использовать с завидной регулярностью при создании собственных приложений в будущем. Также отличительной чертой данной части является большая практическая работа по созданию первого интерактивного приложения.
- ❑ **Приложение.** Изменения и нововведения Swift 3.0. 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.

НАЧИНАЮЩИМ Если вы начинающий программист или не имеете опыта программирования вовсе, то справочная информация, приведенная в этих блоках, предназначена специально для вас. Она поможет вам глубже понять основы программирования и принципы разработки на языке Swift.

Задание

Задания для выполнения вы найдете в отдельных блоках.

Часть I

Подготовка к разработке Swift-приложений

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

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

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

1

Подготовка к разработке в macOS

1.1. Вам необходим компьютер Mac

Прежде чем приступить к разработке программ на языке Swift в macOS, вам потребуется несколько вещей. Для начала понадобится компьютер iMac, MacBook, Mac mini или Mac Pro с установленной операционной системой Mavericks (OS X 10.9), Yosemite (OS X 10.10), El Capitan (OS X 10.11) или Sierra (macOS 10.12). Это первое и базовое требование связано с тем, что среда разработки приложений Xcode создана компанией Apple исключительно с ориентацией на собственную платформу.

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

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

1.2. Зарегистрируйтесь как Apple-разработчик

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

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.

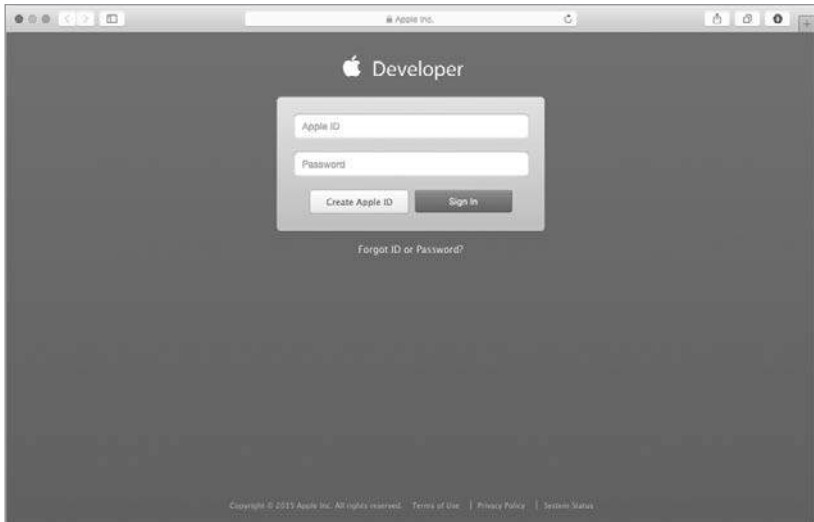


Рис. 1.1. Страница входа в Центр разработчиков

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 **Developer**, расположенной в верхней части страницы, после чего откроется страница, содержащая ссылки на основные ресурсы для 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. Таким образом, каждый может начать разрабатывать приложения, не заплатив за это ни копейки (если не учитывать стоимость компьютера). Тем не менее за 99 долларов в год вы можете участвовать в платной программе iOS-разработчиков (iOS Developer Program), которую вам предлагает Apple. Это не обязательно, но если вы хотите распространять свои приложения, то есть использовать мобильный магазин приложений App Store, то участие в программе становится обязательным.



Рис. 1.2. Перечень ресурсов, доступных в Центре разработчиков

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

1.3. Установите Xcode

Теперь все, что вам необходимо, — скачать Xcode. Для этого перейдите в раздел Xcode, после чего перед вами откроется страница, содержащая доступные для скачивания версии данной среды разработки (рис. 1.3).

ПРИМЕЧАНИЕ Как вы можете видеть, Apple предоставляет 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 **Download** произойдет автоматический переход в магазин приложений Mac App Store на страницу Xcode. Здесь вы можете увидеть краткое изложение всех основных возможностей среды разработки, обзор последнего обновления и отзывы пользователей. Для скачивания Xcode просто щелкните на кнопке **Загрузить** и при необходимости введите данные своей учетной записи Apple ID.

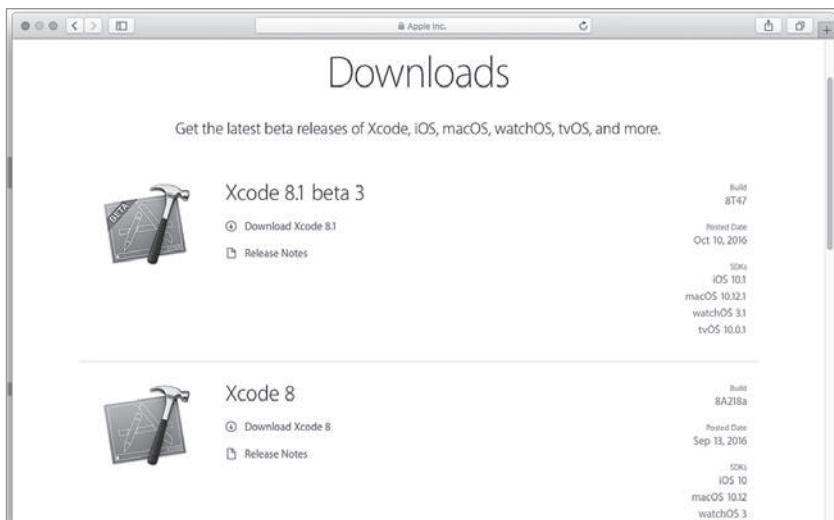


Рис. 1.3. Страница Xcode в Центре разработчиков

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.

НАЧИНАЮЩИМ 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 macOS. Xcode — это наиболее важный инструмент, который использует разработчик. Среда Xcode удивительна! Она предоставляет широкие

возможности, и изучать их следует постепенно, исходя из поставленных и возникающих задач. Внешний вид рабочей среды приведен на рис. 1.4.

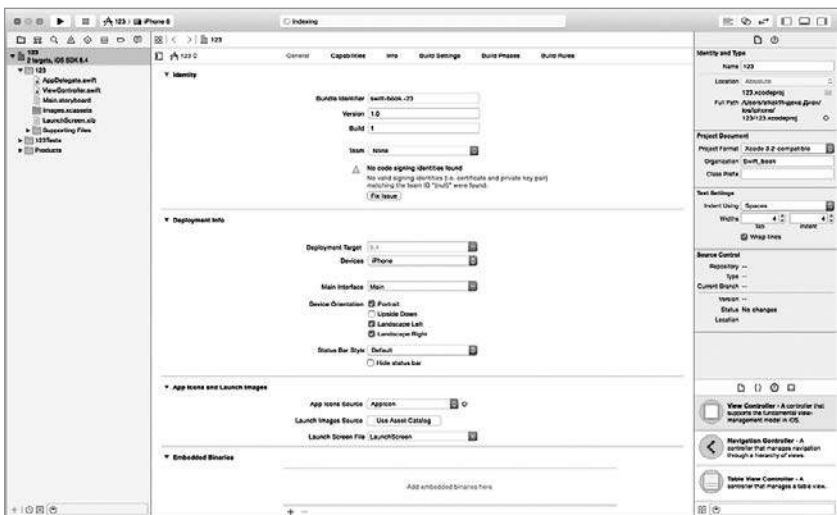


Рис. 1.4. Интерфейс Xcode

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

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

Я надеюсь, что вы уже имеете на своем компьютере последнюю версию Xcode, а значит, мы можем перейти к первому знакомству с этой замечательной средой. Для начала необходимо запустить Xcode. При первом запуске, возможно, вам придется установить некоторые допол-

нительные пакеты (все пройдет в автоматическом режиме при щелчке на кнопке **install** в появившемся окне).

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

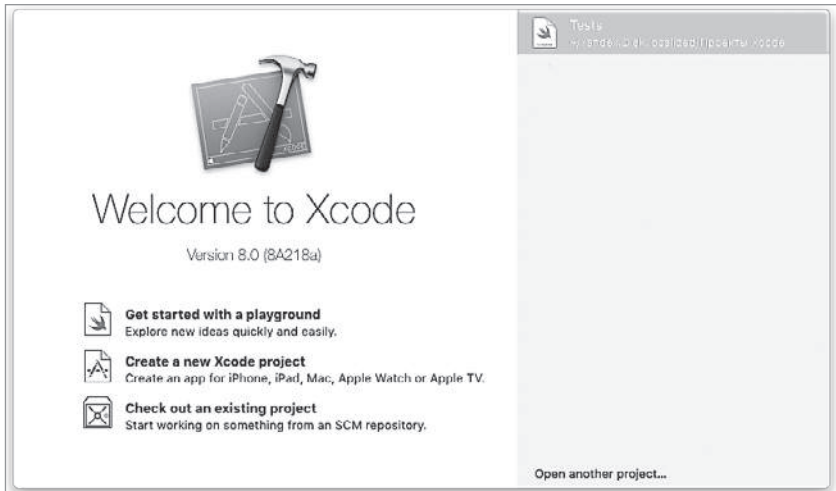


Рис. 1.5. Стартовое окно Xcode

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

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

ПРИМЕЧАНИЕ Идея SCM (software configuration management) базируется на удаленном хранилище, содержащем 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. Сейчас вам не стоит погружаться в этот вопрос, так как он совершенно не повлияет на качество изучения приведенного в книге материала.

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 (предыдущая версия среды разработки) 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.

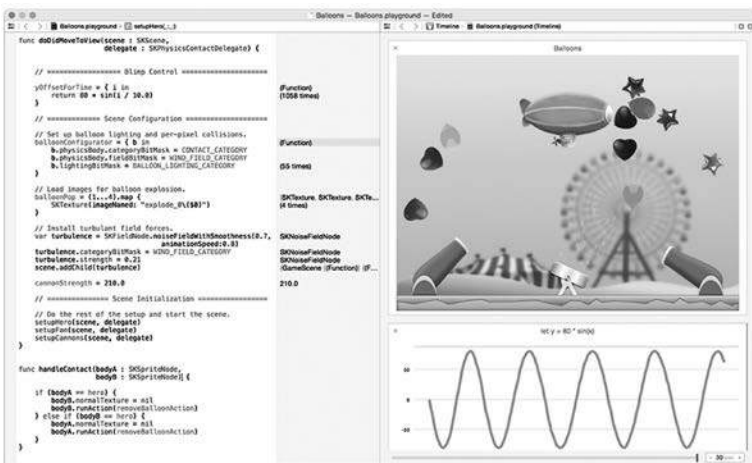


Рис. 1.6. Пример playground-проекта

Как вы можете видеть, внешний вид интерфейса playground-проекта значительно отличается от рабочей области Xcode, которую вы видели ранее в книге. Повторю, что playground-проект позволяет писать код и незамедлительно видеть результат его исполнения, хотя и не служит для создания полноценных самостоятельных программ. Каждый playground-проект хранится в системе в виде особого файла с расширением playground. Хочу отметить, что при разработке полноценных приложений вы можете использовать playground-проекты в качестве их составных частей и использовать реализованные в них механизмы.

1.5. Интерфейс playground-проекта

Нет способа лучше для изучения языка программирования, чем написание кода. Playground-проект предназначен именно для этого. Выберите вариант **Get started with a playground** в стартовом окне для создания нового playground-проекта. Далее Xcode попросит вас ввести имя создаваемого playground-проекта, а также выбрать платформу, для которой вы будете писать код. На выбор предлагается две платформы: iOS и macOS. 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).

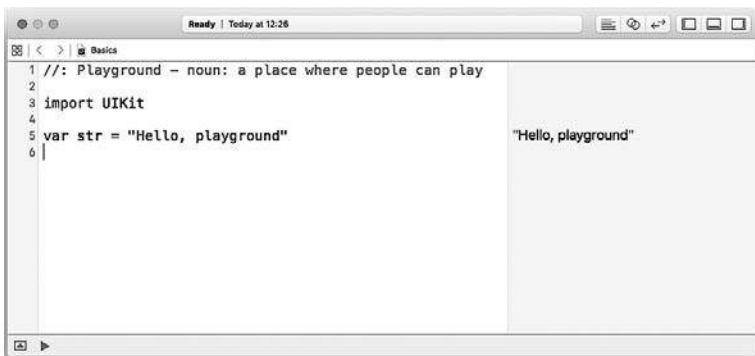


Рис. 1.7. Рабочее окно нового playground-проекта

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.
- ❑ 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).

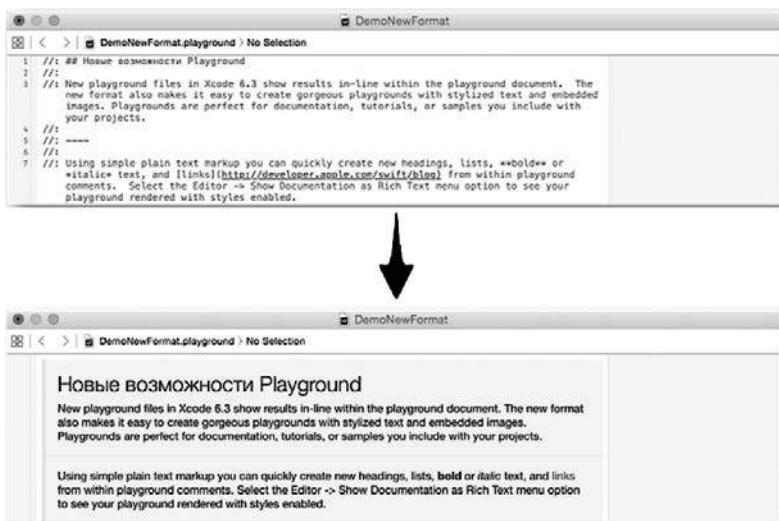


Рис. 1.9. Форматированный комментарий

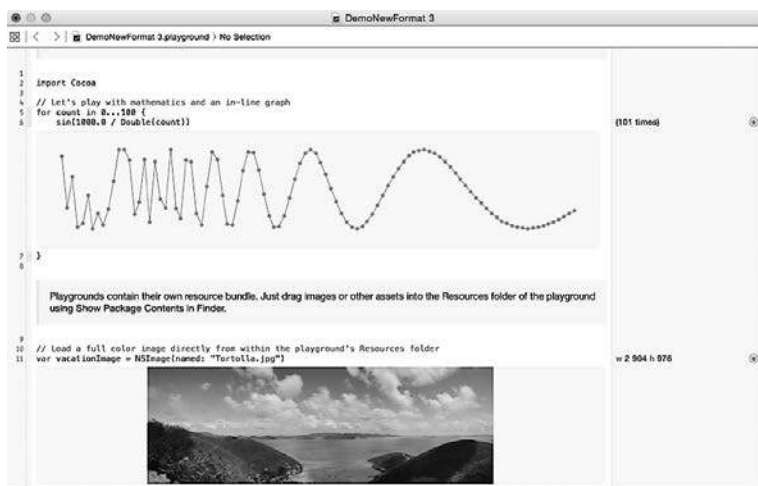


Рис. 1.10. Пример вывода результирующей информации

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.

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).

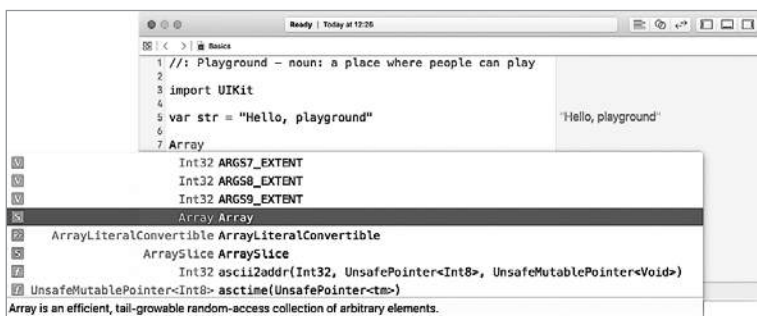


Рис. 1.11. Окно автодополнения в Xcode

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.

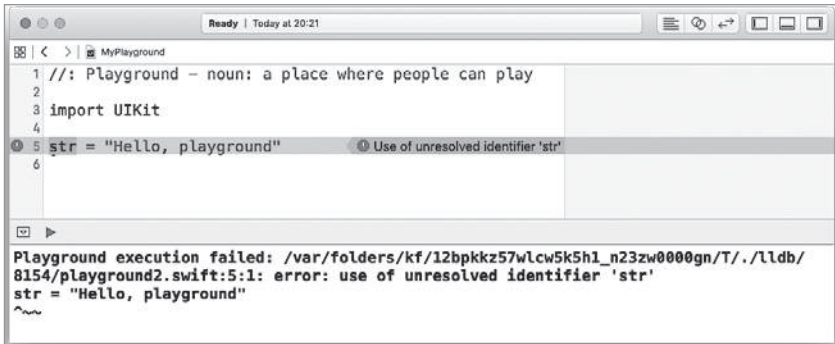


Рис. 1.12. Отображение ошибки в окне playground-проекта

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

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).

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

Теперь введите в терминале следующую команду:

```
gedit .profile
```

```

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

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

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

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

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

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
```

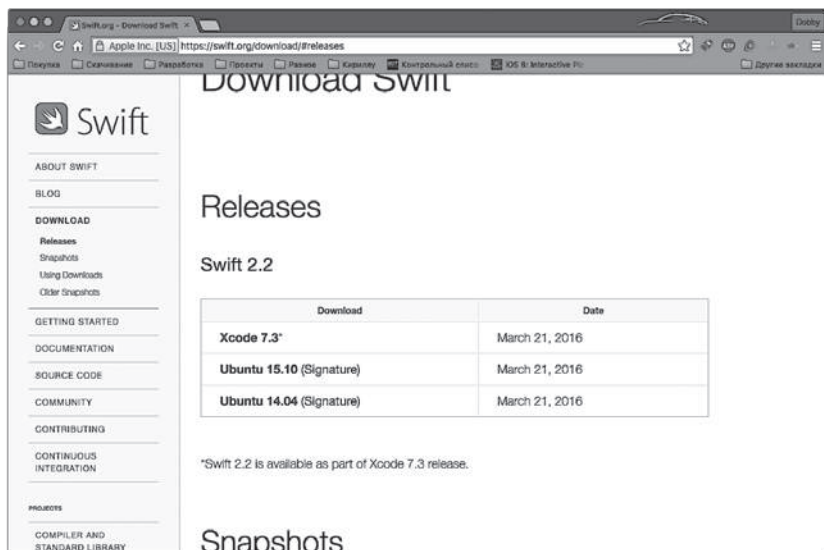


Рис. 2.2. Доступные релизы Swift



Рис. 2.3. Распакованный архив в папке Home

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'
```

```
parallels@ubuntu: ~
parallels@ubuntu:~$ export PATH=/home/parallels/swift-2.2-RELEASE-ubuntu14.04/usr/bin:${PATH}
parallels@ubuntu:~$ swift --version
Swift version 2.2 (swift-2.2-RELEASE)
Target: x86_64-unknown-linux-gnu
parallels@ubuntu:~$
```

Рис. 2.4. Вывод информации о версии Swift

где

/home/parallels/my.swift

Это путь к созданному файлу.

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

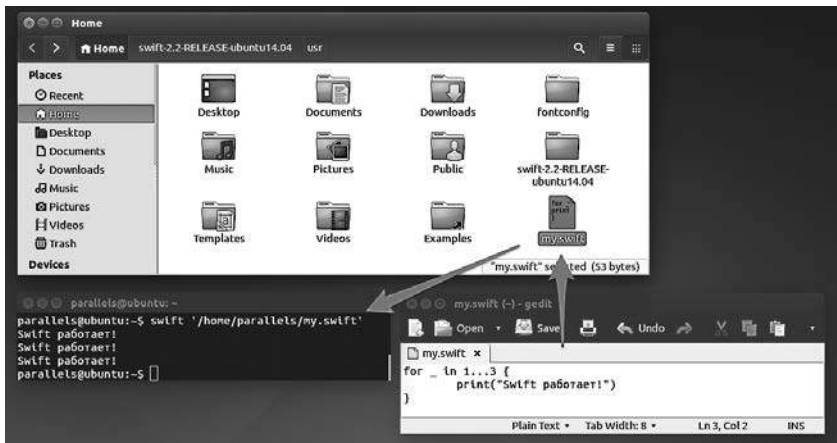


Рис. 2.5. Пример выполненного swift-кода

Вот и 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.

НАЧИНАЮЩИМ Рассмотрим следующий упрощенный пример: представьте, что оперативная память вашего компьютера состоит из миллионов отдельных ячеек. Ячейки объединены в группы с различным количеством ячеек в каждой. У каждой группы есть название. Другими словами, можно сказать, что объявлены (существуют) хранилища с именами «Группа 1», «Группа 2» и т. д. Что будет храниться в ячейках, заранее неизвестно, но можно сказать точно, что к определенному хранилищу (набору ячеек) можно обратиться по его имени (рис. 3.1).

Теперь мы решили записать в ячейки группы 1 некоторую информацию. Другими словами, решили инициализировать группе 1 определенное значение, в данном случае 23 (рис. 3.2).



Рис. 3.1. Пустые группы ячеек

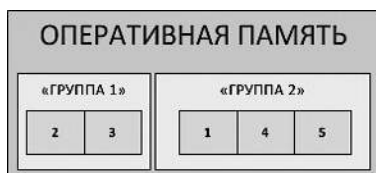


Рис. 3.2. Группы ячеек с данными

Теперь, чтобы найти значение 23, нет необходимости сканировать все ячейки оперативной памяти на предмет записанных в них данных — достаточно лишь обратиться к группе 1 и получить записанное в ней значение. Хранилище «Группа 1» — это объект, который обладает набором свойств, например: записанное значение, количество ячеек, адрес в оперативной памяти, по которому хранится группа, и т. д.

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.

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 данных. Значения, которые операторы затрагивают в своей работе, называются *операндами*. Другими словами, можно сказать, что операторы производят операции над операндами.

НАЧИНАЮЩИМ Рассмотрим простейший арифметический пример: 23×145 . Данный пример является *выражением*, то есть законченной командой на языке математики. В нем можно выделить один *оператор* (умножение) и два *операнда* (23 и 145).

Вернемся к рассмотренному ранее примеру с оперативной памятью. Чтобы произвести операцию умножения, выполняют следующую последовательность действий:

1. Загрузить значение из группы 1.
2. Загрузить значение из группы 2.
3. Перемножить загруженные значения.
4. Объявить новое хранилище с именем «Группа 3» и записать в него результат операции.

В результате мы получим новую группу ячеек памяти с проинициализированным ей значением (рис. 3.3).

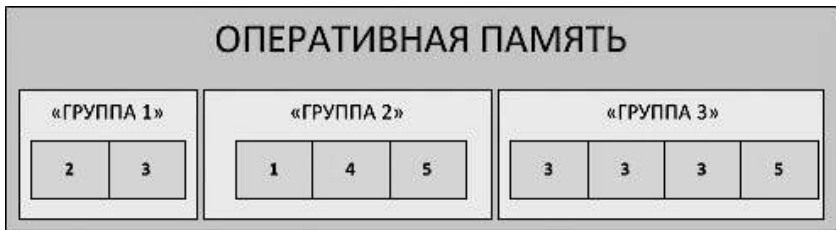


Рис. 3.3. Новая группа ячеек с записанным в нее значением

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. Среди них отдельно можно выделить *тернарный оператор*, который выполняет операции с тремя операндами. В Swift имеется всего один тернарный оператор — оператор условия $a ? b : c$, с ним мы познакомимся в дальнейшем.

Оператор присваивания

Оператор присваивания (`=`) — это особый бинарный оператор. Он используется в типовом выражении $a = b$, инициализируя значение объекта `a` значением объекта `b`.

НАЧИНАЮЩИМ Если обратиться к рассмотренному ранее примеру, то хранилищу «Группа 1» было проинициализировано значение 23, то есть `Группа1 = 23`.

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

Левая и правая части оператора присваивания должны быть однотипными (то есть иметь одинаковый тип, предназначаться для хранения данных одного и того же типа).

НАЧИНАЮЩИМ Представьте, что мы создали новое хранилище «Группа 4» и записали в него текстовые данные (слово «SWIFT») (рис. 3.4). Обратите внимание на то, что ранее мы записывали целые числа. Таким образом, можно сказать, что хранилища могут различаться по типу находящейся в них информации: первые три группы предназначены для хранения чисел, а четвертая — текста.



Рис. 3.4. Новое хранилище с текстовыми данными

Вы можете присвоить определенному хранилищу содержимое только в соответствии с его типом, то есть если хранилище предназначено для хранения, например, текста, то ничто иное, кроме текста, в него не может быть записано. Каждый из типов предназначен для хранения определенного вида значений (числа, текст, логические и другие значения). Тип информации, которая может быть записана в хранилище, называется *типом данных*.

ПРИМЕЧАНИЕ Тип определяет содержимое хранилища.

3.2. Переменные и константы

Переменные

Наиболее важное базовое понятие в любом языке программирования — переменная.

Переменная — это некая именованная область (хранилище), в которой может храниться значение определенного типа данных. Значение переменной может меняться с течением времени. Процесс изменения значения происходит не сам по себе, а инициируется извне при инициализации нового значения.

НАЧИНАЮЩИМ В предыдущем примере все четыре группы — это переменные, содержащие числовые и текстовые значения. Вы можете удалить хранящееся в любой из первых трех групп значение и записать в него новое число. —Таким образом, значение переменной изменится, но ее название останется прежним: новое число будет доступно по старому имени группы. И процесс инициализации, и процесс записи нового значения может повторяться так долго, как того требует программа.

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 имяПеременной = значениеПеременной
```

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 groupOne = 23
```

В данном примере создается переменная с именем `groupOne` и числовым значением 23. То есть код можно прочитать следующим образом: «Объявить переменную с именем `groupOne` и присвоить ей значение "23"».

В ходе работы над программой можно объявлять произвольное количество переменных.

Чтобы изменить значение переменной, необходимо присвоить ей новое значение, оператор `var` при этом 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 groupOne = 23
3 // изменяем значение объявленной ранее переменной
4 groupOne = 46
```

В результате выполнения кода в переменной `groupOne` будет храниться значение 46.

Константы

Для любого хранилища информации в Swift существует возможность указать правило, запрещающее изменение первоначально установленного значения. Такие хранилища будут содержать переданное им значение до момента уничтожения и не изменяться ни при каких условиях. Такие хранилища называются *константами*.

НАЧИНАЮЩИМ Так, например, мы могли бы объявить ранее созданную группу 4 как константу, и записанное в нее значение «SWIFT» не могло бы измениться в течение работы программы.

Константы объявляются с помощью оператора `let`.

Синтаксис

```
let имяКонстанты = значениеКонстанты
```

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 groupFour = "Swift"
```

В результате будет объявлена константа `groupFour`, содержащая значение `"Swift"`. Данное значение нельзя изменить ни при каких обстоятельствах, но оно может быть удалено вместе с константой.

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

Листинг 3.4

```
1 let groupFour = "Swift"
2 groupFour = "Apple Swift" // ОШИБКА: попытка изменить константу
```

Раньше, программируя на других языках, вы, вероятно, не очень активно использовали константы. Swift — это язык, который позволяет очень гибко управлять ресурсами, помогая вам в этом. Константы — это самое первое и базовое средство оптимизации используемых мощностей. В связи с этим вам следует прибегать к использованию констант всегда, 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 константе 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.

При необходимости объявить несколько параметров вы можете писать каждый из них с новой строки, используя при этом оператор `var` или `let` для каждого параметра отдельно или написав оператор однажды и перечислив после него через запятую создаваемые параметры (листинг 3.5).

Листинг 3.5

```
1 // объявим переменные по одной в строке
2 var groupOne = 23
3 var groupTwo = 145
4 // объявим переменные в одну строку
5 var groupThree = 3335, groupFour = "Swift"
```

Использование переменных и констант приносит 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 *каamelкейс-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 (заглавной) буквы. Например: `myBestText`, `theBestCountry`, `highScore`.

Хотя их имена и могут содержать любые Unicode-символы (листинг 3.6), их использование только мешает читабельности кода.

Листинг 3.6

```
1 // Попробуйте прочитать имя этой переменной или ее значение
2 var dŧw = "Rǫĩ"
```

- ❑ 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 какое-либо служебное ключевое слово, то следует написать его в апострофах (`'`), но я настоятельно рекомендую избегать этого, чтобы не нарушать читабельность кода.

В листинге 3.7 приведен пример создания переменной с именем, соответствующим занятому ключевому слову `var`. Для того чтобы создать переменную с таким же именем, необходимо использовать апострофы.

Листинг 3.7

```

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

```

*"Пример переменной
в апострофах"*
*"Пример переменной
в апострофах"*

Обратите внимание на то, 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. Глобальные и локальные объекты

Любая программа содержит большое количество различных объектов, которые в свою очередь содержат другие объекты. Примерами объекта могут служить в том числе и переменные с константами.

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

Каждый созданный объект может использоваться только в соответствии со строго определенными правилами, одно из которых называется областью видимости. Область видимости определяет, где именно в коде можно использовать тот или иной объект. По принадлежности к области видимости объекты делятся на локальные и глобальные.

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

НАЧИНАЮЩИМ Рассмотрим следующий пример: у вас есть программа, в которой существуют несколько объектов. У каждого объекта есть собственное имя, по которому он доступен (рис. 3.5).

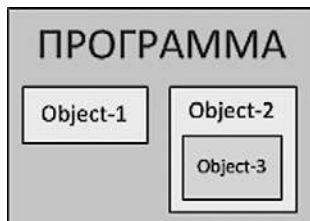


Рис. 3.5. Вариант иерархии объектов внутри программы

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

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

Рассмотрим другой вариант организации объектов (рис. 3.6).

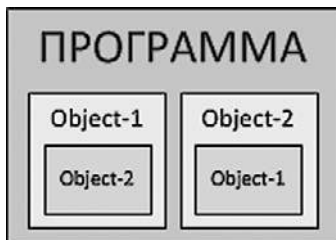


Рис. 3.6. Другой вариант иерархии объектов внутри программы

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

Запомните, что одноименные локальный и глобальный объекты — это разные объекты, которые могут иметь совершенно разные значения и характеристики. Их объединяет только общее имя.

Вот такая запутанная, но очень важная для правильного понимания истории.

Время жизни локальных объектов равно времени жизни их родительских объектов. Также стоит отметить, что локальные объекты при повторном вызове родительского объекта не сохраняют свое состояние: если объект был вызван и уничтожен, то при следующем вызове старые значения локальных объектов будут недоступны.

3.5. 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 3.8).

Листинг 3.8

```
1 // объявим переменную и присвоим ей значение
2 var groupFour = "Swift"                                "Swift"
3 // отобразим значение в области результатов
4 groupFour                                              "Swift"
```

При разработке проектов (будь то playground-проект или полноценное приложение для iOS или macOS) любые данные можно отобразить на отладочной консоли (с ней мы знакомимся ранее), которая находится в нижней части рабочей области. Вывод на консоль осуществляется с помощью глобальной функции `print(_)`.

НАЧИНАЮЩИМ Напомню, что элемент называется глобальным в том случае, если он доступен в любой точке программы. Так как функция `print(_)` является неотъемлемой частью языка Swift, она как раз и является глобальной.

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. То есть функция — это такая же группа ячеек, как мы рассматривали ранее, только вместо значения в ней находится программный код.

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 в функцию можно передать некоторые данные, которые будут использоваться в записанном в нее программном коде (в ее внутренней реализации). Такие данные называются входными параметрами, или аргументами функции. Они указываются в скобках после имени самой функции.

ПРИМЕЧАНИЕ 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 3.9.

Листинг 3.9

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

"Привет, ученик!\n"

Консоль:

Привет, ученик!

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(_:)`. С тех пор 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 3.10.

Листинг 3.10

```
1 // объявим переменную и присвоим ей значение
2 var foo = "Текст для консоли"
```

"Текст для консоли"

```
3 // Выводим значение созданной ранее переменной
4 print(foo)
```

"Текст для консоли\n"

Консоль:

Текст для консоли

Созданная переменная `foo` передается в функцию `print(_:)` в качестве входного аргумента (входного параметра), и ее значение выводится 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 3.11.

Листинг 3.11

```
1 // объявим переменную и присвоим ей значение
2 var bar = "Swift"
3 // Выводим значение созданной ранее переменной
4 print("Я изучаю \( bar)")
```

"Swift"
"Я изучаю Swift\n"

Консоль:

Я изучаю Swift

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

3.6. 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 3.12.

Листинг 3.12

```
1 // это – однострочный комментарий
2 /* это -
3    многострочный
4    комментарий */
```

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

В предыдущей главе упоминалось, что playground-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. 3.7.`

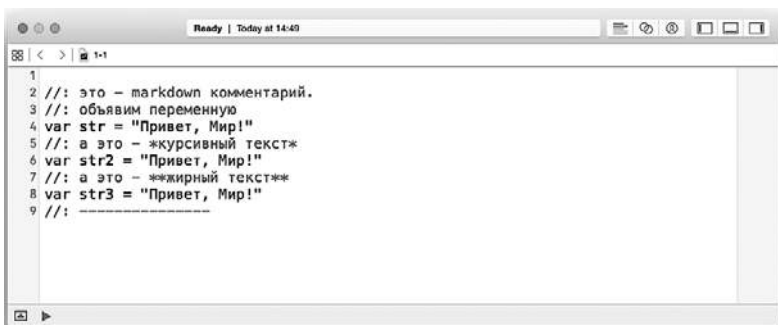


Рис. 3.7. Неформатированные markdown-комментарии

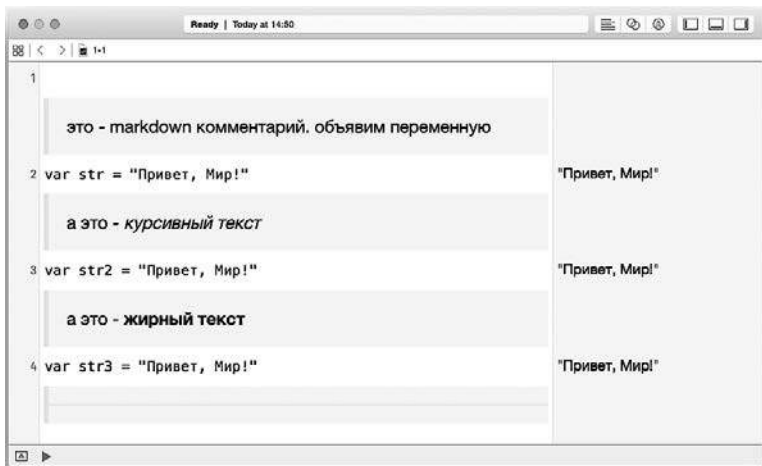


Рис. 3.8. Отформатированные markdown-комментарии

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. 3.8.

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

Обращаю ваше внимание на то, что данный способ форматирования комментариев гарантированно возможен лишь в Xcode. Если вы разрабатываете в ОС Linux, то данная функция определяется возможностями выбранного вами редактора кода.

Рассмотренные выше комментарии предназначены собственно для разработчика программного кода, то есть 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 3.13).

Листинг 3.13

```
1  /**
2   This func say hello to user
3   - parameter name:String Name of user
4   - returns: Absolutely nothing
5   - throws: Error when name is array.
6   - authors: Bilbo Baggins
7   - bug: This is very simple function
8  */
9  func sayHello(name: String) {
10     print("hello, \(name)!")
11 }
12 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. 3.9).

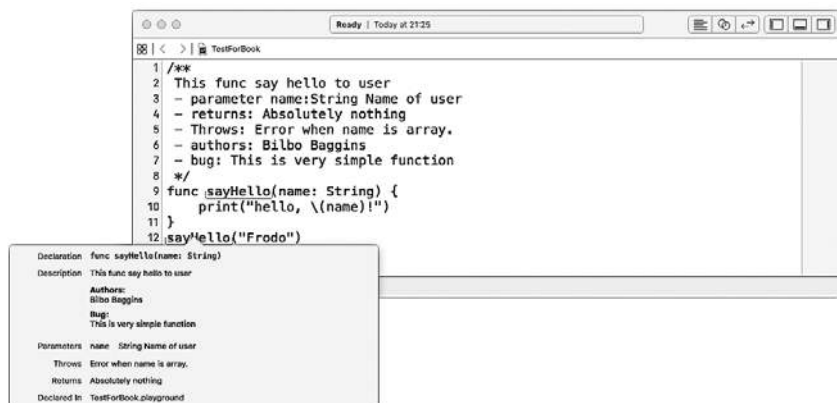


Рис. 3.9. Справочное окно

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.

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 на то, что перед каждым ключевым словом ставится символ «тире».

ПРИМЕЧАНИЕ С большой долей вероятности в настоящее время для вас остается много непонятого 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/>.

3.7. 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 3.14.

Листинг 3.14

```
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 очень дружелюбен. Этот замечательный язык программирования не требует от вас писать лишние символы, давая при этом широкие возможности для того, чтобы код был понятным и прозрачным. В последующих главах вы увидите 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, знакомое нам хранилище «Группа 1» может содержать числовые данные, а «Группа 4» — текстовые. Со значениями каждого типа данных можно совершать только определенные операции: так, например, числа можно складывать или вычитать, строки объединять между собой и т. д.

ПРИМЕЧАНИЕ Запомните, что хранилище определенного типа никогда не сможет хранить данные другого типа.

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 по переданному ему значению. Рассмотрим пример из листинга 4.1.

Листинг 4.1

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

Данный код можно прочитать следующим образом:

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

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 `sometext` и присвоить ей *строковое* значение "Я учу Свифт".

Или еще вариант:

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

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

```
1 // объявляем пустую переменную
2 var dragonsName: String
3 // создаем непустую переменную orksName с неявным определением типа
4 var orksName = "Орк "
5 // создаем непустую переменную elvesName с явным определением типа
6 var elvesName: String = "Эльф "
```

Все три переменные имеют строковый тип данных **String**, то есть могут хранить в себе строковые значения. Переменная `dragonsName` — пустая, а переменные `orksName` и `elvesName` хранят в себе текстовые значения.

У любой объявленной переменной или константы должен быть определен тип. Это значит, что вы должны либо инициализировать значение, либо явно указать тип данных. Игнорирование этого правила ведет к ошибке (листинг 4.3).

Листинг 4.3

```
1 var hobbitsName // 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. В первом случае мы получаем два независимых параметра, а во втором — две ссылки на один и тот же параметр.

НАЧИНАЮЩИМ Представьте, что хранилище `groupOne` — значимого типа. При передаче значения, хранящегося в `groupOne`, в новую переменную `groupOneCopy` произойдет создание полной независимой копии. Никакие изменения, вносимые в `groupOneCopy`, не повлияют на значение, хранящееся в `groupOne` (листинг 4.4).

Листинг 4.4

```
1 var groupOne = 23
2 var groupOneCopy = groupOne
3 print(groupOneCopy)
4 groupOneCopy = 25
5 print(groupOneCopy)
6 print(groupOne)
```

Консоль:

```
23
25
23
```

Из примеров видно, что, при изменении значения, хранящегося в переменной `groupOneCopy`, значение переменной `groupOne` не изменяется.

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

В дальнейшем 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» или «плюс». Так, например, "2", "-64", "+18" — все это числовые литералы.

Для объявления переменной или константы целочисленного типа необходимо использовать ключевые слова `Int` (для знаковых) и `UInt` (для беззнаковых). Пример инициализации целочисленных переменных приведен в листинге 4.5.

Листинг 4.5

```
1 // объявим переменную знакового целочисленного типа
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 выше целочисленные типы данных — это разные типы данных (как строковый и числовой), и значения в этих типах не могут взаимодействовать между собой напрямую. Все операции в Swift должны происходить между значениями одного и того же типа данных! Это важно!

В самом начале книги говорилось об одной из особенностей Swift: всё в этом языке программирования является объектами. *Объект* — это некоторая сущность, реализованная с помощью программного кода. 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 "выдать товар".

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

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

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 рассмотренных ранее числовых типов данных (а их много) описывает сущность «целое число». Разница между ними состоит в том, какие максимальное и минимальное числа могут хранить переменные этих типов данных. Вы можете получить доступ к данным характеристикам через свойства `min` и `max`. Как говорилось ранее, эти значения зависят от разрядности системы, на которой выполняется приложение. Для примера получим макси-

мально и минимально возможные значения для типов `Int8` и `UInt8` (листинг 4.6).

Листинг 4.6

```

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

```

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

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

ПРИМЕЧАНИЕ Запомните, что среди целочисленных типов данных 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. Обаим переменным должны быть присвоены значения.
5. Поменяйте значения переменных местами. Для этого вам придется использовать еще одну переменную, которая будет служить буфером.
6. 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 при работе с целочисленными типами. Для объявления параметров, которые могут хранить числа с плавающей точкой, используются два ключевых слова: **Float** и **Double**. Оба типа данных являются знаковыми, то есть могут хранить положительные и отрицательные значения.

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

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

Разница этих типов данных состоит в точности: поскольку **Double** позволяет хранить больше знаков в дробной части, то и используется он лишь при необходимости выполнения расчетов высокой точности. Пример объявления параметров, содержащих такие числа, приведен в листинге 4.7.

Листинг 4.7

```

1 // дробное число типа Float с явным указанием типа
2 var numFloat: Float = 104.3
3 // пустая константа типа Double
4 let numDouble: Double
5 // дробное число типа Double с неявным указанием типа
6 var someNumber = 8.36
```

104,3

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 времен занятий по математике, когда с помощью минуса вы могли как определить отрицательное число, так и выполнить операцию вычитания.

- * 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.

ПРИМЕЧАНИЕ В Swift 3 был изменен подход к использованию некоторых операторов. Например, довольно часто используемые в программировании операторы инкремента ($++$) и декремента ($--$), увеличивающие и уменьшающие значение на единицу соответственно, были удалены из языка.

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

Чтобы продемонстрировать использование данных операторов, создадим две целочисленные переменные и две переменные типа `Double` (листинг 4.8).

Листинг 4.8

```
1 // целочисленные переменные
2 var numOne = 19
3 var numTwo = 4
4 // переменные типа числа с плавающей точкой
5 var numThree = 3.13
6 var numFour = 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 4.9.

Листинг 4.9

```
1 // целочисленные переменные
2 var numOne = 19
3 var numTwo = 4
4 // операция сложения
5 var sum = numOne + numTwo
```

```

6 // операция вычитания
7 var diff = numOne - numTwo           15
8 // операция умножения
9 var mult = numOne * numTwo           76
10 // операция деления
11 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 4.10).

Листинг 4.10

```

1 // переменные типа числа с плавающей точкой
2 var numThree = 3.13
3 var numFour = 1.1
4 // операция сложения
5 var sumD = numThree + numFour           4,23
6 // операция вычитания
7 var diffD = numThree - numFour           2,03
8 // операция умножения
9 var multD = numThree * numFour           3,443
10 // операция деления
11 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.

Помимо рассмотренных в предыдущем листинге операций в Swift существует функция вычисления остатка от деления, при которой

первый операнд делится на второй и возвращается остаток от этого деления. Или, другими словами, определяет, как много значений второго операнда поместится в первом, и возвращает значение, которое осталось, — оно называется остатком от деления. Тип результирующего значения соответствует типу операндов (листинг 4.11).

Листинг 4.11

```

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

```

Данный листинг не будет корректно выполняться в Xcode, так код выполнения операции вычисления остатка от деления будет приведен ниже. Вместо него в данном случае используется эмулирующий комментарий /* операция вычисления остатка от деления */.

В данном примере рассматриваются три варианта операции целочисленного деления, отличающиеся 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.

При использовании оператора остатка от деления есть одно ограничение: он используется только для целочисленных значений. Для дробных существует другой подход.

Ранее мы рассмотрели использование свойств `min` и `max` для получения минимальных и максимальных значений целочисленных типов данных. Теперь перейдем к рассмотрению использования одного метода, позволяющего выполнить операцию вычисления остатка от деления.

Для выполнения операции вычисления остатка от деления используется метод `truncatingRemainder(dividingBy:)`, который применяется к делимому (то есть пишется через точку после числа, которое требуется разделить). Имя `dividingBy` внутри скобок говорит о том, что данный метод имеет входной аргумент, то есть значение, которое будет передано ему при его вызове и использовано в ходе работы метода (листинг 4.12).

Листинг 4.12

```

1 // дробные переменные
2 var numOne: Float = 3.14                                3.14
3 var numTwo: Float = 1.01                                1.01
4 // операция получения остатка от деления
5 var res1 = numOne.truncatingRemainder(dividingBy:
    numTwo)                                                0.1100001
6 var res2 = -numOne.truncatingRemainder(dividingBy:
    numTwo)                                                -0.1100001
7 var res3 = numOne.truncatingRemainder(dividingBy:
    -numTwo)                                                0.1100001

```

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

Приведение числовых типов данных

Как неоднократно говорилось, 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 типы данных в другие. Данный механизм называется

приведением (от слова «привести»), и выполнен он в виде глобальных функций.

ПРИМЕЧАНИЕ Справедливости ради стоит отметить, что на самом деле приведенные далее 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 он может быть вызван следующим образом:

```
ИмяТипаДанных.init(_:)
```

В результате вызова данного метода будет создан новый объект, описывающий некую сущность, соответствующую типу данных (число, строка и т. д.). Swift позволяет не писать имя инициализатора при создании объекта, а использовать сокращенный синтаксис:

```
ИмяТипаДанных(_:)
```

В результате выполнения данного кода также будет создан объект, описывающий ту же самую сущность, что и в предыдущем примере.

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 типа `Int` и дробное типа `Double` (листинг 4.13).

Листинг 4.13

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

19

3,13

59,47

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`, так как во втором случае дробная часть отбрасывается и игнорируется при расчетах.

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

Составной оператор присваивания

Swift позволяет максимально сократить объем кода. И чем глубже вы будете постигать этот замечательный язык, тем больше приемов, способных облегчить вам жизнь, узнаете. Одним из таких приемов является совмещение оператора арифметической операции (+, -, *, /, %) и оператора присваивания (=). Рассмотрим пример в листинге 4.14, в котором создадим целочисленную переменную и с помощью составного оператора присваивания будем изменять ее значение, используя минимум кода.

Листинг 4.14

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

19

24

```

8  someNumInt                                     24
9  // умножим его на число 3
10 someNumInt *= 3                                72
11 /* эта операция аналогична выражению
12 someNumInt = someNumInt*3 */
13 // выведем результат операции
14 someNumInt                                     72
15 // вычтем из него число 3
16 someNumInt -= 3                                69
17 /* эта операция аналогична выражению
18 someNumInt = someNumInt-3 */
19 // выведем результат операции
20 someNumInt                                     69
21 // найдем остаток от деления на 8
22 someNumInt %= 8                                 5
23 /* эта операция аналогична выражению
24 someNumInt = someNumInt%8 */
25 // выведем результат операции
26 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 4.15).

Листинг 4.15

```

1 // переменная типа Int
2 var someNumInt = 19                               19
3 // увеличим ее значение с использованием арифметической операции
  сложения
4 someNumInt = someNumInt + 1                         20
5 // увеличим ее значение с использованием составного оператора
  присваивания
6 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

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.

Задание 2

1. Объявите переменные `a` и `b` целочисленного типа данных и присвойте им значения 2 и 3.
2. Вычислите значение уравнения $(a+4b)(a-3b)+a^2$ и выведите результат на консоль.

Задание 3

1. Объявите переменные `c` и `d` присвойте им значения 1.75 и 3.25.
2. Вычислите значение уравнения $2(c+1)^2+3(d+1)$ и выведите результат на консоль.

Задание 4

1. Объявите константу `length` и запишите в нее произвольное целочисленное значение.
2. Вычислите следующие значения, запишите каждое из них в новую константу и выведите на консоль:
 - площадь квадрата с длинной стороны, равной `length`;
 - длину окружности радиусом `length`.

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 4.16 показан пример записи числа 17 в различных видах.

Листинг 4.16

```

1 // 17 в десятичном виде
2 let decimalInteger = 17
3 // 17 в двоичном виде
4 let binaryInteger = 0b10001
5 // 17 в восьмеричном виде
6 let octalInteger = 0o21
7 // 17 в шестнадцатеричном виде
8 let hexadecimalInteger = 0x11
```

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 $1,25 \times 10^2$, или 125,0.

Для шестнадцатеричных чисел экспонента указывает на степень двойки:

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

В листинге 4.17 приведен пример записи десятичного числа 12,1875 в различных системах счисления и с использованием экспоненты.

Листинг 4.17

```

1 // десятичное число
2 let decimalDouble = 12.1875                                12,1875
3 // десятичное число с экспонентой /* соответствует выражению
   exponentDouble = 1.21875*101 */
4 let exponentDouble = 1.21875e1                              12,1875
5 // шестнадцатеричное число с экспонентой /* соответствует
   выражению hexadecimalDouble = 0xC.3*20 */
6 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 4.18).

Листинг 4.18

```

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

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

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

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 4.19). Всего один символ!

Листинг 4.19

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

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

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`.

Таким образом, тип данных `String` — это упорядоченный набор символов, позволяющий хранить в параметре строку произвольной длины.

ПРИМЕЧАНИЕ Swift — все еще идет по пути становления, и в каждой новой версии в нем происходят значительные изменения. Так, в Swift 2.0 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 4.20).

Листинг 4.20

```
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 4.21).

Листинг 4.21

```
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, в данном случае в `String`.

Подумайте, что будет в переменной типа `String`, если ее объявить, но не передать значение?

При явном указании типа `String` без инициализации значения в переменной или в константе не окажется вообще никакого значения, в том числе и пустого строкового литерала, — он также является значением. Прежде чем использовать параметр, объявленный таким путем, требуется инициализировать значение. Такой подход применим к любому типу данных. Рассмотрим пример из листинга 4.22.

Листинг 4.22

```
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 4.23).

Листинг 4.23

```
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 4.24).

Листинг 4.24

```
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 4.25).

Листинг 4.25

```
1 // переменная типа String
2 var name = "Дракон"                                   "Дракон"
3 // константа типа String с использованием интерполяции
4 let hello = "Привет, меня зовут \(name)!"             "Привет, меня зовут
                                                         Дракон!"

5 // интерполяция с использованием выражения
6 var meters: Double = 10                               10
7 let text = "Моя длина \(meters * 3.28) футов"         "Моя длина 32.8 футов"
```

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 4.26).

Листинг 4.26

```
1 // константа типа String
2 let firstText = "Мой вес "           "Мой вес "
3 // переменная типа Double
4 var weight = 12.4                   12,4
5 // константа типа String
6 let secondText = " тонны"          " тонны"
7 // конкатенация строк при инициализации
  значения новой переменной
8 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 получить набор входящих в него символов с помощью свойства `characters`. Как и любое свойство в Swift, `characters` пишется через точку после имени строкового параметра (листинг 4.27).

Листинг 4.27

```
1 // переменная типа String
2 var str = "Hello, Troll!"           "Hello, Troll!"
3 // получаем коллекцию символов
4 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 собственные свойства. Для примера получим количество символов в строке, используя созданную коллекцию `collection` и ее свойство `count` (листинг 4.28).

Листинг 4.28

```
1 // количество символов в строке
2 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

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.

Задание 2

1. Выведите на консоль букву `W`, составленную из символов «звездочка» (*), расположенных в три строки.

```
  *   *   *
 * * * *
  *   *
```

Задание 3

1. Объявите переменную и запишите в нее название текущую дату.

2. Объявите переменную и запишите в нее название текущего месяца.
3. Объявите переменную и запишите в нее текущий год.
4. Выведите на консоль текущую дату в формате 2017 январь 01, используя данные, записанные в объявленные ранее переменные.

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 4.29).

Листинг 4.29

```
1 // логическая переменная с неявно заданным типом
2 var isDragon = true                                true
3 // логическая константа с явно заданным типом
4 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 4.30).

Листинг 4.30

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

```

6  }else{
7  print("Привет, Троль!")
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, Троль!».

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 4.31).

Листинг 4.31

```

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 4.32).

Листинг 4.32

```
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 4.33).

Листинг 4.33

```
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 4.34).

Листинг 4.34

```
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 4.35).

Листинг 4.35

```

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

```

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 4.36).

Листинг 4.36

```

1 // определяем псевдоним для типа UInt8
2 typealias ageType = UInt8
3 /* создаем переменную типа UInt8,
4 используя псевдоним */
5 var myAge: ageType = 29

```

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

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

У типа может быть произвольное количество псевдонимов. И все псевдонимы вместе с оригинальным названием типа можно использовать в программе (листинг 4.37).

Листинг 4.37

```
1 // определяем псевдоним для типа String
2 typealias textType = String
3 typealias wordType = String
4 typealias charType = String
5 //создаем переменные каждого типа
6 var someText: textType = "Это текст"           "Это текст"
7 var someWord: wordType = "Слово"              "Слово"
8 var someChar: charType = "Б"                  "Б"
9 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 4.38).

Листинг 4.38

```
1 // объявляем псевдоним
2 typealias ageType = UInt8
3 /* используем свойство типа
4 UInt8 через его псевдоним */
5 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 два и более подтипа. 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:

$$\text{ИМТ} = \text{вес}[\text{кг}] / \text{рост}[\text{м}]^2$$
5. Выведите на консоль текст, содержащий ваши имя и индекс массы тела.

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 4.39.

Листинг 4.39

```
1 // Утверждение "1 больше 2"
2 1 > 2                                false
3 // вернет false, так как оно ложно
4 // Утверждение "2 не равно 2"
5 2 != 2                               false
6 // вернет false, так как оно ложно
7 // Утверждение "1 плюс 1 меньше трех"
8 (1+1) < 3                            true
9 // вернет true, так как оно истинно
10 // Утверждение "5 больше или равно 1"
11 5 >= 1                               true
12 // 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 следующей главы.

4.7. Операторы диапазона

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.

Скоро мы рассмотрим примеры использования данных операторов.

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!

- ✓ Глава 5. Кorteжи
- ✓ Глава 6. Опциональные типы данных
- ✓ Глава 7. Операторы условий
- ✓ Глава 8. Коллекции
- ✓ Глава 9. Циклы
- ✓ Глава 10. Функции
- ✓ Глава 11. Замыкания
- ✓ Глава 12. Вспомогательные функциональные элементы
- ✓ Глава 13. Ленивые вычисления

5

Кортежи

Возможно, вы никогда не встречались в программировании с таким понятием, 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 =                (.0 200, .1 "In Work", .2 true)
        (200, "In Work", 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 некоей 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 входящих в него 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 заданным типом
2  let myProgramStatus: (Int, String, Bool) = (.0 200, .1 "In Work",
        (200, "In Work", true)                .2 true)
3  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

```

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

```

В кортеже `numbersTuple` 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

```

1 // объявляем кортеж
2 let myProgramStatus =                (.0 200, .1 "In Work",
   (200, "In Work", true)              .2 true)
3 // записываем значения кортежа в переменные
4 var (statusCode, statusText,
   statusConnect) = myProgramStatus
5 // выводим информацию
6 print("Код ответа - \(statusCode)")  "Код ответа - 200\n"

```

```

7  print("Текст ответа - \(statusText)")           "Текст ответа -
In Work\n"
8  print("Связь с сервером - \(statusConnect)")    "Связь с сервером -
true\n"

```

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 набору переменных или констант можно присвоить произвольные значения (листинг 5.5).

Листинг 5.5

```

1  /* объявляем сразу несколько
2  переменных и устанавливаем
3  для них значения */
4  var (myName, myAge) = ("Троль", 140)
5  // выводим их значения
6  print("Меня зовут \(myName),           "Меня зовут Троль,
      и мне \(myAge) лет")              и мне 140 лет\n"

```

Консоль:

Меня зовут Троль, и мне 140 лет

Переменные `myName` и `myAge` инициализированы соответствующими значениями элементов кортежа ("Троль", 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

```

1  // объявляем кортеж
2  let myProgramStatus: (Int, String, Bool) = (.0 404, .1 "Error",
      (404, "Error", true))           .2 true)
3  /* получаем только необходимые
4  значения кортежа */
5  var (statusCode, _, _) = myProgramStatus
6  // выводим информацию
7  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 далее.

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

```
1 // объявляем кортеж
2 let myProgramStatus: (Int, String, Bool) = (.0 200, .1 "In Work",
   (200, "In Work", true)           .2 true)
3 // выводим информацию с использованием индексов
4 print(" Код ответа — \(myProgramStatus.0)")    "Код ответа — 200\n"
5 print(" Текст ответа —                          "Текст ответа —
   \(myProgramStatus.1)")                      "In Work\n"
6 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

```
1 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

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

```
1 /* объявляем кортеж с
2 указанием имен элементов
3 в описании типа */
4 let myProgramStatus: (statusCode: Int,      (.0 200, .1
   statusText: String, statusConnect: Bool) = "In Work", .2
   (200, "In Work", true)                  true)
5 /* выводим значение элемента
6 кортежа с помощью имени
7 этого элемента*/
8 myProgramStatus.statusCode                200
9 /* объявляем кортеж с
10 указанием имен элементов
11 при инициализации их значений */
```

```

12 let myNewProgramStatus = (statusCode: 404,      (.0 404, .1 "Error",
    statusText:"Error", statusConnect:true)      .2 true)
13 /* выводим значение элемента
14 кортежа с помощью имени
15 этого элемента*/
16 myNewProgramStatus.statusText                  "Error"

```

Изменение значений кортежей

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

Листинг 5.11

```

1  // объявляем пустой кортеж
2  var myFirstTuple: (Int, String)
3  // создаем кортеж со значением
4  var mySecondTuple = (100, "Код")           (.0 100, .1 "Код")
5  // передаем значение кортежа
6  myFirstTuple = mySecondTuple              (.0 100, .1 "Код")
7  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

```

1  // объявляем кортеж
2  var someTuple: (200, true)                 (.0 200, .1 true)
3  // изменяем значение отдельного элемента
4  someTuple.0 = 404
5  someTuple.1 = false
6  someTuple                                  (.0 404, .1 false)

```

Кортежи позволяют делать то, чего вам, возможно, не доставало 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 будут обнаружены неидентичные элементы (листинг 5.13).

Листинг 5.13

```
1 (1, "alpha") < (2, "beta")                                true
2 // истина, так как 1 меньше 2.
3 // вторая пара элементов не учитывается
4 (4, "beta") < (4, "gamma")                                true
5 // истина, так как "beta" меньше "gamma".
6 (3.14, "pi") == (3.14, "pi")                              true
7 // истина, так как все соответствующие элементы идентичны
```

ПРИМЕЧАНИЕ Встроенные механизмы Swift позволяют сравнивать кортежи с количеством элементов менее 7. При необходимости сравнения кортежей с большим количеством элементов вам необходимо реализовать 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 ноль — 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`, обозначаемый 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

```

1  // переменная с числовым строковым литералом
2  let possibleString = "1945"                                "1945"
3  /* при попытке конвертации
4  преобразуется в целочисленный тип */
5  let convertPossibleString = Int(possibleString)             1 945
6  // переменная со строковым литералом
7  let impossibleString = "Привет, Дракон!"                     "Привет, Дракон!"
8  /* при попытке конвертации
9  не преобразуется в целочисленный тип */
10 let convertImpossibleString = Int(impossibleString)         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

```

1  /* переменная с опциональным типом Int
2  и с установленным значением */
3  var dragonAge: Int? = 230                                    230
4  // уничтожаем значение переменной
5  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 сочтет ее равной `nil`.

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

Листинг 6.3

```
1 // опциональная переменная с установленным значением
2 var optionalVar = Optional("stringValue")           "stringValue"
3 optionalVar                                           "stringValue"
4 // уничтожаем значение опциональной переменной
5 optionalVar = nil                                    nil
6 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 извлечь.

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

```

1  /* опциональная переменная
2  с установленным значением */
3  var trollAge: Int? = 95
4
5  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

```

1  /* опциональная переменная
2  с установленным значением */
3  var trollAge: Int? = 95
4  // проведение арифметической операции
5  trollAge = trollAge! + 10
6  trollAge

```

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 основной 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

```
1 var type: String
2 // принудительное извлечение опционального значения
3 let monsterOneType: String? = "Дракон"
4 type = monsterOneType!
5 type
6 // косвенное извлечение опционального значения
7 let monsterTwoType: String! = "Троль"
8 type = monsterTwoType
9 type
```

"Дракон"
"Дракон"
"Дракон"
"Троль"
"Троль"
"Троль"

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

Листинг 6.7

```
1 let pointCoordinates: (Int, Int)! = nil
2 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

Операторы условий

Представьте себе реку: ее течение, плавные повороты и заводи. Человечество обладает технологиями, благодаря которым можно управлять течением реки: 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, останавливать его, пропускать блоки кода или, наоборот, многократно их повторять и т. д.

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.

7.1. 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 этап разработки программы, на котором обнаруживаются и устраняются ошибки.

Утверждение представляет собой специальную функцию `assert(_:file:line:)`, проверяющую некоторое условие на предмет 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 `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
4  // проверка корректности условия
5  assert( dragonAge >= 225, "Ваш дракон
    слишком молод, чтобы продолжить" )
6  // выводим значение переменной
7  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).

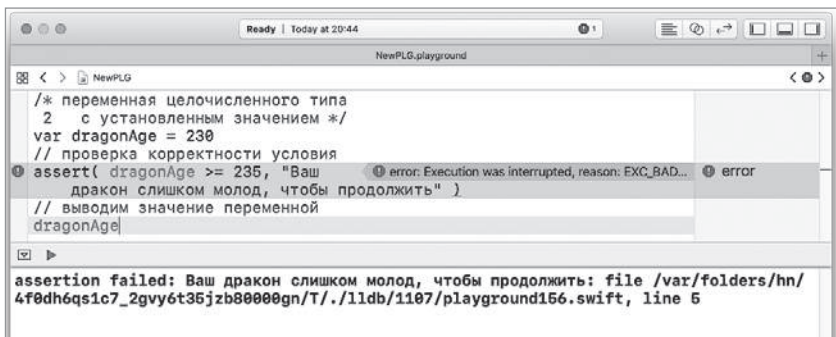


Рис. 7.1. Ошибка при невыполнении переданного в утверждение условия

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

Листинг 7.2

```
1  /* переменная целочисленного типа
2  с установленным значением */
3  var dragonAge = 230                                230
4  // проверка корректности условия
5  assert( dragonAge >= 225 )
6  // выводим значение переменной
7  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.

7.2. Оператор условия if

Синтаксис оператора if

Утверждения, с которыми вы только что познакомились, 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. Данная конструкция используется практически в каждой программе независимо от ее сложности.

Синтаксис

```
if проверяемое_условие {  
    // выполняемый код (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 кодом, следующим за оператором условия.

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 7.4.

Листинг 7.4

```
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 7.5).

Листинг 7.5

```
1 var intVar = 1
2 if intVar { // ОШИБКА
3     // код в теле оператора не будет выполнен
4 }
```

Если вам необходимо выполнить блок кода при ложности переданного условия, то вы можете использовать оператор логического отрицания (листинг 7.6).

Листинг 7.6

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

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 {
    // выполняемый код (при истинности проверяемого условия)
} else {
    // выполняемый код (при ложности проверяемого условия)
}
```

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 7.7 приведен пример использования стандартного синтаксиса логического оператора `if`.

Листинг 7.7

```
1 // переменная типа Bool
2 var logicVar = false
3 // проверка значения переменной
4 if logicVar {
5     print("Переменная logicVar истинна")
6 } else {
7     print("Переменная logicVar ложна")
8 }
```

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 7.8).

Листинг 7.8

```
1 // переменные типа Bool
2 var firstLogicVar = true
3 var secondLogicVar = false
4 // проверка значения переменных
5 if firstLogicVar || secondLogicVar {
6     print("Одна из переменных истинна")
7 } else {
8     print("Обе переменные ложны")
9 }
```

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 условия. 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 7.9 продемонстрирован пример использования расширенного синтаксиса логического оператора условия.

Листинг 7.9

```
1  // переменные типа Bool
2  var firstLogicVar = true
3  var secondLogicVar = true
4  // проверка значения переменных
5  if firstLogicVar && secondLogicVar {
6      print("Обе переменные истинны")
7  } else if firstLogicVar || secondLogicVar {
8      print("Одна из переменных истинна")
9  } else {
10     print("Обе переменные ложны")
11 }
```

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 7.10). Программа получает количество жильцов в доме и в зависимости от стоимости аренды на одного жильца возвращает общую сумму собираемых средств.

Листинг 7.10

```

1 // количество жильцов в доме
2 var tenantCount = 6
3 // стоимость аренды на человека
4 var rentPrice = 0
5 /* определение цены на одного
6 человека в соответствии с условием */
7 if tenantCount < 5 {
8     rentPrice = 1000
9 } else if tenantCount >= 5 && tenantCount <= 7 {
10     rentPrice = 800
11 } else {
12     rentPrice = 500
13 }
14 // вычисление общей суммы
15 var allPrice = rentPrice * tenantCount
```

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 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 7.11.

Листинг 7.11

```
1 // константы
2 let a = 1
3 let b = 2
4 // сравнение значений констант
5 a <= b ? print("А меньше или равно В"):print("А больше В")
```

Консоль:

А меньше или равно В

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 7.12).

Листинг 7.12

```
1 // переменная типа Int
2 var height = 180
3 // переменная типа Bool
4 var isHeader = true
5 // вычисление значения константы
6 let rowHeight = height + (isHeader ? 20 : 10 )
7 // вывод значения переменной
8 rowHeight
```

180

true

200

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 7.13). Приведенный код позволяет определить наличие положительных оценок, о чем говорит значение переменной `fiveMarkCount`. Если пятерок нет, на консоль выводится соответствующее сообщение. В ином случае вычисляется и выводится количество котлов для троллей, которые необходимо приобрести.

Листинг 7.13

```

1  /* переменная опционального типа
2  с предустановленным значением */
3  var fiveMarkCount: Int? = 8
4  // определение наличия значения
5  if fiveMarkCount == nil {
6      print(" Пятерки отсутствуют")
7  } else {
8      // количество пирожных за каждую пятерку
9      var cakeForEachFiveMark = 2
10     // общее количество пирожных
11     var allCakeCount = cakeForEachFiveMark * fiveMarkCount!
12 }
```

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

Обратите внимание на то, что при вычислении значения `allCakeCount` необходимо использовать принудительное извлечение опционального значения переменной `fiveMarkCount`.

ПРИМЕЧАНИЕ Данный способ проверки существования значения опционала работает исключительно при принудительном извлечении опционального значения, так как косвенно извлекаемое значение не может быть равно `nil`, а значит, и сравнивать его с `nil` не имеет смысла.

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` выполняется. 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 (его область видимости не ограничена). Если объект является локальным, то он доступен только в том блоке кода (и во всех вложенных в него блоках), для которого он является локальным. Вне этого блока объект просто не виден.

Рассмотрим пример опционального связывания (листинг 7.14).

Листинг 7.14

```

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

```

Консоль:

Всего 8 оценок

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

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

Обратите внимание на то, что переменная `marks`, созданная вне оператора условия, и одноименная переменная `marks`, 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 `marks` не затрагивается, о чем нам говорит вывод значения этой переменной в конце листинга. Если бы в переменной `markCount` не существовало значения, то управление перешло бы к блоку `else`.

Хорошим примером использования опционального связывания является уже известная нам функция `Int(_)`.

ПРИМЕЧАНИЕ Напомню, что функция `Int(_)` по возможности преобразует строковый литерал в число и возвращает опциональный тип `Int?`.

Вернемся к фэнтезийным примерам. Все любят драконов! Все драконы любят золото! Представьте, что у вас есть группа драконов. У большинства драконов есть свой сундук с золотом, и количество золотых монет в каждом из этих сундуков 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 7.15).

Листинг 7.15

```
1  /* переменная типа String,
2  содержащая количество золотых монет
3  в сундуке нового дракона */
4  var coinsInNewChest = "140"
5  /* переменная типа Int,
6  в которой будет храниться общее
7  количество монет у всех драконов */
8  var allCoinsCount = 1301
9  // проверяем существование значения
10 if Int(coinsInNewChest) != nil{
11     allCoinsCount += Int(coinsInNewChest)!
12 } else {
13     print("У нового дракона отсутствует золото")
14 }
```

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 7.16.

Листинг 7.16

```

1  /* переменная типа String,
2  содержащая количество золотых монет
3  в сундуке нового дракона */
4  var coinsInNewChest = "140"                                "140"
5  /* переменная типа Int,
6  в которой будет храниться общее
7  количество монет у всех драконов */
8  var allCoinsCount = 1301                                    1 301
9  /* извлекаем значение опционала
10 в новую переменную */
11 var coins = Int(coinsInNewChest)                             140
12 /* проверяем существование значения
13 с использованием созданной переменной */
14 if coins != nil{
15     allCoinsCount += coins!                                    1 441
16 } else {
17     print("У нового дракона отсутствует золото")
18 }
```

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 7.17).

Листинг 7.17

```

1  /* переменная типа String,
2  содержащая количество золотых монет
3  в сундуке нового дракона */
4  var coinsInNewChest = "140"                                "140"
5  /* переменная типа Int,
6  в которой будет храниться общее
7  количество монет у всех драконов */
8  var allCoinsCount = 1301                                    1 301
9  /* проверяем существование значения
10 с использованием опционального связывания */
11 if var coins = Int(coinsInNewChest){
12     allCoinsCount += coins                                    1 441
13 } else {
14     print("У нового дракона отсутствует золото")
15 }

```

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

7.3. 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.

7.4. 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 поэтому является плохой практикой программирования. Для решения этой 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 7.18). Данный код выводит на консоль определенный текст в зависимости от оценки, полученной учеником.

Листинг 7.18

```
1  /* переменная типа Int
2  содержит оценку, полученную
3  пользователем */
4  var userMark = 4
5  /* используем конструкцию if-else
6  для определения значения
7  переменной userMark и вывода
8  необходимого текста */
9  if userMark == 1 {
10     print("Единица на экзамене! Это ужасно!")
11 } else if userMark == 2 {
12     print("С двойкой ты останешься на второй год!")
13 } else if userMark == 3 {
14     print("Ты плохо учил материал в этом году!")
15 } else if userMark == 4 {
16     print("Неплохо, но могло быть и лучше")
17 } else if userMark == 5 {
18     print("Бесплатное место в университете тебе обеспечено!")
19 }
```

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 7.19).

Листинг 7.19

```
1  /* переменная типа Int
2  содержит оценку, полученную
3  пользователем */
4  var userMark = 4
5  /* используем конструкцию if-else
6  для определения значения
7  переменной userMark и вывода
8  необходимого текста */
9  switch userMark {
10     case 1:
11         print("Единица на экзамене! Это ужасно!")
```

```

12     case 2:
13         print("С двойкой ты останешься на второй год!")
14     case 3:
15         print("Ты плохо учил материал в этом году!")
16     case 4:
17         print("Неплохо, но могло быть и лучше")
18     case 5:
19         print("Бесплатное место в университете тебе обеспечено!")
20     default:
21         break
22 }

```

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 7.20).

Листинг 7.20

```

1  /* переменная типа Int
2  содержит оценку, полученную
3  пользователем */
4  var userMark = 4
5  // проверка значения с использованием диапазона
6  switch userMark {
7      case 1..<3:
8          print("Экзамен не сдан!")
9      case 3...5:
10         print("Экзамен сдан!")
11     default:
12         assert(false, "Оценка \(userMark) вне доступного диапазона")
13 }

```

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 7.21). 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 7.21

```
1  /* переменная типа Character
2  содержит уровень
3  готовности */
4  var level: Character = "Б"
5  // определение уровня готовности
6  switch level {
7      case "А":
8          print("Выключить все электрические приборы ")
9          fallthrough
10     case "Б":
11         print("Закрыть входные двери и окна ")
12         fallthrough
13     case "В":
14         print("Соблюдать спокойствие")
15     default:
16         break
17 }
```

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 7.22).

Листинг 7.22

```
1 /* переменная типа Float
2  содержит вес дракона */
3 var dragonWeight: Float = 2.4
4 /* переменная типа Float
5  содержит цвет дракона */
6 var dragonColor = "зеленый"
7 // определение загона для поступившего дракона
```

```
8  switch dragonColor {
9  case "зеленый" where dragonWeight < 2:
10 print("Поместите дракона в загон 1")
11     case "красный" where dragonWeight < 2:
12         print("Поместите дракона в загон 2")
13     case "зеленый", "красный" where dragonWeight >= 2:
14         print("Поместите дракона в загон 3")
15     default:
16         break
17 }
```

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 7.23).

Листинг 7.23

```
1  var userMark = 4
2  switch userMark {
3      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-case` 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 7.24).

Листинг 7.24

```
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 7.25.

Листинг 7.25

```
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")
14     default:
15         print("Дракон с неизвестными параметрами")
16 }
```

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 7.26).

Листинг 7.26

```
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 let(color, weight) where (color == "зеленый" || color ==
        "красный") && weight >= 2:
11         print("Поместите дракона в загон 3")
12     default:
13         print("Дракон с неизвестными параметрами")
14 }
```

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`.

8

Коллекции

Swift имеет механизмы, позволяющие выстроить в ряд произвольное количество элементов, пронумеровать их и работать с ними как с единым объектом. Такие механизмы (конструкции) называются коллекциями.

Коллекция — это множество элементов, позволяющее обратиться к каждому отдельному элементу этого множества.

НАЧИНАЮЩИМ Представьте строй солдат. Строй в данном контексте считается коллекцией, а каждый отдельный солдат — элементом коллекции. Вы можете вызвать каждый элемент по его порядковому номеру или значению (фамилии и имени солдата).

Swift предлагает три фундаментальных типа данных, обеспечивающих функциональность 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.

8.1. Массивы

Объявление массива

Массив — это упорядоченная коллекция однотипных элементов, для доступа к которым используются индексы (номера). 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 (не с единицы!). У массива, содержащего 5 элементов, индекс первого элемента равен 0, а последнего — 4. Индексы всегда последовательны и неразрывны.

Значение элемента — это произвольное значение 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 8.1.

Листинг 8.1

```
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 со-

держат 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 8.2).

Листинг 8.2

```

1 // неизменяемый массив
2 let immutableArray = ["one", "two", "three"] ["one", "two", "three"]
3 // копируем массив из константы в переменную
4 var newArray = immutableArray           ["one", "two", "three"]
5 newArray                                           ["one", "two", "three"]
6 // изменяем значение нового массива
7 newArray[1] = "four"                       "four"
8 // выводим значение исходного массива
9 immutableArray                               ["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, вы можете создать массив с помощью функции `Array(arrayLiteral:)`, или, точнее, инициализатор типа данных `Array`.

Синтаксис

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

Литерал массива передается в виде входных параметров функции `Array(arrayLiteral:)`.

Пример создания массива приведен в листинге 8.3.

Листинг 8.3

```

1 // создание массива с помощью передачи списка значений
2 let alphabetArray = Array(arrayLiteral: ["a", "b", "c"])
   "a", "b", "c")

```

В результате создается массив из указанных значений.

ПРИМЕЧАНИЕ Возможно, вы обратили внимание на то, что, по логике, мы передали значение входного аргумента `arrayLiteral` как «а» и еще два безымянных аргумента. На самом деле все это значение одного аргумента `arrayLiteral`. О том, как Swift позволяет выполнять такой прием, будет рассказано в одной из следующих глав.

При необходимости создания массива, состоящего из нескольких последовательных целочисленных значений, можно использовать оператор диапазона.

Синтаксис

Array(значение_1...значение_N)
Array(значение 1..<значение N)

В этом случае указываются начальное и конечное значения, разделяемые оператором диапазона.

Создадим массив, состоящий из 10 значений от 0 до 9, включая концы (листинг 8.4).

Листинг 8.4

```
1 // создание массива с помощью оператора диапазона  
2 let lineArray = Array(0...9)
```

[0, 1, 2, 3, 4,
5, 6, 7, 8, 9]

Третий способ создания массива используется в том случае, когда необходимо объявить массив, состоящий из нескольких одинаковых значений.

Синтаксис

Array(repeating: значение, count: количество)

Аргумент `repeating` определяет значение, которое будет присутствовать в массиве столько раз, сколько указано в качестве значения аргумента `count`.

Создадим массив, состоящий из 5 повторяющихся строковых значений (листинг 8.5)

Листинг 8.5

```
1 // создание массива с повторяющимися значениями
2 let repeatArray = Array(repeating: ["Swift",
  "Swift", "Swift",
  "Swift", "Swift"]
```

Тип данных массива

Тип данных массива всегда основывается на типе данных входящих в него элементов. Существует два способа написания типа данных массива.

Синтаксис

Полная форма записи:

```
Array<ТипДанных>
```

Краткая форма записи:

```
[ТипДанных]
```

Обе формы имеют один и тот же смысл: они указывают, что массив имеет тип данных [типДанных] или Array<типДанных> и состоит из элементов, тип данных которых — типДанных.

Рассмотрим пример из листинга 8.6.

Листинг 8.6

```
1 // Массив с типом данных [String] или Array<String>
2 Array(arrayLiteral: "a", "b", "c")
3 // Массив с типом данных [Int] или Array<Int>
4 Array(1..5)
```

Доступ к элементам массива

Для доступа к отдельному элементу массива необходимо использовать индекс данного элемента, заключенный в квадратные скобки и указанный сразу после имени массива (листинг 8.7).

Листинг 8.7

```
1 // неизменяемый массив
2 let alphabetArray = ["a", "b", "c"]           ["a", "b", "c"]
3 // изменяемый массив
4 var mutableArray = [2, 4, 8]                  [2, 4, 8]
5 // доступ к элементам массивов
6 alphabetArray[1]                             "b"
7 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 8.8).

Листинг 8.8

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

ПРИМЕЧАНИЕ 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 8.9).

Листинг 8.9

```
1 // изменяемый массив
2 var mutableArray = ["one", "two",          ["one", "two",
   "three", "four"]                        "three", "four"]
3 // копируем подмассив в отдельную переменную
4 var subArray = mutableArray[1...2]         ["two", "three"]
5 /* заменим несколько элементов
6    новым массивом */
7 mutableArray[1...2] = ["five"]             ["five"]
8 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 массива

Ранее мы рассмотрели вариант *неявного определения типа массива* и, соответственно, типа его элементов. Тип данных массива основывался на типе значений элементов, то есть автоматически определялся на

основе инициализируемого значения. При необходимости можно *явно указать тип данных массива*.

Для указания типа необходимо передать тип массива через двоеточие после его имени (по аналогии с определением типа переменной или константы).

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>. Напомню, что оба обозначения типа массива эквивалентны. Типом каждого отдельного элемента таких массивов 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 8.10). Подобный подход вы встречали при создании переменных и констант фундаментальных типов данных.

Листинг 8.10

```
1  /* одним выражением объявляем массив,
2  явно указываем тип его элементов
3  и инициализируем его значение */
4  var firstArray: [String] = ["x", "y", "z"]    ["x", "y", "z"]
5  // объявляем массив без элементов
6  var secondArray: Array<Int>
7  // инициализируем его значение
8  secondArray = [1, 2, 3, 4, 5]                [1, 2, 3, 4, 5]
9  // выводим значения обоих массивов
10 firstArray                                  ["x", "y", "z"]
11 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 8.11.

Листинг 8.11

```
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.

Конструкция `[ТипДанных]()` также позволяет создать массив, состоящий из определенного числа одинаковых значений. Для этого в качестве входных параметров необходимо передать параметр `repeating`, указывающий на значение элементов, и `count`, указывающий на количество элементов (листинг 8.12).

Листинг 8.12

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

Тип значения параметра `repeating` должен совпадать с типом элементов массива, указанным в квадратных скобках.

Сравнение массивов

Массивы, так же как и значения фундаментальных типов данных, можно сравнивать друг с другом. Два массива являются эквивалентными:

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

Рассмотрим пример сравнения двух массивов (листинг 8.13).

Листинг 8.13

```
1  /* три константы, которые
2  станут элементами массива */
3  let a1 = 1
4  let a2 = 2
5  let a3 = 3
6  if [1, 2, 3] == [a1, a2, a3] {
7      print("Массивы эквивалентны")
8  } else {
9      print("Массивы не эквивалентны")
10 }
```

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.

Слияние массивов

Со значением массива, как и со значениями фундаментальных типов данных, можно проводить различные операции. Одной из них является операция слияния, при которой значения двух массивов сливаются в одно, образуя новый массив. Обратите внимание на несколько моментов:

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

Операция слияния производится с помощью уже известного оператора сложения (+), как показано в листинге 8.14.

Листинг 8.14

```

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"]
9  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 8.15.

Листинг 8.15

```
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 8.16).

Листинг 8.16

```

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

```

Конструкция `arrayOfArrays[2]` возвращает третий вложенный элемент массива `arrayOfArrays`, а `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 8.17).

Листинг 8.17

```
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 8.18).

Листинг 8.18

```
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 8.19).

Листинг 8.19

```
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 8.20).

Листинг 8.20

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

Свойства `first` и `last` возвращают первый и последний элементы массива (листинг 8.21).

Листинг 8.21

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

С помощью метода `append(_:)` можно добавить новый элемент в конец массива (листинг 8.22).

Листинг 8.22

```
1 var someArray = [1, 2, 3, 4, 5]           [1, 2, 3, 4, 5]
2 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 8.23).

Листинг 8.23

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

При этом индексы массива пересчитываются, чтобы обеспечить их последовательность.

Так же как в случае изменения массива, методы `remove(at:)`, `removeFirst()` и `removeLast()` позволяют удалять требуемые элементы. При этом они возвращают значение удаляемого элемента (листинг 8.24).

Листинг 8.24

```
1 var someArray = [1, 2, 3, 4, 5]           [1, 2, 3, 4, 5]
2 // удаляем третий элемент массива (с индексом 2)
3 someArray.remove(at: 2)                   3
4 // удаляем первый элемент массива
5 someArray.removeFirst()                   1
6 // удаляем последний элемент массива
7 someArray.removeLast()                   5
8 /* итоговый массив содержит
9 всего два элемента */
10 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 входного аргумента ничего не передавать, то из результата удаляется один элемент. В противном случае — столько элементов, сколько передано (листинг 8.25).

Листинг 8.25

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

При использовании данных методов основной массив `someArray`, с которым выполняются операции, не меняется (у нас `let`, а не `var`). Они лишь возвращают получившееся значение, которое при необходимости может быть записано в новый параметр.

Метод `contains(_:)` определяет факт наличия некоторого элемента в массиве и возвращает `Bool` в зависимости от результата (листинг 8.26).

Листинг 8.26

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

Метод `index(of:)` сообщает индекс первого вхождения искомого элемента в рассматриваемом массиве. Так как искомый элемент может отсутствовать, метод возвращает опциональное значение (листинг 8.27). Если элемент отсутствует, то возвращается `nil`.

Листинг 8.27

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

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

Для поиска минимального или максимального элемента в массиве применяются методы `minElement()` и `maxElement()`. Данные методы работают только в том случае, если элементы массива можно сравнить между собой (листинг 8.28).

Листинг 8.28

```
1 let intArray = [3, 2, 4, 5, 6, 4, 7, 5, 6]      [3, 2, 4, 5,
                                                6, 4, 7, 5, 6]
2 // поиск минимального элемента
3 intArray.minElement()                        2
4 // поиск максимального элемента
5 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 встроенным в 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 8.29.

Листинг 8.29

```
1 var someArray = [1, 3, 5, 7, 9]              [1, 3, 5, 7, 9]
2 someArray.reverse()                         [9, 7, 5, 3, 1]
```

8.2. 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 8.30 продемонстрированы все возможные способы создания наборов.

Листинг 8.30

```

1  /* набор, созданный путем
2  явного указания типа */
3  var dishes: Set<String> = ["хлеб", "овощи",      {"овощи", "тушенка",
                                                    "вода", "хлеб"}
                                тушенка", "вода"]
4  /* набор, созданный без явного
5  указания типа данных */
6  var dishesTwo: Set = ["хлеб", "овощи",          {"овощи", "тушенка",
                                                    "вода", "хлеб"}
                        "тушенка", "вода"]
7  /* набор, созданный с помощью
8  функции при явном
9  указании типа данных*/
10 var members = Set<String>(arrayLiteral:        {"Оби Ван",
                                                    "Энекин", "Оби Ван", "Йода"})
11 /* набор, созданный с помощью
12 функции без явного указания
13 типа данных */
14 var membersTwo = Set(arrayLiteral:             {"Оби Ван",
                                                    "Энекин", "Оби Ван", "Йода"})

```

В переменных `members`, `membersTwo`, `dishes`, `dishesTwo` хранятся наборы уникальных значений. При этом в области вывода порядок значений может не совпадать с тем порядком, который существует в литерале. Это связано с тем, что наборы — неупорядоченные коллекции элементов.

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 8.31).

Листинг 8.31

```

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 8.32).

Листинг 8.32

```
1 // создаем пустой набор
2 var musicStyleSet: Set<String> = []           Set([])
3 // добавляем к нему новый элемент
4 musicStyleSet.insert("Jazz")                  (.0 true, .1 "Jazz")
5 musicStyleSet                                "Jazz"
```

В результате выполнения метода `insert(_)` возвращается массив, первый элемент которого содержит значение типа `Bool`, характеризующее успешность проведенной операции, и значение добавляемого элемента. В качестве первого элемента массива может быть возвращено `false` в том случае, когда производилась попытка добавить уже существующий в наборе элемент.

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 8.33).

Листинг 8.33

```
1 // создание набора со значениями
2 var musicStyleSet: Set<String> = ["Jazz",      {"Jazz", "Hip-Hop",
        "Hip-Hop", "Rock"}]                  "Rock"}
3 // удаляем один из элементов
4 musicStyleSet.remove("Hip-Hop")              "Hip-Hop"
5 musicStyleSet                                {"Jazz", "Rock"}
6 // удаляем несуществующий элемент
7 musicStyleSet.remove("Classic")              nil
8 // удаляем все элементы набора
9 musicStyleSet.removeAll()                    Set([])
```

Проверка факта наличия значения в наборе осуществляется методом `contains(_)`. Данный метод возвращает значение типа `Bool` в зависимости от результата проверки (листинг 8.34).

Листинг 8.34

```
1 // создаем набор
2 var musicStyleSet: Set<String> = ["Jazz", "Hip-Hop",
   "Rock", "Funk"]
3 // проверка существования значения в наборе
4 if musicStyleSet.contains("Funk") {
5     print("ты любишь хорошую музыку")
6 } else {
7     print("послушай то, что я слушаю")
8 }
```

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 8.35 создаются три различных целочисленных набора (рис. 8.1). Один из наборов содержит четные числа, второй — нечетные, третий — те и другие.

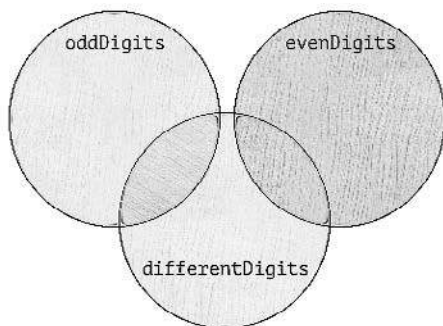


Рис. 8.1. Три набора целочисленных значений

Листинг 8.35

```
1 // набор с четными цифрами
2 let evenDigits: Set = [0, 2, 4, 6, 8]
3 // набор с нечетными цифрами
```

```

4 let oddDigits: Set = [1, 3, 5, 7, 9]
5 // набор со смешанными цифрами
6 var differentDigits: Set = [3, 4, 7, 8]

```

В наборах существуют как уникальные, так и общие элементы.

Для двух любых наборов можно произвести следующие операции (рис. 8.2):

- ❑ получить все общие для обоих наборов элементы (`intersection`);
- ❑ получить все непересекающиеся (не общие) для обоих наборов элементы (`symmetricDifference`);
- ❑ получить все элементы обоих наборов (`union`);
- ❑ получить разницу элементов, то есть элементы, которые входят в первый набор, но не входят во второй (`subtracting`).

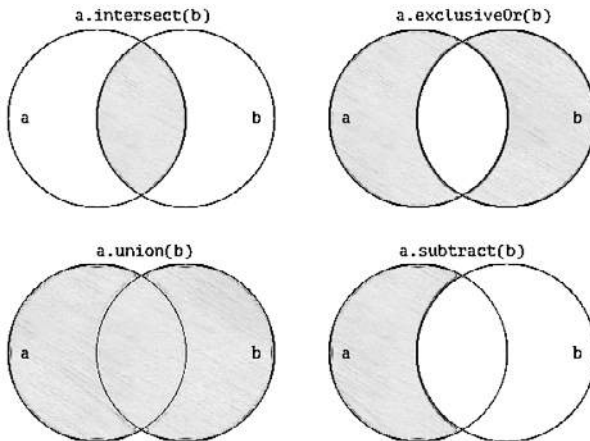


Рис. 8.2. Операции, проводимые с наборами

При использовании метода `intersection(_)` создается новый набор, содержащий значения, общие для двух наборов (листинг 8.36).

Листинг 8.36

```

1 var inter = differentDigits.intersection(oddDigits).      [3, 7]
  sorted()

```

Обратите внимание, что в данном примере впервые в одном выражении использована цепочка вызовов методов `intersection(_)` и `sorted()`.

Цепочка вызовов — очень полезный функциональный механизм Swift. С ее помощью можно не записывать возвращаемое значение функции в переменную для ее дальнейшей обработки новой функцией, а сразу же применять эту функцию в одном выражении с предыдущей.

В результате вызова метода `sorted()` 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 `symmetricDifference(_:)`, представленный в листинге 8.37.

Листинг 8.37

```
1 var exclusive = differentDigits.  
  symmetricDifference(oddDigits).sorted()           [1, 4, 5, 8, 9]
```

Для получения всех элементов из обоих наборов применяется объединяющий метод `union(_:)`, как показано в листинге 8.38.

Листинг 8.38

```
1 var union = evenDigits.union(oddDigits).sorted()   [0, 1, 2, 3, 4,  
                                                    5, 6, 7, 8, 9]
```

Метод `subtracting(_:)` возвращает все элементы первого множества, которые не входят во второе множество (листинг 8.39).

Листинг 8.39

```
1 var subtract = var subtract = differentDigits.  
  subtracting(evenDigits).sorted()                 [3, 7]
```

Эквивалентность наборов

На рис. 8.3 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 поднабор (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 8.40.

Листинг 8.40

```
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 }
```

Консоль:

Наборы эквивалентны

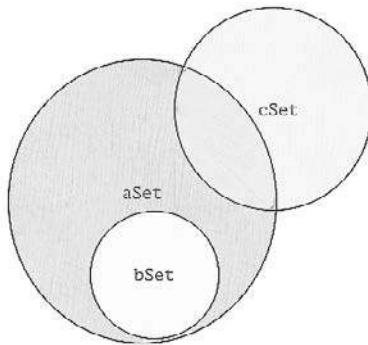


Рис. 8.3. Три набора значений с различными отношениями друг с другом

Самое важное, чтобы при создании набора вы не забывали использовать ключевое слово `Set`, иначе в результате будет создан массив. Метод `isSubset(of:)` определяет, является ли один набор поднабором другого, как bSet для aSet (листинг 8.41).

Листинг 8.41

```
1 var aSet: Set = [1, 2, 3, 4, 5]
2 var bSet: Set = [1, 3]
3 if bSet.isSubset(of: aSet) {
4     print("bSet – это поднабор для aSet")
5 }
```

Консоль:

bSet – это поднабор для aSet

Метод `isSuperset(of:)` вычисляет, является ли набор супернабором для другого набора (листинг 8.42).

Листинг 8.42

```
1 var aSet: Set = [1, 2, 3, 4, 5]
2 var bSet: Set = [1, 3]
3 if aSet.isSuperset(of: bSet) {
4     print("aSet – это супернабор для bSet")
5 }
```

Консоль:

aSet – это супернабор для bSet

Метод `isDisjoint(with:)` определяет, существуют ли в двух наборах общие элементы, и в случае их отсутствия возвращает `true` (листинг 8.43).

Листинг 8.43

```
1 var bSet: Set = [1, 3]
2 var cSet: Set = [6, 7, 8, 9]
3 if bSet.isDisjoint(with: cSet) {
4     print("наборы bSet и cSet не пересекаются")
5 }
```

Консоль:

наборы bSet и cSet не пересекаются

Методы `isStrictSubset(of:)` и `isStrictSuperset(of:)` определяют, является набор поднабором или супернабором, не равным указанному множеству (листинг 8.44).

Листинг 8.44

```
1 var aSet: Set = [1, 2, 3, 4, 5]
2 var bSet: Set = [1, 3]
3 if bSet.isStrictSubset(of: aSet) {
4     print("bSet – поднабор для aSet")
5 }
```

```

5 }
6 if aSet.isStrictSuperset(of: bSet) {
7     print("aSet — супернабор для bSet")
8 }

```

Консоль:

bSet — поднабор для aSet

aSet — супернабор для bSet

8.3. 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 8.45.

Листинг 8.45

```
1 var dictionary = ["one": "один",      ["one": "один", "three": "три",
   "two": "два", "three": "три"]      "two": "два"]
```

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 8.46).

Листинг 8.46

```
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 устанавливаемое 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(_:forKey:)`. Как показано в листинге 8.47, при установке нового значения данный метод возвращает опциональное старое значение (`nil`, если значения по изменяемому ключу не существует).

Листинг 8.47

```
1 var countryDict = ["RUS": "Россия", "BEL":      ["BEL": "Беларусь",
   Беларусь", "UKR": "Украина"] "UKR": "Украина",
                                   "RUS": "Россия"]
```

```

2 // изменяем значение элемента
3 countryDict.updateValue("Российская Федерация", forKey: "RUS")
4 countryDict

```

"Россия"

*["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 8.48).

Листинг 8.48

```

1 var countryDict = ["RUS": "Россия",
2 "BEL": "Беларусь", "UKR": "Украина"]
3
4 // создание нового элемента словаря
5 countryDict["TUR"] = "Турция"
6 countryDict

```

*["BEL": "Беларусь",
"UKR": "Украина",
"RUS": "Россия"]*

"Турция"

*["BEL": "Беларусь", "UKR":
"Украина", "RUS":
"Россия", "TUR": "Турция"]*

Для удаления некоторого элемента (пары «ключ-значение») достаточно присвоить удаляемому элементу `nil` или использовать метод `removeValue(forKey:)`, указав ключ элемента (листинг 8.49).

Листинг 8.49

```

1 var countryDict = ["RUS": "Россия",
2 "BEL": "Беларусь", "UKR": "Украина"]
3 // удаление элемента словаря
4 countryDict["UKR"] = nil
5 countryDict.removeValue(forKey: "BEL")
6 countryDict

```

*["BEL": "Беларусь", "UKR":
"Украина", "RUS": "Россия"]*

nil

"Беларусь"

["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 8.50).

Листинг 8.50

```
1 var countryDict = ["RUS": "Российская Федерация", "BEL": "Беларусь",  
                    "UKR": "Украина"]  
2 // получим значение элемента  
3 let myCountry: String = countryDict["RUS"]! "Российская Федерация"
```

```
[ "BEL": "Беларусь",  
  "UKR": "Украина",  
  "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 8.51).

Листинг 8.51

```
1 var emptyDictionary: [String: Int] = [ : ]  
2 var AnotherEmptyDictionary: Dictionary<String, Int> = [ : ]
```

С помощью конструкции `[ : ]` можно также уничтожить все элементы словаря, если присвоить ее словарю в качестве значения (листинг 8.52).

Листинг 8.52

```
1 var countryDict = ["RUS": "Российская Федерация", "BEL": "Беларусь", "UKR": "Украина"]
2 countryDict = [ :]
3 countryDict
```

["BEL": "Беларусь", "UKR": "Украина", "RUS": "Российская Федерация"]
[:]
[:]

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 8.53).

Листинг 8.53

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

[1, 2, 3, 4, 5]
5

Если свойство `count` равно нулю, то свойство `isEmpty` возвращает `true` (листинг 8.54).

Листинг 8.54

```
1 var someDictionary: [String: Int] = [ :]
2 someDictionary.count
3 someDictionary.isEmpty
```

[:]
0
true

При необходимости вы можете получить все ключи или все значения словаря с помощью свойств `keys` и `values` (листинг 8.55).

Листинг 8.55

```
1 var countryDict = ["RUS": "Российская Федерация", "BEL": "Беларусь", "UKR": "Украина"]
```

["BEL": "Беларусь", "UKR": "Украина", "RUS": "Российская Федерация"]

```
2 // все ключи словаря countryDict
3 var keys = countryDict.keys
                                     LazyMapCollection
                                     <Dictionary<String,
                                     String>, String>

4 // все значения словаря countryDict
5 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`, которое представляет собой ленивую коллекцию. О том, что это такое, мы поговорим в одной из следующих глав.

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.

9

Циклы

Программисты всегда стараются писать код, который будет выполнять как можно больше задач и вычислять как можно больше выражений. При этом самого кода должно быть как можно меньше. Программы с небольшим количеством кода проще отлаживать и поддерживать, да и разрабатывать — тоже.

Ранее мы рассмотрели операторы, позволяющие выполнять различные участки кода в зависимости от условий, возникших в ходе работы программы. В данной главе описываются механизмы, позволяющие циклично (многократно) выполнять различные блоки кода и управлять данными циклами. Другими словами, основная идея циклов состоит в том, чтобы программа многократно выполняла одни и те же выражения, не требуя от программиста много раз писать код этих выражений.

Циклическое 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 определенное количество раз, либо пока какое-то условие не выполнится. Каждое следующее выполнение блока кода называется *итерацией*.

9.1. Оператор повторения `for`

Оператор `for` предназначен для циклического выполнения блоков кода. Swift предлагает одну форму использования данного оператора: `for-in`.

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 9.1. Данный код складывает все числа от 1 до 10 и выводит итоговый результат в области результатов.

Листинг 9.1

```
1  var totalSum = 0                                0
2  for i in 1...10 {
3      totalSum += i                                (10 times)
4  }
5  totalSum                                         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 т. д.

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 9.2).

Листинг 9.2

```
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. При попытке указать диапазон от большего к меньшему, например (10...1), Xcode сообщит об ошибке. Для этого используйте метод `reversed()` (листинг 9.3).

Листинг 9.3

```
1 var totalSum = 0
2 for i in (1...10).reversed() {
3     totalSum += i
4 }
5 totalSum
```

0
(10 times)
55

ВНИМАНИЕ Ранее, при изучении коллекций, мы обращались к методу `reverse()`, который изменяет порядок произвольной коллекции на обратный. В отличие от него метод `reversed()`, с окончанием `-ed`, не изменяет исходную последовательность, а возвращает очередной элемент, начиная с конца коллекции.

Если необходимо пройти определенный числовой интервал с некоторым шагом, можно использовать два одноименных метода: `stride(from:through:by:)` и `stride(from:to:by:)`. Рассмотрим каждый из них (листинг 9.4).

Листинг 9.4

```
1 for i in stride(from:1, through:5, by:2) {
2     print(i)
3 }
4 for i in stride(from:1, to:5, by:2) {
5     print(i)
6 }
```

Консоль:

```
1
3
5
1
3
```

Как видно из вывода на консоли, аргумент `to` указывает начальное значение диапазона, `by` — шаг прохода по диапазону, `through` и `to` — конечное значение диапазона. При использовании `to` указанное значение исключается (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 9.5).

Листинг 9.5

```
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 9.6).

Листинг 9.6

```
1 var countriesAndBlocks = ["Россия": "ЕАЭС", "США": "НАТО",
2     "Франция": "ЕС"]
3 for (countryName, blockName) in countriesAndBlocks {
4     print("\(countryName) вступила в \(blockName)")
5 }
```

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 9.7).

Листинг 9.7

```
1 var countriesAndBlocks = ["Россия": "ЕАЭС", "США": "НАТО",
2     "Франция": "ЕС"]
3 for (countryName, _) in countriesAndBlocks {
4     print("страна – \(countryName)")
5 }
```

Консоль:

страна — Россия
 страна — Франция
 страна — США

С данным приемом вы познакомились еще во время изучения кортежей.

В Swift существует специальный метод `enumerated()`, который позволяет преобразовать массив таким образом, чтобы с помощью цикла `for-in` получить в виде кортежа каждую отдельную пару «индекс-значение» (листинг 9.8).

Листинг 9.8

```
1 var myMusicStyles = ["Rock", "Jazz", "Pop"]
2 for (index, musicName) in myMusicStyles.enumerated() {
3     print("\(index+1). Я люблю \(musicName)")
4 }
```

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 9.9 значение типа `String` представляется в виде коллекции символов, и элементы этой коллекции по одному выводятся на консоль.

Листинг 9.9

```
1 let myName = "Troll"
2 for oneChar in myName.characters {
3     print(oneChar)
4 }
```

Консоль:

T
 r
 o
 l
 l

Для обработки многомерных конструкций вроде вложенных коллекций вы можете вкладывать одни циклы в другие. В листинге 9.10

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 9.10

```
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` — локальная, но в ее область видимости попадает вложенный цикл `for-in`, поэтому в данном цикле можно использовать эту переменную для вывода ее значения.

ПРИМЕЧАНИЕ В Swift 2.2 (а также в более ранних версиях языка) существовали две формы оператора повторения: `for` и `for-in`.

Классическая форма `for` имела следующий синтаксис:

```
for стартовое_выражение; условие_окончания; действие {
    блок_кода
}
```

Если вы занимались программированием ранее, то наверняка знакомы с ней. Тем не менее в Swift 3 разработчики Apple полностью удалили данную форму цикла.

9.2. 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 т. д.

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 9.11, в котором складываются все числа от 1 до 10 и выводится результат.

Листинг 9.11

```
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 9.12, в котором складываются все числа от 1 до 10 и выводится результат.

Листинг 9.12

```
1  // начальное значение  
2  var i = 1  
3  // хранилище результата сложения  
4  var resultSum = 0  
5  // цикл для подсчета суммы  
6  repeat{  
7      resultSum += i  
8      i++  
9  } while i <= 10  
10 resultSum
```

1
0

(10 times)
(10 times)

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.

9.3. 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 9.13 представлена программа, в которой переменная поочередно принимает значения от 1 до 10, причем когда значение нечетное, оно выводится на консоль.

Листинг 9.13

```
1 for i in 1...10 {
2     if i%2 == 0 {
3         continue
4     } else {
5         print(i)
6     }
7 }
```

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 9.14 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 9.14

```
1 import Foundation
2 for i in 1...10 {
3     var randNum = Int(arc4random_uniform(10))
4     if randNum == 5 {
5         print("Итерация номер \(i)")
6         break
7     }
8 }
```

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 одной из следующих глав.

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 9.15).

Листинг 9.15

```
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.
2. Тип данных словаря должен быть `[String:[String:UInt]]`.
3. В вашем электронном журнале должно находиться по две оценки для каждого из трех учеников. Фамилии, даты занятий и оценки придумайте сами.
4. 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 8, в котором вы описывали шахматную фигуру и создавали конструкции `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`.

10

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. Функция при обращении к ней может принимать входные данные, обрабатывать их и возвращать получившееся значение. 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.

10.1. Объявление функций

Функции помогают структурировать код, объединяя его блоки для многократного использования. Функции позволяют следовать принципу «не писать один и тот же код дважды».

ПРИМЕЧАНИЕ У программистов существует шутка, гласящая, 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 (также называемые аргументами), 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 и должно быть записано в нижнем камелкейс-стиле. 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 позволяют передать в функцию значения, которые ей требуются для корректного выполнения возложенных на нее задач. Указанные параметры являются локальными для тела функции, то есть они доступны только в теле функции. Количество входных параметров может быть произвольным (также они могут вовсе отсутствовать). Например:

```
func myFirstFunc
(someValue: Int, anotherValue: String)
```

Аргументы `someValue` и `anotherValue` будут существовать только в пределах тела функции. По окончании ее работы данные параметры будут уничтожены и станут недоступными.

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` работа функции завершается и происходит выход из нее. Например:

```
func myFirstFunc
(someValue: Int, anotherValue: String)
-> String {
  return String(someValue) + anotherValue
}
```

В данном случае тело функции состоит всего из одного выражения, в котором содержится оператор `return`. После его выполнения функция завершится и вернет сформированное значение.

В представленных ранее примерах объявление функции разнесено на разные строки для удобства восприятия кода. Вам не обязательно делать это, можете писать элементы объявления функции в одну строку. Например:

```
func myFirstFunc (someValue: Int) -> String {  
    return String(someValue)  
}
```

Перейдем к практическому примеру. 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 10.1).

Листинг 10.1

```
1 func printMessage() -> Void {  
2     print("Сообщение принято")  
3 }  
4 // вызываем функцию по ее имени  
5 printMessage()
```

Консоль:

Сообщение принято

Процесс обращения к объявленной функции с целью получения результата ее работы называется *вызовом функции*. 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.

ПРИМЕЧАНИЕ Вывод информации на консоль, если он присутствует, не является возвращаемым значением. Это всего лишь выполняемая в ходе работы функции операция.

Если функция не возвращает никакого значения (как в примере ранее), то в качестве типа данных возвращаемого значения необходимо указать тип `Void`. Подобный C-образный подход используется во многих языках программирования.

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")
}
```

10.2. Аргументы функции и возвращаемое значение

Функция может принимать аргументы в качестве входных значений и возвращать результат своей работы в виде возвращаемого значения. И у входных, и у возвращаемого значений должны быть определены типы данных.

Предположим, что требуется разработать функцию, которая складывает два значения, потом делит их на число π (3,14) и возвращает результат. В такой функции входными параметрами будут два неизвестных заранее числа. Не имеет никакого смысла передавать в качестве входного аргумента и число π , так как оно известно заранее и его можно жестко зафиксировать в теле функции. Результат выполненных вычислений как раз и будет возвращаемым значением функции.

Это был теоретический пример, теперь перейдем к более простым практическим примерам работы с функциями.

Простые входные и выходные параметры

Напишем функцию, которая имеет один входной параметр, используемый в ее теле (листинг 10.2). В качестве аргумента функция принимает код ответа от сервера и выводит на консоль строку, содержащую вспомогательную информацию. Возвращаемое значение отсутствует.

Листинг 10.2

```
1 func printCodeMessage(requestCode: Int) -> () {
2     print("Код ответа - \(requestCode)")
3 }
```

```
4 // вызываем функцию
5 printCodeMessage(requestCode: 200)
6 printCodeMessage(requestCode: 404)
```

Консоль:

Код ответа — 200

Код ответа — 404

Функция `printCodeMessage(requestCode:)` 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 данного значения эти параметры необходимо передавать в функцию при ее вызове, указывая поочередно имя каждого входного параметра и его значение, разделив их двоеточием. Аргументы передаются в том же порядке, в каком описывались при объявлении функции.

При вызове функции необходимо указывать имя для каждого параметра при передаче его значения. В листинге 10.3 мы реализуем функцию, которая принимает на вход три значения, складывает их и выводит результат на консоль.

Листинг 10.3

```
1 func sum(a:Int, b: Int, c:Int) -> () {
2     print("Сумма — \(a+b+c)")
3 }
4 sum(a: 10, b: 51, c: 92)
```

Консоль:

Сумма — 153

При вызове функции `sum(a:b:c:)` указываются имена и значения для всех входных параметров. При этом значения для параметров передаются точно в том же порядке, в каком были описаны параметры при определении функции.

ПРИМЕЧАНИЕ Имена каких параметров необходимо указывать, вам подскажет Xcode, а точнее — механизм автозавершения кода. С ним мы познакомились в первой части книги.

Существует способ сообщить Swift, что для определенных параметров нет необходимости указывать имена при функции. Для этого перед именем таких параметров необходимо поставить символ нижнего подчеркивания, отделив его от имени пробелом (листинг 10.4).

Листинг 10.4

```

1 func sum(a:Int, _ b: Int, c:Int) -> () {
2     print("Сумма - \(a+b+c)")
3 }
4 sum(a: 15, 12, c: 9)

```

Консоль:

Сумма - 36

Как видите, в функции `sum(a:_:c:)` перед аргументом `b` стоит символ нижнего подчеркивания и при вызове функции указываются только имена параметров `a` и `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 10.5).

Листинг 10.5

```

1 func generateString(code: Int, _ text: String) -> String {
2     var mutableText = text
3     mutableText += String(code)
4     return mutableText
5 }
6 generateString(code: 200, "Код:")                                     "Код:200"

```

Функция `generateString(code:_:)` 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 входного параметра. В Swift 3 данная возможность удалена.

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 описанием его типа необходимо указать модификатор `inout`. Сквозной параметр передается в функцию, изменяется в ней и сохраняет свое значение при завершении работы функции, заменяя собой исходное значение. При вызове функции перед передаваемым значением аргумента необходимо ставить символ амперсанда (&), указывающий на то, что параметр передается по ссылке. Функция в листинге 10.6 получает на входе два параметра и меняет местами их значения.

Листинг 10.6

```
1 func changeValues(_ a: inout Int, _ b: inout 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
```

Функция принимает на входе две переменные, `a` и `b`. Эти переменные передаются в функцию как сквозные параметры, что позволяет изменить их значения внутри функции и сохранить эти изменения после завершения ее работы.

ПРИМЕЧАНИЕ 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 функцию, возвращающую значение этого типа (листинг 10.7).

Листинг 10.7

```
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(code: 200, message: "Сервер доступен"))
```

Консоль:

Получено сообщение "Сервер доступен" с кодом 200

Функция `generateString(code: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 и принимает на входе, то можно указать в качестве входного аргумента функции `print(_:)` функцию `generateString(code:message:)`.

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 10.8. 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 10.8

```
1 func printRequestString(codes: Int...) -> () {
2     var codesString = ""
3     for oneCode in codes {
4         codesString += String(oneCode) + " "
5     }
6     print("Получены ответы - \(codesString)")
7 }
8 printRequestString(codes: 600, 800, 301)
9 printRequestString(codes: 101, 200)
```

Консоль:

Получены ответы - 600 800 301

Получены ответы - 101 200

Параметр `codes` может содержать произвольное количество значений указанного типа. Внутри функции он трактуется как 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 10.9 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 10.9

```
1 func getCodeDescription(code: Int) -> (Int, String){
2     let description: String
3     switch code {
4         case 1...100:
5             description = "Error"
6         case 101...200:
7             description = "Correct"
8         default:
9             description = "Unknown"
10    }
11    return (code, description)
12 }
13 print(getCodeDescription(code: 150))
```

Консоль:

```
(150, "Correct")
```

В качестве типа возвращаемого значения функции `getCodeDescription(code:)` указан тип кортежа, содержащего два значения: код и его описание.

Функцию `getCodeDescription(code:)` можно улучшить, если указать не просто тип возвращаемого кортежа, а названия его элементов (листинг 10.10).

Листинг 10.10

```
1 func getCodeDescription(code: Int)
2 -> (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(code: 48)      (.0 45, .1 "Error")
15 request.description                            "Error"
16 request.code                                 45

```

Полученное в ходе работы функции `getCodeDescription(code:)` 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 10.11).

Листинг 10.11

```

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,
    5000, 50, 100]                                50, 1 000, 5 000,
                                                    50, 100]

10 // сосчитаем сумму всех купюр
11 sumWallet(wallet: 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: wallet)                        7 950

```

В любой момент вы можете вызвать функцию `sumWallet(wallet:)` 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 10.12).

Листинг 10.12

```

1  func sumWallet( wallet: [Int]? = nil ) -> Int? {
2      var sum = 0
3      if wallet == nil {
4          return nil
5      }
6      for oneBanknote in wallet! {
7          sum += oneBanknote
8      }
9      return sum
10 }
11 // кошелек с купюрами
12 var wallet = [50, 100, 100, 500, 50,           [50, 100, 100, 500,
13     1000, 5000, 50, 100]                        50, 1 000, 5 000,
14                                                     50, 100]
15
13 // считаем сумму всех купюр
14 sumWallet(wallet: wallet)           6 950
15 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 10.13).

Листинг 10.13

```
1 func sumWallet(banknotesArray wallet: [Int]? = nil ) -> Int? {
2     var sum = 0
3     if wallet == nil {
4         return nil
5     }
6     for oneBanknote in wallet! {
7         sum += oneBanknote
8     }
9     return sum
10 }
11 // сосчитаем сумму всех купюр
12 sumWallet(banknotesArray: [50, 100, 100, 500, 50,
                                1000, 5000, 50, 100])
```

6 950

Входной параметр `wallet` теперь имеет внешнее имя `banknotesArray`, поэтому при вызове функции `sumWallet(banknotesArray:)` 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.

10.3. Возможности функций

Функциональный тип

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 двумя парами круглых скобок, разделенных стрелкой:

`() -> ()`

В первых скобках всегда описываются типы данных входных параметров, а вместо вторых указывается тип данных выходного значения, если, конечно, оно существует. Если функция принимает на входе массив целочисленных значений, а возвращает опциональный кортеж из двух элементов типа `String`, то ее тип данных будет выглядеть следующим образом:

```
([Int]) -> (String,String)?
```

Напомню, что в левой части конструкции указываются типы входных параметров, в правой — тип выходного значения.

Функция в качестве аргумента

Возможности входных параметров не ограничиваются передачей им значений фундаментальных типов, массивов и кортежей. Ранее мы 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 она передана. Для передачи функции в качестве значения необходимо указать функциональный тип принимаемой функции в качестве типа входного параметра.

Напишем новую функцию `generateWallet(walletLength:)`, которая случайным образом генерирует массив банкнот. Она должна принимать на входе требуемое количество банкнот в кошельке. Также мы перепишем функцию `sumWallet(wallet:)` таким образом, чтобы она принимала на входе не массив значений, а функцию `generateWallet(walletLength:)` и использовала данную функцию внутри своего тела для генерации массива банкнот, после чего подсчитывала их общую сумму (листинг 10.14).

Листинг 10.14

```
1 import Foundation
2 // функция генерации случайного массива банкнот
3 func generateWallet(walletLength: Int)
4 -> [Int] {
5     // существующие типы банкнот
6     let typesOfBanknotes = [50, 100, 500, 1000, 5000]
7     // массив банкнот
8     var wallet: [Int] = []
9     // цикл генерации массива случайных банкнот
```

```

10     for _ in 1...walletLength {
11         let randomIndex = Int( arc4random_uniform(
12             UInt32( typesOfBanknotes.count-1 ) ) )
13         wallet.append( typesOfBanknotes[randomIndex] )
14     }
15     return wallet
16 }
17 // функция подсчета денег в кошельке
18 func sumWallet(banknotesFunction wallet: (Int)->([Int]) )
19 -> Int? {
20     // вызов переданной функции
21     let myWalletArray = wallet( Int( arc4random_uniform(10) ) )
22     var sum: Int = 0
23     for oneBanknote in myWalletArray {
24         sum += oneBanknote
25     }
26     return sum
27 }
28 // передача функции в функцию
29 sumWallet(banknotesFunction: generateWallet)

```

2 700

Значение в области результатов, вероятно, будет отличаться от того, что показано в примере. Это связано с использованием глобальной функции `arc4random_uniform()`, генерирующей и возвращающей случайное число.

Функция `generateWallet(walletLength:)` создает массив купюр такой длины, которая передана ей в качестве входного параметра. В массиве `typesOfBanknotes` содержатся все 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 функции `generateWallet(walletLength:)`. 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 расходимые ресурсы компьютера. В качестве типа входного параметра `banknotesFunction` функции `sumWallet(banknotesFunction:)` указан функциональный тип `(Int)->([Int])`. Он полностью соответствует типу функции `generateWallet(walletLength:)`.

При вызове `sumWallet(banknotesFunction:)` необходимо указать лишь имя передаваемой 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)
```

Сложный функциональный тип

Рассмотренные ранее функции имели простой функциональный тип, определяющий ряд входных и выходных параметров. В некоторых случаях выходное значение функции также является функцией, которая, в свою очередь, может возвращать значение. В результате этого функциональный тип становится сложным, то есть содержащим несколько указателей на возвращаемое значение (несколько стрелок ->).

В самом простом варианте функция, возвращающая другую функцию, которая ничего не возвращает, будет иметь функциональный тип, состоящий из трех пар круглых скобок:

```
() -> () -> ()
```

Представим, что некоторая функция принимает на вход значение типа `Int` и возвращает функцию, которая принимает на вход значение типа `String` и возвращает значение типа `Bool`. Ее функциональный тип будет выглядеть следующим образом:

```
(Int) -> (String) -> Bool
```

Каждый блок, описывающий типы данных входных аргументов, заключается в круглые скобки. Таким образом можно определить, где начинается функциональный тип очередной функции.

Функция в качестве возвращаемого значения

Так как функция имеет определенный функциональный тип, его можно указывать не только для входных аргументов, но и для выходного значения. В листинге 10.15 объявлена функция `returnPrintTextFunction()`, которая возвращает значение типа `() -> String`, то есть функцию.

Листинг 10.15

```
1 // функция вывода текста  
2 func printText()  
3 -> (String) {
```

```

4     return "Очень хорошая книга"           "Очень хорошая книга"
5 }
6 // функция, которая возвращает функцию
7 func returnPrintTextFunction()
8 -> () -> String {
9     return printText                         (( )) -> String
10 }
11
12 let newFunctionInLet =                     (( )) -> String
    returnPrintTextFunction()
13 newFunctionInLet()                         "Очень хорошая книга"

```

Тип возвращаемого значения функции `returnPrintTextFunction()` соответствует типу функции `printText()`, которая является глобальной. В связи с этим достаточно просто указать имя возвращаемой функции после оператора `return`.

В результате присвоения возвращенной функции константе `newFunctionInLet` ее тип данных неявно определяется как `() -> String`, а сама она хранит в себе функцию, которую можно вызывать уже знакомым нам способом (указав имя хранилища с круглыми скобками после него). На рис. 10.1 отображено справочное окно, описывающее новое хранилище для функции `printText()`.



Рис. 10.1. Справочное окно для константы, хранящей функцию в качестве значения

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 теле родительской функции. Представьте бесконечную плоскость и точку на этой плоскости. Точка имеет некоторые координаты. Она может перемещаться по плоскости. Создадим

функцию, которая принимает на входе координаты точки и направление перемещения, после чего передвигает точку и фиксирует ее новые координаты (листинг 10.16).

Листинг 10.16

```

1  func oneStep( coordinates: inout (Int, Int), stepType: String ) {
2      func up( coords: inout (Int, Int)) {
3          coords = (coords.0+1, coords.1)
4      }
5      func right( coords: inout (Int, Int)) {
6          coords = (coords.0, coords.1+1)
7      }
8      func down( coords: inout (Int, Int)) {
9          coords = (coords.0-1, coords.1)
10     }
11     func left( coords: inout (Int, Int)) {
12         coords = (coords.0, coords.1-1)
13     }
14
15     switch stepType {
16         case "up":
17             up(coords: &coordinates)
18         case "right":
19             right(coords: &coordinates)
20         case "down":
21             down(coords: &coordinates)
22         case "left":
23             left(coords: &coordinates)
24         default:
25             break;
26     }
27 }
28 var coordinates = (10, -5)                                (.0 10, .1 -5)
29 oneStep(coordinates: &coordinates,                        (.0 11, .1 -5)
   stepType: "up")
30 oneStep(coordinates: &coordinates,                        (.0 11, .1 -4)
   stepType: "right")
31 coordinates                                              (.0 11, .1 -4)

```

Функция `oneStep(coordinates:stepType:)` осуществляет перемещение точки по плоскости. В ней определено несколько вложенных функций, которые вызываются в зависимости от значения аргумента `stepType`. Данный набор функций доступен только внутри родительской функции `oneStep(coordinates:stepType:)`.

Входной параметр `coordinates` является сквозным, поэтому все изменения, производимые в нем, сохраняются и после окончания работы функции.

Перегрузка функций

Swift позволяет *перегружать функции* (overloading), то есть в одной и той же области видимости создавать функции с одинаковыми именами. Различия функций должны заключаться лишь в типах и именах входных параметров и типе возвращаемого значения. В листинге 10.17 представлены функции, которые могут сосуществовать одновременно в одной области видимости.

Листинг 10.17

```
1 func say(what: String){}
2 func say(what: Int){}
```

У данных функций одно и то же имя `say(what:)`, но различные типы входных аргументов. В результате Swift определяет обе функции как различные и позволяет им сосуществовать одновременно. Это связано с тем, что функциональный тип первой функции (`String`) -> (), а второй — (`Int`) -> (). Если вы имеете один и тот же список входных параметров (их имена и типы идентичны), то для перегрузки необходимо, чтобы функции имели различные типы выходных значений. Рассмотрим пример из листинга 10.18. Представленные в нем функции также могут сосуществовать одновременно.

Листинг 10.18

```
1 func cry() -> String {
2     return "one"
3 }
4 func cry() -> Int {
5     return 1
6 }
```

В данном случае можно сделать важное замечание: возвращаемое значение функции не может быть передано переменной или константе без явного указания типа объявляемого параметра (листинг 10.19).

Листинг 10.19

```
1 let resultOfFunc = say() // ошибка
```

В данном случае Swift просто не знает, какой тип данных у константы, поэтому не может определить, какую функцию вызвать. В результате Xcode сообщит об ошибке.

Если каким-либо образом указать тип данных константы, согласуемый с типом возвращаемого значения одной из функций, то код отработает корректно (листинг 10.20).

Листинг 10.20

```
1 let resultString: String = say()
2 let resultInt = say() + 100
```

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 постоянно вызывать саму себя. При корректном использовании рекурсий функция всегда в конце концов будет завершать свою работу.

Пример рекурсии приведен в листинге 10.21.

Листинг 10.21

```
1 func countdown(firstNum num: Int) {
2     print(num)
3     if num > 0 {
4         // рекурсивный вызов функции
5         countdown(firstNum:num-1)
6     }
7 }
8 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.

10.4. Безымянные функции

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 10.22).

Листинг 10.22

```
1 // безымянная функция в качестве значения константы
2 let functionInLet = {return true}
3 functionInLet()
```

true

Созданная константа `functionInLet` имеет функциональный тип `() -> Bool` и хранит в себе тело функции.

Безымянные функции также 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
```

Теперь константы `trueFunc` и `anotherTrueFunc` указывают на одну и ту же функцию.

Задание 1

Вернемся к заданию 2 из предыдущей главы. Объедините написанный код анализа коллекции шахмат, хранящейся в переменной `Chessmans`, в функции `chessAnalyzer(figures:Chessman)`. 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.

11

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.

11.1. 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 из этого кошелька (листинг 11.1).

Листинг 11.1

```
1 // функция отбора купюр
2 func handle100(wallet: [Int]) -> [Int] {
3     var returnWallet = [Int]()
```

```

4     for banknot in wallet {
5         if banknot==100{
6             returnWallet.append(banknot)
7         }
8     }
9     return returnWallet
10 }
11 // электронный кошелек
12 var wallet = [10,50,100,100,5000,100,50,50,500,100]
13 handle100(wallet: wallet)

```

При каждом вызове функция `handle100(wallet:)` будет возвращать массив сторублевых купюр. Здесь `handle100(wallet:)` — 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 знакомимся, когда передавали функцию 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 11.2).

Листинг 11.2

```

1 func handleMore1000(wallet: [Int]) -> [Int] {
2     var returnWallet = [Int]()
3     for banknot in wallet {
4         if banknot>=1000{
5             returnWallet.append(banknot)
6         }
7     }
8     return returnWallet
9 }
10 var wallet = [10,50,100,100,5000,100,50,50,500,100]
11 handleMore1000(wallet: wallet)

```

В результате получается, что при написании двух функций в значительной мере происходит дублирование кода. Разница функций `handle100(wallet:)` и `handleMore1000(wallet:)` лишь в проверяемом условии. Остальной код в функциях один и тот же.

Для решения этой проблемы можно пойти двумя путями:

- ❑ реализовать весь функционал в пределах одной функции и передавать условие в виде аргумента;
- ❑ реализовать три функции. Первая будет группировать повторяющийся код и принимать в виде аргумента одну из оставшихся

функций, которые, в свою очередь, будут производить операцию сравнения.

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 11.3).

Листинг 11.3

```

1  // единая функция формирования результирующего массива
2  func handle(wallet: [Int], closure: (Int) -> Bool) -> [Int] {
3      var returnWallet = [Int]()
4      for banknot in wallet {
5          if closure(banknot) {
6              returnWallet.append(banknot)
7          }
8      }
9      return returnWallet
10 }
11 // функция сравнения с числом 100
12 func compare100(banknot: Int) -> Bool {
13     return banknot==100
14 }
15 // функция сравнения с числом 1000
16 func compareMore1000(banknot: Int) -> Bool {
17     return banknot>=1000
18 }
19 var wallet = [10,50,100,100,5000,100,50,50,500,100]
20 handle(wallet: wallet, closure: compare100)
21 handle(wallet: wallet, closure: compareMore1000)

```

Функция `handle(wallet:closure:)` получает в качестве входного параметра `closure` одну из функций проверки условия и в операторе `if` вызывает переданную функцию. Функции проверки принимают на входе анализируемую купюру и возвращают `Bool` в зависимости от результата сравнения.

Чтобы получить купюры определенного достоинства, необходимо вызвать функцию `handle(wallet:closure:)` и передать в нее имя одной из функций проверки.

В итоге мы получим очень качественный код, который достаточно легко расширять.

11.2. 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(wallet: closure:)`, передавая ей замыкающее выражение в качестве входного параметра (листинг 11.4).

Листинг 11.4

```
1 // отбор купюр достоинством выше 1000 рублей
2 handle(wallet: wallet: wallet, closure: {(banknot: Int) -> Bool in
3   return banknot >= 1000
4 })
5 // отбор купюр достоинством 100 рублей
6 handle(wallet: wallet: wallet, closure: {(banknot: Int) -> Bool in
7   return banknot == 100
8 })
```

В результате необходимость в существовании функций `compare100(banknot:)` и `compareMore1000(banknot:)` 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(wallet:closure:)` указывается его функциональный тип (он принимает функцию типа `(Int) -> Bool`), поэтому при передаче замыкающего выражения нет необходимости дублировать данную информацию (листинг 11.5).

Листинг 11.5

```
1  // отбор купюр достоинством выше 1000 рублей
2  handle(wallet: wallet, closure: {banknot in
3      return banknot>=1000
4  })
5  // отбор купюр достоинством 100 рублей
6  handle(wallet: wallet, closure: {banknot in
7      return banknot==100
8  })
```

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.

11.3. 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 11.6).

Листинг 11.6

```
1  // отбор купюр достоинством выше 1000 рублей
2  handle(wallet: wallet,
3      closure: {banknot in banknot>=1000})
4  // отбор купюр достоинством 100 рублей
```

```
5 handle(wallet: wallet,  
6 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.

11.4. Сокращенные имена параметров

Продолжим оптимизацию используемых нами замыканий. Для однострочных замыкающих выражений Swift автоматически предоставляет доступ к 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(wallet:closure:)` с использованием сокращенных имен (листинг 11.7).

Листинг 11.7

```
1 // отбор купюр достоинством выше 1000 рублей  
2 handle(wallet: wallet,  
3 closure: {$0>=1000})  
4 // отбор купюр достоинством 100 рублей  
5 handle(wallet: wallet,  
6 closure: {$0==100})
```

Здесь `$0` — это входной аргумент `banknot` входного аргумента-замыкания `closure` в функции `handle(wallet: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(wallet:closure:)`, где параметр `closure` является последним), Swift позволяет вынести его значение (замыкающее выражение) за круглые скобки (листинг 11.8).

Листинг 11.8

```
1 // отбор купюр достоинством выше 1000 рублей  
2 handle(wallet: wallet)
```

```
3  {$0>=1000}
4  // отбор купюр достоинством 100 рублей
5  handle(wallet: wallet)
6  {$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 11.9 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 11.9

```
1  let successbanknot = [100, 500]
2  handle(wallet: wallet)
3  {banknot in
4      for number in successbanknot {
5          if number == banknot {
6              return true
7          }
8      }
9      return false
10 }
```

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 принесут никаких изменений в исходный массив.

11.5. 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, но при этом ограничились лишь присвоением значения, упустив вопросы указания функционального типа.

Рассмотрим пример из листинга 11.10, в котором определяется константа `closure`, содержащая в виде значения замыкающее выражение.

Листинг 11.10

```

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 11.11.

Листинг 11.11

```

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.

11.6. Метод сортировки массивов

Swift предлагает большое количество функций и методов, позволяющих значительно упростить разработку приложений. Одним из таких методов является `sorted(by:)`, предназначенный для сортировки массивов, как строковых, так и числовых. Он принимает на входе массив, который необходимо отсортировать, и условие сортировки.

Принимаемое условие сортировки — это обыкновенное замыкающее выражение, которое вызывается внутри метода `sorted(by:)`, принимает

на входе два очередных элемента сортируемого массива и возвращает значение `Bool` в зависимости от результата их сравнения. Для того чтобы получить отсортированный массив, необходимо передать соответствующее замыкание. В листинге 11.12 мы отсортируем массив `myArray` таким образом, чтобы элементы были расположены по возрастанию. Для этого в метод `sorted(by:)` передадим такое замыкающее выражение, которое возвращает `true`, когда второе из сравниваемых чисел больше.

Листинг 11.12

```
1 var array = [1,44,81,4,277,50,101,51,8]
2 array.sorted(by: { (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 11.13. Как и в предыдущем примере, здесь тоже необходимо отсортировать массив `myArray` таким образом, чтобы элементы были расположены по возрастанию. Для этого в метод `sorted(by:)` передается такое замыкающее выражение, которое возвращает `true`, когда второе из сравниваемых чисел больше.

Листинг 11.13

```
1 var array = [1,44,81,4,277,50,101,51,8]
2 var sortedArray = array.sorted(by: {$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 11.14).

Листинг 11.14

```
1 var array = [1,44,81,4,277,50,101,51,8]
2 var sortedArray = array.sorted(by: <)
```

Надеюсь, вы приятно удивлены потрясающими возможностями Swift!

11.7. Каррирование функций

Одной из непростых для понимания, но в то же время очень важных для разработки тем является каррирование (от англ. *curring*). Несмотря на всю его сложность, знать и использовать этот механизм обязательно, не зря при подготовке Swift 3 в Apple обратили на него особое внимание и внесли некоторые правки, направленные на повышение читабельности кода при использовании каррирования.

Каррирование — это процесс, при котором функция от нескольких аргументов преобразуется в функцию (или набор функций) от одного аргумента. Это становится возможным благодаря тому, что в качестве выходного значения функции может выступать другая функция.

НАЧИНАЮЩИМ Предположим, что у нас есть функция, которая принимает на вход три параметра. При каррировании мы получим функцию, которая принимает на вход один аргумент, а возвращает функцию, которая также принимает на вход один аргумент и в свою очередь также возвращает функцию, которая принимает на вход один аргумент и возвращает некоторый результат. То есть получается своеобразная линейка функций.

Разложим все по полочкам.

- У нас есть функция с типом $(Int, Int, Int) \rightarrow Int$, которая зависит от трех входных аргументов типа Int и возвращает значение типа Int .
- При каррировании мы получим функцию типа $(Int) \rightarrow (Int) \rightarrow (Int) \rightarrow Int$. Для этого выполним следующие шаги:
 - ♦ обозначим функциональный тип $(Int) \rightarrow Int$ последней функции в цепочке как A . Тогда каррированная функция будет выглядеть как $(Int) \rightarrow (Int) \rightarrow A$;
 - ♦ обозначим функциональный тип $(Int) \rightarrow A$ как B . Тогда каррированная функция будет выглядеть как $(Int) \rightarrow B$.
- Получается, что наша функция принимает на вход одно целое число и возвращает значение типа B .
- Значение типа B , в свою очередь, также является функцией, которая принимает на вход одно целое число и возвращает значение типа A .
- Значение типа A также является функцией, которая принимает на вход одно целое число и возвращает одно целое число.
- В результате одну функцию, зависящую от трех параметров, мы реорганизовали (каррировали) к трем взаимозависимым функциям, каждая из которых зависит всего от одного значения.

Рассмотрим пример. Существует функция с типом $(Int, Int) \rightarrow Int$, которая получает на вход два целочисленных значения, производит некоторую операцию и возвращает ответ в виде целого числа (листинг 11.15).

Листинг 11.15

```

1 func sum(x: Int, y: Int) -> Int {
2     return x + y
3 }
4 sum(x: 1, y: 4) // вернет 5

```

С целью каррирования напишем новую функцию, которая принимает на вход всего один целочисленный параметр, а возвращает замыкание типа `(Int) -> Int` (листинг 11.16).

Листинг 11.16

```

1 func sum2(_ x: Int) -> (Int) -> Int {
2     return { return $0+x }
3 }
4 var closure = sum2(1)
5 closure(12)

```

`(Int) -> Int`
13

Переменная `closure` получает в качестве значения замыкание, которому мы можем передать входной параметр. Прелесть каррирования в том, что мы можем объединить вызов функции `sum2(x:)` и передачу значения в возвращаемое ею замыкание (листинг 11.17).

Листинг 11.17

```

1 sum2(5)(12) // вернет 17

```

В результате мы получаем прекрасно читаемую и удобную в использовании функцию.

Если говорить о пользе, то каррирование хорошо именно тем, что устраняет зависимость функции от нескольких параметров. Мы можем вызвать функцию, как только получим первое нужное значение, и далее при необходимости многократно вызывать возвращенное ею замыкание при поступлении второго требуемого параметра (листинг 11.18).

Листинг 11.18

```

1 var closure = sum2( 1)
2 closure(12) // вернет 13
3 closure(19) // вернет 20

```

11.8. Захват переменных

Swift позволяет зафиксировать значения внешних по отношению к замыканию параметров, которые они имели на момент его определения.

Синтаксис захвата переменных

Обратимся к примеру в листинге 11.19. Существуют два параметра, **a** и **b**, которые не передаются в качестве входных аргументов в замыкание, но используются им в вычислениях. При каждом вызове такого замыкания оно будет определять значения данных параметров, прежде чем приступить к выполнению операции с их участием.

Листинг 11.19

```
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 переменных **a** и **b**. При изменении их значений возвращаемое замыканием значение не меняется.

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 11.20).

Листинг 11.20

```
1 var a = 1
2 var b = 2
3 let closureSum : () -> Int = {
4     [a,b] in
5     return a+b
6 }
7 closureSum() // 3
8 a = 3
9 b = 4
10 closureSum() // 3
```

Замыкание, хранящееся в константе `closureSum`, складывает значения переменных **a** и **b**. При изменении этих значений возвращаемое замыканием значение также меняется.

Захват вложенной функцией

Другим способом захвата значения внешнего параметра является вложенная функция, написанная в теле другой функции. Вложенная функция может захватывать произвольные переменные, константы и даже входные аргументы, объявленные в родительской функции.

Рассмотрим пример из листинга 11.21.

Листинг 11.21

```
1 func makeIncrementer(forIncrement amount: Int) -> () -> Int {
2     var runningTotal = 0
3     func incrementer() -> Int {
4         runningTotal += amount
5         return runningTotal
6     }
7     return incrementer
8 }
```

Функция `makeIncrement(forIncrement:)` возвращает значение с функциональным типом `()->Int`. Это значит, что вернется замыкание, не имеющее входных аргументов и возвращающее целочисленное значение.

Функция `makeIncrement(forIncrement:)` определяет два параметра:

- ❑ `runningTotal` — переменную типа `Int`, объявляемую в теле функции. Именно ее значение является результатом работы всей конструкции;
- ❑ `amount` — входной аргумент, имеющий тип `Int`. Он определяет, насколько увеличится значение `runningTotal` при очередном обращении.

Вложенная функция `increment()` не имеет входных или объявляемых параметров, но при этом обращается к `runningTotal` и `amount` внутри своей реализации. Она делает это в автоматическом режиме путем захвата значений обоих параметров по ссылке. Захват значений по ссылке гарантирует, что измененные значения параметров не исчезнут после окончания работы функции `makeIncrement(forIncrement:)` и будут доступны при повторном вызове функции `increment()`.

ПРИМЕЧАНИЕ В целях оптимизации используемых ресурсов Swift может самостоятельно решить создать копию захваченного значения, а не ссылку на него. Это делается в том случае, если захватываемое значение не изменяется в пределах вложенной функции.

Теперь обратимся к листингу 11.22.

Листинг 11.22

```
1 var incrementByTen = makeIncrementer(forIncrement: 10)
2 var incrementBySeven = makeIncrementer(forIncrement: 7)
3 incrementByTen() // вернется 10
4 incrementByTen() // вернется 20
5 incrementByTen() // вернется 30
6 incrementBySeven() // вернется 7
7 incrementBySeven() // вернется 14
8 incrementBySeven() // вернется 21
```

В переменных `incrementByTen` и `incrementBySeven` хранятся возвращаемые функцией `makeIncrement(forIncrement:)` замыкания. В первом случае значение `runningTotal` увеличивается на 10, а во втором — на 7. Каждая из переменных хранит свою копию захваченного значения `runningTotal`, именно поэтому при их использовании увеличиваемые значения не пересекаются и увеличиваются независимо друг от друга.

ВНИМАНИЕ Так как в переменных `incrementByTen` и `incrementBySeven` хранятся замыкания, то при доступе к ним после их имени необходимо использовать скобки (по аналогии с доступом к функциям).

11.9. Замыкания — это тип-ссылка

Функциональный тип данных — это тип-ссылка (reference type). Это значит, что замыкания передаются не копированием, а с помощью ссылки на область памяти, где хранится это замыкание.

Рассмотрим пример, описанный в листинге 11.23.

Листинг 11.23

```
1 var incrementByFive = makeIncrementer(forIncrement: 5)
2 var copyIncrementByFive = incrementByFive
```

В данном примере используется функция `makeIncrement(forIncrement:)`, объявленная ранее. Напомним, она возвращает замыкание типа `()->Int`, которое в данном случае предназначено для увеличения значения на 5. Возвращаемое замыкание записывается в переменную `incrementByFive`, после чего копируется в переменную `copyIncrementByFive`. В результате можно обратиться к одному и тому же замыканию, используя как `copyIncrementByFive`, так и `incrementByFive` (листинг 11.24).

Листинг 11.24

```
1 incrementByFive() // вернет 5
2 copyIncrementByFive() // вернет 10
3 incrementByFive() // вернет 15
```

Как вы можете видеть, какую бы функцию мы ни использовали, происходит модификация одного и того значения (каждое последующее значение больше предыдущего на 5). Это обусловлено тем, что замыкания — ссылочный тип данных, или тип-ссылка, или *reference type*.

11.10. Автозамыкания

Автозамыкания — это замыкания, которые автоматически создаются из переданного выражения. Иными словами, может существовать функция, имеющая один или несколько входных параметров, которые при ее вызове передаются как значения, но во внутренней реализации функции используются как самостоятельные замыкания. Рассмотрим пример из листинга 11.25.

Листинг 11.25

```
1 var arrayOfNames = ["Helga", "Bazil", "Alex"]
2 func printName(nextName: String ) {
3     // какой-либо код
4     print(nextName)
5 }
6 printName(arrayOfNames.remove(at: 0))
```

Консоль:

Helga

При каждом вызове функции `printName(nextName:)` в качестве входного значения ей передается результат вызова метода `remove(at:)` массива `arrayOfNames`.

Независимо от того, в какой части функции будет использоваться переданный параметр (или не будет использоваться вовсе), значение, возвращаемое методом `remove(at:)`, будет вычислено в момент вызова функции `printName(nextName:)`. Получается, что передаваемое значение вычисляется независимо от того, нужно ли оно в ходе выполнения функции.

Отличным решением данной проблемы станет использование ленивых вычислений, то есть таких вычислений, которые будут выполняться лишь в тот момент, когда это понадобится. Для того чтобы реализо-

вать этот подход, можно передавать в функцию `printName(nextName:)` замыкание, которое будет вычисляться в тот момент, когда к нему обратятся (листинг 11.26).

Листинг 11.26

```
1 var arrayOfNames = ["Helga", "Bazil", "Alex"]
2 func printName(nextName: ()->String) {
3     // какой-либо код
4     print(nextName())
5 }
6 printName({arrayOfNames.remove(at: 0)})
```

Консоль:

Helga

Для решения этой задачи потребовалось изменить тип входного параметра `nextName` на `()->String` и заключить передаваемый метод `remove(at:)` в фигурные скобки. Теперь внутри реализации функции `printName(nextName:)` к входному аргументу `nextName` необходимо обращаться как к самостоятельной функции (с использованием круглых скобок после имени параметра). Таким образом, значение метода `remove(at:)` будет вычислено именно в тот момент, когда оно понадобится, а не в тот момент, когда оно будет передано. Единственным недостатком данного подхода является то, что входной параметр должен быть заключен в фигурные скобки, а это несколько усложняет использование функции и чтение кода.

С помощью автозамыканий можно использовать положительные функции обоих рассмотренных примеров: отложить вычисление переданного значения и передавать значение в виде значения (без фигурных скобок).

Для того чтобы реализовать автозамыкание, требуется, чтобы выполнялись следующие требования.

- ❑ Входной аргумент должен иметь функциональный тип.

В примере, приведенном ранее, аргумент `nextName` уже имеет функциональный тип `()->String`.

- ❑ Функциональный тип не должен определять типы входных параметров.

В примере типы входных параметров не определены (пустые скобки).

- ❑ Функциональный тип должен определять тип возвращаемого значения.

В примере тип возвращаемого значения определен как `String`.

- ❑ Переданное выражение должно возвращать значение того же типа, которое определено в функциональном типе замыкания.

В примере передаваемая в качестве входного аргумента функция возвращает значение типа `String` точно так же, как определено функциональным типом входного аргумента.

- ❑ Перед функциональным типом необходимо использовать атрибут `@autoclosure`.
- ❑ Передаваемое значение должно указываться без фигурных скобок.

Перепишем код из предыдущего листинга в соответствии с указанными требованиями (листинг 11.27).

Листинг 11.27

```
1 var arrayOfNames = ["Helga", "Bazil", "Alex"]
2 func printName(nextName: @autoclosure ()->String) {
3     // какой-либо код
4     print(nextName())
5 }
6 printName(arrayOfNames.remove(at: 0))
```

Консоль:

Helga

Теперь метод `remove(at:)` передается в функцию `printName(nextName:)` как обычный аргумент, без использования фигурных скобок, но внутри тела используется как самостоятельная функция.

Ярким примером глобальной функции, входящей в стандартные возможности Swift и использующей механизм автозамыканий, является функция `assert(condition:message:file:line:)`. Аргументы `condition` и `message` — это автозамыкания, первое из которых вычисляется только в случае активного debug-режима, а второе — только в случае, когда `condition` соответствует `false`.

11.11. Выходящие замыкания

Как вы уже неоднократно убеждались и убедитесь еще не раз, Swift — очень умный язык программирования. Он старается экономить ваши ресурсы там, где вы можете об этом даже не догадываться.

По умолчанию все переданные в функцию замыкания имеют ограниченную этой функцией область видимости, то есть если вы решите

сохранить замыкание для дальнейшего использования, то получите сообщение об ошибке. Другими словами, все переданные в функцию замыкания называются не выходящими за пределы ее тела. Если Swift видит, что область, где замыкание доступно, ограничена, он при первой же возможности удалит его, чтобы освободить и не расходовать оперативную память.

Для того чтобы позволить замыканию выйти за пределы области видимости функции, необходимо указать атрибут `@escaping` перед функциональным типом при описании входных параметров функции.

ПРИМЕЧАНИЕ В Swift 3.0 был изменен подход к передаче замыканий в качестве параметра в функции. Ранее для того, чтобы запретить замыканию выход за пределы области видимости функции, требовалось указать атрибут `@noescaping`, теперь же все передаваемые в функцию замыкания автоматически помечаются как не выходящие.

Рассмотрим пример. Предположим что в программе есть специальная переменная, предназначенная для хранения замыканий, то есть являющаяся коллекцией замыканий (листинг 11.28).

Листинг 11.28

```
1 var arrayOfClosures: [()->Int] = []
```

Пока еще пустой массив `arrayOfClosures` может хранить в себе замыкания с функциональным типом `()->Int`. Реализуем функцию, добавляющую в этот массив переданные ей в качестве аргументов замыкания (листинг 11.29).

Листинг 11.29

```
1 func addNewClosureInArray(_ newClosure: ()->Int){
2     arrayOfClosures.append(newClosure) // ошибка
3 }
```

Xcode сообщит вам об ошибке. И у этого есть две причины:

- ❑ Замыкание — это тип-ссылка, то есть оно передается по ссылке, но не копированием.
- ❑ Замыкание, которое будет храниться в параметре `newClosure`, будет иметь ограниченную телом функции область видимости, а значит, не может быть добавлено в глобальную (по отношению к телу функции) переменную `arrayOfClosures`.

Для решения этой проблемы необходимо указать, что замыкание, хранящееся в переменной `newClosure`, является выходящим. Для

этого перед описанием функционального типа данного аргумента укажите атрибут `@escaping`, после чего вы сможете передать в функцию `addNewClosureInArray(_:)` произвольное замыкание (листинг 11.30).

Листинг 11.30

```
1 func addNewClosureInArray(_ newClosure: @escaping ()->Int){
2     arrayOfClosures.append(newClosure)
3 }
4 addNewClosureInArray({return 100})
5 addNewClosureInArray{return 1000}
6 arrayOfClosures[0]() // 100
7 arrayOfClosures[1]() // 1000
```

Обратите внимание на то, что в одном случае замыкание передается с круглыми скобками, а в другом — без них. Так как функция `addNewClosureInArray(_:)` имеет один входной аргумент, то допускаются оба варианта.

ПРИМЕЧАНИЕ Если вы передаете замыкание в виде параметра, то можете использовать модификатор `inout` вместо `@escaping`.

12

Вспомогательные функциональные элементы

Цель этой книги не только в том, чтобы вы узнали синтаксис и основы разработки на «яблочном» языке программирования. Очень хочется, чтобы вы начали глубже понимать принципы разработки в Xcode на Swift. В этой главе приведены различные вспомогательные функциональные элементы, которыми так богат язык программирования. В данной главе приведены важные функциональные элементы, которые смогут значительно облегчить вашу практику программирования. Используя методы `map(_:)`, `flatMap(_:)`, `reduce(_:)` и `filter(_:)`, вы можете очень успешно обрабатывать коллекции вроде массивов, наборов и словарей. Если у вас уже есть опыт программирования, то, вероятно, вы привыкли заниматься обработкой этих типов с помощью циклов. Однако в настоящее время это неверный подход. Советую вам активно использовать описанные функции.

12.1 Метод `map(_:)`

Данный метод позволяет применить переданное в него замыкание для каждого элемента коллекции. В результате его выполнения возвращается новая последовательность, тип элементов которой может отличаться от типа исходных элементов (рис. 12.1).

Рассмотрим пример, описанный в листинге 12.1.

Листинг 12.1

```
1 var myArray = [1, 2, 4, 5, 7]
2 var newArray = myArray.map{$0}
3 newArray // [2, 4, 5, 7]
```

Метод `map(_:)` принимает замыкание и применяет его к каждому элементу массива `myArray`. Переданное замыкание `{$0}` не производит

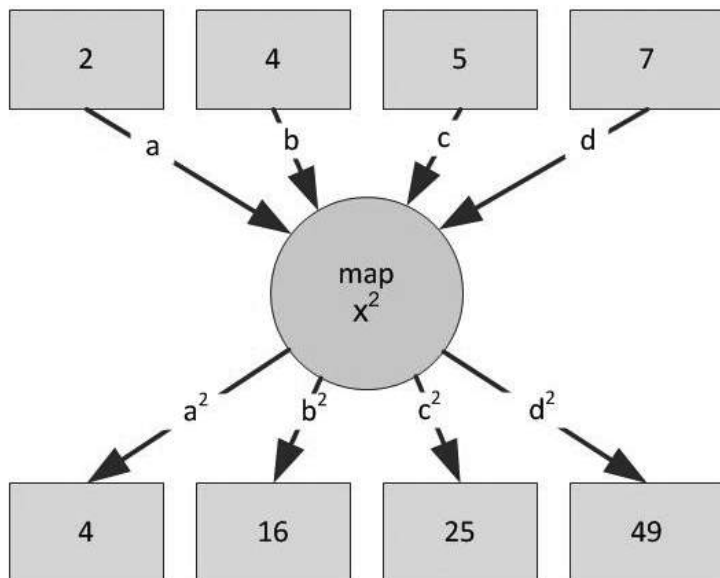


Рис 12.1. Принцип работы метода map

каких-либо действий над элементами, поэтому результат, содержащийся в переменной `newArray`, не отличается от исходного.

ПРИМЕЧАНИЕ В данном примере используются сокращенные имена параметров, а если точнее, то `$0`. Данная тема была изучена в главе, посвященной замыканиям. Давайте повторим, каким образом функция `map(_:)` лишилась круглых скобок и приобрела вид `map{$0}`.

Метод `map(_:)` позволяет передать в него замыкание, которое имеет один входной аргумент того же типа, что и элементы обрабатываемой коллекции, и один выходной параметр. Если не использовать сокращенный синтаксис, то вызов метода будет выглядеть следующим образом:

```
var array = [2, 4, 5, 7]
let newArray = array.map({
    (value: Int) -> Int in
    return value
})
```

Замыкание никак не изменяет входной аргумент — просто возвращает его.

Оптимизируем замыкание.

- Сократим код перед ключевым словом `in`, так как передаваемое замыкание имеет всего один входной аргумент.

- Уберем круглые скобки, так как метод `map(_:)` имеет один входной аргумент.
- Уберем оператор `return`, так как тело замыкания помещается в одно выражение.

В результате получим следующий код:

```
let newArray = array.map{value in value}
```

Теперь можно убрать ключевое слово `in` и заменить `value` на сокращенное имя `$0`:

```
let newArray = array.map{$0}
```

Теперь внесем изменения в замыкание. Сделаем так, чтобы результирующий массив содержал элементы, равные возведенным в квадрат элементам исходного массива (листинг 12.2).

Листинг 12.2

```
1 var array = [ 2, 4, 5, 7]
2 var newArray = array.map{$0*$0}
3 newArray // [4, 16, 25, 49]
```

Как говорилось ранее, тип значений результирующей последовательности может отличаться от типа элементов исходной последовательности. Так, например, в листинге 12.3 количество элементов массивов `intArray` и `boolArray` одинаково, но тип элементов различается (`Int` и `Bool` соответственно).

Листинг 12.3

```
1 var intArray = [1, 2, 3, 4]
2 var boolArray = array.map{$0 > 2}
3 boolArray // [false, false, true, true]
```

Каждый элемент последовательности сравнивается с двойкой, в результате чего возвращается логическое значение. `13. = [1, 2, 3, 4] = ){$0 > 2} // [ , , , ]`. Другой особенностью `map(_:)` является возможность создания многомерных массивов (листинг 12.4).

Листинг 12.4

```
1 let numbers = [1, 2, 3, 4]
2 let mapped = numbers.map { Array(repeating: $0, count: $0) }
3 mapped // [[1], [2, 2], [3, 3, 3], [4, 4, 4, 4]]
```

В данном примере вместо каждого элемента исходного массива вставляется полноценный массив типа `Array<Int>`.

Помимо массивов метод `map(_:)` может быть применен к словарям. В листинге 12.5 этот метод используется для перевода расстояния, указанного в милях, в километры.

Листинг 12.5

```
1 let milesToDest = ["Moscow":120.0,"Dubai":50.0,"Paris":70.0]
2 var kmToDest = milesToDest.map { name,miles in
  [name:miles * 1.6093] }
3 kmToDest // [{"Dubai": 80.465}, {"Paris": 112.651},
  ["Moscow": 193.116}]
```

12.2 Метод filter(_:)

Метод `filter(_:)` используется, когда требуется отфильтровать элементы коллекции по определенному правилу (рис. 12.2).

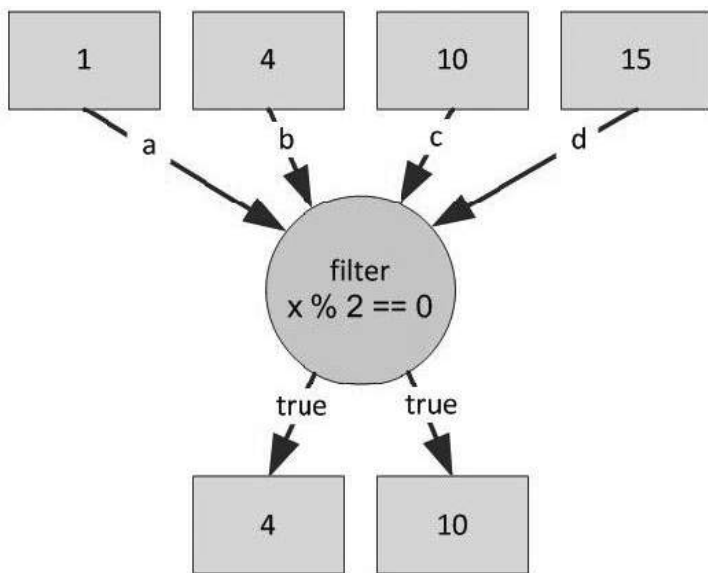


Рис. 12.2. Принцип работы метода `filter`

В листинге 12.6 показана фильтрация всех целочисленных элементов исходного массива, которые делятся на 2 без остатка (то есть всех четных чисел).

Листинг 12.6

```
1 let numbers = [1, 4, 10, 15]
2 let even = numbers.filter{ $0 % 2 == 0 }
3 even // [4, 10]
```

12.3 Метод `reduce(_:_:)`

Метод `reduce(_:_:)` позволяет объединить все элементы коллекции в одно значение в соответствии с переданным замыканием. Помимо самих элементов, метод принимает первоначальное значение, которое служит для выполнения операции с первым элементом коллекции.

Предположим, необходимо определить общее количество имеющихся у вас денег. На вашей карте 210 рублей, а в кошельке 4 купюры различного достоинства. Эта задача легко решается с помощью метода `reduce(_:_:)` (рис. 12.3 и листинг 12.7).

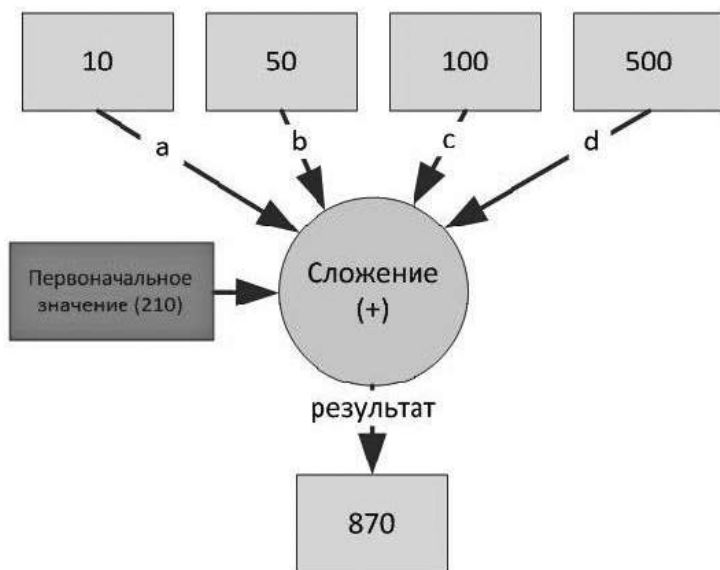


Рис. 12.3. Принцип работы метода `reduce`

Листинг 12.7

```
1 let cash = [10, 50, 100, 500]
2 let total = cash.reduce(210, +) // 870
```

Первый аргумент — это начальное значение, второй — замыкание, обрабатывающее каждую пару элементов. Первая операция сложения производится между начальным значением и первым элементом массива `cash`. Результат этой операции складывается со вторым элементом массива и т. д.

Замыкание, производящее операцию, может быть произвольным — главное, чтобы оно обрабатывало операцию для двух входящих аргументов (листинг 12.8).

Листинг 12.8

```
1 let cash = [10, 50, 100, 500]
2 let total = cash.reduce(210, {$0*$1}) // 5250000000
3 let totalTwo = cash.reduce(210, {x,y in x - y}) // -450
```

12.4 Метод flatMap(_:)

Главным различием функций `map(_:)` и `flatMap(_:)` является то, что `flatMap(_:)` всегда возвращает плоский одномерный массив. Так, пример, приведенный в листинге 12.4, но с использованием `flatMap(_:)`, вернет одномерный массив (листинг 12.9).

Листинг 12.9

```
1 let numbers = [1, 2, 3, 4]
2 let flatMapped = numbers.flatMap { Array(repeating: $0,
3   count: $0) }
3 flatMapped // [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
```

Вся мощь `flatMap(_:)` проявляется тогда, когда в многомерном массиве требуется найти все подпадающие под некоторое условие значения (листинг 12.10).

Листинг 12.10

```
1 let someArray = [[1, 2, 3, 4, 5], [11, 44, 1, 6],
2   [16, 403, 321, 10]]
3 let filterSomeArray = someArray.flatMap{$0.filter{ $0 % 2 == 0}}
3 filterSomeArray // [2, 4, 44, 6, 16, 10]
```

13

Ленивые вычисления

13.1. Понятие ленивых вычислений

«Ленивый» в Swift звучит как *lazy*. Можно сказать, что *lazy* — синоним производительности. Хорошо оптимизированные программы практически всегда используют *lazy*-вычисления. В одной из предыдущих глав вы встречались с типом `LazyMapCollection`, который является не чем иным, как ленивой коллекцией. Возможно, вы работали с *lazy* и в других языках. В любом случае внимательно изучите приведенный далее материал.

В программировании ленивыми называются такие элементы, вычисление значений которых откладывается до момента обращения к ним. Таким образом, пока значение не потребуется и не будет использовано, оно будет храниться в виде исходных данных. С помощью ленивых вычислений достигается экономия процессорного времени, то есть компьютер не занимается ненужными в данный момент вычислениями.

Существует два типа ленивых элементов:

- ❑ *lazy-by-name* — значение элемента вычисляется при каждом доступе к нему;
- ❑ *lazy-by-need* — элемент вычисляется один раз при первом обращении к нему, после чего фиксируется и больше не изменяется.

Swift позволяет работать с обоими типами ленивых элементов, но в строгом соответствии с правилами.

Вернемся к примеру, с которым мы встречались при изучении коллекций. С помощью свойства `keys` можно получить коллекцию всех ключей словаря. Но при этом вместо ожидаемого набора Swift возвращает значение неизвестного типа `LazyMapCollection` (листинг 13.1).

Листинг 13.1

```
1 var countryDict = ["RUS": "Российская  
   Федерация", "BEL": "Беларусь",  
   "UKR": "Украина"]
```

```
["BEL": "Беларусь",  
 "UKR": "Украина", "RUS":  
 "Российская Федерация"]
```

```

2 // все ключи словаря countryDict
3 var keys = countryDict.keys

```

*LazyMapCollection
<Dictionary<String,
String>, String>*

Из названия `LazyMapCollection` видно, что данный элемент является ленивой коллекцией. Как вы считаете, какой смысл возвращать ленивый элемент в результате обращения к свойству `keys`? Дело в том, что Swift не знает заранее, в каком типе хранилища вы будете содержать полученные значения, полученную коллекцию элементов: возможно, это будет массив, или набор, или что-то иное, а может быть, данные значения и вовсе не понадобятся! Именно для того, чтобы можно было работать с ними, используются ленивые вычисления.

Swift помогает вам экономить ресурсы компьютера. Так происходит и в данном случае. Получив значение типа `LazyMapCollection`, вы можете делать с ним все, что посчитаете нужным, например преобразовать в массив или набор (листинг 13.2).

Листинг 13.2

```

1 var mySet = Set(keys)
2 var myArray = Array(keys)

```

*{"BEL", "UKR", "RUS"}
["BEL", "UKR", "RUS"]*

В результате выполнения вы получите уже знакомые массив и набор. Этот пример отлично демонстрирует всю важность `lazy` в Swift. Перейдем к рассмотрению других примеров использования `lazy`.

13.2. Замыкания в ленивых вычислениях

Как говорилось ранее, ленивые вычисления позволяют оптимизировать расход ресурсов компьютера. С помощью замыканий мы можем создавать ленивые конструкции типа `lazy-by-name`, значение которых высчитывается при каждом обращении к ним.

Рассмотрим пример из листинга 13.3.

Листинг 13.3

```

1 var arrayOfNames = ["Helga", "Bazil",
2 "Alex"]
3 print(arrayOfNames.count)
4 let nextName = { arrayOfNames.remove(at: 0) }
5 print(arrayOfNames.count)
6 print(arrayOfNames.count)

```

*["Helga", "Bazil",
"Alex"]
"3\n"
"3\n"
"Helga"
"2\n"*

В константе `nextName` хранится замыкание, удаляющее первый элемент массива `arrayOfNames`. Несмотря на то что константа объявлена, а ее значение проинициализировано, количество элементов массива не уменьшается до тех пор, пока ее значение не будет вызвано непосредственно. Это связано с тем, что замыкание, хранящееся в константе, — ленивое. Таким образом, можно сказать, что любая функция или метод являются *lazy-by-name*, так как их значение высчитывается при каждом обращении к ним.

13.3. Свойство *lazy*

Некоторые конструкции языка Swift (например, массивы и словари) имеют свойство *lazy*, позволяющее преобразовать их в ленивые. Наиболее часто это происходит, когда существуют цепочки вызова свойств или методов и выделение памяти и вычисление промежуточных значений является бесполезной тратой ресурсов, так как эти значения никогда не будут использованы.

Рассмотрим следующий пример: существует массив целых чисел, значения которого непосредственно не используются в работе программы. Вам требуется лишь результат его обработки методом `map(_ :)`, и то не в данный момент, а позже (листинг 13.4).

Листинг 13.4

```
1 var myCollection = [1,2,3,4,5,6].lazy.map{$0 + 1}
   LazyMapRandomAccessCollection<Array<Int>, Int>
```

В результате выполнения возвращается ленивая коллекция. При этом память под отдельный массив целочисленных значений не выделяется, а вычисления метода `map(_ :)` не производятся до тех пор, пока не произойдет обращение к переменной `myCollection`.

Вся прелесть такого подхода в том, что вы можете увеличивать цепочки вызовов, но при этом лишнего расхода ресурсов не будет (листинг 13.5).

Листинг 13.5

<pre>1 var myCollection = [1,2,3,4,5,6] .lazy .map{\$0 + 1} .filter{\$0 % 2 == 0} 2 Array(myCollection)</pre>	<pre>LazyFilterBidirectionalCollection <LazyMapRandomAccessCollection <Array<Int>, Int>> [2, 4, 6]</pre>
---	--

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 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 «всё — это объект», то любой параметр (переменная или константа) с определенным типом данных — это объект. Для реализации новых объектов вы уже изучили множество различных типов данных, но, как отмечалось ранее, Swift обладает функционалом создания собственных объектных типов. Для этого существует три механизма: *перечисления* (enum), *структуры* (struct) и *классы* (class). В чем разница между ними? Как их создавать и использовать? Данная часть рассказывает 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 14. ООП как фундамент
- ✓ Глава 15. Перечисления
- ✓ Глава 16. Структуры
- ✓ Глава 17. Классы

- ✓ Глава 18. Свойства
- ✓ Глава 19. Сабскрипты
- ✓ Глава 20. Наследование
- ✓ Глава 21. Псевдонимы Any и AnyObject
- ✓ Глава 22. Инициализаторы и деинициализаторы
- ✓ Глава 23. Удаление экземпляров и ARC
- ✓ Глава 24. Опциональные цепочки
- ✓ Глава 25. Расширения
- ✓ Глава 26. Протоколы
- ✓ Глава 27. Разработка первого приложения
- ✓ Глава 28. Универсальные шаблоны
- ✓ Глава 29. Обработка ошибок
- ✓ Глава 30. Нетривиальное использование операторов

14 ООП как фундамент

Объектно-ориентированное программирование (ООП) является основой разработки программ на языке Swift. Мы уже неоднократно встречались с одним фундаментальным для данного языка правилом «всё — это объект». Пришло время приступить к изучению наиболее интересных и сложных механизмов, доступных в Swift. Данная глава расскажет вам о довольно важных терминах и понятиях, которые помогут вам освоить последующий за ней 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 в некоторой степени отличаются от таковых для языков Pascal, Visual Basic, Python, PHP и т. д.

14.1. Экземпляры

Основные конструкции, с которыми мы будем работать в этой части, — перечисления, структуры и классы. Открою вам один удивительный секрет: вы уже давно работаете с некоторыми из перечисленных конструкций! Например, тот же целочисленный тип данных `Int` — это структура!

Перечисления, структуры и классы имеют свои особенности и возможности, и именно они делают Swift таким, какой он есть. Тем не менее у них есть одно общее базовое свойство — для них могут быть созданы *экземпляры*. Когда вы самостоятельно определяете объектный тип (перечисление, структуру или класс), вы лишь создаете новый тип данных. В свою очередь, создание экземпляра объектного типа — это создание хранилища (переменной или константы) для данных определенного типа. Так, например, упомянутый ранее тип `Int` — это структура, а переменная `myInteger`, хранящая в себе значение этого типа, — экземпляр данной структуры.

ПРИМЕЧАНИЕ Несмотря на то что мы много раз говорили об объектах, правильнее называть их экземплярами.

Объектами в других языках программирования назывались экземпляры классов, а экземпляры структур и перечислений — просто экземплярами. Так как функционал структур, перечислений и классов очень близок по своим возможностям, в Swift соответствующие объекты называются просто экземплярами.

Представьте, что определен класс `Automobile` (автомобиль). Этот класс является типом данных. Данный «Автомобиль» является не каким-то конкретным автомобилем, а лишь конструкцией, с помощью которой можно определить этот конкретный автомобиль. Если создать переменную типа `Automobile`, то в результате мы получим экземпляр этого класса (рис. 14.1).

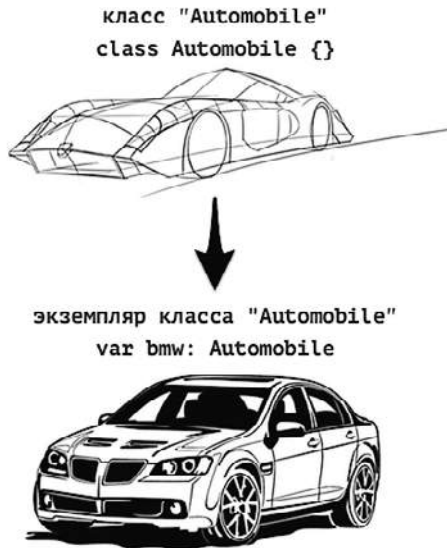


Рис. 14.1. Класс и его экземпляр

Сам класс на рисунке не имеет выраженных черт, так как еще неизвестно, какой же определенный объект реального (или нереального) мира он будет определять. Но когда создана переменная `bmw` типа `Automobile`, мы уже знаем, что с экземпляром в этой переменной мы будем работать как с реальным авто марки BMW:

```
bmw: Automobile
```

или

```
bmw = Automobile()
```

Такой тип данных может наделять экземпляр какими-либо характеристиками. Для класса `Automobile` это могли бы быть марка, модель, цвет, максимальная скорость, объем двигателя и т. д. Характеристики объектных типов называются *свойствами*, с ними мы встречались неоднократно. Для переменной `bmw` значения этих свойств могли бы быть следующими:

```
bmw.brand = "BMW"
bmw.type = "X3"
bmw.maxSpeed = 210
bmw.engineCapacity = 1499
```

Свойства представляют собой хранилища данных, то есть это те же самые переменные и константы, но с ограниченным доступом: они доступны только через доступ к экземпляру. Иначе говоря, вначале вы получаете доступ к экземпляру, а уже потом (через точку) — к его свойству. Помимо свойств у экземпляра могут быть определены методы. Методы, а с ними мы также неоднократно встречались, — это функции, которые определены внутри объектных типов. Класс `Automobile` мог бы иметь следующие методы: издать сигнал, завестись, ускориться:

```
bmw.startEngine()
bmw.accelerate()
bmw.beep()
```

Таким образом, создавая экземпляр, мы можем наполнять его свойствами информацией и использовать его методы. И свойства и методы определяются типом данных.

Способ разработки программ с использованием объектных типов называется объектно-ориентированным программированием. Этот стиль программирования позволяет достичь очень многого при разработке программ. Несмотря на то что вместо термина «объект» используется термин «экземпляр», аббревиатура ООП является устоявшейся в программировании, и в Swift она не трансформируется в ЭОП.

14.2. Пространства имен

Пространства имен (namespaces) — это именованные фрагменты программ. Пространства имен имеют одно очень важное свойство — они скрывают свою внутреннюю реализацию и не позволяют получить

доступ к объектам внутри пространства имен без доступа к самому пространству имен. Это замечательная черта, благодаря которой вы можете иметь объекты с одним и тем же именем в различных пространствах имен.

Мы уже неоднократно говорили об областях видимости переменных и функций. Пространства имен как раз и реализуют в приложении различные области видимости.

Простейшим примером ограничения области видимости может служить функция. Все переменные, объявленные в ней, вне функции недоступны. Но при этом функция не является пространством имен, так как не позволяет получить доступ к объектам внутри себя извне.

К пространствам имен относятся *перечисления*, *структуры* и *классы*, о которых мы говорили ранее. Именно их изучением мы и займемся в этой главе. Также к пространствам имен относятся модули, но их рассмотрение не является темой данной книги, мы познакомимся с ними лишь поверхностно. Вообще, модули — это верхний уровень пространств имен. В простейшем варианте ваша программа — это модуль, а значит, это отдельное пространство имен, именем которого является название вашего приложения. Также модулями являются различные фреймворки. С одним из них, кстати, мы уже работали, когда выполняли операцию импорта: `import Foundation`.

Этот фреймворк называется Cocoa's Foundation Framework и содержит большое количество функциональных механизмов, позволяющих расширить возможности ваших программ.

Одни пространства имен могут включать другие: так, модуль UIKit, ориентированный на разработку iOS-приложений, в своем коде выполняет импорт модуля Cocoa's Foundation Framework.

14.3. API Design Guidelines

Одной из главных проблем предыдущих версий Swift была нестандартизированность и неоднозначность написания имен функциональных элементов. Каждый разработчик сам определял, как он хочет называть создаваемые структуры, классы, перечисления, переменные и т. д. С одной стороны, это, конечно, хорошо, но если в своем проекте вы использовали библиотеки сторонних производителей, то синтаксис вашей программы мог превратиться в невятное месиво. А если еще библиотеки были написаны на Objective-C, то разработку вообще хотелось забросить, настолько неудобным могло стать использование Swift.

Но в третьей версии Swift был разработан и внедрен документ, определяющий правила именования любых элементов, будь то переменная, константа, функция, класс, перечисление, структура или что-то иное. Он получил название *Swift API Design Guidelines*.

Swift ADG — это своеобразная дорожная карта, собравшая правила, благодаря которым синтаксис языка стал четким, понятным и приятным. Когда вы достигнете определенного уровня в программировании и приступите к разработке собственных API-библиотек, то изучение приведенного в *API Design Guidelines* станет отличной базой для создания удобного и функционального продукта.

Хочу отметить, что между выходом второго и третьего изданий была проделана огромная работа, и все именно из-за Swift ADG: из-за него были внесены изменения практически в каждый элемент языка, а значит, практически в каждую главу, каждый раздел и листинг книги!

До разработки *Apple Design Guidelines Swift* был очень изменчив. Далее приведены некоторые наиболее важные правила.

1. Комментарии необходимо писать для каждого объявляемого экземпляра в вашем коде. Комментарии должны быть максимально полными.
2. Имена всех экземпляров должны быть ясными и краткими. Избегайте дублирования и избыточности. Избегайте пустых слов, не несущих смысловой нагрузки.
3. Четкость и ясность именования экземпляров важнее краткости.
4. Имена экземпляров должны исключать неоднозначность.
5. Именуйте экземпляры в соответствии с их ролью и предназначением в программе.
6. Именуйте экземпляры с использованием понятных и максимально простых фраз на английском языке.
7. Названия типов данных указывайте в верхнем камелкейс-стиле (*UpperCamelCase*).
8. Названия свойств, методов, переменных и констант указывайте в нижнем камелкейс-стиле (*camelCase*).

Используя этот небольшой набор правил, вы уже можете создавать приятные для чтения программы. Старайтесь! Включайте фантазию, и у вас все получится!

15

Перечисления

Перейдем к изучению механизмов создания объектных типов данных. Начнем с простейшего из них — перечисления. Это очень важная и интересная тема в Swift.

15.1. 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 15.1). Для того чтобы решить поставленную задачу с помощью изученных ранее типов данных, можно использовать тип `String`. В этом случае потребуется вести учет всех возможных значений для описания денежных единиц.

Листинг 15.1

```
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 потребуются добавить обработку нового значения денежной единицы?

Альтернативой этому способу может служить создание коллекции, например массива (листинг 15.2). Массив содержит все возможные значения, которые доступны в программе. При необходимости происходит получение требуемого элемента массива.

Листинг 15.2

```
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 camel кейс.

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 15.3). Перечисление подобно массиву. Оно

содержит список значений, одно из которых мы можем присвоить некоторому параметру.

Листинг 15.3

```
1 enum CurrencyUnit {
2     case rouble
3     case dollar
4     case euro
5 }
```

Несколько членов перечисления можно писать в одну строку через запятую (листинг 15.4).

Листинг 15.4

```
1 enum CurrencyUnit {
2     case rouble, dollar, euro
3 }
```

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. Далее доступ к значениям происходит уже без указания его имени.

Создадим несколько параметров, которые будут содержать значения созданного ранее перечисления (листинг 15.5).

Листинг 15.5

```
1 // способ 1
2 let roubleCurrency = CurrencyUnit.rouble           rouble
3 // способ 2
4 let dollarCurrency: CurrencyUnit
5     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"
```

Задание

1. Создайте перечисление с именем `Chessmen`, членами которого являются шахматные фигуры. Имена членов перечисления должны соответствовать названиям шахматных фигур на английском языке.
2. Объявите переменную и проинициализируйте ей значение одного из членов (произвольного) созданного ранее перечисления.

15.2. 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 описанного ранее перечисления `CurrencyUnit`, добавив параметры, с помощью которых можно указать список стран, в которых данная валюта используется, а также кратких наименований валюты (листинг 15.6).

Листинг 15.6

```
1 enum CurrencyUnit {
2     case rouble(countries: [String], shortName: String)
3     case dollar(countries: [String], shortName: String)
4     case euro(countries: [String], shortName: String)
5 }
```

Параметр `countries` является массивом, так как валюта может использоваться не в одной, а в нескольких странах: например, евро используется во всем Европейском союзе.

Теперь для того, чтобы создать переменную или константу типа `CurrencyUnit`, необходимо указать значения для всех ассоциированных параметров (листинг 15.7).

Листинг 15.7

```
1 var roubleCurrency: CurrencyUnit
2 roubleCurrency = .rouble(countries:      rouble(["Russia"], "RUB")
   ["Russia"], shortName: "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: США, Австралии, Канады 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 15.8).

Листинг 15.8

```
1 // страны, использующие доллар
2 enum DollarCountries {
3     case usa
4     case canada
5     case australia
6 }
7 // тип данных "валюта"
8 enum CurrencyUnit {
9     case rouble(countries: [String],
10                  shortName: String)
11     case dollar(countries: [String], shortName:
12                  String, national: DollarCountries)
13     case euro(countries: [String], shortName:
14                String)
15 }
16 var dollarCurrency: CurrencyUnit
17 dollarCurrency = .dollar(countries:      dollar(["USA"], "USD",
   ["USA"], shortName: "USD", national: DollarCountries.usa)
```

Для параметра `national` перечисления `CurrencyUnit` используется тип данных `DollarCountries`. При указании этого параметра его тип уже известен (задан при описании перечисления `CurrencyUnit`), поэтому его название можно опустить.

Задание

1. Создайте перечисление `Color`, содержащее два возможных цвета шахматных фигур — белый и черный.
2. Для каждого члена перечисления `Chessmen` определите ассоциированные параметры: название на русском и цвет. Значение параметра «цвет» должно использовать тип `Color`.
3. Объявите константу, содержащую в качестве значения член перечисления `Chessmen`.

15.3. Вложенные перечисления

Перечисления могут быть частью других перечислений, то есть могут быть определены в области видимости родительских перечислений. Так как перечисление `DollarCountries` используется исключительно в перечислении `CurrencyUnit` и создано для него, его можно перенести внутрь этого перечисления (листинг 15.9).

Листинг 15.9

```
1  // тип данных "валюта"
2  enum CurrencyUnit {
3      // страны, использующие доллар
4      enum DollarCountries {
5          case usa
6          case canada
7          case australia
8      }
9      case rouble(countries: [String], shortName: String)
10     case dollar(countries: [String], shortName: String, national:
        DollarCountries)
11     case euro(countries: [String], shortName: String)
12 }
```

Теперь перечисление `DollarCountries` обладает ограниченной областью видимости и доступно только через родительское перечисление. Можно сказать, что это подтип типа, или вложенный тип. Тем не менее при необходимости вы можете создать параметр, содержащий

значение этого перечисления, и вне перечисления `CurrencyUnit` (листинг 15.10).

Листинг 15.10

```
1 var someVar: CurrencyUnit.DollarCountries
2 someVar = .australia australia
```

Так как перечисление `DollarCountries` 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 `DollarCountries` в `CurrencyUnit` код продолжает работать, а Xcode дает корректные подсказки в окне автодополнения.

Задание

Перенесите перечисление `Color` в `Chessmen`.

15.4. Оператор `switch` для перечислений

Мы научились устанавливать в качестве значений переменных и констант члены перечисления и их ассоциированные параметры. Для анализа и разбора значений используется оператор `switch`.

Рассмотрим пример из листинга 15.11, в котором анализируется переменная, содержащая член перечисления в качестве значения.

Листинг 15.11

```
1 // создание переменной
2 var someCurrency = CurrencyUnit.rouble(countries: ["Russia",
3 "Ukraine", "Belarus"], shortName: "RUB")
4 // анализ переменной
5 switch someCurrency {
6     case .rouble :
7         print("Рубль.")
8     case let . euro (countries, shortname):
9         print("Евро. Страны: \(String(describing: countries)),
10 краткое наименование: \(shortname)")
11     case .dollar(let countries, let shortname, let national):
12         print("Доллар \(national). Страны: \(String(describing:
13 countries)),
14 краткое наименование: \(shortname) ")
15 }
```

Консоль:

Рубль.

В операторе `switch` описан каждый элемент перечисления `CurrencyUnit`, поэтому использовать оператор `default` не обязательно. Доступ к ассоциированным параметрам реализуется связыванием значений: после ключевого слова `case` и указания значения в скобках объявляются константы, которым будут присвоены ассоциированные с членом перечисления значения. Так как для всех ассоциированных параметров создаются константы со связываемым значением, оператор `let` можно ставить сразу после ключевого слова `case` (это продемонстрировано для члена `Euro`).

Обратите внимание на то, что при передаче в функцию `String` массива `countries` используется имя входного аргумента `describing`.

15.5. Связанные значения членов перечисления

Как альтернативу ассоциированным параметрам для членов перечислений им можно задать связанные значения некоторого типа данных (например, `String`, `Character` или `Int`). В результате вы получаете член перечисления и постоянно привязанное к нему значение.

ПРИМЕЧАНИЕ Связанные значения также называют исходными, или сырыми. Но в данном случае термин «связанные» значительно лучше отражает их предназначение.

Указание связанных значений

Для задания связанных значений членов перечисления необходимо указать тип данных самого перечисления, соответствующий значениям членов, и определить значения для каждого отдельного члена перечисления (листинг 15.12).

Листинг 15.12

```
1 enum Smile: String {
2     case joy = ":)"
3     case laugh = ":D"
4     case sorrow = ":("
5     case surprise = "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 15.13 представлено перечисление, содержащее список планет Солнечной системы в порядке удаленности от Солнца.

Листинг 15.13

```
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 объявленного ранее перечисления `Smile` и получим исходное значение установленного в этом экземпляре члена (листинг 15.14).

Листинг 15.14

```
1 var iAmHappy = Smile.Joy
2 iAmHappy.rawValue
```

joy
":.)"

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 15.15). В данном примере создается экземпляр, содержащий в себе указатель на третью планету от Солнца. Для этого служит созданное ранее перечисление `Planet`.

Листинг 15.15

```
1 var myPlanet = Planet.init(rawValue: 3)
2 var anotherPlanet = Planet.init(rawValue: 11)
```

earth
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 и закреплен в исходном коде языка Swift. Инициализатор `init(rawValue:)` возвращает опционал, поэтому если вы укажете несуществующее связанное значение, возвратится `nil`.

ПРИМЕЧАНИЕ Инициализаторы вызываются каждый раз при создании нового экземпляра какого-либо перечисления, структуры или класса. Для некоторых конструкций их можно и нужно создавать самостоятельно, а для некоторых, вроде перечислений, они существуют по умолчанию.

Инициализатор проводит процесс инициализации, то есть выполняет установку всех требуемых значений для параметров с непосредственным созданием экземпляра и помещением его в хранилище.

Инициализатор — это всегда метод с именем `init`. Но даже если вы не вызываете его напрямую, он все равно срабатывает.

С инициализаторами мы познакомимся подробнее в следующих главах книги.

Задание

1. Для перечисления `Color` укажите связанные значения, определяющие цвет на русском языке.
2. Определите переменную, содержащую произвольный член перечисления `Chessmen`.
3. С помощью оператора `switch` выведите ассоциированные значения созданной ранее переменной (для параметра `color` требуется вывести связанное значение).

15.6. Свойства в перечислениях

Благодаря перечислениям можно смоделировать ситуацию, в которой существует ограниченное количество исходов. У таких ситуаций помимо возможных результатов (членов перечисления) могут существовать и некоторые свойства.

Свойства позволяют хранить в перечислении вспомогательную информацию. Мы уже неоднократно встречались со свойствами в процессе изучения Swift.

Свойство в перечислении — это хранилище, аналогичное переменной или константе, объявленное в пределах перечисления и доступное через доступ к его экземпляру. 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 15.16). За основу возьмем перечисление `Smile` и создадим вычисляемое перечисление, которое возвращает связанное с текущим членом перечисления значение.

Листинг 15.16

```
1 enum Smile: String {
2     case joy = ":"}
```

```

3     case laugh = ":D"
4     case sorrow = ":("
5     case surprise = "o_o"
6     var description: String {return self.rawValue}
7 }
8 var mySmile: Smile = .sorrow
9 mySmile.description

```

Вычисляемое свойство должно быть объявлено как переменная (**var**). В противном случае (если используете оператор **let**) вы получите сообщение об ошибке.

С помощью оператора **self** вы получаете доступ к текущему члену перечисления. Данный оператор будет очень активно использоваться вами при разработке приложений. С ним мы познакомимся подробнее уже в ближайших разделах.

15.7. Методы в перечислениях

Перечисления могут группировать в себе не только свойства, члены и другие перечисления, но и методы. Ранее мы говорили об инициализаторах **init()**, которые являются встроенными в перечисления методами. *Методы* — это функции внутри перечислений, поэтому их синтаксис и возможности идентичны синтаксису и возможностям изученных ранее функций.

Вернемся к примеру с перечислением **Smile** и создадим метод, который выводит на консоль справочную информацию о предназначении перечисления (листинг 15.17).

Листинг 15.17

```

1  enum Smile: String {
2      case joy = ":"
3      case laugh = ":D"
4      case sorrow = ":("
5      case surprise = "o_o"
6      // метод для вывода описания
7      func description(){
8          print("Перечисление содержит список используемых смайликов:
9              их названия и графические обозначения")
10     }
11 }
12 var mySmile = Smile.joy
13 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.

15.8. 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 требуется написать два метода, один из которых будет возвращать сам член перечисления, а второй — его связанное значение. Используем для этого перечисление `Smile` (листинг 15.18).

Листинг 15.18

```

1  enum Smile: String {
2      case joy = ":"
3      case laugh = ":D"
4      case sorrow = ":("
5      case surprise = "o_o"
6      func description(){
7          print("Перечисление содержит список используемых смайлов:
            их названия и графические обозначения")
8      }
9      func descriptionValue() -> Smile{
10         return self
11     }
12     func descriptionRawValue() -> String{
13         return self.rawValue
14     }
15 }
16 var mySmile = Smile.joy
17 mySmile.descriptionValue()
18 mySmile.descriptionRawValue()
```

joy
joy
":"

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.

15.9. 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 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 15.19). Для простоты рассмотрим только две операции: сложение и вычитание.

Листинг 15.19

```
1  enum ArithmeticExpression{
2      // операция сложения
3      case addition(Int, Int)
4      // операция вычитания
5      case subtraction(Int, Int)
6  }
7  var expr = ArithmeticExpression.addition(10, 14)    addition(10, 14)
```

Данное перечисление имеет два члена, как и положено в арифметических выражениях: арифметические операции в Swift — это бинарные операторы, то есть операторы, проводящие операцию с двумя операндами. В связи с этим каждая из операций может принимать два ассоциированных параметра.

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 метод, который определяет наименование члена и возвращает результат данной операции (листинг 15.20).

Листинг 15.20

```

1  enum ArithmeticExpression{
2      case addition(Int, Int)
3      case subtraction(Int, Int)
4      // метод подсчета
5      func evaluate() -> Int {
6          switch self{
7              case .addition(let left, let right):
8                  return left + right
9              case .subtraction(let left, let right):
10                 return left - right
11          }
12      }
13 }
14 var expr = ArithmeticExpression.Addition(10, 14)  Addition(10, 14)
15 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 значения типа `ArithmeticExpression`.

Получается, что перечисление, описывающее выражение, должно давать возможность выполнять операции само с собой. Данный механизм реализуется в *рекурсивном перечислении*. Для того чтобы разрешить членам перечисления обращаться к этому перечислению, используется ключевое слово `indirect`, которое ставится:

- ❑ либо перед оператором `enum` — в этом случае каждый член перечисления может обратиться к данному перечислению;
- ❑ либо перед оператором `case` того члена, в котором необходимо обратиться к перечислению.

Если в качестве ассоциированных параметров перечисления указывать значения типа самого перечисления `ArithmeticExpression`, то возникает вопрос: а где же хранить числа, над которыми совершаются операции? Такие числа также необходимо хранить в самом перечислении, в его отдельном члене.

Рассмотрим пример из листинга 15.21. В данном примере вычисляется значение выражения $20 + 10 - 34$.

Листинг 15.21

```

1  enum ArithmeticExpression {
2      // хранилище для операнда
3      case number(Int)
4      indirect case addition(ArithmeticExpression,
5                             ArithmeticExpression)
6      indirect case subtraction(ArithmeticExpression,
7                                ArithmeticExpression)
9      func evaluate(expression: ArithmeticExpression? =
10         nil) -> Int{
11         let expression = (expression == nil ? self : expression)
12         switch expression! {
13             case .number( let value ):
14                 return value
15             case .addition( let valueLeft, let valueRight ):
16                 return self.evaluate( expression: valueLeft ) +
17                     self.evaluate( expression: valueRight )
18             case .subtraction( let valueLeft, let valueRight ):
19                 return self.evaluate( expression: valueLeft ) -
20                     self.evaluate( expression: valueRight )
21         }
22     }
23 }
24 var expr = ArithmeticExpression.addition( .number(20),
25     .subtraction( .number(10), .number(34) ) )
26 expr.evaluate()
```

-4

Здесь у перечисления появился новый член `number`. Он определяет целое число — операнд для проведения операции. Для членов арифметических операций использовано ключевое слово `indirect`, позволяющее передать значение типа `ArithmeticExpression` в качестве ассоциированного параметра.

Метод `evaluate(expression:)` принимает на входе опциональное значение типа `ArithmeticExpression?`. Опционал в данном случае позволяет вызвать метод, не передавая ему экземпляр, из которого этот метод был вызван. В противном случае последняя строка листинга выглядела бы следующим образом:

```
expr.evaluate(expression: expr)
```

Согласитесь, что существующий вариант значительно удобнее.

Оператор `switch`, используя принудительное извлечение, определяет, какой член перечисления передан, и возвращает соответствующее значение.

В результате данное перечисление позволяет смоделировать любую операцию, в которой присутствуют операторы сложения и вычитания. Перечисления в Swift мощнее, чем аналогичные механизмы в других языках программирования. Вы можете создавать свойства и методы, применять к ним расширения и протоколы, а также делать многое другое. Обо всем этом мы вскоре поговорим.

Задание

Допишите перечисление `ArithmeticExpression` таким образом, чтобы оно могло реализовать любое выражение с использованием операций сложения, вычитания, умножения, деления и возведения в степень.

16

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 разработчиками Swift. Все фундаментальные типы данных — коллекции — это структуры. Структуры в некоторой степени похожи на перечисления и во многом сходны с классами (с ними мы познакомимся в скором времени).

16.1. 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 16.1). Доступ к значениям будет осуществляться через элементы кортежа.

Листинг 16.1

```
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 как называются, я просто не представляю.

Соответственно нам нужен механизм, служащий своего рода 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 камелкейс.

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 16.2).

Листинг 16.2

```
1 struct PlayerInChess {  
2     var oleg: PlayerInChess
```

Так как структура `PlayerInChess` является типом данных, то можно объявить новое хранилище (переменную) данного типа.

16.2. 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 показанным ранее кортежем (листинг 16.3).

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 16.3

```

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 `init`, который не требуется объявлять. Данный инициализатор принимает на входе значения всех свойств структуры, производит их инициализацию и создает экземпляр структуры (листинг 16.4).

Листинг 16.4

```

1 var oleg = PlayerInChess(name: "Олег", wins: 32)

```

В результате создается новый экземпляр структуры `PlayerInChess`, содержащий свойства с определенными значениями.

ВНИМАНИЕ Значения свойств обязательно должны быть 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. 16.1).

На рис. 16.1 изображены два разных инициализатора, доступных при создании экземпляра: один не требует указывать значения свойств, поскольку использует их значения по умолчанию, другой, наоборот,

требуется указать эти значения. Инициализатор, который не требует указывать какие-либо значения, называется *пустым инициализатором*.

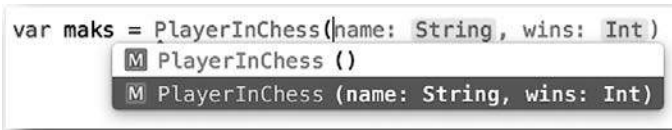


Рис. 16.1. Два инициализатора в окне автодополнения

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 позволяет определять значения по умолчанию лишь для некоторых свойств только в случае создания собственного инициализатора. Об этом мы поговорим чуть позже.

Объявим значения по умолчанию для структуры `PlayerInChess` (листинг 16.5).

Листинг 16.5

```
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.

16.3. Структура как пространство имен

Структура образует отдельное пространство имен, поэтому для доступа к элементам этого пространства имен необходимо в первую очередь получить доступ к самому пространству.

В предыдущем примере была создана структура `PlayerInChess` с двумя свойствами. Каждое из свойств имеет некоторое значение, но от них не будет никакого толка, если не описать механизмы доступа к данным свойствам.

Доступ к элементам структур происходит с помощью экземпляров данной структуры (листинг 16.6).

Листинг 16.6

```

1 struct PlayerInChess {
2     var name: String = "Игрок"
3     var wins: UInt = 0
4 }
5 var oleg = PlayerInChess( )
6 // доступ к свойству
7 oleg.name
```

Игрок

Данный способ доступа обеспечивает не только чтение, но и изменение значения свойства экземпляра структуры (листинг 16.7).

Листинг 16.7

```

1 var ilya = PlayerInChess(name: "Илья", wins: 32)
2 ilya.wins
3 ilya.wins = 33
4 ilya.wins
```

32
33
33

ВНИМАНИЕ Если свойство структуры или ее экземпляр указаны как константа (let), при попытке изменения значения свойства Xcode сообщит об ошибке.

16.4. 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. Несмотря на то что инициализатор — метод, он объявляется без использования ключевого слова `func`. При этом одна структура может содержать произвольное количество инициализаторов, каждый из которых должен иметь уникальный набор входных параметров.

Создадим инициализатор для структуры `PlayerInChess`, который принимает значение только для свойства `name` (листинг 16.8).

Листинг 16.8

```
1 struct PlayerInChess {  
2     var name: String  
3     var wins: UInt = 0  
4     init(name: String){  
5         self.name = name  
6     }  
7 }  
8 var oleg = PlayerInChess(name: "Олег")
```

Как вы можете видеть, нами был разработан такой инициализатор, после использования которого все свойства экземпляра будут иметь значения, несмотря на то что значения по умолчанию заданы не для всех свойств. При создании нового экземпляра вам будет доступен только разработанный вами инициализатор. В него передается значение параметра `name`.

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 они уже имеют встроенные инициализаторы.

ВНИМАНИЕ Повторю: если экземпляр структуры хранится в константе, модификация его свойств невозможна. Если же он хранится в переменной, то возможна модификация тех свойств, которые объявлены с помощью оператора `var`.

ВНИМАНИЕ Структуры — это типы-значения. При передаче экземпляра структуры от одного параметра в другой происходит его копирование. В следующем примере создаются два независимых экземпляра одной и той же структуры:

```
var olegMuhin = PlayerInChess(name: "Олег")  
var olegLapin = olegMuhin
```

Задание

1. Объявите перечисление `Color`, содержащее члены, описывающие возможные цвета шахматных фигур. Члены перечисления должны иметь связанные значения.

2. Объявите перечисление `ChessFigures`, содержащее все возможные типы шахматных фигур. Члены перечисления должны иметь связанные значения.
3. Объявите структуру `Chessmen`, описывающую шахматную фигуру. Эта структура должна содержать следующие свойства: тип фигуры (тип `ChessFigures`), цвет фигуры (тип `Color`), координаты на игровой площадке (тип `(String, UInt)?`).
4. Подумайте, почему координаты фигуры указаны в виде опционала? Причина в том, что фигура может находиться на игровой площадке, а может быть убрана с нее (или еще не размещена на ней).
5. В структуре `Chessmen` создайте инициализатор, который принимает значения для свойств «тип» и «цвет». Координаты по умолчанию должны устанавливаться в `nil`.

16.5. Методы в структурах

Объявление методов

Помимо свойств структуры, как и перечисления, могут содержать методы. Синтаксис объявления методов в структурах аналогичен объявлению методов в перечислениях. Они, как и обычные функции, могут принимать входные параметры. Для доступа к собственным свойствам структуры используется оператор `self`.

Напишем метод, который выводит справочную информацию об игроке в шахматы на консоль (листинг 16.9).

Листинг 16.9

```
1 struct PlayerInChess {
2     var name: String
3     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 16.10).

Листинг 16.10

```
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.

Задание

1. В структуру `Chessmen` добавьте метод `setCoordinates(char:num:)`, позволяющий установить новые координаты фигуры.
2. В структуру `Chessmen` добавьте метод `kill()`, позволяющий убрать фигуру с игровой площадки (поставить координаты в `nil`).
3. Создайте фигуру «Белый слон». Установите ее координаты как `A1`, после чего уберите фигуру с игровой площадки.

17

Классы

Классы являются наиболее функциональными конструкциями в разработке приложений. Данная глава призвана познакомить вас с их замечательными возможностями. Если вы ранее разрабатывали приложения на других языках программирования, то, возможно, уже осведомлены о классах. В этом случае имеющийся опыт пригодится вам при их изучении в Swift.

Классы 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–subclass» друг к другу. При этом, в отличие от наборов (`sets`), где супернабор включает все элементы субнабора, здесь ситуация обратная: subclass 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.

17.1. 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 камелкейс-стиле.

Тело класса заключается в фигурные скобки и может содержать методы и свойства, а также другие элементы, с которым мы еще не знакомы.

ПРИМЕЧАНИЕ В момент объявления нового класса, как и при объявлении перечисления или структуры, в программе создается новый тип данных.

17.2. 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 определенными в классе типами 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 каж-

дое свойство должно иметь определенное значение, а любое свойство, для которого не указано значение по умолчанию, должно получать значение с помощью разработанного вами инициализатора.

При выполнении заданий в предыдущих главах мы описывали сущность «шахматная фигура» вначале с помощью перечисления `Chessmen`, а потом с помощью структуры `Chessmen`. Использование класса для 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 17.1).

Листинг 17.1

```

1  class Chessman {
2      let type: String
3      let color: String
4      var coordinates: (String, Int)? = nil
5      let figureSymbol: Character
6      init(type: String, color: String, figure: Character){
7          self.type = type
8          self.color = color
9          self.figureSymbol = figure
10     }
11 }
12 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 17.2). Созданные перечисления должны найти место в качестве типов соответствующих свойств класса `Chessman`. Не забывайте, что и входные аргументы инициализатора должны измениться соответствующим образом.

Листинг 17.2

```

1  // типы фигур
2  enum ChessmanType {
3      case king
4      case castle
5      case bishop
6      case pawn
7      case knight
8      case queen
9  }
10 // цвета фигур
11 enum ChessmanColor {
12     case black
13     case white
14 }
15 class Chessman {
16     let type: ChessmanType
17     let color: ChessmanColor
18     var coordinates: (String, Int)? = nil
19     let figureSymbol: Character
20     init(type: ChessmanType, color: ChessmanColor, figure:
        Character){
21         self.type = type
22         self.color = color
23         self.figureSymbol = figure
24     }
25 }
26 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.

17.3. 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 17.3). 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 17.3

```
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,
7         figure: Character){
8         self.type = type
9         self.color = color
10        self.figureSymbol = figure
11    }
12    // метод установки координат
13    func setCoordinates(char c:String, num n: Int){
14        self.coordinates = (c, n)
15    }
16    // 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.

17.4. 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 выражением создать фигуру и выставить ее на поле. Сейчас для этого используются два независимых выражения. Давайте разработаем второй инициализатор, который будет дополнительно принимать координаты фигуры (листинг 17.4).

Листинг 17.4

```

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:
7          Character){
8          self.type = type
9          self.color = color
10         self.figureSymbol = figure
11     }
12     init(type: ChessmanType, color: ChessmanColor, figure:
13         Character, coordinates: (String, Int)){
14         self.type = type
15         self.color = color
16         self.figureSymbol = figure
17         self.setCoordinates(char: coordinates.0, num:
18             coordinates.1)

```

```

16     }
17     func setCoordinates(char c:String, num n: Int){
18         self.coordinates = (c, n)
19     }
20     func kill(){
21         self.coordinates = nil
22     }
23 }
24 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, объявленные 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.

17.5. 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 17.5).

Листинг 17.5

```

1 class Chessman {
2     enum ChessmanType {
3         case king
4         case castle
5         case bishop
6         case pawn
7         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:
19         Character){
20         self.type = type
21         self.color = color
22         self.figureSymbol = figure
23     }
24     init(type: ChessmanType, color: ChessmanColor, figure:
25         Character, coordinates: (String, Int)){
26         self.type = type
27         self.color = color
28         self.figureSymbol = figure
29         self.setCoordinates(char: coordinates.0, num:
30             coordinates.1)
31     }
32     func setCoordinates(char c:String, num n: Int){
33         self.coordinates = (c, n)
34     }
35 }

```

Структуры `ChessmanColor` и `ChessmanType` теперь являются вложенными в класс `Chessman` и существуют только в пределах области видимости данного класса.

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 17.6). В этом примере мы получаем доступ к одному из членов перечисления `ChessmanType`, объявленного в контексте класса `Chessman`.

Листинг 17.6

```

1 var linkToEnumType = Chessman.ChessmanType.Bishop

```

18

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.

18.1. 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.

Ранее 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 18.1).

Листинг 18.1

```
1 class AboutMan{
2     let firstName = "Ероп"
3     var 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 требуется.

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.

Ленивые свойства являются `lazy-by-need`, то есть вычисляются однажды и больше не меняют своего значения. Это продемонстрировано в листинге 18.2.

Листинг 18.2

```
1 var Me = AboutMan()  
2 Me.wholeName  
3 Me.secondName = "Иванов"  
4 Me.wholeName
```

*Егор Петров**Егор Петров***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 имяСвойства: ТипДанных { тело_замыкающего_выражения }
```

Вычисляемые свойства могут храниться исключительно в переменных (var). После указания имени объявляемого свойства и типа возвращаемого замыкающим выражением значения без оператора присваивания указывается замыкание, в результате которого должно быть сгенерировано возвращаемое свойством значение.

Для того чтобы свойство возвращало некоторое значение, в теле замыкания должен присутствовать оператор return.

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 18.3).

Листинг 18.3

```
1 class AboutMan{  
2     let firstName = "Ероп"  
3     var 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, за тем лишь исключением, что оно будет вычислять результат каждый раз при доступе к нему (листинг 18.4).

Листинг 18.4

```

1 var Me = AboutMan()
2 Me.wholeName
3 Me.secondName = "Иванов"
4 Me.wholeName

```

*Егор Петров**Егор Иванов*

18.2. 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 18.5).

Листинг 18.5

```
1  struct Circle{
2      var coordinates: (x: Int, y: Int)
3      var radius: Float
4      var perimeter: Float {
5          get{
6              return 2.0*3.14*self.radius
7          }
8          set(newPerimeter){
9              self.radius = newPerimeter / (2*3.14)
10         }
11     }
12 }
13 var myNewCircle = Circle(coordinates: (0,0), radius: 10)
14 myNewCircle.perimeter // выводит 62.8
15 myNewCircle.perimeter = 31.4
16 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 18.6.

Листинг 18.6

```
1  struct Circle{
2      var coordinates: (x: Int, y: Int)
3      var radius: Float
4      var perimeter: Float {
```

```

5      get{
6          return 2.0*3.14*self.radius
7      }
8      set{
9          self.radius = newValue / (2*3.14)
10     }
11 }
12 }

```

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 18.7).

Листинг 18.7

```

1  struct Circle{
2      var coordinates: (x: Int, y: Int)
3      var radius: Float {
4          didSet (newValueOfRadius) {
5              print("Вместо значения \(self.radius) устанавливается
6                  значение \(newValueOfRadius)")
7          }
8          didSet (oldValueOfRadius) {
9              print("Вместо значения \(oldValueOfRadius) установлено
10                 значение \(self.radius)")
11          }
12      }
13      var perimeter: Float {
14          get{
15              return 2.0*3.14*self.radius
16          }
17          set{
18              self.radius = newValue / (2*3.14)
19          }
20      }
21  }
22  var myNewCircle = Circle(coordinates: (0,0), radius: 10)
23  myNewCircle.perimeter // выведет 62.8
24  myNewCircle.perimeter = 31.4
25  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.

18.3. 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 SomeEnumeration{
    static var storedTypeProperty = "Some value"
    static var computedTypeProperty: Int {
        return 2
    }
}
class SomeClass{
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 3
    }
    class var overrideableComputedTypeProperty: Int {
        return 4
    }
}
```

Свойства типа объявляются с использованием ключевого слова `static` для 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 18.8). Класс `AudioChannel` моделирует аудиоканал, у которого есть два параметра:

- ❑ максимально возможная громкость ограничена для всех каналов в целом;
- ❑ текущая громкость ограничена максимальной громкостью.

Листинг 18.8

```
1 struct AudioChannel {
2     static var maxVolume = 5
3     var volume: Int {
4         didSet {
5             if volume > AudioChannel.maxVolume {
6                 volume = AudioChannel.maxVolume
7             }
8         }
9     }
10 }
11 var LeftChannel = AudioChannel(volume: 2)
12 var RightChannel = AudioChannel(volume: 3)
13 RightChannel.volume = 6
14 RightChannel.volume // выводит 5
15 AudioChannel.maxVolume // выводит 5
16 AudioChannel.maxVolume = 10
17 AudioChannel.maxVolume // выводит 10
18 RightChannel.volume = 8
19 RightChannel.volume // выводит 8
```

Мы использовали тип `AudioChannel` для создания двух каналов: левого и правого. Свойству `volume` не удастся 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.

19

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.

19.1. 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.

19.2. 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 19.1). 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 19.1

```
1 class gameDesk {
2     var desk: [Int:[String:Chessman]] = [:]
3     init(){
4         for i in 1...8 {
5             desk[i] = [:]
6         }
7     }
8     func setChessman(chess: Chessman, coordinates: (String, Int)){
9         if let oldCoordinates = chess.coordinates {
10             desk[oldCoordinates.1]![oldCoordinates.0] = nil
11         }
12         self.desk[coordinates.1]![coordinates.0] = chess
13         chess.setCoordinates(char: coordinates.0, num:
            coordinates.1)
14     }
15 }
16 var game = gameDesk()
17 var QueenBlack = Chessman(type: .queen, color: .black, figure:
    "♛", coordinates: ("A", 6))
18 game.setChessman(chess: QueenBlack, coordinates: ("B",2))
19 game.setChessman(chess: 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 19.2). 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 19.2

```
1 class gameDesk {
2     var desk: [Int:[String:Chessman]] = [:]
3     init(){
4         for i in 1...8 {
5             desk[i] = [:]
6         }
7     }
8     subscript(alpha: String, number: Int) -> Chessman? {
9         get{
10             if let chessman = self.desk[number][alpha] {
11                 return chessman
12             }
13             return nil
14         }
15     }
16     func setChessman(chess: Chessman, coordinates: (String, Int)){
17         if let oldCoordinates = chess.coordinates {
18             desk[oldCoordinates.1][oldCoordinates.0] = nil
19         }
20         self.desk[coordinates.1][coordinates.0] = chess
21         chess.setCoordinates(char: coordinates.0, num:
            coordinates.1)
22     }
23 }
24 var game = gameDesk()
```

```

25 var QueenBlack = Chessman(type: .queen, color: .black, figure:
    "♚", coordinates: ("A", 6))
26 game.setChessman(chess: QueenBlack, coordinates: ("A",3))
27 game["A",3]?.coordinates    (.0 "A", .1 3)

```

Реализованный сабскрипт имеет только геттер, причем в данном случае можно было использовать краткий синтаксис записи (без ключевого слова `get`).

Так как сабскрипт возвращает опционал, перед доступом к свойству `coordinates` возвращенной шахматной фигуры необходимо выполнить извлечение опционального значения.

Теперь мы имеем возможность установки фигур на шахматную доску с помощью метода `setChessman(_: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 19.3).

Листинг 19.3

```

1  class gameDesk {
2      var desk: [Int:[String:Chessman]] = [:]
3      init(){
4          for i in 1...8 {
5              desk[i] = [:]
6          }
7      }
8      subscript(alpha: String, number: Int) -> Chessman? {
9          get{
10             if let chessman = self.desk[number]![alpha] {
11                 return chessman
12             }
13             return nil
14         }
15         set{
16             self.setChessman(chess: newValue!, coordinates: (alpha,
17                 number))
18         }
19     }
20     func setChessman(chess: Chessman, coordinates: (String, Int)){
21         if let oldCoordinates = chess.coordinates {
22             desk[oldCoordinates.1]![oldCoordinates.0] = nil
23         }
24         self.desk[coordinates.1]![coordinates.0] = chess
25         chess.setCoordinates(char: coordinates.0, num:
26             coordinates.1)
27     }
28 }

```

```

27 var game = gameDesk()
28 var QueenBlack = Chessman(type: .queen, color: .black, figure:
   "♛", coordinates: ("A", 6))
29 game.setChessman(chess: QueenBlack, coordinates: ("A",3))
30 game["C",5] = QueenBlack
31 QueenBlack.coordinates                (.0 "C", .1 5)

```

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

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.

20 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).

20.1. 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 20.1). 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 20.1

```
1  // суперкласс
2  class Quadruped {
3      var type = ""
4      var name = ""
5      func walk(){
6          print("walk")
7      }
8  }
9  // подкласс
10 class Dog: Quadruped {
11     func bark(){
12         print("woof")
13     }
14 }
15 var dog = Dog()
16 dog.type = "dog"
17 dog.walk() // выводит walk
18 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 20.2).

Листинг 20.2

```

1  // подкласс
2  class Dog: Quadruped {
3      func bark(){
4          print("woof")
5      }
6      func printName(){
7          print(self.name)
8      }
9  }
10 var dog = Dog()
11 dog.name = "Dragon Wan Helsing"
12 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.

20.2. 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 20.3).

Листинг 20.3

```

1  class NoisyDog: Dog{
2      override func bark(){
3          print ("woof")

```

```

4         print ("woof")
5         print ("woof")
6     }
7 }
8
9 var badDog = NoisyDog()
10 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 20.4).

Листинг 20.4

```

1 class NoisyDog: Dog{
2     override func bark(){
3         for _ in 1...3 {
4             super.bark()
5         }
6     }
7 }
8 var badDog = NoisyDog()
9 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 20.5).

Листинг 20.5

```
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.

20.3. 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.

20.4. Подмена экземпляров классов

Наследование, помимо всех перечисленных возможностей, позволяет заменять требуемые экземпляры определенного класса экземплярами одного из подклассов.

Рассмотрим пример из листинга 20.6. В нем объявим массив элементов типа `Quadruped` и добавим в него несколько элементов.

Листинг 20.6

```
1 var animalsArray: [Quadruped] = []
2 var someAnimal = Quadruped()
3 var myDog = Dog()
4 var badDog = NoisyDog()
5 animalsArray.append(someAnimal)
6 animalsArray.append(myDog)
7 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`.

20.5. 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 20.7).

Листинг 20.7

```
1 for item in animalsArray {
2     if item is Dog {
3         print("Yap")
4     }
5 }
6 // 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 `NoisyDog` (листинг 20.8).

Листинг 20.8

```
1  for item in animalsArray {
2      if var animal = item as? NoisyDog {
3          animal.bark()
4      }else if var animal = item as? Dog {
5          animal.bark()
6      }else{
7          item.walk()
8      }
9  }
```

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()`.

21 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.

21.1. Псевдоним Any

Благодаря псевдониму **Any** можно создавать хранилища неопределенного типа данных. Объявим массив, который может содержать элементы произвольных типов (листинг 21.1).

Листинг 21.1

```
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 21.2).

Листинг 21.2

```
1  for thing in things {
2      switch thing {
3          case let someInt as Int:
4              print("an integer value of \(someInt)")
5          case let someDouble as Double where someDouble > 0:
6              print("a positive double value of \(someDouble)")
7          case let someString as String:
8              print("a string value of \(someString)")
9          case let (x, y) as (Double, Double):
10             print("an (x, y) point at \(x), \(y)")
11          case let stringConverter as (String) -> String:
12             print(stringConverter("Troll"))
13          default:
14             print("something else")
15      }
16 }
```

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).

21.2. 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 21.3).

Листинг 21.3

```
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 21.4).

Листинг 21.4

```
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 21.5).

Листинг 21.5

```
1 for object in someObjects as! [Dog]{  
2     print("This is bad Dog")  
3 }
```

22 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.

22.1. 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, при-

нимающий на входе значение для наследуемого свойства `type` (листинг 22.1).

Листинг 22.1

```
1 class Dog: Quadruped {
2     override init(){
3         super.init()
4         self.type = "dog"
5     }
6     convenience init(text: String){
7         self.init()
8         print(text)
9     }
10    func bark(){
11        print("woof")
12    }
13    func printName(){
14        print(self.name)
15    }
16 }
17 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 инициализатор должен вызвать назначенный инициализатор того же объектного типа.
- ❑ Вспомогательный инициализатор в конечном счете должен вызвать назначенный инициализатор.

Иллюстрация всех трех правил представлена на рис. 22.1.

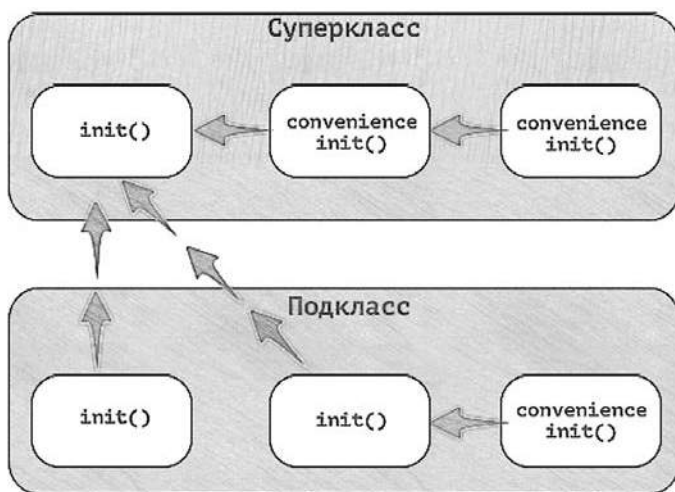


Рис. 22.1. Отношения между инициализаторами

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, вы-

званной 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 22.2).

Листинг 22.2

```
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         }  
10    }  
11 }  
12 var rectangle = Rectangle(height: 56, weight: -32) // возвращает nil
```

Инициализатор принимает и проверяет значения двух параметров. Если хотя бы одно из них не больше нуля, то возвращается `nil`. Обратите внимание на то, что, прежде чем вернуть `nil`, инициализатор устанавливает значения всех хранимых свойств.

ВНИМАНИЕ В классах проваливающийся инициализатор может вернуть `nil` только после установки значений всех хранимых свойств. В случае структур данное ограничение отсутствует.

Назначенный инициализатор в подклассе может переопределить проваливающийся инициализатор суперкласса, а проваливающийся инициализатор может вызывать назначенный инициализатор того же класса.

Не забывайте, что в случае использования проваливающегося инициализатора возвращается опционал. Поэтому прежде чем работать с экземпляром, необходимо выполнить извлечение опционального значения.

Вы можете использовать проваливающийся инициализатор для выбора подходящего члена перечисления, основываясь на значениях входных аргументов. Рассмотрим пример из листинга 22.3. 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 22.3

```

1  enum TemperatureUnit {
2      case Kelvin, Celsius, Fahrenheit
3      init?(symbol: Character) {
4          switch symbol {
5              case "K":
6                  self = .Kelvin
7              case "C":
8                  self = .Celsius
9              case "F":
10                 self = .Fahrenheit
11             default:
12                 return nil
13         }
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 22.4).

Листинг 22.4

```

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.

22.2. 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 22.5).

Листинг 22.5

```
1 class SuperClass {
2     init?(isNil: Bool){
3         if isNil == true {
4             return nil
5         }else{
6             print("Экземпляр создан")
7         }
8     }
9     deinit {
10        print("Деинициализатор суперкласса")
11    }
12 }
13 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.

23

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.

23.1. Уничтожение экземпляров

Как мы отмечали в предыдущей главе, непосредственно перед уничтожением экземпляра класса вызывается деинициализатор, при этом остался нерассмотренным вопрос о том, как удаляется экземпляр. Любой экземпляр может быть удален одним из двух способов:

- ❑ его самостоятельно уничтожает разработчик;
- ❑ его уничтожает Swift.

Область видимости

Ранее мы самостоятельно уничтожали созданный экземпляр опционального типа `SubClass?`, передавая ему в качестве значения `nil`. Теперь обратимся к логике работы Swift. Для этого разработаем класс `myClass`, который содержит единственное свойство `description`. Данное свойство служит для того, чтобы отличать один экземпляр класса от другого.

Необходимо создать два экземпляра, один из которых должен иметь ограниченную область видимости (листинг 23.1).

Листинг 23.1

```
1 class myClass{
2     var description: String
3     init(description: String){
4         print("Экземпляр \(description) создан")
5         self.description = description
6     }
7     deinit{
8         print("Экземпляр \(self.description) уничтожен")
9     }
10 }
11 var myVar1 = myClass(description: "ОДИН")
12 if true {
13     var myVar2 = myClass(description: "ДВА")
14 }
```

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 23.2).

Листинг 23.2

```
1 class myClass{
2     var description: String
3     init(description: String){
4         print("Экземпляр \(description) создан")
5         self.description = description
6     }
7     deinit{
8         print("Экземпляр \(self.description) уничтожен")
9     }
}
```

```

10 }
11 var myVar1 = myClass(description: "ОДИН")
12 var myVar2 = myVar1
13 myVar1 = myClass(description: "ДВА")
14 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.

23.2. 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 23.3).

Листинг 23.3

```

1 class Human {
2     let name: String
3     var child = [Human?]()
4     var father: Human?
5     var mother: Human?
6     init(name: String){
7         self.name = name
8     }
9     deinit {
10         print(self.name+" - удален")
11     }
12 }
13 if 1==1 {
14     var Kirill = Human(name: "Кирилл")

```

```
15     var Olga = Human(name: "Ольга")
16     var Aleks = Human(name: "Алексей")
17     Kirill.father = Aleks
18     Kirill.mother = Olga
19     Aleks.child.append(Kirill)
20     Olga.child.append(Kirill)
21 }
```

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 `owned`. 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 23.4).

Листинг 23.4

```

1  class Human {
2      let name: String
3      var child = [Human?]()
4      weak var father: Human?
5      weak var mother: Human?
6      init(name: String){
7          self.name = name
8      }
9      deinit {
10         print(self.name+" — удален")
11     }
12 }
13 if 1==1 {
14     var Kirill = Human(name: "Кирилл")
15     var Olga = Human(name: "Ольга")
16     var Aleks = Human(name: "Алексей")
17     Kirill.father = Aleks
18     Kirill.mother = Olga
19     Aleks.child.append(Kirill)
20     Olga.child.append(Kirill)
21 }

```

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.

23.3. 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 23.5 создается пустое опциональное замыкание, которому в зоне ограниченной области видимости передается значение (тело замыкания).

Листинг 23.5

```
1 class Human{
2     var name = "Человек"
3     deinit{
4         print("Объект удален")
5     }
6 }
7 var closure : (() -> ())?
8 if true{
9     var human = Human()
10    closure = {
11        print(human.name)
12    }
13    closure!()
14 }
15 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 23.6).

Листинг 23.6

```
1 class Human{
2     var name = "Человек"
3     deinit{
4         print("Объект удален")
5     }
6 }
7 var closure : (() -> ())?
8 if true{
9     var human = Human()
10    // измененное замыкание
11    closure = {
12        [unowned human] in
13        print(human.name)
14    }
15    closure!()
16 }
```

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.

24

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.

24.1. 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 24.1).

Листинг 24.1

```
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 24.2).

Листинг 24.2

```
1 var man = Person()
2 if let manResidence = man.residence {
3     print("Есть дом с \$(manResidence.numberOfRooms) комнатами")
4 }else{
5     print("Нет дома")
6 }
7 // 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 24.3).

Листинг 24.3

```
1 class Person {
2     var residence: Residence?
3 }
4 class Residence {
5     var numberOfRooms = 1
6     var room: Room?
7 }
8 class Room {
9     var square: Int = 10
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.

Продemonстрируем это на примере (листинг 24.4). В нем создается трехуровневая структура вложенности, в которой требуется получить доступ к свойству `room`. При этом оба свойства (и `residence`, и `room`) — опционалы.

Листинг 24.4

```
1 var man = Person()
2 var residence = Residence()
```

```
3 var room = Room()
4 man.residence = residence
5 residence.room = room
6 if let square = man.residence?.room?.square {
7     print("Площадь \(square) кв.м.")
8 }else{
9     print("Комнаты отсутствуют")
10 }
11 // 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.

24.2. 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 24.5). Пусть нам необходимо изменить значение свойства `square`.

Листинг 24.5

```
1 man.residence?.room?.square = 36
2 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.

24.3. 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 24.6).

Листинг 24.6

```
1 class Residence {
2     var numberOfRooms = 1
3     var room: Room?
4     func returnNumberOfRooms() {
5         return numberOfRooms
6     }
7 }
```

Для вызова данного метода требуется использовать опциональную последовательность (листинг 24.7).

Листинг 24.7

```
1 man.residence?.printNumberOfRooms()
2 // вывод количества комнат
```

Принцип доступа к методу точно такой же, как к свойству.

25

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.

25.1. 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 25.1) и обеспечив тип `Double` возможностью работы с единицами длины.

Листинг 25.1

```
1 extension Double {
2     var km: Double { return self * 1000.0 }
3     var m: Double { return self }
4     var cm: Double { return self / 100.0 }
5     var mm: Double { return self / 1000.0 }
6     var ft: Double { return self / 3.28084 }
7 }
8 let oneInch = 25.4.mm
9 print("Один фут — это \(oneInch) метра")
10 // выводит "Один фут — это 0.0254 метра"
11 let threeFeet = 3.ft
12 print("Три фута — это \(threeFeet) метра")
13 // 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.

25.2. 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 25.2). 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 25.2

```
1 import UIKit
2 // сущность "линия"
3 struct Line{
4     var pointOne: (Double, Double)
5     var pointTwo: (Double, Double)
6 }
7 // расширения для Double
8 extension Double {
9     init(line: Line){
10         self = sqrt(pow((line.pointTwo.0 - line.pointOne.0),2) +
11                     pow((line.pointTwo.1 - line.pointOne.1),2))
12     }
13 var myLine = Line(pointOne: (10,10), pointTwo: (14,10))
14 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.

25.3. Методы в расширениях

Следующей рассматриваемой функцией расширений является создание новых методов в расширяемом типе данных. Рассмотрим пример (листинг 25.3). 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 25.3

```
1 extension Int {
2     func repetitions(task: () -> ()) {
3         for _ in 0..
```

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 25.4).

Листинг 25.4

```
1 extension Int {
2     mutating func square() {
3         self = self * self
4     }
5 }
6 var someInt = 3
7 someInt.square()
```

25.4. Сабскрипты в расширениях

Помимо свойств, методов и инициализаторов, расширения позволяют создавать новые сабскрипты.

Создаваемое в листинге 25.5 расширение типа `Int` реализует новый сабскрипт, который позволяет получить определенную цифру собственного значения экземпляра. В сабскрипте указывается номер позиции числа, которое необходимо вернуть.

Листинг 25.5

```
1  extension Int {
2      subscript(var digitIndex: Int) -> Int {
3          var decimalBase = 1
4          while digitIndex > 0 {
5              decimalBase *= 10
6              -digitIndex
7          }
8          return (self / decimalBase) % 10
9      }
10 }
11 746381295[0]
12 // возвращает 5
13 746381295[1]
14 // возвращает 9
```

Если у числа отсутствует цифра с запрошенным индексом, возвращается 0, что не нарушает логику работы.

26

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. Другими словами, он содержит требования к наличию определенных элементов внутри типа данных. Протокол сам непосредственно не реализует какой-либо функционал, он лишь является своеобразным набором правил и требований к типу. Любой объектный тип данных может принимать протокол. Наиболее важной функцией протокола является обеспечение целостности объектных типов путем указания требований к их реализации.

Протоколы объявляются независимо от каких-либо элементов программы, так же как и объектные типы данных.

Синтаксис

```
protocol ИмяПротокола {  
    // тело протокола  
}
```

Для объявления протокола используется ключевое слово `protocol`, после которого указывается имя создаваемого протокола.

Для того чтобы принять протокол к исполнению каким-либо объектным типом, необходимо написать его имя через двоеточие сразу после имени реализуемого типа:

```
struct ИмяПринимающейСтруктуры: ИмяПротокола{  
    // тело структуры  
}
```

После указания имени протокола при объявлении объектного типа данный тип обязан выполнить все требования протокола. Вы можете указать произвольное количество принимаемых протоколов.

Если класс не только принимает протоколы, но и наследует некоторый класс, то имя суперкласса необходимо указать первым, а за ним через запятую — список протоколов:

```
class ИмяПринимающегоКласса: ИмяСуперКласса,  
    Протокол1, Протокол2{  
    // тело класса  
}
```

26.1. 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 26.1).

Листинг 26.1

```
1 protocol SomeProtocol {  
2     var mustBeSettable: Int { get set }  
3     var doesNotNeedToBeSettable: Int { get }  
4 }
```

Протокол `SomeProtocol` требует, чтобы в принимающем типе были реализованы два изменяемых (**var**) свойства типа `Int`. При этом свойство `mustBeSettable` должно быть и доступным 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 26.2).

Листинг 26.2

```
1 protocol AnotherProtocol {  
2     static var someTypeProperty: Int { get set }  
3 }
```

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 26.3).

Листинг 26.3

```
1 protocol FullyNamed {  
2     var fullName: String { get }  
3 }  
4 struct Person: FullyNamed {  
5     var fullName: String  
6 }  
7 let john = Person(fullName: "John Wick")
```

В данном примере определяется протокол `FullyNamed`, который обязывает структуру `Person` иметь доступное свойство `fullName` типа `String`.

26.2. 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 26.4).

Листинг 26.4

```

1 protocol RandomNumberGenerator {
2     func random() -> Double
3     static func getRandomString()
4     mutating func changeValue(newValue: Double)
5 }

```

В данном примере реализован протокол `RandomNumberGenerator`, который содержит требования реализации трех методов. Метод экземпляра `random()` должен возвращать значение типа `Double`. Метод `getRandomString()` должен быть методом типа, при этом требования к возвращаемому им значению не указаны. Метод `changeValue(_:)` должен быть изменяющим и должен принимать в качестве входного параметра значение типа `Double`.

Данный протокол не делает никаких предположений относительно того, как будет вычисляться случайное число, ему важен лишь факт выполнения требований.

26.3. 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 26.5).

Листинг 26.5

```

1 protocol Named{
2     init(name: String)
3 }
4 class Person : Named {

```

```

5     var name: String
6     required init(name: String){
7         self.name = name
8     }
9 }

```

26.4. 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.

26.5. 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 26.6).

Листинг 26.6

```

1 protocol TextRepresentable {
2     func asText() -> String
3 }
4 extension Int: TextRepresentable {
5     func asText() -> String {
6         return String(self)
7     }
8 }
9 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 26.7).

Листинг 26.7

```
1 extension TextRepresentable {  
2     func description() -> String {  
3         return "Данный тип поддерживает протокол TextRepresentable"  
4     }  
5 }  
6 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`.

26.6. Наследование протоколов

Протокол может наследовать один или более других протоколов. При этом он может добавлять новые требования поверх наследуемых, — тогда тип, принявший протокол к реализации, будет вынужден выполнить требования всех протоколов в структуре. При наследовании

протоколов используется тот же синтаксис, что и при наследовании классов.

Работа с наследуемыми протоколами продемонстрирована в листинге 26.8.

Листинг 26.8

```
1 protocol SuperProtocol {  
2     var someValue: Int { get }  
3 }  
4 protocol SubProtocol: SuperProtocol {  
5     func someMethod()  
6 }  
7 struct SomeStruct: SubProtocol {  
8     let someValue: Int = 10  
9     func someMethod() {  
10         // тело метода  
11     }  
12 }
```

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.

26.7. 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 26.9. В нем мы изменим протокол `SubProtocol` таким образом, чтобы его мог принять исключительно класс.

Листинг 26.9

```
1 protocol SubProtocol: class, SuperProtocol {  
2     func someMethod()  
3 }
```

26.8. Композиция протоколов

Иногда бывает удобно требовать, чтобы тип соответствовал не одному, а нескольким протоколам. В этом случае, конечно же, можно создать новый протокол, наследовать в него несколько необходимых протоколов и задействовать имя только что созданного протокола. Однако для решения данной задачи лучше воспользоваться *композицией протоколов*, то есть скомбинировать несколько протоколов.

Синтаксис

Протокол1 & Протокол2 ...

Для композиции протоколов необходимо указать имена входящих в данную композицию протоколов, разделив их оператором & (амперсанд).

В листинге 26.10 приведен пример, в котором два протокола комбинируются в единственное требование.

Листинг 26.10

```

1  protocol Named {
2      var name: String { get }
3  }
4  protocol Aged {
5      var age: Int { get }
6  }
7  struct Person: Named, Aged {
8      var name: String
9      var age: Int
10 }
11 func wishHappyBirthday(celebrator: Named & Aged) {
12     print("С Днем рождения, \(celebrator.name)! Тебе уже \(
13         celebrator.age)!")
14 }
15 let birthdayPerson = Person(name: "Джон Уик", age: 46)
16 wishHappyBirthday(celebrator: birthdayPerson)
17 // выводит "С Днем рождения, Джон Уик! Тебе уже 46!"

```

В данном примере объявляются два протокола: `Named` и `Aged`. Созданная структура принимает оба протокола и в полной мере выполняет их требования.

Входным аргументом функции `wishHappyBirthday(celebrator:)` должно быть значение, которое удовлетворяет обоим протоколам. Таким значением является экземпляр структуры `Person`, который мы и передаем.

27

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 в данной главе.

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.

27.1. Важность работы с документацией

Думаю, 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 «всё — это объект», и каждый раз этому приводились подтверждения, например, когда у обычного целого числа появлялись свойства и методы, позволяющие

работать с ним уже не просто как с числом, а как 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 `Collection`. 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 27.1).

Листинг 27.1

```
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.

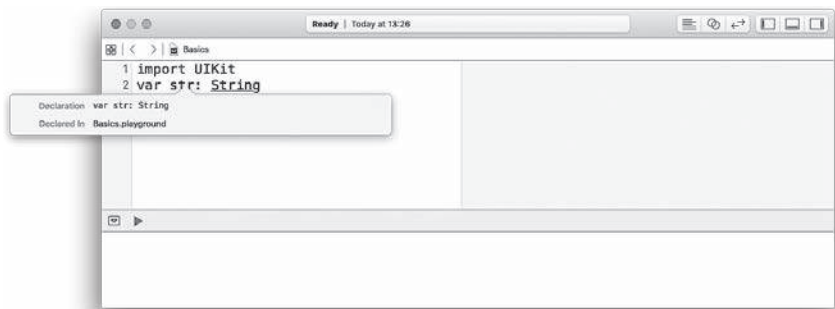


Рис. 27.1. Окно со справочной информацией о переменной

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. 27.1).

В строке **Declaration** щелкните на слове `String`, после чего Xcode откроет окно документации для типа данных `String` (рис. 27.2).

ПРИМЕЧАНИЕ Для загрузки документации необходим доступ в Интернет. Кроме того, в меню настроек Xcode вы можете загрузить документацию для автономного доступа к ней.

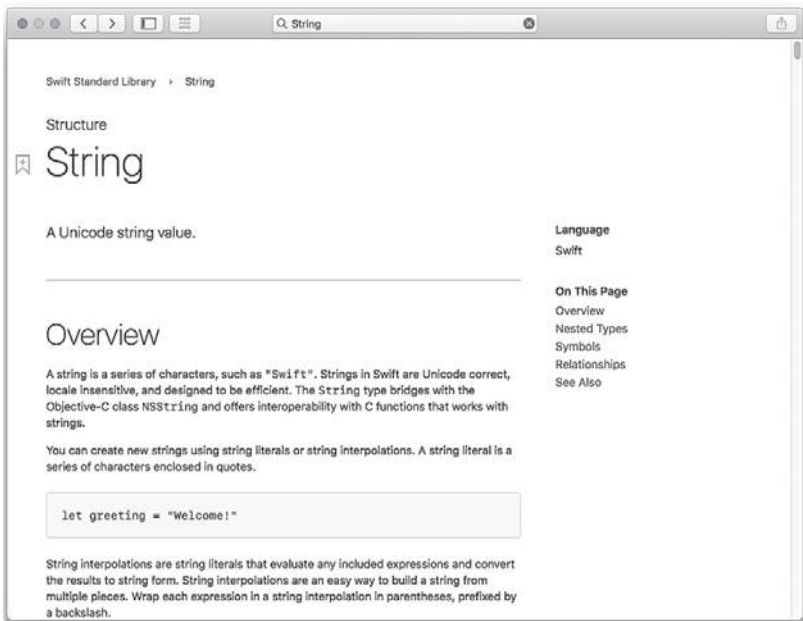


Рис. 27.2. Окно документации о типе данных `String`

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** (в самом низу страницы документации) 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 27.2).

Листинг 27.2

```
1 var countryDict = ["RUS": "Российская
    Федерация", "BEL": "Беларусь",
    "UKR": "Украина"]
2 // все ключи словаря countryDict
3 var keysOfDict = countryDict.keys
4 // все значения словаря countryDict
5 var valuesOfDict = countryDict.values
```

*["BEL": "Беларусь", "UKR":
"Украина", "RUS":
"Российская Федерация"]*

*LazyMapCollection
<Dictionary<String,
String>, String>*

*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 ленивую коллекцию, которая имеет тип данных:

`LazyMapCollection<Dictionary<String, String>, String>`

Мы уже работали с ней ранее, но рассмотрим, что сказано о ней в документации к языку. 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. 27.3).

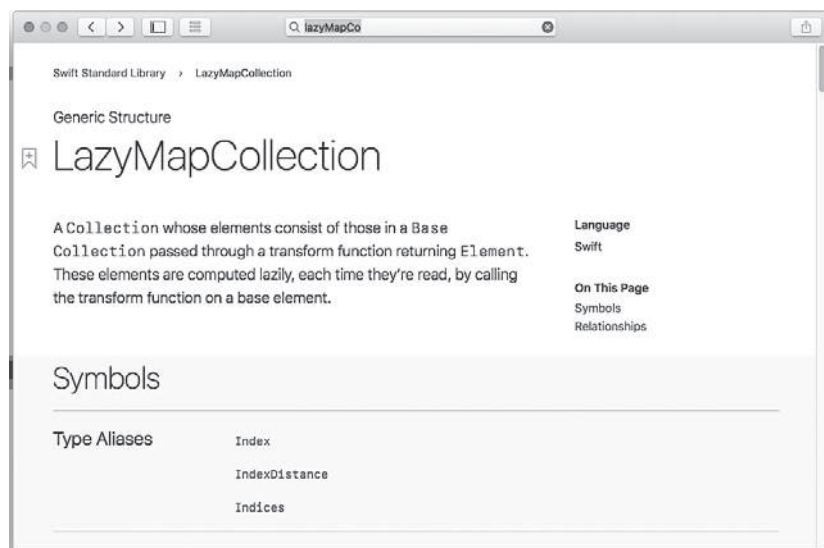
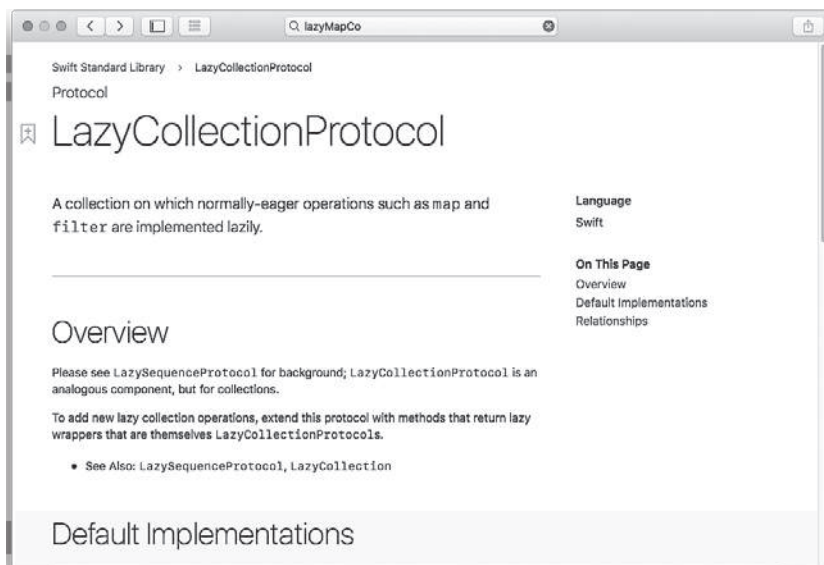
Слово `Collection` в названии типа говорит о том, что данный тип — это коллекция элементов.

Вверху страницы приведено полное наименование типа данных:

Если ознакомиться с описанием, которое расположено на странице документации, то становится ясно, что значение данного типа — не что иное, как коллекция, которая состоит из элементов базовой коллекции. Элементы данной коллекции преобразуются специальной функцией.

Обратимся к разделу `Conforms To`. В нем указан протокол `LazyCollectionProtocol`. Щелкните на этом имени, чтобы перейти к соответствующей странице документации (рис. 27.4).

Обратите внимание на раздел `Inherits From`: в нем указан протокол `Collection`, который наследуется в `LazyCollectionProtocol`.

Рис. 27.3. Документация для типа `LazyMapCollection`Рис. 27.4. Документация для протокола `LazyCollectionProtocol`

`Collection` — 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 `Collection` (рис. 27.5).

В разделе **Adopted By** представлены конструкции, которые выполняют требования данного протокола. Среди них присутствуют и `Dictionary`, и `Set`. Тип данных `Array` наследует протокол `Collection` не напрямую, а через протокол `MutableCollection`.

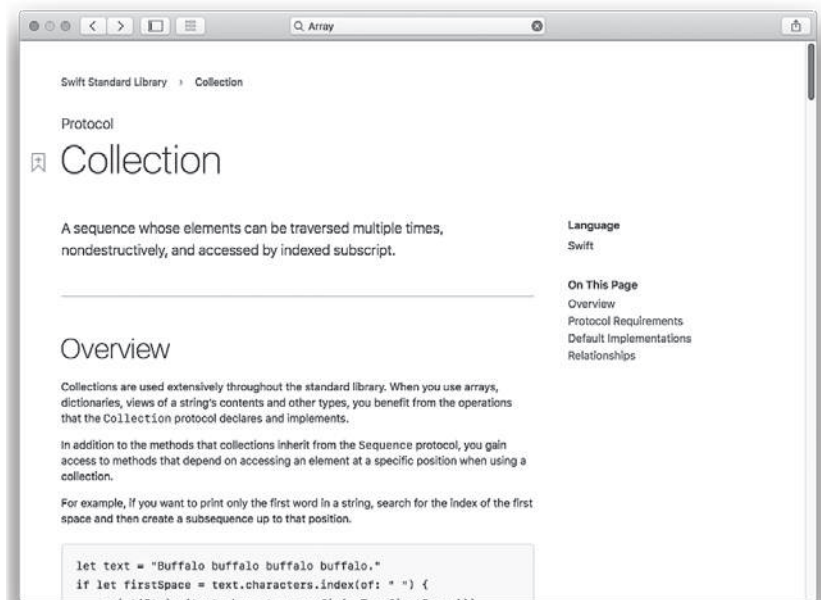


Рис. 27.5. Документация для протокола `Collection`

В результате получается незамысловатая схема зависимостей протоколов и типов данных, изображенная на рис. 27.6.

Получается, что конструкции `Array`, `Dictionary` и `Set`, как и конструкция `LazyMapCollection`, реализуют один и тот же протокол `Collection`, а значит, являются коллекциями.

Вернемся в окно кода Xcode и повторно вызовем справочную информацию о переменной `keysofDict`. Вы увидите следующие данные:

```
var keys: LazyMapCollection<Dictionary<String, String>, String>
```

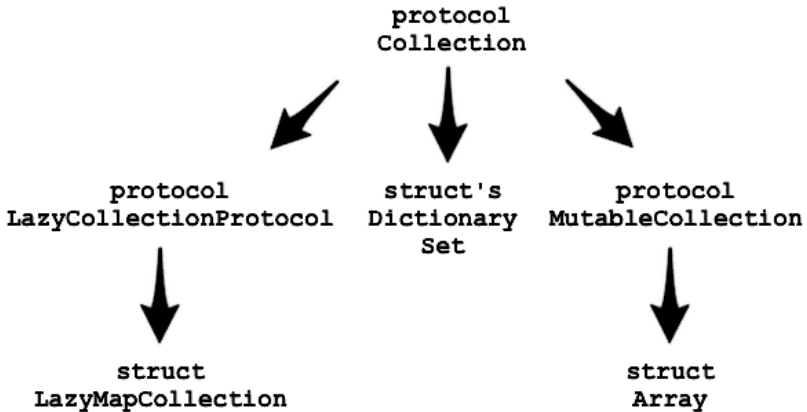


Рис. 27.6. Схема зависимостей протоколов и типов данных

Получается, что тип данных этой коллекции выглядит так:

`LazyMapCollection<Dictionary<String, String>, String>`

Исходя из описания `LazyMapCollection`, первый параметр (`Dictionary<String, String>`) описывает базовую коллекцию `countryDict`, а второй (`String`) — тип объектов в результирующей коллекции.

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 27.3).

Листинг 27.3

```

1 var countryDict = ["RUS": "Российская Федерация", "BEL": "Беларусь",
2   "UKR": "Украина"]
3 var keys = Set(countryDict.keys)
4 var values = Array(countryDict.values)
5
6 ["BEL": "Беларусь",
7  "UKR": "Украина", "RUS":
8  "Российская Федерация"]
9 {"BEL", "UKR", "RUS"}
10 ["Беларусь", "Украина",
11  "Российская Федерация"]
  
```

Тот же самый принцип можно применить и к результату работы метода `reversed()`, который возвращает реверсивный массив. Если посмотреть справку для примера из листинга 27.4, 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 27.4

```
1 var someArray = [1, 3, 5, 7, 9]           [1, 3, 5, 7, 9]
2 var reverseSomeArray =                   ReverseRandomAccess
    someArray.reversed()                   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 27.5).

Листинг 27.5

```
1 var someArray = [1, 3, 5, 7, 9]           [1, 3, 5, 7, 9]
2 let resArray: Array<Int> = someArray.reversed() [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 большую часть возникающих у вас вопросов.

27.2. 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, позволяющие настраивать рабочую зону Xcode (рис. 27.7).

Кнопки разбиты на две панели, по три кнопки в каждой. Левая панель отвечает за внешний вид рабочей зоны вашего окна и содержит следующие кнопки:



Рис. 27.7. Кнопки настройки рабочей зоны

❑ Standart Editor

Активна по умолчанию. Позволяет отобразить рабочую область в привычном нам виде: с редактором кода и областью результатов.

❑ 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. 27.8).

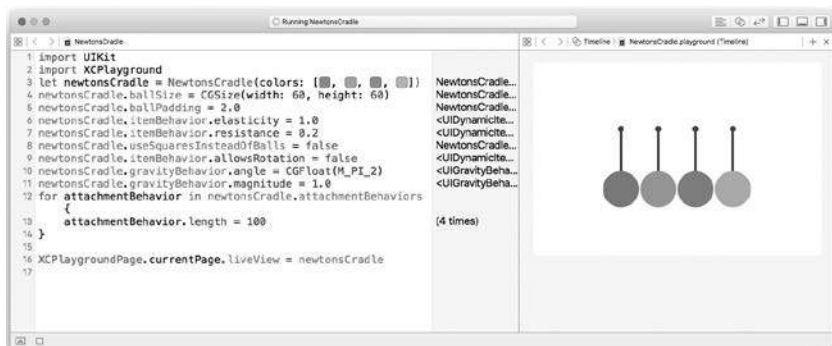


Рис. 27.8. Пример работы в режиме Assistant Editor

❑ 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. 27.9).



Рис. 27.9. Панель Project Navigator

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.

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 27.6.

Листинг 27.6

```
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.

27.3. Разграничение доступа

Уровни доступа

База системы разграничения доступа при разработке приложений строится на основе понятия «модуль».

Всего Swift предлагает пять различных уровня доступа для объектов вашего кода:

❑ `open` и `public`

Открытый и публичный. Данные уровни доступа очень похожи, они открывают полную свободу использования объекта. Вы можете импортировать модуль и свободно использовать его `public`-объекты в своем коде. Разница между `open` и `public` описывается далее.

❑ `internal`

Внутренний. Данный уровень используется в случаях, когда необходимо ограничить использование объекта самим модулем. Таким образом, объект будет доступен во всех исходных файлах модуля, исключая его использование за пределами модуля.

❑ `fileprivate`

Частный в пределах файла. Данный уровень позволяет использовать объект только в пределах данного исходного файла.

❑ `private`

Частный. Данный уровень позволяет использовать объект только в пределах конструкции, в которой он объявлен. Например, объявленный в классе параметр не будет доступен в его расширениях.

Открытый уровень доступа является самым высоким и наименее ограничивающим, а частный — самым низким и максимально ограничивающим.

Открытый уровень применяется только к классам и членам класса (свойствам, методам и т. д.) и отличается от публичного следующими характеристиками:

- ❑ Класс, имеющий уровень доступа `public` (или более строгий), может иметь подклассы только в том модуле, где он был объявлен.
- ❑ Члены класса, имеющие уровень доступа `public` (или более строгий), могут быть переопределены (с помощью оператора `override`) в подклассе только в том модуле, где он объявлен.
- ❑ Класс, имеющий уровень доступа `open`, может иметь подклассы внутри модуля, где он определен, и в модулях, импортированных в данном модуле.
- ❑ Члены класса, имеющие уровень доступа `open`, могут быть переопределены (с помощью оператора `override`) в подклассе в том модуле, где он объявлен, а также в модулях, импортируемых в данном модуле.

Главное правило определения уровня доступа в Swift звучит следующим образом: объект с более низким уровнем доступа не может определить объект с более высоким уровнем доступа (в контексте модуля).

По умолчанию все объекты вашего кода имеют уровень доступа `internal`. Для того чтобы изменить его, необходимо явно указать уровень. При этом если вы разрабатываете фреймворк, то для того, чтобы сделать некоторый объект частью доступного API, вам необходимо изменить его уровень доступа на `public`.

Синтаксис определения

Чтобы определить уровень доступа к некоторому объекту, необходимо указать соответствующее ключевое слово (`open`, `public`, `internal`, `private`) перед определением объекта (`func`, `property`, `class`, `struct` и т. д.). Пример приведен в листинге 27.7.

Листинг 27.7

```
1 open class SomePublicClass {}
2 internal class SomeInternalClass {}
3 fileprivate 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 27.8.

Листинг 27.8

```
1 public class SomePublicClass { // public класс
2     public var somePublicProperty = 0 // public свойство
3     var someInternalProperty = 0 // internal свойство
4     fileprivate func somePrivateMethod() {} // fileprivate 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 SomePrivateClass { // private класс
13     var somePrivateProperty = 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 результирующим уровнем доступа будет `private`, то есть самый строгий.

ПРИМЕЧАНИЕ Запомните, что Swift не позволяет явно указать тип данных кортежа. Он вычисляется автоматически.

Уровень доступа к функции определяется самым строгим уровнем типов аргументов функции и типа возвращаемого значения. Рассмотрим пример из листинга 27.9.

Листинг 27.9

```
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 27.10).

Листинг 27.10

```
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.

27.4. 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 шарiki и сталкивать их друг с другом.

Interactive Playgrounds, UIKit и PlaygroundSupport

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 PlaygroundSupport — модуль, обеспечивающий интерактивное взаимодействие пользователя с Xcode при работе в playground.

Наиболее важным механизмом данной библиотеки является класс PlaygroundPage, предназначенный для создания интерактивного содержимого страниц в playground-проекте. До выхода Swift 3 этими вопросами занимался модуль XCPlayground, но в Swift 3 он был отмечен как устаревший.

PlaygroundSupport позволяет взаимодействовать с некоторыми объектами. Поэтому нам потребуется функционал, обеспечивающий графическое построение этих объектов. И таким функционалом обладает модуль UIKit.

UIKit — это библиотека, обеспечивающая ключевую инфраструктуру, необходимую для построения iOS-приложений. UIKit содержит огромное количество классов, позволяющих строить визуальные элементы, анимировать их, обрабатывать физические законы, управлять механизмами печати, 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 PlaygroundSupport и UIKit (листинг 27.11).

Листинг 27.11

```
1 import PlaygroundSupport
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. 27.10). Всю функциональную начинку мы разместим в этом файле, в то время как главный файл **Balls** будет использовать реализованный функционал.

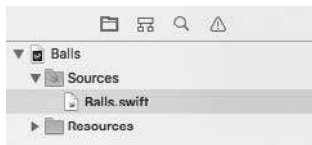


Рис. 27.10. Новый файл в Project Navigator

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 27.12.

Листинг 27.12

```
1 import UIKit
2 public class Balls: UIView {
3 }
```

В качестве уровня доступа класса **Balls** указан **public**. Это связано с тем, что некоторые свойства и методы класса **UIView**, которые будут переопределены в **Balls**, также соответствуют **public**, а, как вы знаете, уровень доступа свойств и методов не может быть выше, чем уровень самого типа.

В состав подключенной библиотеки **UIKit** входит тип данных **UIView**, предназначенный для визуального отображения графических элементов и взаимодействия с ними. Например, мы можем отобразить

прямоугольную рабочую область типа `UIView`, наполнив ее другими графическими элементами, каждый из которых также будет представлять собой отдельный экземпляр типа `UIView`. В этом и будет заключаться решение поставленной нами задачи с шариками (рис. 27.11).

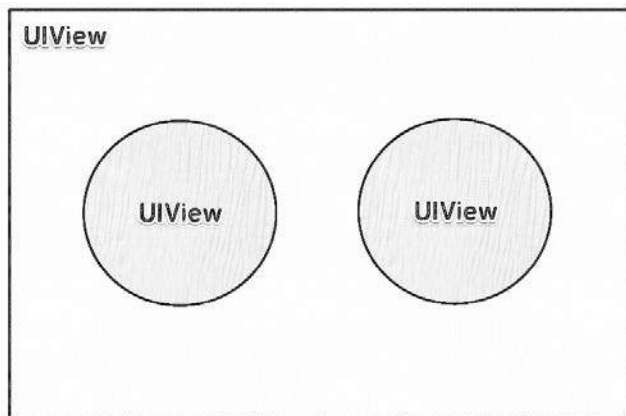


Рис. 27.11. Структура отображений

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. Шарика могут перемещаться.
5. Шарика могут взаимодействовать друг с другом (ударяться).
6. Шарика 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 27.13).

Листинг 27.13

```
1 // список цветов для шариков
2 private var colors: [UIColor]
3 // шарик
4 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 27.14).

Листинг 27.14

```
1 public init(colors: [UIColor]){
2     self.colors = colors
3     super.init(frame: CGRect(x: 0, y: 0, width: 400, height: 400))
4     backgroundColor = UIColor.gray
5 }
```

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 свойство `gray`, которое определяет серый цвет. При необходимости создания произвольного цвета вы можете использовать соответствующие методы (в этом вам поможет окно автодополнения).

Обратите внимание, что Xcode отобразит ошибку, сообщающую об отсутствии инициализатора `init(coder:)` (рис. 27.12).

Вы можете написать код инициализатора самостоятельно (листинг 27.15) или позволить Xcode исправить ошибку.

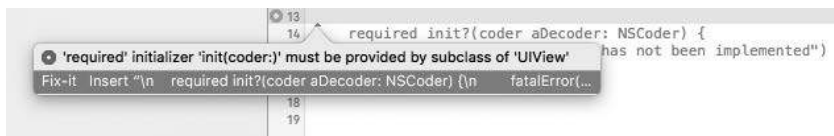


Рис. 27.12. Ошибка, сообщающая об отсутствии инициализатора `init(coder:)`

Листинг 27.15

```
1 required public init?(coder aDecoder: NSCoder) {
2     fatalError("init(coder:) has not been implemented")
3 }
```

Теперь перейдем к файлу `Balls` в `Project Navigator` и создадим экземпляр класса `Balls` (листинг 27.16).

Листинг 27.16

```
1 let balls = Balls(colors: [UIColor.white])
```

В качестве входного параметра инициализатора мы передаем массив объектов типа `UIColor`, содержащий всего один элемент, который определяет белый цвет. В результате в константу `balls` будет записан экземпляр класса `Balls`, описывающий всю разрабатываемую систему целиком.

ПРИМЕЧАНИЕ Xcode позволяет преобразовать некоторые ресурсы из текстового вида в графический. Так, например, описание цвета может быть представлено в виде визуальной палитры цветов прямо в редакторе кода!

Для этого требуется переписать объявление вида в виде специального литерала:

```
#colorLiteral(red: 0, green: 0, blue: 0, alpha: 1)
```

После его написания данный код будет преобразован к графическому квадрату белого цвета. Вы можете вставить как элемент массива `colors` вместо `UIColor.white` и скопировать необходимое количество раз (рис. 27.13).

```
import PlaygroundSupport
import UIKit
let balls = Balls(colors: [□])
```

Рис. 27.13. 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 на то, что такой подход возможен только в файлах, описывающих страницы `playground`. В `Balls.swift` вся информация отображается исключительно в текстовом виде.

Добавьте еще три произвольных цвета в массив `colors`, при этом не забудьте разделить их запятыми.

Теперь наш проект готов для предварительного отображения написанного кода. Для этого добавьте следующую строку в файл `Balls` (листинг 27.17).

Листинг 27.17

```
1 PlaygroundPage.current.liveView = balls
```

В данном листинге используется функционал подключенной ранее библиотеки `PlaygroundSupport` — класс `PlaygroundPage`, который позволяет вывести графические элементы на странице `playground`.

Теперь нажмите кнопку **Assistant Editor**, и в правой части Xcode отобразится вывод экземпляра класса `Balls` (рис. 27.14).

В данный момент отображается только серый прямоугольник — он будет подложкой для отображения набора шариков.

Вернемся к файлу `Balls.swift`.

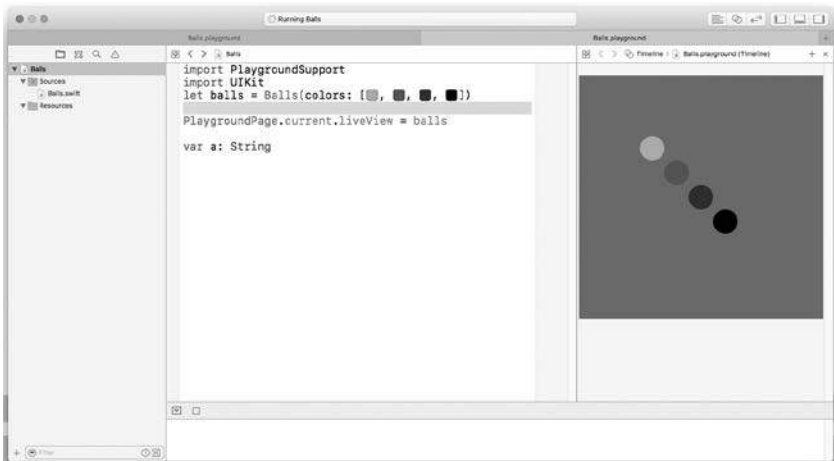


Рис. 27.14. Пример работы в режиме Assistant Editor

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 27.18).

Листинг 27.18

```

1  // размер шариков
2  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 напомним метод, отвечающий за отображение шариков (листинг 27.19).

Листинг 27.19

```

1  func ballsView () {
2      /* производим перебор переданных цветов
3       именно они определяют количество шариков */
4      for (index, color) in colors.enumerated() {
5          /* шарик представляет собой
6           экземпляр класса UIView */
7          let ball = UIView(frame: CGRect.zero)
8          /* указываем цвет шарика
9           он соответствует переданному цвету */
10         ball.backgroundColor = color
11         // накладываем отображение шарика на отображение подложки
12         addSubview(ball)
13         // добавляем экземпляр шарика в массив шариков
14         balls.append(ball)
15         /* вычисляем отступ шарика по осям X и Y, каждый
16          последующий шарик должен быть правее и ниже предыдущего */
17         let origin = 40*index + 100
18         // отображение шарика в виде прямоугольника
19         ball.frame = CGRect(x: origin, y: origin,
20                             width: Int(ballSize.width), height: Int(ballSize.height))
21         // с закругленными углами
22         ball.layer.cornerRadius = ball.bounds.width / 2.0
23     }
24 }

```

Для обработки шариков мы используем свойство `colors`, которое хранит в себе массив переданных цветов. Метод `enumerated()` уже знаком нам — он позволяет перебрать все элементы массива, получая индекс и значение каждого элемента.

Как уже неоднократно говорилось, шарики, как и вся система целиком, это отображения. Но если подложка — это экземпляр потомка класса `UIView`, то каждый шарик — экземпляр самого `UIView`. Изначально в качестве отображения мы используем конструкцию `CGRect.zero`, которая соответствует `CGRect(x: 0, y: 0, width: 0, height: 0)`, то есть прямоугольнику с нулевыми размерами.

Для того чтобы подложка и шарики выводились совместно, необходимо использовать функцию `addSubview(_:)`, которая накладывает одно отображение на другое.

Для того чтобы шарики обрисовывались так же, как и их подложка, необходимо добавить вызов метода `ballsView()` в инициализатор `init(colors:)` (листинг 27.20).

Листинг 27.20

```
1 public init(colors: [UIColor]){
2     self.colors = colors
3     super.init(frame: CGRect(x: 0, y: 0, width: 400, height: 400))
4     backgroundColor = UIColor.blueColor()
5     // вызов функции отрисовки шариков
6     ballsView()
7 }
```

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 27.21).

Листинг 27.21

```
1 // аниматор графических объектов
2 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 27.22).

Листинг 27.22

```
1 public init(colors: [UIColor]){
2     self.colors = colors
3     super.init(frame: CGRect(x: 0, y: 0, width: 400, height: 400))
4     backgroundColor = UIColor.blueColor()
```

```

5      // подключаем аниматор с указанием на сам класс
6      animator = UIDynamicAnimator(referenceView: self)
7      ballsView()
8  }

```

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 27.23).

Листинг 27.23

```

1  // обработчик перемещений объектов
2  private var snapBehavior: UISnapBehavior?

```

Класс `UISnapBehavior` позволяет обрабатывать касания экрана устройства (или щелчки мышки). Для этого в состав `UISnapBehavior` включены три метода:

❑ `touchesBegan(_:with:)`

Метод вызывается в момент касания экрана.

❑ `touchesMoved(_:with:)`

Метод срабатывает при каждом перемещении пальца, уже коснувшегося экрана.

❑ `touchesEnded(_:with:)`

Метод вызывается по окончании взаимодействия с экраном (когда палец убран).

Все методы уже определены в `UISnapBehavior`, поэтому при создании собственной реализации этих методов необходимо их переопределять, то есть использовать ключевое слово `override`.

Реализуем метод `touchesBegan(_:with:)` (листинг 27.24).

Листинг 27.24

```

1  override public func touchesBegan(_ touches: Set<UITouch>,
    with event: UIEvent?) {

```

```
2     if let touch = touches.first {
3         let touchLocation = touch.location( in: self )
4         for ball in balls {
5             if (ball.frame.contains( touchLocation )) {
6                 snapBehavior = UISnapBehavior(item: ball,
7                 snapTo: touchLocation)
8                 snapBehavior?.damping = 0.5
9                 animator?.addBehavior(snapBehavior!)
10            }
11        }
12    }
```

Коллекция `touches` содержит данные обо всех касаниях. Это связано с тем, что таच-панель современных устройств поддерживает мульти-тач, то есть одновременное касание несколькими пальцами. В начале метода извлекаются данные о первом элементе набора `touches` и помещаются в константу `touch`.

Константа `touchLocation` содержит координаты касания относительно всего отображения.

С помощью метода `ball.frame.contains()` мы определяем, входят ли координаты касания в какой-либо из шариков. Если находится соответствие, то в свойство `snapBehavior` записываются данные об объекте, с которым происходит взаимодействие, и о координатах, куда данный объект должен быть перемещен.

Свойство `damping` определяет плавность и затухание при движении шарика.

Далее, используя метод `addBehavior(_:)` аниматора, указываем, что обрабатываемое классом `UISnapBehavior` поведение объекта должно быть анимировано. Таким образом, все изменения состояния объекта, производимые в свойстве `snapBehavior`, будут анимированы.

После обработки касания необходимо обработать перемещение пальца. Для этого реализуем метод `touchesMoved(_:with:)` (листинг 27.25).

Листинг 27.25

```
1  override public func touchesMoved(_ touches: Set<UITouch>,
2  with event: UIEvent?) {
3      if let touch = touches.first {
4          let touchLocation = touch.location( in: self )
5          if let snapBehavior = snapBehavior {
6              snapBehavior.snapPoint = touchLocation
7          }
8      }
```

Так как в свойстве `snapBehavior` уже содержится указание на определенный шарик, с которым происходит взаимодействие, нет необходимости проходить по всему массиву шариков снова. Единственной задачей данного метода является изменение свойства `snapPoint`, которое указывает на координаты объекта.

Для завершения обработки перемещения объектов касанием необходимо переопределить метод `touchesEnded(_:with:)` (листинг 27.26).

Листинг 27.26

```
1 public override func touchesEnded(_ touches: Set<UITouch>,
   with event: UIEvent?) {
2     if let snapBehavior = snapBehavior {
3         animator?.removeBehavior(snapBehavior)
4     }
5     snapBehavior = nil
6 }
```

Данный метод служит для решения одной очень важной задачи — очистки используемых ресурсов. После того как взаимодействие с шариком окончено, нет необходимости хранить информацию об обработчике поведения в `snapBehavior`.

ПРИМЕЧАНИЕ Возьмите за привычку удалять ресурсы, пользоваться которыми вы уже не будете. Это сэкономит изрядное количество памяти и уменьшит вероятность возникновения ошибок.

Перейдите в файл `Balls`, и в окне `Assistant Editor` вы увидите изображение четырех шариков, но, в отличие от предыдущего раза, сейчас вы можете указателем мыши перемещать любой из шариков (рис. 27.15)! 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 27.27).

Листинг 27.27

```
1 // обработчик столкновений
2 private var collisionBehavior: UICollisionBehavior
```

Следующим шагом будет редактирование инициализатора (листинг 27.28).

Листинг 27.28

```
1 public init(colors: [UIColor]){
2     self.colors = colors
```

```

3      // создание значения свойства
4      collisionBehavior = UICollisionBehavior(items: [])
5      /* указание на то, что границы отображения
6         также являются объектами взаимодействия */
7      collisionBehavior.setTranslatesReferenceBoundsIntoBoundary( with:
      UIScreenInsets(top: 1, left: 1, bottom: 1, right: 1))
8      super.init(frame: CGRect(x: 0, y: 0, width: 400, height: 400))
9      backgroundColor = UIColor.grayColor()
10     animator = UIDynamicAnimator(referenceView: self)
11     /* добавляем обработчик поведения при столкновении
12        к аниматору */
13     animator?.addBehavior(collisionBehavior)
14     ballsView()
15 }

```

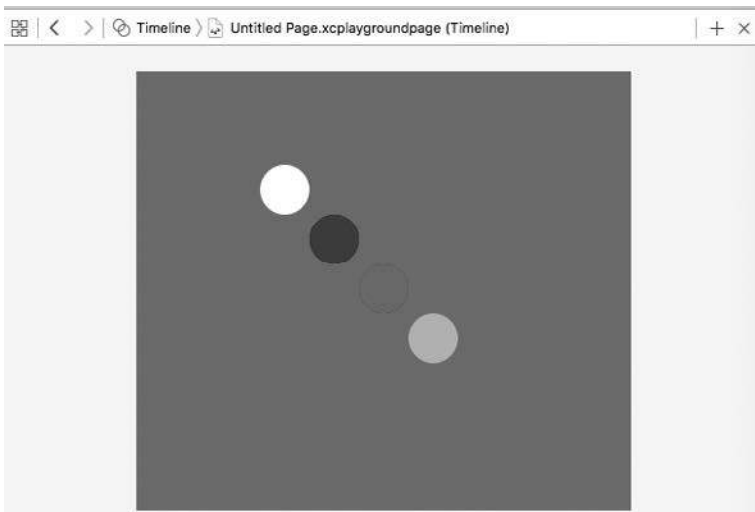


Рис. 27.15. Шарики, расположенные внутри подложки

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 `setTranslatesReferenceBoundsIntoBoundary(with:)`. Он устанавливает грани-

цы 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 27.29).

Листинг 27.29

```
1 func ballsView(){
2     for (index, color) in colors.enumerated(){
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:
9             Int(ballSize.width), height: Int(ballSize.height))
10        ball.layer.cornerRadius = ball.bounds.width / 2.0
11        // добавим шарик в обработчик столкновений
12        collisionBehavior.addItem(ball)
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 столкнуть их они разлетаются.

UIKit и Foundation предоставляет еще огромное количество возможностей помимо рассмотренных. Так, например, можно создать гравитационное, магнитное и любое другое взаимодействие элементов, смоделировать силу тяжести или турбулентности, установить параметры скорости и ускорения. Все это и многое-многое другое позволяет вам создавать поистине функциональные приложения, которые обязательно найдут отклик у пользователей.

28 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 **Dictionary** существует требование, чтобы тип данных ключа соответствовал протоколу **Hashable** (его предназначение мы обсуждали ранее).

28.1. Универсальные функции

Разработаем функцию, с помощью которой можно поменять значения двух целочисленных переменных (листинг 28.1).

Листинг 28.1

```
1 func swapTwoInts( a: inout Int,  b: inout Int) {  
2     let temporaryA = a  
3     a = b  
4     b = temporaryA  
5 }
```

```
6 var firstInt = 4010
7 var secondInt = 13
8 swapTwoInts(a: &firstInt, b: &secondInt)
```

Функция `swapTwoInts(a:b:)` использует сквозные параметры, чтобы обеспечить доступ непосредственно к параметрам, передаваемым в функцию, а не к их копиям. В результате выполнения значения в переменных `firstInt` и `secondInt` меняются местами.

Данная функция является крайне полезной, но очень ограниченной в своих возможностях. Для того чтобы поменять значения двух переменных других типов, придется писать отдельную функцию: `swapTwoStrings()`, `swapTwoDoubles()` и т. д. Если обратить внимание на то, что тела всех функций должны быть практически одинаковыми, то мы просто-напросто займемся дублированием кода, хотя ранее в книге неоднократно рекомендовалось всеми способами этого избегать.

Для решения задачи было бы намного удобнее использовать универсальную функцию, позволяющую передать в качестве аргумента значения любого типа с одним лишь требованием: типы данных обоих аргументов должны быть одинаковыми.

Универсальные функции объявляются точно так же, как и стандартные, за одним исключением: после имени функции в угловых скобках указывается заполнитель имени типа, то есть литерал, который далее в функции будет указывать на тип данных переданного аргумента.

Преобразуем функцию `swapTwoInts(a:b:)` к универсальному виду (листинг 28.2).

Листинг 28.2

```
1 func swapTwoValues<T>( a: inout T,  b: inout T) {
2     let temporaryA = a
3     a = b
4     b = temporaryA
5 }
6 var firstString = "one"
7 var secondString = "two"
8 swapTwoValues(a: &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(a:b:)` может вызываться точно так же, как и определенная ранее функция `swapTwoInts(a:b:)`.

Используемый заполнитель называется параметром типа. Как только вы его определили, можете применять его для указания типа любого параметра или значения, включая возвращаемое функцией значение. При необходимости можно задать несколько параметров типа, вписав их в угловых скобках через запятую.

28.2. Универсальные типы

В дополнение к универсальным функциям универсальные шаблоны позволяют создать универсальные типы данных. К универсальным типам относятся, например, упомянутые ранее массивы и словари. Создадим универсальный тип данных `Stack` (стек) — упорядоченную коллекцию элементов, подобную массиву, но со строгим набором доступных операций:

- ❑ метод `push(_:)` служит для добавления элемента в конец коллекции;
- ❑ метод `pop()` служит для возвращения элемента из конца коллекции с удалением его оттуда.

Никаких иных доступных операций для взаимодействия со своими элементами стек не поддерживает.

В первую очередь создадим неуниверсальную версию данного типа (листинг 28.3).

Листинг 28.3

```
1 struct IntStack {
2     var items = [Int]()
3     mutating func push(_ item: Int) {
4         items.append(item)
5     }
6     mutating func pop() -> Int {
7         return items.removeLast()
8     }
9 }
```

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 28.4).

Листинг 28.4

```
1 struct Stack<T> {  
2     var items = [T]()  
3     mutating func push(_ item: T) {  
4         items.append(item)  
5     }  
6     mutating func pop() -> T {  
7         return items.removeLast()  
8     }  
9 }
```

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 28.5).

Листинг 28.5

```
1 var stackOfStrings = Stack<String>()  
2 stackOfStrings.push("uno")  
3 stackOfStrings.push("dos")  
4 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 28.6). Для реализации этой задачи опишем инициализатор, принимающий в качестве входного аргумента массив значений.

Листинг 28.6

```
1 struct Stack<T> {  
2     var items = [T]()  
3     init(){}  
4     init(_ elements: T...){  
5         self.items = elements  
6     }  
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)           Stack<Int>
15 var stackOfStrings = Stack<String>()       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.

28.3. 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 28.7).

ПРИМЕЧАНИЕ Для обеспечения функционала сравнения двух значений в Swift существует специальный протокол **Equatable**. Он обяывает поддерживающий его тип данных реализовать функционал сравнения двух значений с помощью операторов равенства (**==**) и неравенства (**!=**). Другими словами, если тип данных поддерживает этот протокол, то его значения можно сравнивать между собой.

Листинг 28.7

```

1 func findIndex<T: Equatable>(array: [T], valueToFind: T) -> Int? {

```

```

2     for (index, value) in array.enumerated() {
3         if value == valueToFind {
4             return index
5         }
6     }
7     return nil
8 }
9 var myArray = [3.14159, 0.1, 0.25]
10 let firstIndex = findIndex(array: myArray, valueToFind: 0.1) // 1
11 let secondIndex = findIndex(array: myArray, valueToFind: 31) // nil

```

Параметр типа записывается как `<T: Equatable>`. Это означает «любой тип, поддерживающий протокол `Equatable`». В результате поиск в переданном массиве выполняется без ошибок, поскольку тип данных `Int` поддерживает протокол `Equatable`, а значит, значения данного типа могут быть приняты к обработке.

28.4. 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 28.8).

Листинг 28.8

```

1 extension Stack {
2     var topItem: T? {
3         return items.isEmpty ? nil : items[items.count - 1]
4     }
5 }

```

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`.

28.5. Связанные типы

При определении протокола бывает удобно использовать связанные типы, указывающие на некоторый, пока неизвестный, тип данных. Связанный тип позволяет задать заполнитель типа данных, который будет использоваться при заполнении протокола. Фактически тип данных

не указывается до тех пор, пока протокол не будет принят каким-либо объектным типом. Связанные типы указываются с помощью ключевого слова `associatedtype`, за которым следует имя связанного типа.

Определим протокол `Container`, использующий связанный тип `ItemType` (листинг 28.9).

Листинг 28.9

```
1 protocol Container {
2     associatedtype ItemType
3     mutating func append(item: ItemType)
4     var count: Int { get }
5     subscript(i: Int) -> ItemType { get }
6 }
```

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 28.10.

Листинг 28.10

```
1 struct Stack<T>: Container {
2     typealias ItemType = T
3     var items = [T]()
4     var count: Int {
5         return items.count
6     }
7     init(){}
8     init(_ elements: T...){
9         self.items = elements
10    }
11    subscript(i: Int) -> T {
12        return items[i]
13    }
14    mutating func push(item: T) {
15        items.append(item)
16    }
17    mutating func pop() -> T {
18        return items.removeLast()
19    }
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 **typealias** указывает на то, какой тип данных является связанным в данном объектном типе.

ПРИМЕЧАНИЕ Обратите внимание на то, что при описании протокола используется ключевое слово **associatedtype**, а при описании структуры — **typealias**.

Так как заполнитель имени использован в качестве типа аргумента **item** свойства **append** и возвращаемого значения сабскрипта, Swift может самостоятельно определить, что заполнитель **T** указывает на тип **ItemType**, соответствующий типу данных в протоколе **Container**. При этом указывать ключевое слово **associatedtype** не обязательно: если вы его удалите, то тип продолжит работать без ошибок.

29

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.

29.1. Выбрасывание ошибок

В Swift для создания перечня возможных ошибок служат перечисления, где каждый член перечисления соответствует отдельной ошибке. Само перечисление при этом должно поддерживать протокол `Error`, который, хотя и является пустым, сообщает о том, что данный объектный тип содержит варианты ошибок.

Не стоит создавать одно перечисление на все случаи жизни. Группируйте возможные ошибки по их смыслу в различных перечислениях.

В следующем примере объявляется тип данных, который описывает ошибки в работе торгового автомата по продаже еды (листинг 29.1).

Листинг 29.1

```
1 enum VendingMachineError: Error {  
2     case InvalidSelection  
3     case InsufficientFunds(coinsNeeded: Int)  
4     case OutOfStock  
5 }
```

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 29.2).

Листинг 29.2

```
1 throw VendingMachineError.InsufficientFunds(coinsNeeded: 5)
```

29.2. 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 29.3 приведен пример объявления двух функций, которые передают возникающие в них ошибки выше.

Листинг 29.3

```

1 func anotherFunc() throws {
2     // тело функции
3     var value = try someFunc()
4     // ...
5 }
6 func someFunc() throws -> String{
7     // тело функции
8     try anotherFunc()
9     // ...
10 }
11 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 29.4.

Листинг 29.4

```

1 struct Item {
2     var price: Int
3     var count: Int
4 }
5 class VendingMachine {
6     var inventory = [
7         "Candy Bar": Item(price: 12, count: 7),
8         "Chips": Item(price: 10, count: 4),
9         "Pretzels": Item(price: 7, count: 11)
10    ]
11     var coinsDeposited = 0
12     func dispenseSnack(snack: String) {
13         print("Dispensing \(snack)")
14     }
15     func vend(itemNamed name: String) throws {
16         guard var item = inventory[name] else {
17             throw VendingMachineError.InvalidSelection

```

```

18         }
19         guard item.count > 0 else {
20             throw VendingMachineError.OutOfStock
21         }
22         guard item.price <= coinsDeposited else {
23             throw VendingMachineError.InsufficientFunds(coinsNeeded:
24                 item.price - coinsDeposited)
25         }
26         coinsDeposited -= item.price
27         item.count -= 1
28         inventory[name] = item
29         dispenseSnack(snack: name)
30     }

```

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(snack:)` сообщает о том, что аппарат выдает некий товар. Метод `vend(itemNamed:)` непосредственно обслуживает покупку товара через аппарат.

При определенных условиях (запрошенный товар недоступен, его нет в наличии или количества внесенных монет недостаточно для покупки) метод `vend(itemNamed:)` может выбросить ошибку, соответствующую перечислению `VendingMachineError`. Сама реализация метода использует оператор `guard` для реализации преждевременного выхода с помощью оператора `throw`. Оператор `throw` мгновенно изменяет ход работы программы, в результате 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 29.5). В данном примере словарь `favoriteSnacks` содержит указатель на любимое блюдо каждого из трех человек.

Листинг 29.5

```

1 let favoriteSnacks = [
2     "Alice": "Chips",
3     "Bob": "Licorice",

```

```

4     "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(person:vendingMachine:)` не может выбросить ошибку, но так как она вызывает метод `vend(itemNamed:)`, для передачи ошибки выше необходимо использовать операторы `throws` и `try`.

Оператор do-catch

Выброс и передача ошибок вверх в конце концов должна вести к их обработке таким образом, чтобы это принесло определенную пользу пользователю и разработчику. Для этого вы можете задействовать оператор `do-catch`.

Синтаксис

```

do {
    try 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 29.6).

Листинг 29.6

```

1 var vendingMachine = VendingMachine()

```

```

2  vendingMachine.coinsDeposited = 8
3  do {
4      try buyFavoriteSnack(person: "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(person:vendingMachine:)` вызывается в блоке `do`. Поскольку внесенной суммы монет не хватает для покупки любимой сладости покупателя `Alice`, возвращается ошибка и выводится соответствующее этой ошибке сообщение.

Преобразование ошибки в опционал

Для преобразования выброшенной ошибки в опциональное значение используется оператор `try`, а точнее, его форма `try?`. Если в этом случае выбрасывается ошибка, то значение выражения вычисляется как `nil`.

Рассмотрим пример из листинга 29.7.

Листинг 29.7

```

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 выброс 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 29.8.

Листинг 29.8

```
1 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.

29.3. 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 29.9).

Листинг 29.9

```
1 func processFile(filename: String) throws {  
2     if exists(filename) {  
3         let file = open(filename)  
4         defer {  
5             close(file)  
6         }  
7         while let line = try file.readline() {  
8             // работа с файлом.  
9         }  
10    }  
11 }
```

В данном примере оператор `defer` просто обеспечивает закрытие открытого ранее файла.

30 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.

30.1. 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 30.1).

Листинг 30.1

```
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 30.2.

Листинг 30.2

```
1 func + (left: Vector2D, right: Vector2D) -> Vector2D {  
2     return Vector2D(x: left.x + right.x, y: left.y + right.y)
```

```
3 }  
4 let vector = Vector2D(x: 3.0, y: 1.0)  
5 let anotherVector = Vector2D(x: 2.0, y: 4.0)  
6 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 *перегрузки*. С данным понятием мы уже неоднократно познакомились 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 перегрузки префиксного или постфиксного оператора перед объявлением операторной функции необходимо указать модификатор `prefix` или `postfix` соответственно.

Реализуем префиксный оператор унарного минуса для структуры `Vector2D` (листинг 30.3).

Листинг 30.3

```
1 prefix func - (vector: Vector2D) -> Vector2D {  
2     return Vector2D(x: -vector.x, y: -vector.y)  
3 }  
4 let positive = Vector2D(x: 3.0, y: 4.0)  
5 let negative = -positive // negative – экземпляр Vector2D  
   со значениями (-3.0, -4.0)
```

Благодаря созданию операторной функции мы можем использовать унарный минус для того, чтобы развернуть вектор относительно начала координат.

Составной оператор присваивания

В составных операторах присваивания оператор присваивания (+) комбинируется с другим оператором. Для перегрузки составных операторов в операторной функции первый передаваемый аргумент

необходимо сделать сквозным (`inout`), так как именно его значение будет меняться в ходе выполнения функции.

В листинге 30.4 приведен пример реализации составного оператора присваивания-сложения для экземпляров типа `Vector2D`.

Листинг 30.4

```
1 func += ( left: inout Vector2D, right: Vector2D) {
2     left = left + right
3 }
4 var original = Vector2D(x: 1.0, y: 2.0)
5 let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
6 original += vectorToAdd
7 // original теперь имеет значения (4.0, 6.0)
```

Так как оператор сложения был объявлен ранее, вам нет нужды реализовывать его в теле данной функции. Вы можете просто сложить два значения типа `Vector2D`.

Обратите внимание, что первый входной аргумент функции является сквозным.

Оператор эквивалентности

Пользовательские объектные типы не содержат встроенной реализации оператора эквивалентности, поэтому чтобы сравнить два экземпляра, необходимо перегрузить данный оператор с помощью операторной функции.

В следующем примере приведена реализация оператора эквивалентности и оператора неэквивалентности (листинг 30.5).

Листинг 30.5

```
1 func == (left: Vector2D, right: Vector2D) -> Bool {
2     return (left.x == right.x) && (left.y == right.y)
3 }
4 func != (left: Vector2D, right: Vector2D) -> Bool {
5     return !(left == right)
6 }
7 let twoThree = Vector2D(x: 2.0, y: 3.0)
8 let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
9 if twoThree == anotherTwoThree {
10     print("Эти два вектора эквивалентны.")
11 }
12 // 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.

30.2. 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 30.6).

Листинг 30.6

```
1 prefix operator ++
2 prefix func ++ (inout vector: Vector2D) -> Vector2D
3 {
4     vector += vector
5     return vector
6 }
7 var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
8 let afterDoubling = ++toBeDoubled
9 // toBeDoubled теперь имеет значения (2.0, 8.0)
10 // 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 разработке 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 3.0

Swift 3.0 внес в процесс разработки довольно много изменений. В приложении приведены все нововведения новой версии этого замечательного языка программирования. Лучший способ быть готовым к переходу на новые версии — следить за официальным блогом разработчиков Swift (swift.org), а также проектом Swift programming Language Evolution на [github](https://github.com). Хочу отметить, что все вносимые изменения предлагались именно в рамках работы данного проекта. Всего было представлено более 100 предложений, большинство из которых после обсуждений и доработок были приняты. Как видите, язык Swift все еще активно изменяется, но благодаря активной работе сообщества его синтаксис становится все более устоявшимся.

Ассистент миграции

В состав Xcode 8 входит ассистент миграции, которой поможет вам перейти на Swift 3.0 или Swift 2.3 (поддерживаются обе версии языка). Ассистент миграции выполнит большинство изменений в ваших старых приложениях в автоматическом режиме, но, как показывает практика, некоторые из них вам придется править самостоятельно. При выполнении миграции вы наверняка обратите внимание на то, что изменения будут внесены практически в каждый файл вашего проекта. Это связано с глобальными изменениями имен в Cocoa API.

API Design Guidelines

Одной из основных целей, которую разработчики ставили перед Swift 3, стало утверждение стандартов разработки и использования API-библиотек. В более ранних версиях API-библиотеки очень сильно зависели от самих разработчиков этих библиотек, а также от того языка программирования, на котором данные библиотеки реализовывались. С целью стабилизации и утверждения синтаксиса был разработан документ, стандартизирующий и регламентирующий все моменты разработки API. В результате синтаксис некоторых методов

и функций, которые вы могли использовать в Swift 1 и Swift 2, был изменен.

Актуальная версия Swift API Design Guidelines представлена по адресу <https://swift.org/documentation/api-design-guidelines>.

Транслирование имен API

Серьезной критике сообщества разработчиков подверглась работа Clang, транслирующего имена API, написанные на языках C и Objective-C, при использовании их в Swift. Используемые функции, типы, методы, свойства и многое другое получались слишком многословными, избыточными. Посмотрите на код из листинга П.1. Он написан на Swift 2.2.

Листинг П.1

```
1 content = listItemView.text.stringBytrimmingCharactersInSet
   (NSCharacterSet.whitespaceAndNewLineCharacterSet())
```

В данном примере используются возможности API-библиотеки, написанной на Objective-C и включенной в Swift-проект. Код получился очень сложным и непонятным, но он мог бы быть более «swiftы», например, как в листинге П.2.

Листинг П.2

```
1 content = listItemView.text.trimming(.whitespaceAndNewlines)
```

Как видите, вся избыточность убрана, имена элементов стали более читаемыми, а функциональные возможности при этом совершенно не изменились.

Повлиять на работу Clang стало возможным благодаря созданию упомянутого ранее API Design Guidelines. Разработчики привели его требования в соответствие с требованиями документа Objective-C Coding Guidelines for Cocoa, после чего было написано соответствующее расширение для транслятора. В результате были убраны лишние слова, а удобочитаемость кода при уменьшении его размеров значительно возросла. Примеры приведены в листинге П.3.

Листинг П.3

```
1 let red = UIColor.redColor() // Swift 2
2 let red = UIColor.red() // Swift 3
3
4 let min = numbers.minElement() // Swift 2
5 let min = numbers.min() // Swift 3
```

```
6
7 UIDevice.currentDevice() // Swift 2
8 UIDevice.current() // Swift 3
```

Соответствие стандартных библиотек и API Design Guidelines

API Design Guidelines однозначно определяет правила именования методов и свойств. Это позволило применить описанные правила для входящих в состав Swift API-библиотек. Далее приведены все изменения, с которыми вы, вероятно, встретитесь в ходе разработки:

1. UpperCamelCase-стиль используется теперь только для имен типов (в том числе и протоколов), а для методов и свойств используется lowerCamelCase (листинг П.4).

Листинг П.4

```
1 NSTextAlignment.Right // Swift 2
2 NSTextAlignment.right // Swift 3
```

Из листинга видно, что в новой версии языка свойство `right` пишется со строчной буквы.

2. Суффикс `Type` убран из имен всех протоколов, а в некоторых случаях он заменен суффиксом `Protocol` (листинг П.5).

Листинг П.5

```
1 public protocol CollectionType : ... { ... } // Swift 2
2 public protocol Collection : ... { ... } // Swift 3
3 public protocol LazyCollectionType : ... { ... } // Swift 2
4 public protocol LazyCollectionProtocol : ... { ... } // Swift 3
```

3. Концепт `generator` переименован в `iterator` (листинг П.6).

Листинг П.6

```
1 public struct EmptyGenerator<Element> : ... { ... } // Swift 2
2 public struct EmptyIterator : ... { ... } // Swift 3
3
4 public struct GeneratorType : ... { ... } // Swift 2
5 public struct IteratorProtocol : ... { ... } // Swift 3
```

4. Удален тип данных `Bit`, единственным предназначением которого было использование для индекса структуры `CollectionOfOne`. Вместо него рекомендовано использовать тип `Int`.
5. Были исключены структура `PermutationGenerator` и протокол `MutableSliceable`.

6. В generic-типах `UnsafePointer` и `UnsafeMutablePointer` имя параметра `memory` изменено на `pointee`. Также для данных типов удалены пустые инициализаторы (без аргументов).
7. Для методов, которые возвращают модифицированную копию входного объекта, но при этом сам переданный в метод объект не изменяется, в конце имени метода требуется использовать окончание `-ed` (а в некоторых случаях `-ing`) (листинг П.7).

Листинг П.7

```

1  someArray.sort() // Swift 2
2  someArray.sorted() // Swift 3
3  someArray.reverse() // Swift 2
4  someArray.reversed() // Swift 3
5  someArray.enumerate() // Swift 2
6  someArray.enumerated() // Swift 3

```

8. Благодаря переименованию метода `sort()` в `sorted()` метод `sortInPlace()` был переименован в освободившийся `sort()` (листинг П.8).

Листинг П.8

```

1  someArray.sortInPlace() // Swift 2
2  someArray.sort() // Swift 3

```

9. Методы `minElement()` и `maxElement()` для типа `Sequence` (в Swift 3 это новое имя для `SequenceType`) переименованы в `min()` и `max()` соответственно (листинг П.9).

Листинг П.9

```

1  SequenceType.minElement() // Swift 2
2  Sequence.min() // Swift 3

```

10. Имена некоторых методов и их аргументов были изменены в соответствии с требованиями API Design Guidelines. Несколько примеров вы можете найти в листинге П.10.

Листинг П.10

```

1  func advancedBy(n: Distance) -> Self    // Swift 2
2  func advanced(by n: Distance) -> Self  // Swift 3
3  public mutating func removeAtIndex(index: Index) -> Element
                                                    // Swift 2
4  public mutating func remove(at index: Index) -> Element ()
                                                    // Swift 3
5  someArray.enumerate() //swift 2 someArray.enumerated() // Swift 3

```

В данном случае я не делаю привязку к конкретным типам данных. Из примеров вы сможете понять общий принцип проведенных изменений.

Изменения в функциях

Довольно серьезные изменения коснулись функций. Рассмотрим наиболее важные из них.

Имя первого аргумента функции

Swift 2.x при вызове функции было разрешено не указывать имя первого параметра при передаче его значения "".(", : , : 8). В Swift 3, если не указано иное, имя первого параметра указывается в обязательном порядке (листинг П.11).

Листинг П.11

```
1 "Text".writeToFile("filename", atomically: true, encoding:
   NSStringEncoding) "" "" // Swift 2
2 "Text".write(toFile: "filename", atomically: true, encoding:
   NSStringEncoding) // Swift 3
```

Первый параметр указывается с именем `toFile`. Это позволило значительно улучшить читаемость кода. При необходимости исключения указания имени параметра необходимо использовать нижнее подчеркивание в качестве внешнего имени параметра (листинг П.12).

Листинг П.12

```
1 func hexToColor(_ hex: String) {...}
2 hexToColor("#FF00FF")
```

Описание функционального типа

В некоторых случаях чтение кода функции могло быть крайне затруднительным вследствие указания сложных конструкций входных и выходных параметров. Так, например, код из листинга П.13 довольно тяжело прочитать.

Листинг П.13

```
1 func foo(arg: Int->Int)->Int->Int {...}
```

Далеко не каждый программист сразу сообразит, какие данные являются входными, а какие — выходными.

ПРИМЕЧАНИЕ Если вам это сложно, то функция с именем `foo` принимает на вход функцию типа `(Int) -> Int` и возвращает функцию точно такого же типа.

Swift 3 устраняет эту проблему. Теперь начало каждого отдельного функционального типа заключается в скобки. Таким образом, предыдущий пример в контексте Swift 3 будет выглядеть так, как в листинге П.14.

Листинг П.14

```
1 func foo(arg: (Int)->Int)->(Int)->Int {...}
```

Каррирование функций

Новые правила описания функционального типа непосредственно повлияли на каррирование. В новой версии языка синтаксис описания каждого входного параметра внутри своей пары скобок был упразднен. Теперь требуется явно указывать тип возвращаемого значения (листинг П.15).

Листинг П.15

```
1 func sum(x:Int)(y:Int) -> Int {...} // Swift 2
2 func sum(x:Int) -> (Int) -> Int {...} // Swift 3
```

Модификатор `inout` перенесен к определению типа

Модификатор `inout`, позволяющий указывать сквозные параметры, теперь указывается перед описанием типа (листинг П.16).

Листинг П.16

```
1 func changeValues( a: inout Int, _ b: inout Int) -> () { ... }
```

Удаление `var` для входных аргументов

Ранее мы могли использовать ключевые слова `var` и `inout` для указания того, что переданные в функцию значения можно изменять. При этом только `inout` позволяло непосредственно влиять на сам переданный параметр, а `var` лишь создавало его локальную копию с областью видимости «внутри функции». Это вызывало некоторые неудобства, так как могло привести к путанице.

Теперь возможность использования ключевого слова `var` исключена из языка. Для создания локальной изменяемой копии вам потребуется использовать `var` уже в теле функции (листинг П.17).

Листинг П.17

```
1 func doSomething(x:String){
2     var x = x
3     // какие-либо действия с использованием локальной переменной x
4 }
5 func sum(x:Int) -> (Int) -> Int {...} //swift 3
```

Ключевое слово `inout`, как и ранее, позволяет указать входной параметр как сквозной. Таким образом, все изменения этого параметра внутри функции будут сохранены и после завершения ее работы (листинг П.18).

Листинг П.18

```
1 func doSomethingWithInout(x: inout String){
2     return x+1
3 }
4 var someArg = 12                                12
5 doSomethingWithInout(x: &someArg)                12
6 someArg                                           13
```

ПРИМЕЧАНИЕ Обратите внимание на то, что атрибут `inout` теперь пишется не перед именем аргумента, а перед его типом. Это также одно из изменений Swift 3.0, о котором мы говорили ранее.

Удаление `let` для входных аргументов

Так как аргументы функции по умолчанию неизменяемы, то указывать оператор `let` перед именем входного параметра не имело никакого смысла (листинг П.19).

Листинг П.19

```
1 func foo(let x: Int) {...}
```

Теперь подобный подход использовать запрещено, то есть вам требуется лишь исключить оператор `let` из кода, но результат останется тем же самым. В следующем примере локальный параметр `x` является неизменяемым (листинг П.20).

Листинг П.20

```
1 func foo(x: Int) {...}
```

Убрана неявная передача параметров функции в виде кортежа

Ранее для функции, имеющей несколько входных параметров, существовали два способа передачи этих параметров (листинг П.21).

Листинг П.21

```
1 func foo(a : Int, b : Int) {}
2 // каждый аргумент передается отдельно
3 foo(42, b : 17)
4 // аргументы передаются в виде кортежа
5 let x = (1, b: 2)
6 foo(x)
```

Второй вариант значительно усложнял чтение кода, особенно в тех случаях, когда имелось несколько вариантов функции с разным набором входных параметров. Вследствие этого возможность передачи параметров в виде кортежа удалена из языка.

Входящие аргументы-замыкания по умолчанию @noescaping

Атрибут `@noescaping` удален из-за того, что все аргументы-замыкания по умолчанию являются неуходящими (`noescaping`). Для того чтобы дать замыканию возможность существовать вне родительской функции, необходимо использовать атрибут `@escaping`.

Перемещены атрибуты @escaping и @autoclosure

Теперь атрибуты `@escaping` и `@autoclosure` указываются не перед именем параметра, как это было ранее, а перед типом этого параметра (по аналогии с `inout`). Пример приведен в листинге П.22.

Листинг П.22

```
1 // в Swift 2 было так
2 func foo(@autoclosure closure : () -> Void ) { ... }
3 // в Swift 3 стало так
4 func foo(closure : @autoclosure () -> Void ) { ... }
```

Передача входных параметров в функцию

При вызове функции вы можете передать значения для входных параметров только в том порядке, в каком они определены. При этом, если какие-либо из аргументов имеют значения по умолчанию, то они могут быть пропущены (листинг П.23).

В Swift 2 порядок параметров был не важен.

Листинг П.23

```
1 func foo(a:Int, b:Int = 2, c: Int){ ... }
2 foo(b:2, a:1, c:4) // вызовет ошибку
```

```
3 foo(a:1, b:2, c:3) // корректный вызов функции
4 foo(a:1, c:3) // корректный вызов функции с исключением параметра,
    имеющего значение по умолчанию
```

Удалены операторы инкремента и декремента

Разговоры по поводу операторов инкремента (++) и декремента (—) начались уже давно. Основная причина критики крылась в том, что данные операторы создают измененную копию исходного параметра (уменьшенную или увеличенную на единицу) без ведома разработчика, то есть происходит неконтролируемое расходование ресурсов. В Swift 2.2 данные элементы были помечены как устаревшие, а в Swift 3.0 удалены полностью. Вместо них стоит использовать сокращенные операторы сложения (листинг П.24).

Листинг П.24

```
1 i = 1                                1
2 i += 1                               2
3 i -= 1                               1
4 i                                     1
```

Удалена классическая форма цикла for

Следующим удаленным элементом стал цикл `for` классической С-образной формы (листинг П.25).

Листинг П.25

```
1 // применение такой формы больше недопустимо
2 for var i = 1; i <= 10; i += 1 {
3     print("\(i) green bottles")
4 }
```

Причиной такого решения стали его скудные возможности. Также он поощрял использование операторов инкремента и декремента, которые тоже было решено удалить. Вместо исключенной формы цикла следует использовать `for-in`, который по функциональным возможностям значительно превосходит своего собрата.

Доступ к типу

При написании кода в контексте типа данных (внутри класса или структуры) для осуществления доступа к экземпляру этого типа используется ключевое слово `self` (со строчной буквы). Для получения доступа к самому типу, а не к его экземпляру, требовалось либо ука-

затем имя этого типа, либо использовать конструкцию `self.dynamicType` (листинг П.26).

Листинг П.26

```
1 struct MyStruct {
2     static func staticMethod(){...}
3     func instanceMethod(){
4         //вызываем статический метод через имя структуры
5         MyStruct.staticMethod()
6         //вызываем статический метод через self.dynamicType
7         self.dynamicType.staticMethod()
8     }
9 }
```

В новой версии для получения доступа к типу необходимо использовать конструкцию `Self` (с прописной буквы). Пример приведен в листинге П.27.

Листинг П.27

```
1 struct MyStruct {
2     static func staticMethod(){...}
3     func instanceMethod(){
4         //вызываем статический метод через Self
5         Self.staticMethod()
6     }
7 }
```

Добавлена ленивая реализация flatMap(_:) для последовательности из опционалов

Ранее, если применялся метод `flatMap(_)`, возвращающий последовательность опционалов к lazy-последовательности, то возвращалась классическая последовательность (листинг П.28).

Листинг П.28

```
1 (1..10) [1, 2, 3, 4, 5]
2 .lazy
3 .flatMap { n in n % 2 == 0 ? n/2 : nil }
```

Теперь в подобном случае будет возвращаться lazy-коллекция (листинг П.29).

Листинг П.29

```
1 (1..10)
2   .lazy
3   .flatMap { n in n % 2 == 0 ? n/2 : nil } LazyMapBidirectional
                                         Collection...
```

Добавлены инициализаторы для типов `Int` и `UInt` для конвертации из типов `UnsafePointer` и `UnsafeMutablePointer`

В ранних версиях Swift вы могли конвертировать `Int` и `UInt` в `UnsafePointer` и `UnsafeMutablePointer` соответственно, но не наоборот. Теперь эта проблема решена. Сделано это в первую очередь для лучшей совместимости с API, написанными на Objective-C.

Тип `Unmanaged` использует `UnsafePointer`

Универсальная структура `Unmanaged<T>` — обертка для произвольного типа данных. Используя ее, вы становитесь ответственным за существование созданного экземпляра, так как она не использует механизмы ARC (автоматического подсчета ссылок), а значит, область памяти, содержащая данные, обернутые в `Unmanaged`, не будет удалена при уничтожении последней ссылки на экземпляр.

Члены swift-сообщества обратили внимание разработчиков на ряд проблем, связанных с использованием этого типа данных. В результате он был частично переписан для использования типа `UnsafePointer`.

Тестирование

Процесс разработки программ состоит не только из написания кода: проектирования, прототипирования, разработки, тестирования, релиза и т. д. Xcode позволяет вам не только создавать проекты, написав их код, но и тестировать отдельные модули в ходе разработки. В новой версии Xcode для тест-файлов в рамках вашего проекта создаются отдельные папки.

Изменены модификаторы области видимости

В предыдущей версии Swift не хватало модификатора, который бы ограничивал область видимости некоторого объекта текущим контекстом. В этой версии система разграничения слегка переработана, теперь существуют четыре модификатора:

- `public` — объект виден вне реализации модуля;
- `internal` — объект виден внутри реализации модуля;
- `fileprivate` — бывший `private`-объект может быть использован только в пределах одного исходного файла;

- ❑ `private` — объект может быть использован только в пределах конструкции, в которой объявлен. Например, объявленный в классе параметр не будет доступен в его расширениях.

Ранее для функции, имеющей несколько входных параметров, существовали два способа передачи этих параметров (листинг П.30).

Добавлен метод `first(where:)` для типа `Sequence`

Для любых последовательностей (конечно же, включая коллекции) добавлен метод `first(where:)`, который возвращает первое попавшееся значение, подпадающее под условие, переданное в аргументе `where`. Пример приведен в листинге П.30.

Листинг П.30

```
1 var array = [1, 2, 3, 4, 5]
2 var result = array.first(where: {$0>3}) // вернет 4
```

Импорт констант Objective-C в виде типов данных

При импорте API, написанных на Objective-C и содержащих константы, в автоматическом режиме будут определяться перечисления или структуры, полностью описывающие импортируемые константы.

Точка перед членом перечисления

В Swift 3 при доступе к члену перечисления перед его именем всегда ставится точка вне зависимости от того, где этот член используется (листинг П.31).

Листинг П.31

```
1 // точка ставится при доступе внутри перечисления
2 enum Coin {
3     case heads, tails
4     func printMe(){
5         switch self {
6             case .heads: print ("Орел")
7             case .tails: print ("Решка")
8         }
9         // ...
10    }
11 }
12 // также при доступе вне перечисления
```

```
13 switch Coin(){
14     case .heads: print ("Орел")
15     case .tails: print ("Решка")
16 }
```

Уточнено взаимодействие комментариев и операторов

Ранее при совместном использовании операторов и комментариев в рамках одного выражения в некоторых случаях Xcode мог сообщить об ошибке, которая по сути ошибкой не являлась. Это было связано с обработкой префиксных, инфиксных и постфиксных операторов. Так, например, код, приведенный в листинге П.32, вызывал ошибку.

Листинг П.32

```
1 if /* комментарий */!foo { ... }
```

В новой версии языка вы можете свободно использовать операторы и комментарии, не боясь того, что такой подход вызовет какие-либо ошибки.

Параметры атрибутов через точку

Для некоторых атрибутов, например `@available`, могут быть указаны параметры, которые уточняют работу этих атрибутов. Так, прежде значения этих параметров писали через знак равенства (=). Теперь он заменен двоеточием (:) (листинг П.33).

Листинг П.33

```
1 @available(*, unavailable, renamed: "MyRenamedProtocol") ...
```

Объявление параметров в конструкции switch-case

Теперь, если вы попытаетесь объявить параметры при указании нескольких шаблонов в case-блоке, вы не получите ошибку, как это было в Swift 2 (листинг П.34).

Листинг П.34

```
1 enum MyEnum
2     case Value1(Int, Float)
3     case Value2(Float, Int)
4 }
```

```

5
6  switch someValue {
7      case let .Value1(x, 2), let .Value2(2, x) : print(x)
                                                // раньше вызывало ошибку
8      case .Value1, .Value2: break
9  }

```

Уведомления о неиспользовании результатов работы функции

Если вы используете функцию, которая возвращает определенный результат, но возвращенное значение не используется нигде в коде, то Xcode сообщит вам об этом и по возможности предложит альтернативу, имеющую тип `Void` в качестве возвращаемого значения. Так, например, при использовании функции `sorted()` вам будет предложено использовать `sort()`, если вы не воспользовались возвращенным результатом ее работы.

Универсальные псевдонимы типов

Универсальные псевдонимы типов являются развитием идеи псевдонимов типов и позволяют значительно улучшить уже известный вам оператор `typealias`. Теперь вы можете использовать синтаксис универсальных шаблонов для определения псевдонимов (листинг П.35).

Листинг П.35

```

1  typealias StringDictionary<T> = Dictionary<String, T>
2  typealias Vec3<T> = (T, T, T)

```

Изменения в композиции протоколов

Конструкция `protocol<p1,p2>` заменена на `p1 & p2` (листинг П.36).

Листинг П.36

```

1  protocol A { }
2  protocol B { }
3
4  let a: A & B = SomeClass()
5
6  func someFunc(x: A & B) { ... }
   () ()-0045

```