

Айк Харазян

Язык **Swift**

Простые типы данных в Swift
Операторы и типы коллекций
Условные конструкции и циклы
Функции и замыкания
Перечисления и кортежи
Классы и структуры
ООП в Swift
Расширения и протоколы
Обобщенные типы



Айк Харазян

с а м о у ч и т е л ь

Язык Swift

Санкт-Петербург
«БХВ-Петербург»
2016

УДК 004.438 Swift
ББК 32.973.26-018.1
Х20

Харазян А. А.

Х20 Язык Swift. Самоучитель. — СПб.: БХВ-Петербург, 2016. — 176 с.: ил. — (Самоучитель)

ISBN 978-5-9775-3572-4

Книга предназначена для самостоятельного изучения Swift — нового языка программирования для iOS и OS X. Описана версия Swift 2.0. Материал построен по принципу от более легкого к сложному, изложение сопровождается большим количеством листингов кода, для тестирования и отладки используется новая среда быстрой разработки Playground. Объяснены основы Swift, синтаксис языка и его особенности. Описаны типы данных, условные выражения, циклы, массивы, функции, кортежи, базовые операторы и другие стандартные конструкции. Кратко даны основы объектно-ориентированного программирования. Подробно рассмотрены более сложные или специфические для Swift конструкции: перечисления, замыкания, опциональные типы, классы, структуры, встроенные и обобщенные типы, расширения, протоколы, расширенные операторы и др.

Для программистов

УДК 004.438 Swift
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	<i>Екатерина Кондукова</i>
Зам. главного редактора	<i>Евгений Рыбаков</i>
Зав. редакцией	<i>Екатерина Капалыгина</i>
Редактор	<i>Григорий Добин</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Корректор	<i>Зинаида Дмитриева</i>
Дизайн обложки	<i>Марины Дамбиевой</i>

Подписано в печать 31.07.15.
Формат 70×100¹/₁₆. Печать офсетная. Усл. печ. л. 14,19.
Тираж 1000 экз. Заказ №
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.
Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9 линия, 12/28

ISBN 978-5-9775-3572-4

© Харазян А. А., 2016
© Оформление, издательство "БХВ-Петербург", 2016

Оглавление

- Введение 9**
 - Для кого предназначена эта книга? 10
 - Какова польза от книги? 10
 - Структура книги..... 10
 - Тестируйте листинги кода! 11
 - Программа Apple для разработчиков..... 11
- ЧАСТЬ I. ОСНОВЫ SWIFT..... 13**
 - Глава 1. Swift: как он появился и с чего начать? 15**
 - 1.1. Как появился Swift? 15
 - 1.2. Что нужно, чтобы начать разрабатывать на Swift? 16
 - 1.3. Playground..... 16
 - 1.4. Как создать новый документ Playground? 16
 - Глава 2. Особенности синтаксиса Swift 20**
 - 2.1. Swift — это C-подобный язык 20
 - 2.2. Отсутствие заголовочных файлов 20
 - 2.3. Точки с запятой ставить не обязательно..... 20
 - 2.4. Набор символов 21
 - Выводы 21
 - Глава 3. Простые типы данных, переменные и константы 22**
 - 3.1. Переменные и константы 22
 - 3.2. Вывод информации в консоль 24
 - 3.3. Комментарии..... 24
 - 3.4. Статическая типизация и вывод типов 25
 - 3.5. Явное указание типов..... 26
 - 3.6. Литералы 27
 - 3.7. Числовые типы..... 27
 - 3.7.1. Целые числа 27
 - 3.7.2. Числа с плавающей точкой..... 28
 - 3.7.3. Способы записи числовых типов 28
 - 3.7.4. Преобразование числовых типов 29

3.8. Строки и символы.....	30
3.8.1. Конкатенация строк.....	30
3.8.2. Преобразование в строку.....	31
3.8.3. Интерполяция строк.....	31
3.9. Логические типы.....	31
3.10. Псевдонимы типов.....	31
Выводы	32
Глава 4. Базовые операторы	33
4.1. Оператор присваивания	34
4.2. Арифметические операторы	34
4.3. Составные операторы присваивания	34
4.4. Операторы инкремента и декремента.....	35
4.5. Операторы унарного минуса и унарного плюса	36
4.6. Операторы сравнения.....	36
4.7. Тернарный условный оператор	37
4.8. Операторы диапазона.....	37
4.9. Логические операторы	38
4.9.1. Логическое НЕ.....	38
4.9.2. Логическое И	38
4.9.3. Логическое ИЛИ.....	38
Выводы	38
Глава 5. Типы коллекций.....	40
5.1. Массивы	40
5.1.1. Объявление массива.....	40
5.1.2. Получение доступа к элементам массива.....	42
5.1.3. Добавление элементов в массив.....	43
5.1.4. Изменение элементов массива	44
5.1.5. Удаление элементов из массива.....	45
5.1.6. Итерация по массиву.....	46
5.2. Множества.....	46
5.2.1. Объявление множеств.....	47
5.2.2. Работа с множествами.....	47
5.2.3. Сочетание и сравнение множеств	49
5.3. Словари.....	50
5.3.1. Объявление словаря	50
5.3.2. Получение доступа к элементам словаря	51
5.3.3. Добавление элементов в словарь	51
5.3.4. Изменение элементов словаря.....	51
5.3.5. Удаление элементов из словаря	52
5.3.6. Итерация по словарю	52
Выводы	53
Глава 6. Ветвление потока	55
6.1. Условия.....	55
6.1.1. Условный оператор <i>if</i>	55
6.1.2. Оператор <i>switch</i>	57

6.2. Циклы	61
6.2.1. Циклы <i>for</i>	61
Стандартный цикл <i>for</i>	61
Цикл <i>for-in</i>	62
6.2.2. Цикл <i>while</i>	63
6.3. Управление потоком цикла.....	64
6.4. Оператор <i>guard</i>	65
6.5. Проверка на доступность API.....	66
Выводы	66

Глава 7. Функции

7.1. Объявление функции.....	67
7.2. Параметры	68
7.2.1. Внешние имена параметров	69
7.2.2. Параметры со значением по умолчанию.....	70
7.2.3. Сквозные параметры	70
7.2.4. Функции с переменным числом параметров	71
7.3. Возвращаемое значение функции	72
7.3.1. Функции с несколькими возвращаемыми значениями	73
7.4. Функции — объекты первого класса	75
7.4.1. Функции, принимающие параметры в виде функции	75
7.4.2. Функции, возвращающие функцию	76
7.4.3. Вложенные функции	77
Выводы	77

ЧАСТЬ II. УГЛУБЛЕННОЕ ИЗУЧЕНИЕ SWIFT

Глава 8. Опциональные типы

8.1. Опциональная привязка	83
8.2. Принудительное извлечение.....	83
8.3. Неявное извлечение	84
8.4. Опциональное сцепление	85
Выводы	85

Глава 9. Кортежи

9.1. Объявление кортежа.....	86
9.2. Получение доступа к элементам кортежа.....	86
9.2.1. Использование индекса элемента	86
9.2.2. Разложение кортежа	87
9.3. Именованное элементов кортежа.....	87
9.4. Использование кортежей	88
9.4.1. Массовое присвоение.....	88
9.4.2. В циклах <i>for-in</i>	88
9.4.3. В качестве возвращаемого значения для функций	88
9.5. Опциональный кортеж	89
Выводы	90

Глава 10. Замыкания.....

10.1. Сокращенные имена параметров замыкания	94
--	----

10.2. Операторы-функции	94
10.3. Последующее замыкание	94
Выводы	96
Глава 11. Перечисления	97
11.1. Объявление перечислений	98
11.2. Перечисления и оператор <i>switch</i>	98
11.3. Связанные значения	99
11.4. Исходные значения	100
11.5. Вложенные перечисления	102
Выводы	103
Глава 12. Классы	104
12.1. Свойства, методы и объекты класса	104
12.2. Объявление классов	105
12.3. Свойства класса	105
12.3.1. Ленивые свойства	107
12.3.2. Вычисляемые свойства	108
12.3.3. Наблюдатели свойств	111
12.3.4. Вычисляемые переменные и наблюдатели для переменных	111
12.3.5. Свойства типа	112
12.4. Инициализаторы	113
12.4.1. Инициализатор по умолчанию	113
12.4.2. Инициализаторы с параметрами	114
12.4.3. Локальные и внешние имена параметров инициализатора	116
12.4.4. Проваливающиеся инициализаторы	117
12.4.5. Деинициализаторы	118
12.5. Методы	118
12.5.1. Создание методов	118
12.5.2. Методы типа	120
12.6. Индексаторы	121
12.6.1. Синтаксис индексаторов	121
12.6.2. Многомерные индексаторы	121
Выводы	124
Глава 13. Наследование	126
13.1. Переопределение	127
13.2. Наследование инициализаторов	127
13.3. Переопределение инициализаторов	131
13.4. Назначенные и удобные инициализаторы	132
13.5. Необходимые инициализаторы	134
Выводы	135
Глава 14. Автоматический подсчет ссылок	136
14.1. Принципы работы автоматического подсчета ссылок	136
14.2. Циклы сильных ссылок внутри объектов классов	137
14.3. Решение проблемы циклов сильных ссылок между объектами классов	138
14.3.1. Слабые ссылки	139
14.3.2. Ссылки без владельца	139
Выводы	140

Глава 15. Структуры	141
15.1. Типы-значения и ссылочные типы	141
15.2. Оператор идентичности	141
15.3. Свойства структур	142
15.4. Свойства типа для структур	143
15.5. Методы структур	144
15.6. Методы типа для структур	144
15.7. Инициализаторы структур	145
Выводы	145
Глава 16. Проверка типов и приведение типов	146
16.1. Проверка типов	146
16.2. Приведение типов	148
16.3. Проверка типов <i>Any</i> и <i>AnyObject</i>	150
16.3.1. Тип <i>AnyObject</i>	150
16.3.2. Тип <i>Any</i>	151
Выводы	152
Глава 17. Расширения	153
17.1. Расширение свойств	153
17.2. Расширение методов	154
17.3. Расширение инициализаторов	155
Выводы	155
Глава 18. Протоколы	156
18.1. Объявление протокола	156
18.2. Требования для свойств	156
18.3. Требования для методов	158
18.4. Требования для инициализаторов	160
18.5. Протоколы как типы	160
18.6. Соответствие протоколу через расширение	161
18.7. Наследование протоколов	161
18.8. Протоколы только для классов	161
18.9. Сочетание протоколов	162
18.10. Проверка объекта на соответствие протоколу	162
18.11. Расширения протоколов	162
Выводы	163
Глава 19. Обобщенные типы	164
19.1. Обобщенные функции	164
19.2. Обобщенные типы	165
19.3. Ограничения типов	166
Выводы	166
Глава 20. Обработка ошибок	167
Выводы	168
Глава 21. Расширенные операторы	169
21.1. Оператор объединения по нулевому указателю	169

21.2. Операторы с переполнением	169
21.2.1. Переполнение значения	170
21.2.2. Потеря значения	170
21.3. Перегрузка операторов.....	170
21.4. Побитовые операторы	171
21.4.1. Побитовый оператор <i>NOT</i>	171
21.4.2. Побитовый оператор <i>AND</i>	171
21.4.3. Побитовый оператор <i>OR</i>	172
21.4.4. Побитовый оператор <i>XOR</i>	172
21.4.5. Побитовые операторы левого и правого сдвига	172
Выводы	172
Заключение.....	173
Изучайте фреймворки Apple.....	173
Вступайте в Apple’s Developer Program.....	173
Вперед, к новым высотам!	173

Введение

2 июня 2014 года на ежегодной конференции разработчиков WWDC компания Apple представила новый язык программирования Swift. С его помощью можно создавать приложения для iOS и OS X с еще большей легкостью, чем ранее. Swift сочетает в себе производительность и эффективность компилируемых языков программирования с простотой и интерактивностью популярных скриптовых языков. Он был вдохновлен такими языками программирования, как JavaScript, Python и Ruby.

При создании языка Swift разработчики руководствовались тремя парадигмами:

- быстрота;
- современность;
- надежность.

По словам Apple, некоторые алгоритмы, написанные на Swift, выполняются в разы быстрее, чем те же алгоритмы на Python. Этим она утверждает, что код, написанный на Swift, не только лаконичен и легко читается, но и может конкурировать по скорости с мощными скриптовыми языками. Современность языка демонстрируют особенности, перенятые из скриптовых языков программирования, — такие как интерактивность и простота восприятия. Разработчики очень тщательно подошли к реализации возможностей, которые помогают противостоять появлению ошибок, тем самым делая код более надежным. Например, переменные всегда должны быть объявлены перед их использованием, а массивы и целые числа проверяются на перегрузку. Память очищается автоматически, а технология опциональных типов позволяет исключить возможные ошибки с отсутствием значения.

Еще одной важной особенностью Swift являются песочницы Xcode Playground, которые позволяют тестировать код и видеть результат его выполнения в реальном времени. До их появления приходилось экспериментировать на кусках кода внутри целых проектов.

Для кого предназначена эта книга?

Необходимость изучения нового языка программирования может сильно пугать новичков. Но эта книга прекрасно подойдет даже тем, кто никогда не использовал языки программирования C, C++ или Objective-C. И хотя книга не рассчитана на тех, кто знакомится с языками программирования впервые, но, все же, в начале каждой главы даны краткие объяснения основ программирования, касающиеся освещаемой темы.

Опытных же разработчиков в первую очередь заинтересуют особенности и новшества языка Swift, которым уделено особое внимание, а также материалы второй части книги по углубленному изучению программирования на Swift.

Изложение каждой новой темы в книге сопровождается подробными примерами на языке Swift. А сами главы построены по принципу последовательного изучения материала от более легкого к более сложному. Это облегчает поиск нужных глав для опытных разработчиков и помогает новичкам изучать Swift постепенно.

Какова польза от книги?

В книге рассмотрены все особенности языка Swift, которые в будущем помогут вам при разработке приложений и игр под iOS и OS X. Изучив материал книги, вы сможете полностью разбираться в коде, написанном на Swift, а также самостоятельно разрабатывать алгоритмы на этом языке. Знания, изложенные в книге, лежат в основе фреймворков Cocoa и Cocoa Touch, служащих для разработки приложений и игр под iPhone, iPad, Mac и т. д. Так что, изучив техники языка программирования Swift, вы смело можете приступить к изучению API и фреймворков и разработке приложений для iOS и OS X.

Структура книги

Книга состоит из двух больших частей.

- **Часть I. Основы Swift.** Первая часть книги знакомит читателя с тем, как реализованы в Swift стандартные структуры вроде циклов, условий и массивов. Она будет полезна для начинающих разработчиков, поскольку расскажет им об основах программирования. А опытным разработчикам эта часть даст понять, какие особенности есть в простых структурах Swift в отличие от других языков.
- **Часть II. Углубленное изучение Swift.** Вторая часть книги описывает более сложные структуры и понятия, которые не встречаются в других языках программирования. Она будет полезна для опытных разработчиков, которые хотят углубиться в разработку на Swift более сложных приложений. Для новичков же важно сначала изучить главы из первой части, и только затем приступить ко второй.

Тестируйте листинги кода!

Каждая глава книги сопровождается большим количеством листингов кода. Они позволяют лучше понять некоторые алгоритмы и операции. Мы рекомендуем вам по ходу изучения материала набивать предлагаемые программные коды непосредственно вручную. Так вы легче запомните синтаксис языка Swift, поскольку будете учиться на своих ошибках.

А помогут вам в этом уже упомянутые песочницы Playground — новый тип документа Xcode, который появился в его шестой версии, вместе с языком Swift. Playground представляет собой самостоятельный файл, куда можно вписать несколько строк кода, которые будут тут же выполнены с выводом полученного результата. Более подробно о Playground рассказано в *главе 1*.

Программа Apple для разработчиков

Чтобы получать последнее программное обеспечение для разработчиков и иметь возможность публиковать свои приложения в магазине цифровой дистрибуции AppStore, вы, вероятно, захотите стать членом Apple Developer Programs. Найти подробную информацию о привилегиях, предоставляемых вступившим в эту программу разработчикам, вы можете на странице сайта Apple по адресу: <https://developer.apple.com/programs>.



ЧАСТЬ I

Основы Swift

В первой части этой книги мы познакомимся с простыми элементами Swift, которые встречаются в большинстве языков программирования. Но сначала мы узнаем, как Swift появился на свет, чем разработчики пользовались до него и с чего следует начать с ним знакомство.

- ❑ Глава 1. Swift: как он появился и с чего начать?
- ❑ Глава 2. Особенности синтаксиса Swift.
- ❑ Глава 3. Простые типы данных, переменные и константы.
- ❑ Глава 4. Базовые операторы.
- ❑ Глава 5. Типы коллекций.
- ❑ Глава 6. Ветвление потока.
- ❑ Глава 7. Функции.

ГЛАВА 1



Swift: как он появился и с чего начать?

1.1. Как появился Swift?

Язык для человека является универсальным средством коммуникации и обмена информацией. С его помощью мы можем высказать наши мысли друзьям, членам семьи или коллегам по работе. И компьютерные языки программирования в этом плане также не являются исключением. Принцип выражения своих мыслей в них тот же. Только на этот раз мы высказываем их компьютеру.

Языки программирования, подобно человеческим языкам, существуют в разных формах и со временем эволюционируют и совершенствуются. Главной задачей языков программирования изначально являлась необходимость взаимодействовать с инструкциями компьютера, чтобы задать набор определенных действий, которые мы хотим от него получить.

Опытные пионеры компьютерной эпохи знают, что центральный процессор компьютера понимает только набор инструкций, состоящий из нулей и единиц. Чтобы разработчики могли разрабатывать свой код на более понятном им языке, чем нули и единицы, существуют программы-компиляторы, которые переводят код, написанный на определенном языке программирования, в инструкции для процессора, состоящие из тех самых нулей и единиц.

Компромисс между возможностями языка программирования и его производительностью обеспечили такие языки программирования, как C и C++. Но пока языки C и C++ распространялись на все большее количество платформ, Apple решила пойти иным путем и разработала язык Objective-C. Построенный на основе C, Objective-C сочетал в себе мощь широко используемого языка программирования с объектно-ориентированными методологиями. На протяжении многих лет Objective-C являлся основой для разработки программ для компьютеров Macintosh и iOS-устройств.

Однако, несмотря на производительность и простоту, Objective-C носил с собой весь багаж, наследованный от C. Для разработчиков, которые разбираются в C, это не вызывает трудностей. Но новички жаловались, что Objective-C труден для понимания и разработки.

Apple прислушалась к мольбам начинающих разработчиков и снизила для них входной барьер, выпустив язык программирования Swift. Теперь писать программы получается намного легче и понятнее.

1.2. Что нужно, чтобы начать разрабатывать на Swift?

Прежде всего, мы настоятельно рекомендуем писать листинги кода вместе с нами. Это позволяет лучше запомнить конструкции языка Swift. Для того чтобы вместе с нами писать и запускать код Swift, вам понадобится компьютер Macintosh с операционной системой не ниже OS X 10.9 Mavericks или 10.10 Yosemite. Вам также потребуется Xcode 7, в котором содержится компилятор Swift и его среда разработки.

1.3. Playground

Во время разработки какого-либо приложения нам часто хочется иметь возможность быстро протестировать несколько строчек кода. Обычно для этого приходится создавать новый проект, который формирует новую папку, генерирует конфигурационные файлы, пишет классы по умолчанию и т. д. И все это делается лишь для того, чтобы мы могли протестировать несколько кодовых строк.

В Swift нам этого делать не придется — чтобы протестировать свои алгоритмы мы можем воспользоваться уже упомянутым во *введении* Playground — новым типом документа Xcode, представляющим собой самостоятельный файл, в котором можно написать несколько строк кода для тестирования и не беспокоиться более о создании новых проектов.

Быстрое создание «площадки для тестирования» — не единственное преимущество Playground. Главным его отличием от проектов является немедленное выполнение любой написанной строки кода с выводом результатов выполнения этой строки. Они автоматически обновляются и выполняются по мере изменения документа.

Из сказанного можно сделать вывод, что Playground является очень удобным средством для углубления в Swift. Используйте его для достижения лучших и быстрых результатов в процессе изучения этого языка.

1.4. Как создать новый документ Playground?

Чтобы создать новый файл Playground, нам нужно запустить Xcode 7. Сразу после запуска откроется окно приветствия, содержащее меню, первым пунктом которого будет: **Get started with a playground** (рис. 1.1). Нам нужно выбрать именно этот пункт.

В результате откроется окно с предложением выбора имени для нового документа и типа платформы для разработки: OS X или iOS (рис. 1.2).

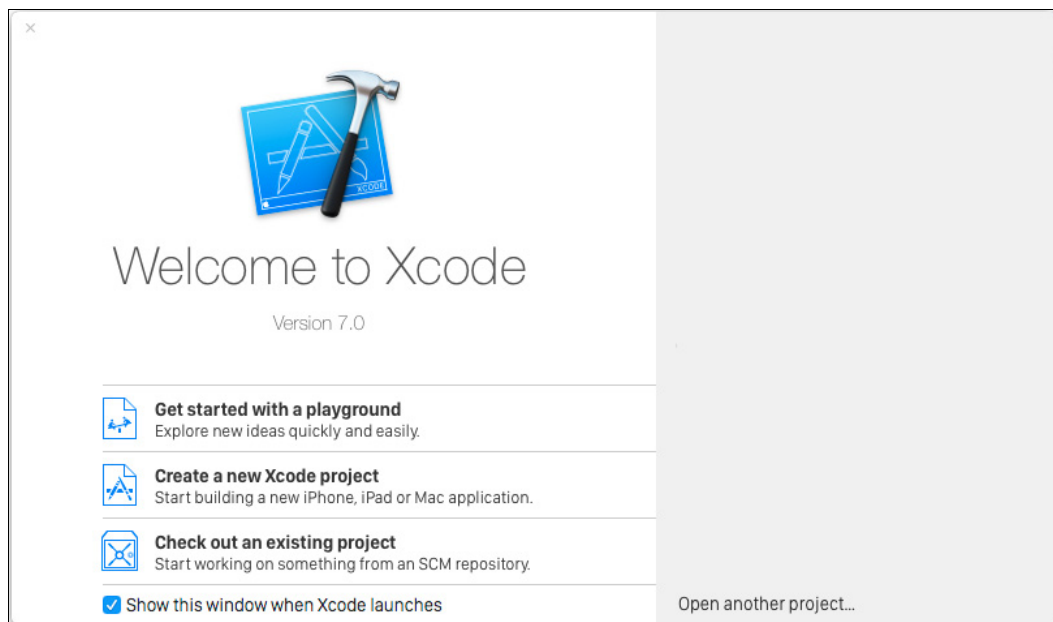


Рис. 1.1. Окно приветствия Xcode

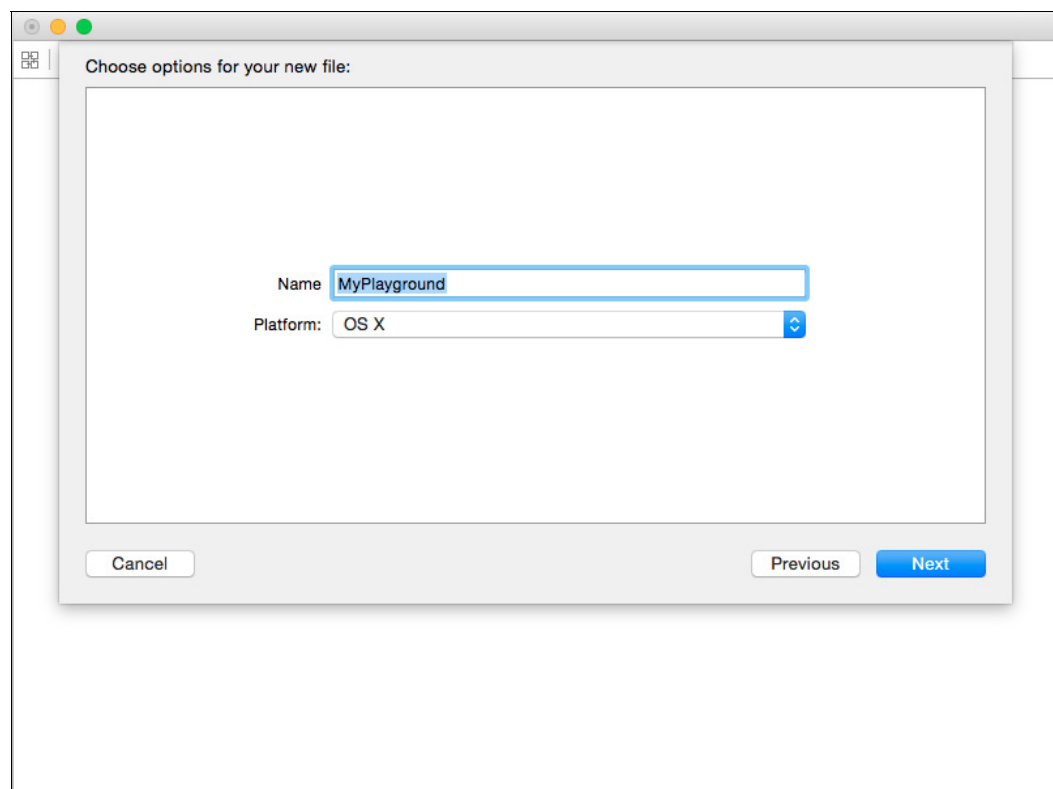


Рис. 1.2. Окно выбора имени и типа платформы

Указав имя и платформу, мы можем нажать кнопку **Next** и переместиться в диалоговое окно выбора места для сохранения нашего Playground. Выберите нужное местонахождение и нажмите на кнопку **Create**, чтобы окончательно его создать.

Если мы все правильно сделали, то должны увидеть новое окно Playground (рис. 1.3).

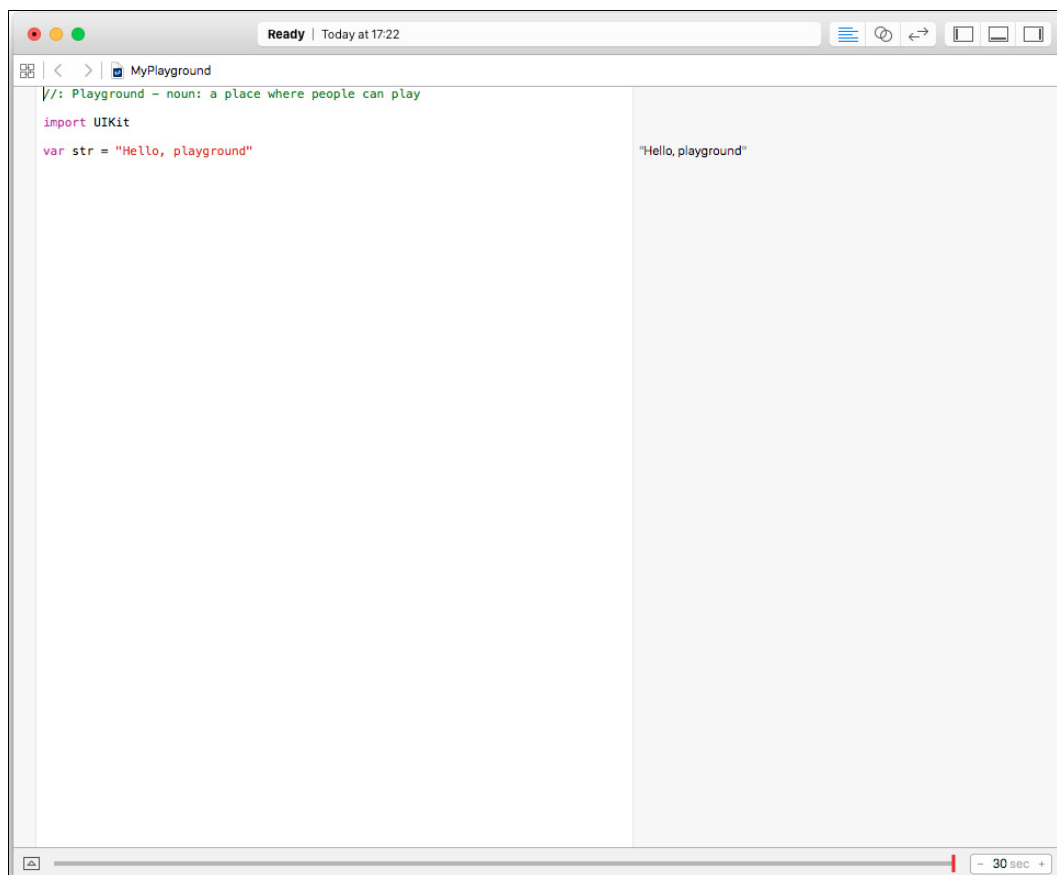



Рис. 1.3. Новое окно Playground

Теперь мы можем писать свои первые строчки кода на Swift. Левая область окна Playground предназначена для нашего кода, а правая — отображает результат выполнения этого кода.

В правой области отображается также любая полезная информация о значениях, содержащихся в переменных, о количествах итераций циклов, о возвращаемых значениях функций. Эти значения носят информативный характер и не совпадают со значениями, которые выводит наша программа. Для того чтобы узнать, какие данные выводит программа, нужно в правом верхнем углу окна найти три кнопки , отвечающие за дополнительные панели с левой, с нижней и правой

его сторон. Нажать нужно на центральную кнопку, отвечающую за консольный вывод (рис. 1.4).

Теперь мы можем приступить к изучению Swift. Пишите и тестируйте листинги кода вместе с нами!

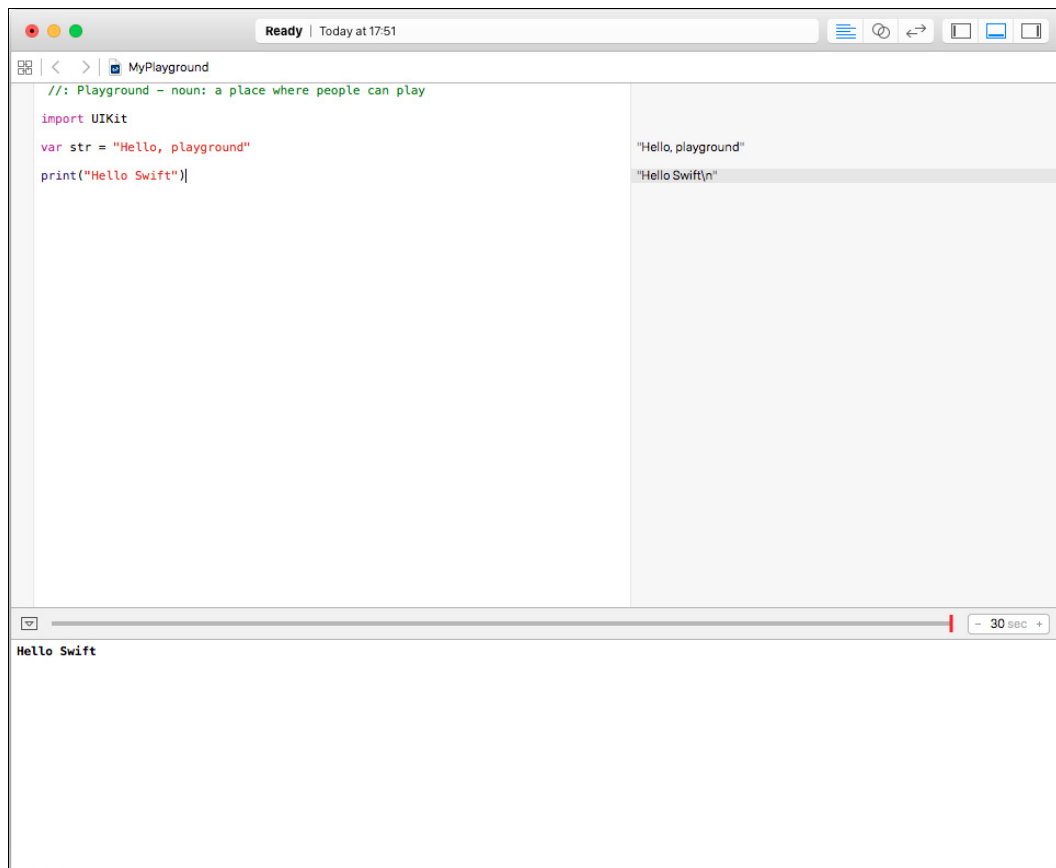


Рис. 1.4. Окно Playground с областью консольного вывода

ГЛАВА 2



Особенности синтаксиса Swift

Синтаксис языка программирования — это набор правил, определяющих, как пишутся программы на этом языке. Он определяет, какие символы будут употреблены для обозначения переменных, как нужно разделять инструкции друг от друга, какие конструкции употребляются в том или ином случае и т. д.

2.1. Swift — это C-подобный язык

Первой особенностью синтаксиса языка Swift является его принадлежность к группе C-подобных языков программирования. Разработчики, которые уже знакомы с C-подобными языками, найдут в Swift много знакомых конструкций. Блоки кода все также разделяются парой фигурных скобок, а циклы, условия и функции содержат знакомые нам ключевые слова `if`, `else`, `for`, `return`. Но, в отличие от Objective-C, мы не можем взять код, написанный на C, и вставить его в программу, написанную на Swift. Если в Objective-C мы могли так сделать, и все бы работало, то со Swift это не пройдет. Это означает, что Swift — заново написанный язык, а не просто надстройка над языком C.

2.2. Отсутствие заголовочных файлов

Следующей особенностью синтаксиса Swift является отсутствие заголовочных файлов. Программа, написанная на Swift, теперь не нуждается в отдельно написанных для него заголовочных файлах. Исполняемые файлы программы теперь имеют только один формат: `.swift`.

2.3. Точки с запятой ставить не обязательно

Одной из особенностей языка Swift является отсутствие необходимости ставить точку с запятой после каждого выражения. Теперь компилятор сам понимает, когда выражение закончилось. Но возможность использовать точки с запятой есть. Так,

в тех случаях, когда нам нужно записать несколько выражений в одну строку, мы должны после каждого выражения поставить точку с запятой. Либо, если вы в прошлом программировали на каком-нибудь C-подобном языке и по привычке ставите после каждого выражения точку с запятой, то можете не отказываться от этой привычки. Это означает, что вставлять точки с запятой не обязательно, но при желании вы их использовать можете.

2.4. Набор символов

При написании программ в Swift можно использовать символы Unicode. Но, в отличие от других языков программирования, где символы Unicode допускается употреблять только для строк и символов, в Swift символы Unicode можно использовать в названиях переменных, функций и других объектов. То есть, например, писать названия переменных на русском языке не возбраняется.

Кроме стандартных символов UTF-8, Swift также поддерживает UTF-16 и UTF-32. Этот факт сильно расширяет набор допустимых символов. Мы можем писать названия для переменных даже символами «эмодзи»¹.

Выводы

- ❑ Swift — заново написанный язык программирования, который относится к семейству C-подобных языков.
- ❑ Исходный код, написанный на Swift, теперь имеет только один тип: `.swift`, и не нуждается в заголовочных файлах.
- ❑ В конце каждой инструкции не обязательно ставить точку с запятой.
- ❑ Переменные, функции и другие объекты можно писать любыми символами Unicode.

¹ Эмодзи — язык идеограмм и смайликов, используемый в электронных сообщениях и на веб-страницах.

ГЛАВА 3



Простые типы данных, переменные и константы

Самым простым понятием в языках программирования являются *переменные*. Они позволяют хранить нужные нам значения в памяти компьютера. В Swift, наряду с переменными, существуют еще и *константы*. Если в переменных значения могут быть изменены, то значения констант после первого присвоения изменить невозможно.

Употребление констант в Swift сильно отличается от их употребления в других языках программирования. Если в других языках мы редко применяли константы, то в Swift константы используются так же часто, как и переменные. Apple рекомендует употреблять константы везде, где это возможно. Например, если нам нужно хранить информацию о дате рождения человека, то, по рекомендациям Apple, это значение лучше всего хранить внутри константы. Поскольку дата рождения человека не должна никогда изменяться, то использование константы позволит обезопасить это значение от случайного изменения. Кроме того, с использованием констант компилятор позволяет генерировать более оптимизированный код.

В этой главе мы рассмотрим, как создаются переменные и константы. Познакомимся с тем, что собой представляет механизм вывода типов, и где полезно явно указывать типы. После этого мы детально изучим каждый из простых встроенных типов значений в Swift. А в конце главы познакомимся с псевдонимами типов.

3.1. Переменные и константы

Давайте начнем с простой конструкции присвоения переменной значения. В Swift переменные объявляются через ключевое слово `var`. Название переменной мы можем придумать любое, но оно не должно начинаться с цифры:

```
var myHome = "Earth"
```

Эта строчка кода является законченной программой. Больше не нужно подключать дополнительные библиотеки и писать всю программу внутри глобальной функции `main`, как мы это делали в Objective-C.

В приведенном примере мы решили использовать название переменной `myHome`. Apple советует при написании имен для переменных, констант и других объектов придерживаться *горбатой нотации*. Согласно ее принципам, мы должны следовать правилу: если название состоит из нескольких слов, мы записываем всех их вместе, но каждое новое слово пишем с большой буквы. По-английски эту нотацию называют CamelCase (верблюжий стиль). Она перекочевала в Swift из Objective-C, где мы в таком же стиле писали названия переменных, функций и т. д. Следует заметить, что горбатая нотация не обязательна для соблюдения, но Apple рекомендует это делать, чтобы код был одинаково читаем для всех разработчиков.

Как мы отмечали в предыдущей главе, при задании имен переменных и других объектов у Swift, по сравнению с Objective-C, имеется одна интересная особенность — имена переменных могут состоять из любых Unicode-символов. А это означает, что названия переменных в Swift мы можем писать также и на русском языке. Да что там говорить — даже символ торговой марки (®) является в Swift приемлемым названием для переменной! А поскольку Swift поддерживает не только UTF-8, но еще UTF-16 и UTF-32, то мы можем использовать в названии переменных и символы «эмодзи»:

```
var 🌍 = "Earth"
```

Если вы заметили, то после приведенного выражения мы не поставили в конце точку с запятой. Действительно, в Swift, как уже и отмечалось ранее, не обязательно ставить точку с запятой после каждой строчки. Единственный случай, когда точку с запятой все же нужно ставить, — это когда мы пишем несколько выражений в одну строчку, например:

```
var myHome = "Earth"; var myShip = "Enterprise";
```

В Swift, наряду с переменными, существуют и константы. Они представляют собой те же переменные, только с постоянным значением. При попытке изменить значение константы, которой уже присвоили значение, Swift выведет ошибку. В остальном константы полностью идентичны переменным. Константы объявляются через ключевое слово `let`:

```
let myName = "James Kirk"
```

В отличие от других языков программирования, в Swift константы используются так же часто, как и переменные. Суть констант проста: когда у нас есть значение, которое никогда не будет изменяться (например, чья-то дата рождения), то настоятельно рекомендуется объявить ее в виде константы.

На первый взгляд вам может показаться непонятным, почему в Swift присутствуют и константы, и переменные? Разве переменная не предоставляет больше возможностей, чем константа? Это хороший вопрос. Ответ на него скрыт в основе работы компилятора. Компилятор в Swift может лучше оптимизировать код, когда знает, что значение, записанное в памяти, никогда не будет изменяться. Поэтому всегда используйте константы там, где вы точно знаете, что значение меняться не будет.

По мере разработки приложений на Swift вы все чаще будете использовать константы. Грубо говоря, с константами меньше головной боли. Мы тем самым защищаем наши значения от внезапного изменения в коде.

3.2. Вывод информации в консоль

В наших примерах мы часто будем пользоваться функцией `print(_:)`. Она выводит в консоль Swift информацию, которую мы передаем ей внутри скобок. Эта информация окажется очень полезна при тестировании функционала нашего кода.

Например, мы можем вывести в консоль значение переменной `myHome` из предыдущего раздела:

```
var myHome = "Earth"
print(myHome)
```

Эти строки кода выведут в консоль слово "Earth".

3.3. Комментарии

По мере увеличения количества строк кода нам иногда становится сложно найти нужный кусок. Или бывает так, что мы смотрим на код, написанный нами неделю назад, и не понимаем, что мы тут написали.

Чтобы решить проблемы такого характера, в языках программирования существуют *комментарии* — блоки текста, которые игнорируются компилятором. Они созданы для нас, чтобы мы могли делать в коде для себя пометки. Например, написав сложный код, мы можем добавить комментарий по поводу его работы, чтобы будущие разработчики не мучались с изучением этого куска кода, а сразу поняли, в чем здесь дело. Составление комментариев является хорошей практикой для любого языка программирования, т. к. они упрощают повторное восприятие информации и помогают лучше организовать код.

В Swift комментарии пишутся после двойного знака косой черты (слеша): `//`. Увидев в коде сочетание двух слешей, компилятор проигнорирует все, что написано правее такого сочетания на этой строке:

```
// эта часть кода будет проигнорирована компилятором
```

Кроме однострочных комментариев, в Swift можно писать еще и многострочные комментарии. Для них нам нужно использовать знак косой черты со знаком звездочки (астериска): `/*` — для начала многострочного комментария, и обратное сочетание: `*/` — для его конца:

```
/*
Эти
строки
кода
будут
проигнорированы
компилятором
*/
```

В отличие от других языков программирования, в Swift многострочные комментарии могут быть вложенными:

```
/*  
Начало первого многострочного комментария  
/*  
Начало второго многострочного комментария  
Конец второго многострочного комментария  
*/  
Конец первого многострочного комментария  
*/
```

Вложенные многострочные комментарии были введены в Swift для облегчения ситуаций, когда нам нужно закомментировать большой фрагмент кода, в котором могут встречаться другие многострочные комментарии. В других языках программирования нужно было бы сначала избавиться от вложенного комментария и лишь потом комментировать нужный фрагмент кода. Но в Swift об этом можно не беспокоиться.

В наших примерах мы будем использовать комментарии для иллюстрации результата той или иной программы. Например, мы напишем:

```
// Напечатает в консоли слово "Earth"
```

Иногда мы также можем воспользоваться условным обозначением возвращения какого-либо значения (`=>`):

```
// => 34
```

Такая строка комментария в наших примерах условно будет означать, что функция или какая-нибудь другая конструкция возвращает число 34.

3.4. Статическая типизация и вывод типов

Чтобы классифицировать в языках программирования разные значения, их делят по *типу*. Когда компилятор знает, какой тип будет храниться в переменной или константе, то он выделит под нее ячейку памяти соответствующего размера. Это означает, что, например, переменная со строковым значением "Hello World" и переменная, хранящая числовое значение 23, в памяти занимают разное количество памяти.

По способу проверки компилятором типов переменных или констант языки программирования делятся на языки со статической типизацией и языки с динамической типизацией. При *статической типизации* типы переменных устанавливаются на момент компиляции. То есть, если переменной не указан тип, во время компиляции программа выдаст ошибку. При *динамической типизации* типы переменных выясняются во время выполнения программы — т. е. указывать тип каждой переменной не нужно.

Из первых строк, написанных нами на Swift, может показаться, что Swift имеет динамическую типизацию, т. к. мы нигде не указывали тип хранимого значения для переменных и констант.

Но Swift — язык со статической типизацией, а это означает, что переменные обязательно должны быть определенного типа, и этот тип не может меняться в дальнейшем. Как же тогда может быть правильной наша конструкция присвоения, приведенная ранее?

В Swift существует механизм, называемый «выводом типов». Когда мы присваиваем переменной какое-либо значение, компилятор на основе присвоенного ей значения сам предугадывает, какой тип назначить этой переменной. То есть, в нашем ранее приведенном примере компилятор понимает, что переменной `myHome` мы присваиваем строковое значение, и, следуя из этого, переменная должна иметь тип строковой переменной (`String`). Но при этом не стоит забывать, что, присвоив один раз тип переменной, мы больше не можем его менять, — даже, если этот тип за нас был предугадан механизмом вывода типов.

3.5. Явное указание типов

Кроме использования вывода типов, в Swift также можно явно указать тип переменной, поставив после названия переменной двоеточие и написав затем название типа:

```
var myHome: String = "Earth"
```

Мы можем объявить и пустую переменную, т. е. не присваивать ей какого-либо значения. Но, так как компилятор должен обязательно знать, сколько памяти выделить под эту переменную, нам обязательно следует указать тип:

```
var myShip: String  
myShip = "Enterprise"
```

В приведенном примере мы сначала создали пустую переменную `myShip` с типом `String`, а на следующей строке присвоили ей значение.

Если мы попробуем объявить переменную и указать для нее только тип — без значения, а затем выведем ее значение в консоль, то получим ошибку:

```
var myShip: String  
print(myShip) // => Ошибка
```

Дело в том, что мы можем использовать переменную только после того, как ей будет присвоено какое-либо значение. И только тогда мы избавимся от ошибки:

```
var myShip: String  
myShip = "Enterprise"  
print(myShip) // => "Enterprise"
```

Мы можем попытаться присвоить этой переменной значение другого типа — чтобы убедиться, что менять тип переменной или константы после первого указания нельзя:

```
myShip = 150 // ошибка компиляции
```

Здесь мы попробовали присвоить числовое значение переменной, которая хранит строковые значения.

Конструкция указания типа очень часто будет встречаться в книге и поэтому очень важно помнить, что двоеточие в Swift означает «имеет тип». Часто этот знак будет употребляться в очень запутанных выражениях, так что запомните: двоеточие — это указание типа.

3.6. Литералы

Литералами в языках программирования называют значения, встречающиеся непосредственно в коде. В переводе с английского языка *literal value* означает «буквальное значение», т. е. значение, как оно есть. Для примера давайте рассмотрим литералы нескольких типов:

```
22      // литерал целого числа
2.3     // литерал числа с плавающей точкой
"Hello" // строковый литерал
"н"     // литерал символа
true    // литерал логического значения
```

3.7. Числовые типы

3.7.1. Целые числа

Одним из базовых встроенных типов данных в Swift является тип `Int` — целое число. Диапазон значений, который может хранить тип `Int`, зависит от платформы, на которой запускается код. Если платформа 32-битная, то `Int` может хранить значения от $-2\,147\,483\,648$ до $2\,147\,483\,647$. А если платформа 64-битная, то диапазон хранимых чисел расширяется от $-9\,223\,372\,036\,854\,775\,808$ до $9\,223\,372\,036\,854\,775\,807$.

В повседневных ситуациях использование простого `Int` предпочтительнее, но мы можем также явно указать нужный тип целого числа: 32-битное целое число: `Int32`, либо 64-битное: `Int64`.

Если использование таких больших чисел не предполагается, то для оптимизации кода мы можем объявлять типы, хранящие меньший диапазон чисел. Для этого в Swift есть типы `Int16` и `Int8`, которые могут хранить числа от $-32\,768$ до $32\,767$ и от -128 до 127 соответственно.

Также, когда в переменной не предполагается хранить отрицательные значения, мы можем использовать беззнаковые целые типы: `UInt`, `UInt32`, `UInt64`, `UInt16` и `UInt8`. За счет отсутствия отрицательных значений, диапазон допустимых значений для беззнаковых целых чисел сдвигается в сторону положительных чисел, что означает, что они могут хранить в два раза большие по размеру числа. Например, диапазон допустимых значений для `UInt32` — от 0 до $4\,294\,967\,295$.

Для того чтобы лучше понять соотношение типов и их диапазонов, можно воспользоваться данными табл. 3.1.

Таблица 3.1. Диапазоны числовых целых типов

Тип	Начало диапазона	Конец диапазона
Int	В зависимости от платформы: 32-бит: -2 147 483 648 64-бит: -9 223 372 036 854 775 808	В зависимости от платформы: 32-бит: 2 147 483 647 64-бит: 9 223 372 036 854 775 807
Int8	-128	127
Int16	-32 768	32 767
Int32	-2 147 483 648	2 147 483 647
Int64	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
UInt	0	В зависимости от платформы: 32-бит: 4 294 967 295 64-бит: 18 446 744 073 709 551 615
UInt8	0	255
UInt16	0	65 535
UInt32	0	4 294 967 295
UInt64	0	18 446 744 073 709 551 615

3.7.2. Числа с плавающей точкой

Кроме целых числовых типов, в Swift есть и дробные типы: Float и Double. Если вы программировали на других языках программирования, то наверняка знаете, чем отличаются типы Float и Double. Переменные с типом Float в Swift имеют 32-битную длину, а переменные с типом Double — 64-битную длину, что означает двойную точность дробного значения. Так как Swift проектировался для современных платформ, то он предпочитает использовать тип Double. Если мы присвоим дробное значение переменной, то Swift выведет для нее тип Double, а не Float:

```
var myDecimal = 3.2 // => Выведен тип Double
```

3.7.3. Способы записи числовых типов

Для того чтобы можно было удобно читать большие числа, в Swift разряды чисел можно разделять знаками подчеркивания:

```
var myNum = 1_000_000
```

Компилятор никак не воспринимает эти знаки подчеркивания. Для него что 1000000, что 1_000_000 — это одни и те же числа. Просто во втором случае нам легче читать числа с разделенными разрядами. Одним из интересных примеров использования разделителей является запись номера кредитной карты:

```
var myCardNumber = 4065_3012_3456_7859
```

Кроме десятичной системы, числа в Swift также можно писать в двоичной, восьмеричной и шестнадцатеричной системах счисления. Для этого нам нужно перед каждым числом из этих систем счисления указать соответствующие префиксы:

- ❑ двоичная система — префикс: `0b`;
- ❑ восьмеричная система — префикс: `0o`;
- ❑ шестнадцатеричная система — префикс: `0x`.

Для примера напишем одно и то же число 22 в двоичной, восьмеричной и шестнадцатеричной системах счисления:

```
var binary = 0b10110
var octal = 0o26
var hex = 0x16
```

3.7.4. Преобразование числовых типов

Нам часто будут попадаться случаи, когда потребуется работать со значениями разных типов, — складывать их, умножать, делить, вычитать и т. д. Когда мы складываем два значения типа `Int`, Swift нам без проблем выводит результат такого сложения:

```
var first = 4
var second = 5
var result = first + second
print(result) // => 9
```

То есть, при работе с одинаковыми типами все проходит без ошибок. Но проблемы начинают возникать, когда мы хотим сложить разные типы значений. Например, пусть мы хотим сложить значения `Int` и `Double`:

```
var first = 4
var second = 5.5
var result = first + second // => Ошибка
```

Дело в том, что Swift не понимает, какого типа должен быть результат этого сложения. Он начинает думать: это должен быть `Double` со всей дробной частью или это должен быть `Int` с отброшенной дробной частью?

Во многих популярных языках такое выражение выполнилось бы без ошибок — переменная `result` неявно имела бы тип `Double` и содержала значение 9.5.

Но такое не проходит в Swift. Все дело в том, что в Swift нет неявного преобразования типов. Мы должны указывать тип явно. То есть, если мы хотим, чтобы результатом выражения было значение с типом `Double`, то мы должны явно преобразовать переменную `first` в тип `Double`, чтобы Swift работал с двумя значениями типа `Double`.

Для явных преобразований числовых типов существуют функции `Int()`, `Double()` и `Float()`. Мы просто передаем такой функции значение, и она возвращает преобразованное значение:

```
var first = 4
var second = 5.5
var result = Double(first) + second
print(result) // => 9.5
```

Как мы видим, с использованием явного преобразования типов наше выражение больше не выводит ошибок.

Здесь важно понять, что функции преобразования не изменяют значения и типы переменных, которые мы ему передаем. Они просто копируют значения, преобразовывают, а затем возвращают. Значение и тип исходной переменной остаются теми же.

Следует обратить внимание на то, что если мы хотим перевести значение с плавающей точкой в целое, то при преобразовании дробная часть числа с плавающей точкой будет просто отброшена, а не округлена. Дробные значения при преобразовании всегда отбрасываются. Поэтому, работая с точными значениями, имейте это в виду.

3.8. Строки и символы

Для хранения текста и символов служат типы `String` и `Character`. Тип `String` представляет собой набор из значений типа `Character`. Когда мы присваиваем переменной строковое значение, окруженное кавычками, Swift выводит для нее тип `String`:

```
var myText = "Hello World" // Выведен тип String
```

Тип `Character` используется для хранения одного любого строкового символа. Но следует заметить, что если мы явно не укажем тип переменной `Character`, то, при присвоении переменной одного символа, Swift будет выводить для этой переменной тип `String`:

```
var myChar = "S" // Выведен тип String
var myChar2: Character = "S" // Имеет тип Character
```

3.8.1. Конкатенация строк

Конкатенацией в языках программирования называют «склеивание» двух каких-либо объектов — обычно, строк. В Swift конкатенация строк происходит так же легко, как если бы мы складывали вместе два числа. Например, мы можем сложить вместе два строковых значения: `"Hello"` и `"World"`:

```
var a = "Hello" + "World"
print(a) // => "HelloWorld"
```

Для того чтобы результат выглядел красивее, мы можем добавить между двумя склеиваемыми словами пробел :

```
var a = "Hello" + " " + "World"
print(a) // => "Hello World"
```

3.8.2. Преобразование в строку

Несмотря на простоту конкатенации строк, мы не можем просто так сложить вместе строковое и числовое значение. Чтобы выполнить эту операцию, нам нужно явно преобразовать числовой тип в строковый. Это можно сделать с помощью функции `String()`:

```
var score = 42
var text = "Количество заработанных очков равно " + String(score)
print(text) // => "Количество заработанных очков равно 42"
```

3.8.3. Интерполяция строк

В предыдущем примере мы использовали строку, содержащую в себе значение переменной. Такие конструкции очень часто встречаются в программировании. И чтобы облегчить их применение, была придумана операция *интерполяции* строк.

Чтобы вложить значение переменной внутрь строки, нам не обязательно обрывать кавычки и использовать конкатенацию строк. Мы просто можем написать переменную в скобках, поставив перед ними знак обратной косой черты:

```
var score = 42
print("Количество заработанных очков равно \(score)")
```

В этом примере можно заметить, что мы не использовали явного преобразования с помощью функции `String()`. При интерполяции строк любые значения автоматически преобразуются в строку.

3.9. Логические типы

Swift имеет два логических типа: `true` и `false` — истина и ложь соответственно. Чтобы объявить переменную логического типа, нам нужно указать ей тип `Bool`:

```
var a: Bool
```

Либо можно сразу присвоить переменной логическое значение. Тогда вывод типов автоматически определит, что тип переменной `Bool`:

```
var a = true
```

После этого переменная может хранить только два значения: `true` или `false`. Логические типы чаще всего используются в условных выражениях или в циклах, которые мы рассмотрим позже.

3.10. Псевдонимы типов

Для типов в Swift можно создавать *псевдонимы*. Например, если мы хотим по-другому назвать тип `UInt8`, то можем это сделать через ключевое слово `typealias`:

```
typealias Bit = UInt8
```

Здесь мы создали псевдоним типа `UInt8`. Теперь мы можем использовать тип `Bit` так же, как и использовали `UInt8`.

Псевдонимы типов очень полезны, когда мы хотим назначить типу более подходящее по контексту имя. А так как новый тип ссылается на старый, то нам доступны все методы и все свойства старого типа.

Выводы

- ❑ В Swift наряду с переменными используются константы.
- ❑ Мы выводим информацию в консоль с помощью функции `print()`.
- ❑ Однострочные комментарии в Swift пишутся через двойную косую черту (`//`), а многострочные — через косую черту и звездочку (`/* ... */`).
- ❑ Swift — статически типизированный язык программирования, и каждый объект в нем имеет свой тип.
- ❑ Вывод типов определяет автоматически тип переменной на основе значения, которого мы ему присваиваем.
- ❑ Мы можем явно указать тип через двоеточие (`:`) после названия переменной или константы.
- ❑ Основные числовые типы: `Int`, `Double`, `Float`.
- ❑ Строковый тип: `String` и символьный тип: `Character`.
- ❑ Логический тип: `Bool`.
- ❑ Типам в Swift мы можем указывать псевдонимы через ключевое слово `typealias`.

ГЛАВА 4



Базовые операторы

Операторами в языках программирования называются специальные символы или фразы, с помощью которых можно проверять, изменять или комбинировать значения. Простейшим примером является оператор присваивания (`=`), который мы использовали, чтобы присвоить значение переменной. Оператор присваивания берет значение правого операнда и сохраняет его внутри левого операнда. Тем самым каждый оператор имеет свой набор действий над операндами. Например, оператор сложения (`+`) берет значения левого и правого операнда и возвращает нам результат сложения.

Большинство операторов в Swift наследовано из C, поэтому они будут понятны опытным разработчикам. Тем не менее некоторые операторы в Swift ведут себя иначе, чем в C. Так как Swift задумывался как современный и безопасный язык программирования, то разработчики языка попытались исключить популярные ошибки, возникающие при работе с операторами.

Аргументы операторов, над которыми производятся действия, называются *операндами*. По количеству операндов, операторы делятся на:

- ❑ унарные;
- ❑ бинарные;
- ❑ тернарные.

Количество операндов у них одно, два или три соответственно.

В этой главе мы рассмотрим:

- ❑ оператор присваивания: `=`
- ❑ арифметические операторы: `+`, `-`, `*`, `/`, `%`
- ❑ составные операторы присваивания: `+=`, `-=`, `*=`, `/=`
- ❑ операторы инкремента: `++` и декремента: `--`
- ❑ оператор унарного минуса: `-`
- ❑ операторы сравнения: `==`, `>`, `<`, `>=`, `<=`, `!=`
- ❑ тернарный условный оператор: `? :`

- ❑ операторы диапазона: ... , ..<
- ❑ логические операторы: &&, ||, !

4.1. Оператор присваивания

Как мы отмечали в начале главы, оператор присваивания в Swift берет значение правого операнда и присваивает его левому:

```
var a = 55
```

Главным отличием оператора присваивания в Swift является то, что он ничего не возвращает после того, как операция присвоения была выполнена. Этим поведением разработчики хотели исключить популярную ошибку, когда в условных выражениях вместо оператора равенства (==) мы использовали оператор присваивания (=). В других языках программирования оператор присваивания возвращал логическое значение (true или false), и условие работало неправильно, а проблему найти было сложно, поскольку никаких сообщений с ошибками программа не выводила.

4.2. Арифметические операторы

В наших примерах мы уже рассмотрели некоторые арифметические операторы. В Swift можно использовать сложение (+), вычитание (-), умножение (*), деление (/) и вычисление остатка от деления (%):

```
var a = 10
var b = 5
print(a + b)    // => 15
print(a - b)    // => 5
print(a * b)    // => 50
print(a / b)    // => 2
print(a % b)    // => 0
```

Все арифметические операторы, кроме последнего, мы знаем из школьной программы. А оператор вычисления остатка от деления очень распространен в языках программирования. Чтобы лучше его понять, рассмотрим пример:

```
var myNum = 45 % 7
print(myNum) // => 3
```

Здесь мы разделили 45 на 7. Как известно, нацело разделить 45 на 7 нельзя. Целых значений 7 внутри 45 всего 6, что дает нам 42, а остаток получается 3. Именно это значение нам выводит оператор остатка от деления.

4.3. Составные операторы присваивания

Допустим, у нас есть переменная `a`, значение которой равно единице. Если мы хотим увеличить ее значение на 4, то можем воспользоваться выражением `a = a + 4`. Например, так:

```
var a = 1  
a = a + 4
```

Используя составные операторы присваивания, мы можем сократить это выражение до вида:

```
a += 4
```

Таким образом, выражение `a += 4` эквивалентно выражению `a = a + 4` из предыдущего примера.

Составные операторы присваивания существуют для всех указанных в *разд. 4.2* арифметических операторов:

```
var a += 4  
var b -= 8  
var c *= 15  
var d /= 16  
var e %= 23
```

Важно заметить, что составные операторы присваивания не возвращают никакого значения. То есть, выражения типа `var b = a += 4` приведут к ошибке.

4.4. Операторы инкремента и декремента

В предыдущем примере, чтобы увеличить значение на 4, мы воспользовались составным оператором присваивания со сложением: `a += 4`. Чаще всего нам будут встречаться случаи, когда нужно увеличить или уменьшить значение только на единицу. Именно для этих случаев существуют операторы инкремента и декремента. Они позволяют увеличить или уменьшить значение на 1:

```
var a = 5  
a++      // => 5  
print(a) // => 6  
a--      // => 6  
print(a) // => 5
```

Здесь в первом случае мы применили оператор инкремента `++`, который увеличил значение переменной `a` на единицу. А во втором случае мы к этой же переменной применили оператор декремента `--`, который уменьшил значение переменной на единицу и вернул ей исходное значение 5.

Операторы инкремента и декремента сами по себе возвращают значение, которое в них хранилось до увеличения или уменьшения. То есть, они сначала возвращают значение, а потом только его увеличивают.

Если нам нужно, чтобы оператор сначала увеличил или уменьшил значение, а потом только вывел его, то можно использовать операторы преинкремента и предекремента, которые пишутся до названия переменной:

```
var a = 5  
++a      // => 6
```

```
print(a) // => 6
--a      // => 5
print(a) // => 5
```

Это первый пример, когда мы рассмотрели унарный оператор. Как можно заметить операторы инкремента и декремента работают только с одним операндом. Следующим примером унарных операторов будут унарный минус и унарный плюс.

4.5. Операторы унарного минуса и унарного плюса

Чтобы поменять знак какого-либо числа, т. е. превратить положительное число в отрицательное или наоборот, нам нужно использовать оператор унарного минуса:

```
var a = 3
var b = -a
print(b) // => "-3"
print(-b) // => "3"
```

В свою очередь, существует еще и оператор унарного плюса, который не изменяет никакого значения:

```
var a = -4
print(+a) // => "-4"
```

Оператор унарного плюса не несет никакой смысловой нагрузки. Его просто иногда используют для наглядного изображения знака числа.

4.6. Операторы сравнения

Для того чтобы сравнить два числа, можно использовать операторы сравнения. Swift поддерживает все стандартные операторы сравнения из языка C:

- ☐ равно: ==
- ☐ не равно: !=
- ☐ больше: >
- ☐ меньше: <
- ☐ больше либо равно: >=
- ☐ меньше либо равно: <=

Для примера сравним несколько чисел:

```
2 == 1 // => false, т. к. 2 не равно 1
2 != 4 // => true, т. к. 2 не равно 4
4 > 3 // => true, т. к. 4 больше 3
9 <= 7 // => false, т. к. 3 не меньше либо равно 7
```

Операторы сравнения преимущественно используются в условных выражениях, когда нужно сравнить два числа. Результатом операторов сравнения всегда является логическое значение: либо `true`, либо `false`.

4.7. Тернарный условный оператор

Тернарный условный оператор является первым и единственным оператором, в котором содержатся три операнда. Конструкция тернарного условного оператора состоит из:

условие ? результат при `true` : результат при `false`

Если значение условия равно `true`, то выполняется первое выражение, а если `false` — второе. Для того чтобы лучше понять этот оператор, рассмотрим пример:

```
var isHot = true
isHot ? "Горячо" : "Не горячо" // => "Горячо"
```

Этот пример проверяет, не равно ли `true` значение переменной `isHot`. Если *да*, то он возвращает нам первое выражение, а если *нет* — второе.

Тернарный условный оператор является краткой формой записи простых условных выражений, с которыми мы познакомимся чуть дальше.

4.8. Операторы диапазона

Операторы диапазона позволяют записать диапазон, который содержит в себе набор чисел. Левым операндом является число, указывающее на начало диапазона, а правым — число, указывающее на его конец. Например, диапазон чисел от 2 до 6 мы можем написать так: `2...6`. Этот диапазон будет включать в себя числа 2, 3, 4, 5 и 6.

Операторы диапазона присутствуют в таких языках программирования, как Perl и Ruby. Поэтому те, кто не знаком с этими языками, столкнутся здесь с диапазонами впервые. В Swift есть два оператора диапазона:

- оператор закрытого диапазона: `...<` описывает диапазон чисел от первого операнда до второго операнда включительно;
- оператор полуоткрытого диапазона: `...<` описывает диапазон чисел от первого операнда до второго, но не включает его. Этот тип оператора диапазона очень полезен при итерации по элементам массива, чей индекс начинается с 0.

Обязательным условием использования операторов диапазона является то, что первый операнд должен всегда быть меньше второго.

Более подробно диапазоны мы рассмотрим в других главах, т. к. они, в основном, используются в циклах или массивах, которые мы еще не изучали.

4.9. Логические операторы

Логические операторы позволяют создавать логические выражения из булевых (Bool) значений. В Swift есть три логических оператора:

- ❑ логическое НЕ: !
- ❑ логическое И: &&
- ❑ логическое ИЛИ: ||

4.9.1. Логическое НЕ

Оператор логического НЕ (!) преобразует значение true в false и наоборот:

```
var a = true
var b = false
!a           // => false
!b           // => true
```

4.9.2. Логическое И

С помощью оператора логического И (&&) можно создавать логические выражения, в которых для того, чтобы все выражение было true, все его элементы тоже обязательно должны равняться true:

```
var a = true
var b = false
var c = true
a && b // false, т. к. хотя бы один из элементов false
a && c // true, т. к. все элементы true
```

4.9.3. Логическое ИЛИ

С помощью оператора логического ИЛИ (||) можно создавать логические выражения, в которых для того, чтобы все выражение равнялось true, хотя бы один из элементов должен быть true:

```
var a = true
var b = false
var c = false
a || b // true, т. к. хотя бы один из элементов true
b || c // false, т. к. все элементы false
```

Выводы

- ❑ Оператор присваивания: =
- ❑ Арифметические операторы: +, -, *, /, %

- ❑ Составные операторы присваивания: `+=`, `-=`, `*=`, `/=`
- ❑ Операторы инкремента: `++` и декремента: `--`
- ❑ Оператор унарного минуса: `-`
- ❑ Операторы сравнения: `==`, `>`, `<`, `>=`, `<=`, `!=`
- ❑ Тернарный условный оператор: `? :`
- ❑ Операторы диапазона: `... , ...<`
- ❑ Логические операторы: `&&`, `||`, `!`

ГЛАВА 5



Типы коллекций

На протяжении всех предыдущих глав мы присваивали переменным только одно значение. Это значение могло иметь тип `Int`, `String`, `Bool` и т. д., но оно было единственным. Однако, кроме единичных значений, в языках программирования существуют и сгруппированные. Такие значения называются *коллекциями*. В Swift предусмотрены три типа коллекций: массивы, множества и словари.

5.1. Массивы

Массив — это сгруппированный однотипный набор значений. Используя массивы, мы можем назначать одной переменной группу значений. То есть, вместо того чтобы создавать множество переменных, мы можем создать массив и по порядку присвоить ему все эти значения. Но как потом получать доступ к этим значениям из массива? А дело в том, что каждый элемент массива имеет свой числовой индекс. По этому индексу мы и получаем доступ к определенному значению внутри массива. Давайте посмотрим, как создаются массивы в Swift.

5.1.1. Объявление массива

Объявить массив в Swift можно простым присвоением переменной группы значений, окруженных квадратными скобками и разделенных запятыми:

```
var myArray = [2, 3, 44, 45]
```

Такое сгруппированное значение, окруженное квадратными скобками и разделенное запятыми, называется *литералом массива*. Эта конструкция аналогична присвоению литерала какого-либо другого типа. Вывод типов понимает, что переменной присваивается значение с типом `Array`, значит, переменной нужно задать именно тип `Array`.

Для массивов в Swift есть одно большое ограничение. В отличие от других языков программирования, массивы в Swift строго типизированы. Это означает, что значения в массиве могут быть только одного типа. Скажем, массив может состоять только из строк или только из целых чисел и т. д.:

```
var myArray1 = [ 2,3,44,45 ]
var myArray2 = ["apple","orange","peach","banana"]
```

Здесь мы в первой строке объявили массив из целых чисел (`Int`), а во второй строке — массив из строк (`String`). Если мы попробуем объявить массив из значений разного типа, то получим ошибку:

```
var myArray3 = [ 2 , 3 ,"peach","banana"] // Ошибка
```

Строгая типизация массивов в Swift введена для обеспечения более надежного кода. Когда массив строго типизирован, то мы от него ожидаем элементы только определенного типа. Поэтому не нужно будет каждый раз проверять, все ли элементы соответствуют тому типу, который мы хотим задать. В Swift мы точно знаем, что если массив имеет тип `Int`, то любое значение его тоже будет `Int`.

У каждого массива в Swift есть очень полезные методы: `count` и `isEmpty`. Метод `count` передает нам количество элементов, хранящихся в массиве, а метод `isEmpty` — логическое выражение `true` или `false`, означающее пустой массив или нет. Если массив пустой, то метод `isEmpty` возвращает `true`, если не пустой — `false`:

```
var arrayOfInts = [ 2,3,44,45 ]
print("Количество значений в массиве равно \(arrayOfInts.count)")
// => "Количество значений в массиве равно 4"

var arrayOfStrings = ["apple","orange","peach","banana"]
if arrayOfStrings.isEmpty {
    print("Этот массив пустой")
} else {
    print("Этот массив не пустой, и количество его элементов равно \(arrayOfInts.count)")
}
// => "Этот массив не пустой, и количество его элементов равно 4"
```

Кроме простого присвоения литерала массива, мы можем также сначала объявить переменную как массив, и только потом присваивать ей значения. Указатель на тип массива пишется в виде названия определенного типа, окруженного квадратными скобками:

```
var firstArray: [Int]
```

Здесь мы объявили массив из `Int` элементов. Но такой способ является сокращенным способом записи типа массивов. Указатель на тип массива в полной форме пишется через слово `Array`, после которого следуют угловые скобки (`<>`) с типом хранимого значения:

```
var secondArray: Array<Int>
```

Это и предыдущее выражения равносильны. Мы чаще будем употреблять сокращенную форму указания типа, т. к. она короче и легче для восприятия.

Давайте попробуем применить метод `isEmpty` для этих двух массивов:

```
var firstArray: [Int]
firstArray.isEmpty // => Ошибка

var secondArray: Array<Int>
secondArray.isEmpty // => Ошибка
```

Как мы видим, оба массива выдали нам ошибку. А все потому, что мы стали применять к массиву методы, до того как присвоить ему начальное значение. Эта ошибка нам знакома из объявления переменных, когда мы никак не могли использовать переменную, пока не присвоили ей начальное значение. С массивами происходит то же самое. Мы не можем использовать массив, пока не присвоим ему какое-нибудь значение.

Альтернативой присвоения литерала для создания массива является создание пустого массива. В предыдущих двух примерах мы просто указали тип переменной как массив, но не присвоили массив. Если мы хотим использовать массив, но пока не назначать ему какие-либо значения, то можем создать пустой массив. Массивы в Swift можно создавать через выражение *инициализации*:

```
var myArray = [Int]()
```

Когда вы изучите, что собой в Swift представляют инициализаторы, то лучше поймете приведенную конструкцию. Пока же вам следует обратить внимание, что, в отличие от предыдущих двух примеров, мы здесь использовали оператор присваивания (`=`), а не указания типа (`:`). Дело в том, что конструкция `[Int]()` возвращает пустой массив, а вывод типов назначает переменной `myArray` тип `[Int]`. Таким образом, с помощью этого выражения мы одновременно и указываем тип переменной — массив, и присваиваем ей пустой массив. Так что теперь этот массив можно использовать.

С помощью выражения инициализации мы также можем создать уже заполненный массив. Инициализатор массива имеет два именованных параметра: `count` и `repeatedValue`. В параметре `count` мы можем написать количество элементов массива, которое хотим создать, а в параметре `repeatedValue` — написать значение, которым нужно заполнить эти элементы:

```
var myArray = [Int](count: 5, repeatedValue: 1 )
```

Этим выражением мы создали массив из пяти единиц. Давайте теперь научимся получать из массива содержащиеся в нем значения.

5.1.2. Получение доступа к элементам массива

Мы научились объявлять массив и заполнять его значениями. Следующий полезный навык — научиться получать из массива значения. Чтобы получить из массива значение под определенным индексом, мы просто пишем название массива и затем индекс искомого значения в квадратных скобках:

```
var myArray = [2.3, 2.4, 2.5]
myArray[2] // => 2.5
```

Здесь в первой строке мы объявили массив из трех значений типа `Double`. А во второй — написали выражение, которое возвращает значение под вторым индексом из массива `myArray`. Но подождите! Оно нам вернуло 2.5, хотя вторым значением по порядку является 2.4. Это еще одна из распространенных ошибок в работе с языками программирования. В большинстве языков программирования порядок индексов в массиве начинается с нуля. И Swift не исключение. Давайте выведем из массива все значения, чтобы было понятнее:

```
var myArray = [2.3, 2.4, 2.5]
myArray[0] // => 2.3
myArray[1] // => 2.4
myArray[2] // => 2.5
```

А при попытке получения доступа к элементу по индексу 3 мы получим ошибку:

```
var myArray = [2.3, 2.4, 2.5]
myArray[3] // => Ошибка
```

И все потому, что элемента с индексом 3 еще не существует.

Это правило нужно очень хорошо запомнить, поскольку подобная ошибка очень распространена среди начинающих разработчиков.

5.1.3. Добавление элементов в массив

Давайте теперь научимся добавлять элементы в уже существующий массив. Мы это можем сделать двумя способами: через метод `append(_)` либо через оператор присваивания со сложением `(+=)`.

В первом способе мы просто после имени существующего массива пишем метод `append(_)` и в скобках — значение, которое хотим поместить в конец массива:

```
var toDoList = ["Построить дом"]
toDoList.append("Посадить дерево")
// теперь массив toDoList состоит из двух элементов
```

Добавлять же значения через оператор присваивания со сложением `(+=)` следует так:

```
toDoList += ["Родить сына"]
// теперь массив toDoList состоит из трех элементов
```

Обратите внимание, что мы использовали не просто строку, а строку, заключенную в квадратные скобки.

В отличие от метода `append(_)`, с помощью оператора `+=` можно присвоить сразу несколько элементов:

```
var townsToVisit = ["Вена", "Сан-Франциско"]
townsToVisit += ["Рио-де-Жанейро", "Венеция", "Лондон"]
```

```
// теперь массив townsToVisit состоит из пяти элементов  
// ["Вена", "Сан-Франциско", "Рио-де-Жанейро", "Венеция", "Лондон"]
```

Рассмотренные два способа добавляют значения в конец массива. То есть, например, если у нашего массива было два элемента с индексами 0 и 1, то, присвоив им дополнительно три элемента, мы заполнили индексы 2, 3 и 4.

Однако бывают ситуации, когда нам нужно добавить элемент на определенный индекс. Мы это можем сделать с помощью метода `insert(_:atIndex:)`, который принимает два параметра: значение, которое нужно вставить, и именованный параметр `atIndex`, отвечающий за индекс, по которому нужно вставить значение:

```
townsToVisit.insert( "Сингапур", atIndex: 8) // Ошибка
```

Здесь мы попытались вставить новый элемент в значение по индексу 8 и получили ошибку. Зная специфику безопасности Swift, вы можете догадаться, почему это случилось. В массиве `townsToVisit` у нас было пять элементов, значит, последним индексом был 4, а мы хотели сразу вставить значение с индексом 8. Это бы привело к пустым значениям с индексами 5, 6 и 7, чего не произошло. Из этой ошибки можно сделать вывод, что Swift в целях надежности кода не позволяет иметь в массивах пустые значения.

Тогда давайте попробуем вставить значение на индекс, следующий за последним. Если у нас в массиве пять элементов и последний индекс 4, то нам нужен 5-й индекс:

```
townsToVisit.insert( "Сингапур", atIndex: 5)  
// массив townsToVisit теперь содержит шесть элементов  
// ["Вена", "Сан-Франциско", "Рио-де-Жанейро", "Венеция", "Лондон", "Сингапур"]
```

Получилось! Метод `insert(_:atIndex:)` добавил значение в конец массива. А что будет, если мы применим метод `insert(_:atIndex:)` для уже существующего индекса:

```
townsToVisit.insert( "Москва", atIndex: 2)  
// массив townsToVisit теперь содержит семь элементов  
// ["Вена", "Сан-Франциско", "Москва", "Рио-де-Жанейро", "Венеция", "Лондон",  
"Сингапур"]
```

Как мы видим, метод `insert(_:atIndex:)` добавил новый элемент по индексу 2, а старые элементы сместил на один индекс вправо. Из этого следует, что метод `insert(_:atIndex:)` полезен, когда нам нужно вставить элемент куда-либо внутри коллекции. А для того чтобы добавить новые значения в конец, хорошо подходят метод `append(_:)` и оператор присваивания со сложением `(+=)`.

5.1.4. Изменение элементов массива

Кроме добавления элементов в массив, важно уметь изменять значение элемента массива. Для этого достаточно написать название массива с нужным нам индексом и присвоить ему другое значение:

```
var fruits = ["Яблоко", "Груша", "Банан", "Апельсин", "Арбуз"]
fruits[1] = "Ананас"
// => ["Яблоко", "Ананас", "Банан", "Апельсин", "Арбуз"]
```

После изменения значения выражение возвращает "Ананас", т. е. новый элемент, которым мы заменили элемент "Груша".

Кроме того, мы можем изменить значения сразу нескольких элементов. Для этого вместо одиночного индекса можно указать диапазон значений:

```
var fruits = ["Яблоко", "Груша", "Банан", "Апельсин", "Арбуз"]
fruits[0..<2] = ["Мандарин", "Манго"]
// => ["Мандарин", "Манго", "Банан", "Апельсин", "Арбуз"]
```

Здесь мы велели заменить значения от нулевого индекса до второго, не включая второго. То есть, программа должна была заменить значения "Яблоко" и "Груша" на "Мандарин" и "Манго". Программа так и сработала — после замены выражение на второй строчке возвращает ["Мандарин", "Манго"], т. е. новые элементы, которые пришли на замену другим.

Возникает интересный вопрос: что будет, если диапазон, в который нужно вставить значения, мы укажем больше, чем вставляемые значения? Давайте проверим это. Оставим все так же, просто диапазон теперь напомним так: $0 \dots 2$, т. е. от 0 до 2, включая 2:

```
var fruits = ["Яблоко", "Груша", "Банан", "Апельсин", "Арбуз"]
fruits[0...2] = ["Мандарин", "Манго"]
// => ["Мандарин", "Манго", "Апельсин", "Арбуз"]
```

В этом случае Swift поступает очень интересно. Он берет три первых элемента, вставляет вместо них два новых элемента, а последний просто удаляет. То есть, теперь у нас нет элемента "Банан". Любопытно... А можно было ожидать, что Swift начнет повторяться и назначит элементу под индексом 2 значение "Мандарин".

5.1.5. Удаление элементов из массива

Добавлять и изменять элементы мы научились, пора приступить к их удалению. Удаление элементов происходит через метод `removeAtIndex(_ :)`. Он принимает параметр со значением индекса элемента, который нужно удалить. Что немаловажно, после удаления элемента метод `removeAtIndex(_ :)` возвращает этот удаленный элемент.

Давайте сразу попробуем удалить значение из середины массива — посмотрим, как Swift поступит с пустыми значениями:

```
var fruits = ["Яблоко", "Груша", "Банан", "Апельсин", "Арбуз"]
fruits.removeAtIndex(2)
// => ["Яблоко", "Груша", "Апельсин", "Арбуз"]
```

И вот опять Swift удалил пробел, который должен был образоваться вместо элемента "Банан", который мы удалили. Ну никак он не позволяет существовать пробелам в массивах!

Кроме метода удаления по индексу, есть еще менее интересный метод `removeLast()`, который удаляет последнее значение в массиве:

```
var fruits = ["Яблоко", "Груша", "Банан", "Апельсин", "Арбуз"]
fruits.removeLast()
// => ["Яблоко", "Груша", "Банан", "Апельсин"]
```

Так же, как и метод `removeAtIndex(_:)`, метод `removeLast()` тоже возвращает удаленный объект.

5.1.6. Итерация по массиву

Для того чтобы перебрать все элементы массива, можно использовать цикл `for-in`. Циклы мы будем рассматривать в следующей главе, а сейчас вам следует просто посмотреть на конструкцию итерации по массиву:

```
var fruits = ["Яблоко", "Груша", "Банан", "Апельсин", "Арбуз"]
for fruit in fruits {
    print(fruit)
}
// "Яблоко"
// "Груша"
// "Банан"
// "Апельсин"
// "Арбуз"
```

В этом примере мы запустили цикл по каждому элементу массива и напечатали их. Если нам нужно иметь доступ к индексу каждого элемента массива внутри цикла, то мы можем использовать глобальную функцию `enumerate()`:

```
var fruits = ["Яблоко", "Груша", "Банан", "Апельсин", "Арбуз"]
for (index, fruit) in fruits.enumerate() {
    print("Фрукт номер \(index) - \(fruit)")
}
// Фрукт номер 1 - Яблоко
// Фрукт номер 2 - Груша
// Фрукт номер 3 - Банан
// Фрукт номер 4 - Апельсин
// Фрукт номер 5 - Арбуз
```

5.2. Множества

Множества — это набор уникальных однотипных и никак не упорядоченных значений. То есть, в отличие от массивов, значения внутри множеств могут встречаться только один раз и выводятся в случайном порядке.

Множества в Swift эквивалентны `NSSet` из Foundation.

5.2.1. Объявление множеств

Тип множеств в Swift пишется только в обобщенной форме: `Set<SomeType>`, где `SomeType` — тип хранимых значений. В отличие от массивов, для множеств не существует сокращенной формы.

Для того чтобы объявить пустое множество, следует поместить пустые скобки после типа множества и присвоить это все переменной или константе:

```
var mySet = Set<Int>()
```

Здесь мы объявили пустое множество из значений типа `Int`. Чтобы заполнить множество значениями, мы можем присвоить ему литерал массива:

```
mySet = [4, 8, 15, 16, 23, 42]
```

При этом следует обратить внимание на то, что мы присвоили множеству литерал массива с уникальными значениями, поскольку элементы множества не могут повторяться.

Впрочем, если мы попробуем присвоить множеству неуникальные значения, то никакой ошибки не будет:

```
mySet = [4, 4, 8, 15, 16, 23, 42]
```

Однако, хотя Swift нам ошибки и не выдаст, но запишет в множество только уникальные значения:

```
mySet = [4, 4, 8, 15, 16, 23, 42]
mySet // => [15, 8, 23, 42, 4, 16]
```

И, поскольку, как было отмечено ранее, множества никак не упорядочены, то и в этом примере элементы множества вывелись беспорядочно.

При создании множества мы можем воспользоваться выводом типов и не указывать тип его элементов. Например, предыдущий пример мы бы могли записать так:

```
var mySet: Set = [4, 8, 15, 16, 23, 42]
```

Здесь мы определили тип переменной просто как `Set` — без угловых скобок и указания типа. Тем не менее компилятор сам увидит, что значения, которые записываются внутри множества, имеют тип `Int`.

5.2.2. Работа с множествами

Работа с множествами во многом похожа на работу с массивами, так что мы быстро рассмотрим здесь простые операции для множеств.

Для того чтобы узнать количество значений внутри множества, стоит воспользоваться методом `count`:

```
var mySet: Set = [4, 8, 15, 16, 23, 42]
mySet.count // => 6
```

Для того чтобы узнать, содержит ли множество значения, или оно вообще пустое, можно воспользоваться логическим методом `isEmpty`, который возвращает `true` — если множество пустое, и `false` — если множество имеет хотя бы один элемент:

```
var mySet: Set = [4,8,15,16,23,42]
mySet.isEmpty // => false
var myEmptySet = Set<Int>()
myEmptySet.isEmpty // => true
```

Для того чтобы добавить значение внутрь множества, мы можем воспользоваться методом `insert(_:)`:

```
var mySet: Set = [4,8,15,16,23,42]
mySet.insert(56)
mySet // => [4,8,15,16,23,42,56]
```

Для того чтобы удалить значение из множества, можно воспользоваться методом `remove(_:)`, который удаляет заданное значение либо возвращает `nil`, если значение найти не удалось:

```
var mySet: Set = [4,8,15,16,23,42,56]
mySet.insert(56)
mySet // => [4,8,15,16,23,42]
```

А если мы хотим удалить все элементы из множества, то можем использовать метод `removeAll()`.

Для итерации по множеству мы, так же, как и в случае массивов, можем воспользоваться циклом `for-in`:

```
var mySet: Set = [4,8,15,16,23,42]
for digit in mySet {
    print(digit)
}
// 16
// 15
// 23
// 4
// 8
// 42
```

Если мы все же хотим иметь какой-нибудь порядок, то можем применить функцию `sort()`:

```
for digit in mySet.sort() {
    print(digit)
}
// 4
// 8
// 15
// 16
// 23
// 42
```

В этом случае, функция `sort()` упорядочила значения в множестве по возрастающей.

Чтобы проверить, содержится ли определенное значение в множестве, можно воспользоваться методом `contains(_)`

```
var mySet: Set = [4, 8, 15, 16, 23, 42]
if mySet.contains(16) {
    print("Множество содержит число 16")
} else {
    print("Множество не содержит число 16")
}
```

Если значение, переданное методу `contains(_)`, содержится внутри множества, то метод возвращает `true`, а если такого значения нет — `false`.

5.2.3. Сочетание и сравнение множеств

В предыдущем разделе мы рассмотрели методы множеств, схожие с методами для массивов. Здесь же мы познакомимся с методами, которые можно использовать только с множествами.

В Swift над множествами допускается производить такие фундаментальные действия, как, например, сочетание, вычитание одного множества из другого и т. д.

Чтобы объединить два множества, нужно воспользоваться методом `union(_:)`, который создает новое множество из двух указанных:

```
var a: Set = [1, 2, 3]
var b: Set = [4, 5, 6]
var c = a.union(b) // [6, 4, 1, 2, 5, 3]
```

Для исключения элементов одного множества из другого существует метод `subtract(_:)`, который создает новое множество из элементов, не содержащихся в указанном множестве:

```
var a: Set = [1, 2, 3, 4, 5, 6]
var b: Set = [4, 5, 6]
var c = a.subtract(b) // [3, 2, 1]
```

Создать новое множество из элементов, общих для двух множеств, можно методом `intersect(_:)`:

```
var a: Set = [1, 2, 3, 4, 5, 6]
var b: Set = [4, 5, 6]
var c = a.intersect(b) // [5, 6, 4]
```

А чтобы создать новое множество из элементов, содержащихся в каждом множестве, но исключить одинаковые для обоих множеств элементы, нужно воспользоваться методом `exclusiveOr()`:

```
var a: Set = [1, 2, 3, 4, 5, 6]
var b: Set = [4, 5, 6, 7, 8, 9]
var c = a.exclusiveOr(b) // [1, 7, 9, 3, 8, 2]
```


Кроме рассмотренных методов сочетания множеств, мы можем также сравнивать два множества. Для того чтобы сравнить, содержат ли два множества одни и те же элементы, следует воспользоваться оператором сравнения `==` :

```
var a: Set = [1,2,3,4,5,6]
var b: Set = [4,5,6,1,2,3]
a == b // => true
```

Если одно из двух сравниваемых множеств является подмножеством или надмножеством другого, то можно прибегнуть к методам `isSubsetOf(_:)` и `isSupersetOf(_:)`:

```
var a: Set = [1,2,3,4,5,6]
var b: Set = [4,5,6]
b.isSubsetOf(a) // => true
a.isSupersetOf(b) // => true
```

Если одно из двух множеств является подмножеством или надмножеством другого, и эти множества не равны, то можно применить методы `isStrictSubsetOf(_:)` и `isStrictSupersetOf(_:)`:

```
var a: Set = [4,5,6]
var b: Set = [4,5,6]
b.isStrictSubsetOf(a) // => false
b.isSubsetOf(a) // => true
a.isStrictSupersetOf(b) // => false
a.isSupersetOf(b) // => true
```

Если два множества не содержат одинаковых элементов, то можно воспользоваться методом `isDisjointWith(_:)`:

```
var a: Set = [1,2,3]
var b: Set = [4,5,6]
a.isDisjointWith(b) // true
```

5.3. Словари

Еще одним типом коллекций в Swift являются *словари*. Словари, как и массивы, группируют одно или несколько значений вместе. Но, в отличие от массива, элементы словаря имеют два отдельных компонента: ключ и значение. Типы ключа и значения могут отличаться друг от друга, но у каждого должен быть единый тип для всего словаря. По большому счету, словари — это те же массивы, только индексы у словарей могут быть любого типа, а не только числового как у массивов.

5.3.1. Объявление словаря

Мы создадим словарь с днями неделями. Ключами для словаря будут являться двухбуквенные сокращенные названия дней недели, а значениями — названия дней недели:

```
var daysOfWeek = ["ПН": "Понедельник", "ВТ": "Вторник", "СР": "Среда", "ЧТ": "Четверг", "ПТ": "Пятница", "СБ": "Суббота", "ВС": "Воскресенье", ]
```

Здесь мы присвоили переменной `daysOfWeek` литерал словаря. Как можно заметить, литералы словаря очень похожи на литералы массива. Только в литералах словаря пишутся еще и ключи, отделенные от значения двоеточиями. Так как мы присвоили переменной литерал, то вывод типов автоматически определил, что это словарь с типом ключа `String` и типом значения `String`. Впрочем, мы бы могли и явно указать тип переменной — как словарь с этими типами:

```
var daysOfWeek: [String: String]
```

Такая конструкция означает, что мы создали переменную с типом словаря, чьи ключи и значения имеют тип `String`.

5.3.2. Получение доступа к элементам словаря

Получение доступа к элементам словаря осуществляется подобно доступу к элементам массива. Отличие составляет только то, что при обращении к элементу словаря в квадратных скобках мы указываем значение ключа, а не индекса, как в массивах:

```
daysOfWeek["ВТ"] // => "Вторник"
```

У возвращаемого значения словаря имеется одна особенность. Так как есть вероятность того, что мы укажем несуществующий ключ, то словарь всегда возвращает нам не обычный тип, а *опциональный*. Опциональные типы мы рассмотрим в следующих главах, поэтому пока просто запомните, что словари всегда возвращают опциональные значения при доступе к их элементам.

5.3.3. Добавление элементов в словарь

Для добавления новых элементов в словарь существуют два способа. Первый способ — это присвоение нового значения элементу с новым ключом:

```
var alphabetFruits = ["А" : "Арбуз", "Б": "Банан", "В": "Виноград"]  
alphabetFruits["Г"] = "Груша"
```

Вторым способом добавления элемента в словарь является метод `updateValue(_ : forKey:)`. По названию метода можно понять, что на самом деле его основное предназначение не в создании новых значений, а в обновлении старых. Просто, если метод по заданному в параметре `forKey` ключу ничего не находит, он создает новый элемент словаря:

```
alphabetFruits.updateValue("Дыня", forKey: "Д")
```

5.3.4. Изменение элементов словаря

Для изменения значения элементов словаря также можно воспользоваться двумя способами. Первый способ — присваивание нового значения для элемента с существующим ключом:

```
alphabetFruits["А"] = "Ананас"
```

Второй же способ — это использование ранее уже рассмотренного метода `updateValue(_:forKey:)` для существующего ключа:

```
alphabetFruits.updateValue("Вишня", forKey: "В") // => "Виноград"
```

После обновления значения метод `updateValue(_:forKey:)` возвращает нам предыдущее значение опционального типа.

5.3.5. Удаление элементов из словаря

Для того чтобы удалить значение из словаря, можно просто присвоить `nil` значению словаря под определенным ключом:

```
alphabetFruits["Д"] = nil
```

Такое выражение не просто присвоит значение `nil` для элемента под ключом "Д", а полностью сотрет этот элемент вместе с ключом и значением.

Альтернативным способом удаления элементов словаря является использование метода `removeValueForKey(_:)`. Этот метод удаляет элемент под ключом, который мы ему передаем, или возвращает `nil`, если значения с таким ключом не существует:

```
if let oldFruit = alphabetFruits.removeValueForKey("Г") {  
    print("Мы удалили фрукт \(oldFruit).")  
} else {  
    print("В словаре алфавитных фруктов нет значения с ключом Г")  
}  
// напечатает "Мы удалили фрукт Груша"
```

Здесь мы использовали конструкцию *опциональной привязки*, которую будем рассматривать в *главе 8*.

5.3.6. Итерация по словарю

Для итерации по словарю мы, подобно тому, как поступали с массивами, используем цикл `for-in`. Единственное отличие составляет то, что для словаря мы одновременно работаем и с ключом, и со значением для каждого элемента:

```
var alphabetFruits = ["А": "Арбуз", "Б": "Банан", "В": "Виноград"]  
for (letter, fruit) in alphabetFruits {  
    print("\(letter): \(fruit)")  
}  
// А: Арбуз  
// В: Виноград  
// Б: Банан
```

Как можно заметить, значения внутри словаря вывелись на экран не в алфавитном порядке. Дело в том, что для элементов словаря не существует определенного порядка, и они располагаются внутри словаря беспорядочно.

Выводы

- ❑ В Swift существуют три типа коллекций: массивы, множества и словари.
- ❑ Массивы и словари строго типизированы.
- ❑ Для создания массива достаточно присвоить переменной литерал массива — например: `[2, 3, 44, 45]`.
- ❑ Метод `count` выводит количество элементов массива, а метод `isEmpty` — проверяет, не пустой ли массив.
- ❑ Указывать тип массива нужно через название типа, окруженное квадратными скобками: `var firstArray: [Int]`.
- ❑ Создание пустого массива: `var myArray = [Int]()`.
- ❑ Для получения значения из массива мы просто указываем индекс искомого значения в квадратных скобках: `myArray[2]`.
- ❑ Для добавления элементов в массив мы используем либо оператор присваивания со сложением (`+=`), либо метод `append(_)`.
- ❑ С помощью метода `insert(_:atIndex:)` мы можем вставить значение в определенный индекс.
- ❑ Для изменения значения элемента массива мы присваиваем новое значение по индексу: `fruits[1] = "Ананас"`.
- ❑ Для удаления элемента массива мы используем методы `removeAtIndex(_)` и `removeLast()`.
- ❑ Множества — это набор уникальных однотипных неупорядоченных значений.
- ❑ Тип множеств в Swift пишется только в обобщенной форме `Set<SomeType>`.
- ❑ Чтобы объявить пустое множество, следует поместить пустые скобки после типа множества `var mySet = Set<Int>()`.
- ❑ Для множеств доступны методы `count` и `isEmpty`.
- ❑ Для того чтобы добавить в множество элемент, нужно воспользоваться методом `insert(_)`, а чтобы удалить — методом `remove(_)`.
- ❑ Итерация по множеству происходит через цикл `for-in`.
- ❑ Множества можно сочетать через методы `union(_)`, `subtract(_)`, `intersect(_)` и `exclusiveOr(_)`.
- ❑ Множества можно сравнивать через оператор сравнения `==`, а также с помощью методов `isSubsetOf(_)`, `isSupersetOf(_)`, `isStrictSubsetOf(_)`, `isStrictSupersetOf(_)` и `isDisjointWith(_)`.
- ❑ Для объявления словаря мы присваиваем переменной литерал словаря: `["ПН": "Понедельник", "ВТ": "Вторник", "СР": "Среда"]`.
- ❑ Указывать тип словаря нужно через названия типа для ключа и значения, окруженные квадратными скобками: `var daysOfWeek: [String: String]`.

- ❑ Для получения значения из словаря мы просто указываем ключ в квадратных скобках: `daysOfWeek["BT"]`.
- ❑ Для добавления элементов в словарь можно просто присвоить новое значение с новым ключом: `alphabetFruits["Г"] = "Груша"` либо воспользоваться методом `updateValue(_:forKey:)` для нового ключа.
- ❑ Для изменения значения элемента словаря мы присваиваем новое значение по существующему ключу: `alphabetFruits["А"] = "Ананас"` либо используем метод `updateValue(_:forKey:)` для существующего ключа.
- ❑ Для удаления элементов из словаря мы присваиваем `nil` значению с определенным ключом либо используем метод `removeValueForKey(_:)`.
- ❑ Для итерации по словарю мы используем цикл `for-in`.



ГЛАВА 6

Ветвление потока

До этого места мы писали лишь по несколько строк кода. Компилятор выполнял этот код строчка за строчкой. Ничего непредсказуемого не происходило. Но программы обычно бывают намного умнее. Нам часто придется сталкиваться с выполнением определенных действий, зависящих от неких условий. Например, если пользователь нажал на левую кнопку, то следует выполнить такой-то блок кода, а если пользователь нажал на правую кнопку — выполнить другой его блок.

Такие конструкции в языках программирования называются *условными выражениями*. Блок кода выполняется, если соблюдено определенное условие, иначе выполняется другой какой-то блок. Мы тем самым меняем ход выполнения программы, и в ней появляются *ветвления потока*.

Кроме условных выражений, еще одним фундаментальным понятием являются *циклы* — блоки кода, которые выполняются определенное количество раз. Например, если у нас есть программа, которая рассылает сообщения группе из 20 пользователей, то будет неразумно писать один и тот же код 20 раз. Это займет у нас много времени и много пространства в коде. Для организации такого процесса имеет смысл использовать циклы. Мы просто помещаем действие отправки сообщения внутрь цикла и говорим, чтобы оно выполнялось 20 раз. Это сэкономит наше время и сделает код намного компактнее.

6.1. Условия

6.1.1. Условный оператор *if*

Условный оператор `if` присутствует абсолютно во всех языках программирования. С его помощью можно записать условие типа: если выполняется условие, то выполни этот блок кода, в противном случае выполни другой его блок. В Swift условный оператор `if` пишется так:

```
if (условие) {  
    // выполнять блок кода, если условие выполнилось
```

```
} else {  
    // выполнять блок кода, если условие не выполнилось  
}
```

Это каркас условного оператора `if` в Swift. Мы пишем ключевое слово `if`, потом в круглых скобках определяем условие, а затем вписываем два блока кода, разделенные ключевым словом `else`. В зависимости от того, выполнится ли условие, будет запущен тот или иной блок кода. В ряде случаев блок `else` мы можем опустить. Тогда при несоблюдении условия компилятор просто пропустит блок с `if` и начнет выполнять строки кода дальше.

У оператора `if` в Swift, по сравнению с другими языками программирования, есть несколько особенностей:

- ❑ круглые скобки, ограничивающие условие, — не обязательны. Мы можем их использовать, если хотим, но строгого требования их присутствия нет;
- ❑ фигурные скобки, отделяющие блоки кода, — необходимы. Это правило обязательно и для блока, который выполняется, когда условие истинно, и для блока `else`;
- ❑ условие обязательно должно вычисляться как логическое значение. То есть, оно должно быть либо `true`, либо `false`. Во многих других языках программирования результатом условия может оказаться 0 или 1, и в зависимости от этого выполняются определенные блоки. Но в Swift этого нет.

Рассмотрим пример простого условного выражения:

```
var a = 3  
if (a < 0) {  
    print("Значение переменной a - отрицательное")  
} else {  
    print("Значение переменной a - положительное либо ноль")  
}  
// => "Значение переменной a - положительное либо ноль"
```

Здесь мы написали условие, которое проверяет, отрицательная ли переменная, положительная она либо вовсе ноль.

Для этого примера было бы красивее написать три блока: значение переменной либо ноль, либо положительное, либо отрицательное. В этом нам поможет блок `else if`, который запускается, когда основное условие не выполнилось и нужно проверить другое условие. Если это условие выполнилось, блок `else if` выполняется, если нет, то ветвление идет либо к следующему `else if`, либо к завершающему блоку `else`.

Давайте допишем новый блок `else if`, который будет проверять, равна ли переменная нулю:

```
var a = 3  
if (a < 0) {  
    print("Значение переменной a - отрицательное")
```

```
} else if ( a == 0 ){
    print("Значение переменной a - ноль")
} else {
    print("Значение переменной a - положительное")
}
// => "Значение переменной a - положительное"
```

Теперь мы поступили более красиво, отделив нулевое значение от положительного и отрицательного. Обратите внимание, что в условии `else if` мы написали `a == 0` с двумя знаками равно. То есть, использовали оператор сравнения (`==`), а не оператор присваивания (`=`), что было бы распространенной ошибкой.

6.1.2. Оператор *switch*

Иногда бывает так, что нам нужно проверить переменную на большее количество условий. Например, когда нам требуется вывести информацию о том, какой сегодня день недели. С использованием условия `else if` нам придется написать очень много кода. Но есть и альтернативный путь — применить оператор `switch`:

```
switch переменная {
case значение 1:
    действие при переменной, равной значению 1
case значение 2:
    действие при переменной, равной значению 2
case значение 3:
    действие при переменной, равной значению 3
default:
    действие при прочих условиях
}
```

Оператор `switch` сопоставляет значение переменной со значением, записанным в блоках `case`. Если оператор `switch` нашел совпадение, он выполняет действие, написанное для этого блока `case`. Если ни один из `case` не соответствует значению переменной, то `switch` выполняет действие, написанное в блоке `default`.

В Swift не требуется писать после каждого блока `case` указатель `break`, т. к. после выполнения подходящего блока `case` поток выполнения выходит из всей конструкции `switch`, а не проваливается к другим `case`, как это происходит в C.

Рассмотрим оператор `switch` на примере с днями недели. Скажем, у нас есть переменная `today`, в которой записан номер сегодняшнего дня недели. Мы будем выводить информацию о том, какой сегодня день недели в зависимости от значения в переменной `today`:

```
var today = 3
switch today {
case 1:
    print("Сегодня понедельник")
```



```
case 2:
    print("Сегодня вторник")
case 3:
    print("Сегодня среда")
case 4:
    print("Сегодня четверг")
case 5:
    print("Сегодня пятница")
case 6:
    print("Сегодня суббота")
case 7:
    print("Сегодня воскресенье")
}
// => Ошибка
```

В чем дело? Мы, вроде, все записали правильно, но компилятор выдает нам ошибку. А дело в том, что одним из обязательных условий использования оператора `switch` является требование, чтобы `switch` охватывал все возможные варианты. Здесь мы перечислили все возможные дни недели, но проблема не в этом, а в том, что переменная `today` случайным образом может равняться 8, или 15, или 110. У нас нет дней недели с такими номерами, и поэтому мы как-то должны ограничить наш код от подобных вариантов. Однако прописать все возможные здесь варианты бессмысленно, поскольку их бесчисленное множество. Поэтому мы воспользуемся блоком `default`, который будет выполняться при условии, что `switch` не нашел совпадения ни с одним значением `case`:

```
var today = 3
switch today {
case 1:
    print("Сегодня понедельник")
case 2:
    print("Сегодня вторник")
case 3:
    print("Сегодня среда")
case 4:
    print("Сегодня четверг")
case 5:
    print("Сегодня пятница")
case 6:
    print("Сегодня суббота")
case 7:
    print("Сегодня воскресенье")
default:
    print("Неправильный номер недели")
}
// => "Сегодня среда"
```

Как видим, наша конструкция успешно вывела нам результат "Сегодня среда", поскольку переменная `today` равняется 3.

Кроме единичных значений, в блоках `case` можно также писать сопоставление по группам значений, разделенных запятыми:

```
var today = 3
switch today {
case 1,2,3,4,5:
    print("Сегодня будний день")
case 6,7:
    print("Сегодня выходной")
default:
    print("Неправильный номер недели")
}
// => "Сегодня будний день"
```

Группы значений также можно заменить на диапазоны:

```
var today = 3
switch today {
case 1...5:
    print("Сегодня будний день")
case 6...7:
    print("Сегодня выходной")
default:
    print("Неправильный номер недели")
}
// => "Сегодня будний день"
```

С использованием диапазонов значений может получиться так, что сопоставление произойдет с двумя или несколькими диапазонами. Например, если мы введем новый диапазон *Середина недели* от 2 до 4, то наше значение попадет и в этот диапазон тоже:

```
var today = 3
switch today {
case 1...5:
    print("Сегодня будний день")
case 2...4:
    print("Середина недели")
case 6...7:
    print("Сегодня выходной")
default:
    print("Неправильный номер недели")
}
// => "Сегодня будний день"
```

Но условие вывело нам только значение "Сегодня будний день" из первого блока `case`. А все потому, что компилятор выходит из оператора `switch` после первого

совпадения. И чтобы явно провалиться в следующий блок `case`, мы в данном случае можем использовать ключевое слово `fallthrough`:

```
var today = 3
switch today {
case 1...5:
    print("Сегодня будний день")
    fallthrough
case 2...4:
    print(" и середина недели")
case 6...7:
    print("Сегодня выходной")
default:
    print("Неправильный номер недели")
}
// => "Сегодня будний день и середина недели"
```

Теперь наше условие находит совпадение в блоке `case 1...5` и выводит его сообщение, а затем проваливается к следующему `case` и выводит также и его сообщение. Здесь следует обратить внимание, что проваливание происходит, даже если блок `case 2...4` не подходит по условию. Поэтому этот пример лучше переписать так:

```
var today = 3
switch today {
case 2...4:
    print("Сегодня середина недели и ")
    fallthrough
case 1...5:
    print("Сегодня будний день")
case 6...7:
    print("Сегодня выходной")
default:
    print("Неправильный номер недели")
}
// => "Сегодня середина недели и Сегодня будний день"
```

Теперь условие с меньшим диапазоном находится выше, и если номер дня недели попадает в диапазон от 2 до 4, то выводится сообщение "Сегодня середина недели и Сегодня будний день", а если номер дня недели 1 или 5, то выводится сообщение "Сегодня будний день".

Если для определенного `case` нам нужно дополнительное условие, мы его можем записать через ключевое слово `where`:

```
var today = 3
switch today {
case 1...5 where today == 3:
    print("Сегодня среда")
case 1...5:
    print("Сегодня будний день")
```

```
case 6...7:
    print("Сегодня выходной")
default:
    print("Неправильный номер недели")
}
// => "Сегодня среда"
```

6.2. Циклы

Циклы в языках программирования призваны облегчить множественное выполнение одинакового набора действий. Большинство циклов в Swift наследованы из языка C. В *главе 5* мы вскользь познакомились с циклом `for-in`, который позволял производить итерацию по массивам и словарям. Здесь мы продолжим изучение итераций и познакомимся с остальными циклами Swift.

6.2.1. Циклы *for*

В Swift есть два типа циклов `for`:

- ❑ традиционный цикл `for`, наследованный из C, который позволяет указывать переменную, условие и инкремент;
- ❑ цикл `for-in`, который позволяет удобно производить итерацию по коллекциям, диапазонам и символам в строке.

Стандартный цикл *for*

Конструкция объявления цикла `for` состоит из трех отдельных частей:

```
for инициализация; условие; инкремент {
}
```

- ❑ В блоке инициализации мы объявляем переменную-итератор, с которой будем работать внутри тела цикла.
- ❑ В блоке условия мы пишем выражение, которое проверяется после каждой итерации цикла. Если выражение внутри условия вычисляется как `true`, то цикл `for` срабатывает еще один раз. И так до тех пор, пока выражение не вычислится до `false`. Тогда итерация по циклу прекратится и станет выполняться код после цикла `for`.
- ❑ В блоке инкремента мы указываем, как должна будет изменяться переменная-итератор. Обычно она либо увеличивается (`++`), либо уменьшается (`--`).

Рассмотрим пример простого цикла на 10 итераций, который с каждым циклом печатает значение итератора:

```
for ( var i = 1; i <= 10; i++ ) {
    print(i)
}
```

```
// 1
// 2
// 3
// 4
// 5
// 6
// 7
// 8
// 9
// 10
```

В этом примере в блоке инициализации мы создали переменную `i` и присвоили ей начальное значение 1. В блоке условия мы записали, что цикл должен проходить до тех пор, пока значение переменной `i` не станет меньше либо равно 10. В блоке инкремента мы указали, что с каждым прохождением цикла переменная `i` будет увеличиваться на единицу. В теле цикла мы написали функцию `print`, которая будет печатать каждое значение переменной `i`.

В отличие от цикла `for` в C, в Swift мы можем не ставить скобки вокруг блоков цикла. То есть, приведенный цикл мы можем переписать так:

```
for var i = 1; i <= 10; i++ {
    print(i)
}
```

Обратите внимание, что при объявлении переменной мы обязательно должны писать ключевое слово `var`. А поскольку эта переменная олицетворяет итератор, который меняется с каждым циклом, то объявлять ее в виде константы нецелесообразно.

Цикл *for-in*

Цикл `for-in` прекрасно подходит для итерации по коллекциям, диапазонам и любым наборам значений. Конструкция объявления цикла `for-in` выглядит следующим образом:

```
for переменная цикла in набор значений {

}
```

В блоке переменной мы указываем переменную, с которой будем работать внутри тела цикла. В блоке набора значений мы можем указать массивы, словари, диапазоны, строки и т. д.

Приведем пример, который мы использовали для итерации по массиву в *главе 5*:

```
var fruits = ["Яблоко", "Груша", "Банан", "Апельсин", "Арбуз"]
for fruit in fruits {
    print(fruit)
}
```

```
// "Яблоко"  
// "Груша"  
// "Банан"  
// "Апельсин"  
// "Арбуз"
```

6.2.2. Цикл *while*

Цикл `while` позволяет выполнить одну и ту же последовательность действий, пока его условие истинно. При выполнении цикла `while` сначала проверяется условие. Если оно истинно, то выполняется код, написанный в теле цикла. Затем условие проверяется снова, и так до тех пор, пока оно не станет ложным. Когда условие становится ложным, то код внутри тела цикла не выполняется, и поток выполнения переходит на строчку, следующую за циклом.

Чтобы создать цикл `while`, мы пишем ключевое слово `while`, затем условие и тело цикла:

```
var i = 1  
while ( i <= 10 ) {  
    print("Swift")  
    i++  
}
```

В этом примере мы поручили циклу `while` выполняться до тех пор, пока `i` меньше либо равно 10. С каждой итерацией цикла мы печатаем слово "Swift", а затем увеличиваем значение параметра `i` на единицу. Как только `i` станет равно 11, условие не выполнится, и поток выйдет из цикла и начнет выполнять следующие за циклом строки. Тем самым мы получим 10 строк со словом "Swift".

Слово `while` переводится с английского языка как «до тех пор, пока». Это означает, что, создавая цикл `while`, мы как бы говорим программе, чтобы она выполняла инструкции внутри тела цикла, «пока» условие цикла истинно.

Кроме цикла `while` можно также создавать цикл `repeat while`, который выполняет проверку условия после выполнения всех инструкций цикла. Для примера перепишем предыдущий код с использованием цикла `repeat while`:

```
var i = 1  
repeat {  
    print("Swift")  
    i++  
} while ( i <= 10 )
```

Здесь программа напечатает слово "Swift" 11 раз вместо 10. Дело в том, что когда `i` достигает 10, то цикл все равно срабатывает еще один раз. И лишь после завершения итерации проверится условие, которое на этот раз вычислится как `false`, и программа выйдет из цикла.

6.3. Управление потоком цикла

С помощью указателей `break` и `continue` мы можем управлять потоком цикла. Указатель `break` останавливает выполнение цикла и выходит из него, а указатель `continue` переводит цикл на следующую итерацию.

Воспользуемся примером цикла `for` и напишем условие: если значение `i = 5`, то немедленно выйти из цикла:

```
for ( var i = 1; i <= 10; i++ ) {  
    if i == 5 {  
        break  
    }  
    print(i)  
}  
// 1  
// 2  
// 3  
// 4
```

Как мы видим, цикл выполнил только четыре успешные итерации, а на пятой выполнилось условие `i = 5`, и `break` остановил работу цикла. Теперь заменим указатель `break` на `continue`:

```
for ( var i = 1; i <= 10; i++ ) {  
    if i == 5 {  
        continue  
    }  
    print(i)  
}  
// 1  
// 2  
// 3  
// 4  
// 6  
// 7  
// 8  
// 9  
// 10
```

Как мы здесь видим, цикл не напечатал значение 5. То есть, указатель `continue` не позволил до конца выполниться пятой итерации и дал команду перейти к следующей.

Если у нас в коде будет много вложенных циклов, то компилятор может запутаться какие `break` или `continue` к какому циклу относятся. Для этого в Swift есть возможность давать циклам имена:

```
mainLoop: for ( var i = 1; i <= 10; i++ ) {  
    if i == 5 {  
        break mainLoop  
    }  
}
```

```
    print(i)
  }
  // 1
  // 2
  // 3
  // 4
```

В этом случае мы явно написали, что указателю `break` нужно остановить выполнение именно цикла `mainLoop`.

6.4. Оператор *guard*

Оператор `guard`, подобно оператору `if`, проверяет выражение внутри скобок и выполняется только при условии, что это выражение истинно. Но, в отличие от оператора `if`, в операторе `guard` всегда присутствует оператор `else`. Конструкции с участием оператора `guard` используются, когда нужно поставить ограничение на выполнение кода после этой конструкции:

```
for ( var i = 1; i <= 10; i++ ) {
  guard i == 5 else {
    continue
  }
  print(i)
}
// 5
```

Здесь мы взяли цикл из предыдущего раздела и вместо оператора `if` написали оператор `guard`. Как можно видеть, цикл напечатал только цифру 5.

Если условие внутри оператора `guard` вычисляется как `true`, то код продолжает выполняться после закрывающей фигурной скобки оператора `guard`. Если условие вычисляется как `false`, то выполняется блок `else`, в котором обычно пишется один из указателей управления потоком: `break` или `continue`, либо — для функций — пишется `return`. Эти указатели переводят поток выполнения в другую часть кода, не давая выполниться коду после оператора `guard`.

Оператор `guard`, так же как и оператор `if`, можно использовать и при опциональной привязке:

```
guard let unwrappedItem = optionalItem else {
  break
}
```

В этом случае, если опциональная привязка выполнилась, и `optionalItem` — не `nil`, то поток выполнения продолжится после закрывающей фигурной скобки оператора `guard`, а иначе выполнится указатель `break`.

6.5. Проверка на доступность API

С приходом версии 2.0 в Swift появилась встроенная возможность проверять, доступен ли тот или иной API. Это бывает полезно, когда мы пишем единый код сразу для нескольких версий платформ — например, для iOS 8 и iOS 9. В более новой операционной системе могут присутствовать новые методы, которых не было в старых. В таких случаях мы просто можем написать конструкцию проверки доступности, которая будет выполняться только при условии присутствия API на платформе, где выполняется этот код. Конструкции проверки доступности пишутся либо через оператор `if`, либо через оператор `guard`:

```
if #available(iOS 9, OSX 10.10, *) {  
    // Здесь можно использовать API, доступные в iOS 9 и OS X 10.10  
} else {  
    // Здесь нужно использовать более ранние API  
}
```

Выводы

- ❑ Условные операторы `if` и `switch` позволяют создавать условия.
- ❑ Для оператора `if` круглые скобки не обязательны, а фигурные — обязательны. Также условие должно возвращать логическое значение `true` или `false`.
- ❑ Оператор `switch` явно не проваливается к следующим значениям, если мы не укажем указатель `break`. Если проваливание все же нужно, мы можем написать указатель `fallthrough`.
- ❑ Циклы `for`, `for-in`, `while` и `repeat while` позволяют в Swift производить действия заданное количество раз.
- ❑ Указатель `break` помогает остановить выполнение цикла, а указатель `continue` — перейти к выполнению следующей итерации.
- ❑ Оператор `guard` помогает останавливать выполнение последующего кода, если его условие не выполнилось.
- ❑ Мы можем проверить API на доступность с помощью конструкции `if #available`.

ГЛАВА 7



Функции

По мере добавления новых строк кода рано или поздно мы приходим к выводу, что наш код получается слишком длинным, а алгоритмы некоторых инструкций повторяются в ряде мест. Слишком длинные листинги кода сложно читать и понимать. Поэтому были придуманы *функции* — блоки кода с определенной функциональностью, которые можно многократно использовать в коде. Для того чтобы объявить функцию, нам нужно дать ей название (имя) и написать в ее теле набор выполняемых инструкций. Затем в тех местах, где нам нужно выполнить эти инструкции, мы просто вызываем содержащую их функцию, обращаясь к ней по ее имени. Использование функций позволяет сократить повторение одинаковых последовательностей инструкций и сделать код более читаемым.

Функции могут иметь параметры. Они позволяют передавать функции значения во время ее вызова. Кроме параметров, функция также может иметь возвращаемое значение.

В этой главе мы сначала узнаем, как в Swift объявить и вызвать функцию. Затем остальная часть главы будет разбита на два больших раздела, посвященных параметрам функций и возвращаемым значениям. В них мы подробно изучим различные техники использования этих возможностей на практике. В конце главы мы еще глубже погрузимся в изучение функций Swift и рассмотрим вложенные функции, передачу функции в качестве параметра другой функции и возврат функции из другой функции.

7.1. Объявление функции

Объявим сначала простую функцию. Для этого нам нужно воспользоваться ключевым словом `func`:

```
func sayTime() {  
    print("It's late")  
}
```

Каждый раз при вызове эта функция будет печатать "It's late". Так же, как и в случае переменных и констант, названия для функций принято писать в *горбатой*

нотации и начинать с маленькой буквы. Скобки после названия функции в Swift — обязательны. Обязательно и наличие фигурных скобок для ограничения тела функции.

Для того чтобы выполнить инструкции, записанные внутри тела функции, нам нужно вызвать эту функцию. Это можно сделать, просто написав название функции, оканчивающееся парой скобок:

```
sayTime() // It's late
```

Теперь эта функция напечатает нам на экране "It's late". Мы сейчас объявили и вызвали самую простую функцию. В ней содержится лишь одна инструкция, которая печатает нам строку на экране. Чтобы иметь возможность передавать значения внутрь тела функции, нам нужно научиться создавать параметры функции.

7.2. Параметры

Параметры позволяют сделать функции более гибкими. С их помощью мы можем передать функции какие-либо значения и влиять на выполнение инструкций внутри тела функции. Для того чтобы лучше понять сказанное, напомним новую функцию, которая складывает два числа и печатает результат на экране через функцию `print`:

```
func addTogether(firstNum: Int, secondNum: Int) {  
    print(firstNum + secondNum)  
}
```

Мы объявили простую функцию `addTogether`, которая принимает два параметра типа `Int`: `firstNum` и `secondNum`. Параметры функций всегда по умолчанию являются константами, и они доступны только в пределах тела функции. То есть, эти параметры мы можем использовать лишь внутри тела функции и не можем изменить их значения. Для изменения значений параметров придется сначала скопировать эти значения в другую переменную и лишь потом производить над ней те или иные действия. Либо можно заранее явно объявить параметры как переменные, написав перед названием параметра ключевое слово `var`:

```
func addTogether( var firstNum: Int, var secondNum: Int) {  
    print(firstNum + secondNum)  
}
```

В любом случае, использование параметров-констант предпочтительнее, т. к. это исключает возможность ошибок. В нашем примере мы будем использовать именно функцию с параметрами-константами.

Объявив функцию с двумя параметрами, мы можем теперь вызвать эту функцию и передать ей соответствующие параметры `Int` в скобках:

```
addTogether(1, secondNum: 2) // => 3
```

Функция напечатает нам результат сложения двух этих параметров. Почему мы не написали имя первого параметра, станет ясно из следующего раздела.

Теперь мы можем вызвать эту функцию в любом месте кода, где хотим напечатать результат сложения двух чисел.

7.2.1. Внешние имена параметров

Для параметров функций в Swift существуют два типа имен: локальные и внешние. Внутри объявления функции мы используем локальные имена параметров, а во время вызова функции — внешние. В нашем предыдущем примере мы воспользовались только локальными именами параметров:

```
func addTogether(firstNum: Int, secondNum: Int) {  
    print(firstNum + secondNum)  
}
```

Здесь `firstNum` и `secondNum` — локальные имена параметров. Они могут использоваться только внутри объявления функции. Чтобы мы могли использовать имена параметров во время вызова функции, мы должны написать для этих параметров *внешние имена*. Внешние имена пишутся перед локальными именами параметров. Напишем, например, для первого параметра внешнее имя `first`, а для второго — `second`:

```
func addTogether(first firstNum: Int, second secondNum: Int) {  
    print(firstNum + secondNum)  
}
```

Теперь при вызове функции мы должны использовать имена этих параметров:

```
addTogether(first: 2, second: 3) // => 5
```

Если мы не хотим использовать внешние имена параметров, то во время объявления функции можем их опустить и на месте имен внешних параметров написать знак подчеркивания `_`:

```
func addTogether(_ firstNum: Int, _ secondNum: Int) {  
    print(firstNum + secondNum)  
}
```

В этом случае при вызове функции мы можем вообще не использовать никаких имен:

```
addTogether(2, 3)
```

Теперь, когда мы знаем, что такое локальные и внешние имена параметров, посмотрим, как ведет себя функция, объявленная только с локальными параметрами, и почему при вызове функции в предыдущем разделе мы для первого параметра имя не использовали, а для второго это сделать пришлось.

С обновлением Swift до версии 2.0, при использовании только локальных имен параметров Swift автоматически опускает внешнее имя первого параметра, а имена остальных локальных параметров автоматически делает и внешними тоже. Это означает, что для первого параметра Swift как бы незаметно ставит знак нижнего подчеркивания на месте имени внешнего параметра:

```
func addTogether( _ firstNum: Int, secondNum secondNum: Int) {  
    print(firstNum + secondNum)  
}
```

Приведенная здесь функция эквивалентна функции без использования имен внешних параметров.

Теперь при вызове функции мы не должны использовать имя первого параметра, а для остальных параметров имя обязательно должно присутствовать:

```
addTogether(2, secondNum: 3) // => 5
```

Вот почему мы не использовали имя для первого параметра в предыдущем разделе.

7.2.2. Параметры со значением по умолчанию

В Swift есть возможность назначать параметрам во время объявления функции *значение по умолчанию*, т. е. значение, которое будет присвоено этому параметру в случае, если ему не присвоили значение при вызове функции.

Напишем функцию `getAreaOfCircle`, которая вычисляет площадь круга. Как мы помним, площадь круга вычисляется умножением π на радиус в квадрате:

```
func getAreaOfCircle(radius: Double, pi: Double = 3.14) {  
    let result = pi * radius * radius  
    print(result)  
}
```

В этом примере мы сделали функцию более гибкой. Теперь те, кто хочет просто вычислить приблизительное значение площади круга с π , равным 3.14, могут пропустить параметр `pi` при вызове функции:

```
getAreaOfCircle(5)
```

В этом случае функция будет использовать то значение параметра `pi`, которое мы указали при объявлении. А те, кто хочет вычислить площадь круга поточнее, могут написать в параметре `pi` более точное значение π :

```
getAreaOfCircle(5, pi: 3.14159)
```

Теперь функция будет использовать то значение параметра `pi`, которое мы укажем при вызове функции.

Стоит обратить внимание, что параметры по умолчанию автоматически становятся внешними. Это означает, что мы обязательно должны указывать название параметров при вызове функции.

7.2.3. Сквозные параметры

Бывают случаи, когда нам нужно, чтобы функция изменяла значение своего параметра. Скажем, требуется передать функции переменную, которую надо возвести в квадрат. Если мы передадим переменную в виде параметра обычным способом,

то функция не сможет изменить хранимое в памяти значение этой переменной. Чтобы функция имела возможность изменять значение переменной, хранящееся в ячейке памяти, нужно указать, что этот параметр является *сквозным*.

Для примера напомним функцию, которая возводит в третью степень число, которое мы ему передадим. Без применения сквозных параметров функция выглядела бы так:

```
var someNumber = 3
func toThirdDegree(number: Int) {
    print( number * number * number )
}
toThirdDegree(someNumber) // => 27
someNumber // => 3
```

Здесь мы просто передали переменную `someNumber`, которую функция возвела в третью степень и вернула нам результат. То есть она скопировала значение, хранящееся по адресу этой переменной, возвела ее в требуемую степень и вернула нам результат. При этом переменная осталась с тем же значением, какое и было.

Но мы хотим, чтобы функция изменила значение, хранящееся в памяти этой переменной, возведя его в третью степень. Давайте напишем тот же пример, только с применением сквозных параметров. Параметры, которые мы хотим сделать сквозными, обозначаются ключевым словом `inout` перед названием параметра. А при вызове функции мы ставим перед названием переменной символ амперсанда (&):

```
var someNumber = 3
func toThirdDegree(inout number: Int) {
    number = number * number * number
}

toThirdDegree(&someNumber) // => Ничего
someNumber // => 27
```

Здесь нам стоит обратить внимание на то, что наша функция ничего не печатает. Ее работа теперь состоит в изменении значения переменной, а не просто в выводе информации. И на последней строчке кода мы видим, что значение переменной теперь поменялось на 27.

Стоит также заметить, что при вызове функции со сквозным параметром мы не можем ей передать константу или литерал, поскольку их невозможно изменить. Соответственно, в этом случае смысл сквозных параметров пропадает.

7.2.4. Функции с переменным числом параметров

В тех случаях, когда количество параметров заранее неизвестно, — например, когда мы хотим вычислить среднее арифметическое неизвестного нам заранее количества чисел, можно объявить функцию с переменным числом параметров. Это означает, что при вызове функции можно писать сколько угодно параметров, —

Swift просто объединит все значения этого множественного параметра в массив, с которым можно работать внутри тела функции.

Чтобы объявить функцию с переменным количеством параметров, нам нужно добавить троеточие (...) после названия типа параметра. После этого в теле функции параметр становится массивом со значениями, которые будут переданы при вызове функции.

Давайте рассмотрим все это на примере. Объявим функцию, которая вычисляет среднее арифметическое чисел, которые мы передадим ей в виде параметров. Назовем эту функцию `getAverage`:

```
func getAverage(nums: Double...) {  
    var result: Double = 0  
    for num in nums {  
        result += num  
    }  
    print( result / Double(nums.count) )  
}
```

```
getAverage(3,2,5,2,5,6,6,5) // => 4.25
```

Здесь мы объявили функцию `getAverage`, которая вычисляет среднее арифметическое из переданных ей параметров. Мы передали ей множественный параметр `nums`, который внутри тела цикла превратился в массив из значений `Double`. Внутри тела функции мы сложили каждое значение из массива и разделили на их общее количество.

В официальной документации Swift параметры с переменным числом значений называют *вариативными параметрами*. Swift также позволяет использовать вариативный параметр вместе с обычными параметрами, но только во время объявления функции нужно вписать вариативный параметр в конце списка параметров. Это позволяет избежать путаницы и вам, и компилятору.

7.3. Возвращаемое значение функции

В начале главы мы говорили, что функции позволяют не только принимать параметры, но и возвращать значения. В предыдущих примерах мы использовали функцию `print()`, которая просто печатала значение на экране. В этом разделе мы научимся создавать функции, которые возвращают значения.

Давайте рассмотрим знакомую нам функцию `addTogether`, складывающую значения своих двух параметров. Только теперь перепишем ее так, чтобы она возвращала результат операции сложения. Для того чтобы указать, что функция может возвращать значения, мы должны добавить ключевое слово `return` в конце тела функции:

```
func addTogether(firstNum: Int, _ secondNum: Int) {  
    return firstNum + secondNum  
}
```

После ключевого слова мы написали выражение, результат которого должна вернуть функция. Но для того, чтобы функция возвращала значения, этого недостаточно. В Swift, если функция возвращает значение, мы обязательно должны указать тип этого возвращаемого значения. Он указывается с помощью знака «стрелка» (\rightarrow) после объявления параметров:

```
func addTogether(firstNum: Int, _ secondNum: Int) -> Int {  
    return firstNum + secondNum  
}
```

Поскольку функция `addTogether` складывает два значения типа `Int`, то и результатом ее будет значение типа `Int`, которое мы и указали после объявления параметров функции.

Теперь давайте вызовем нашу функцию. Для того чтобы показать, что функция возвращает значение, мы присвоим результат функции переменной, а затем напечатаем ее значение:

```
var result = addTogether(3,2)  
print(result) // => 5
```

В этом примере функция возвращает число 5, которое присваивается переменной `result`. А на следующей строчке мы печатаем значение этой переменной — чтобы убедиться, что функция действительно возвращает значения.

Честно говоря, функции, которые не возвращают значения, на самом деле возвращают специальное значение с типом `Void`. Это значение является пустым кортежем (о кортежах — см. далее), которое также можно записать и как `()`. А значение типа `Void` — это просто псевдоним для типа `()`:

```
typealias Void = ()
```

7.3.1. Функции с несколькими возвращаемыми значениями

В Swift функции могут возвращать несколько значений одновременно. Чтобы это сделать, как результат функции нужно передать *кортеж*. С кортежами мы познакомимся во второй части книги, а пока просто запомним, что кортеж — это сгруппированные значения любого типа.

Давайте напишем функцию, выводящую минимальное и максимальное значения из группы чисел, которую мы ему передадим. Функция будет принимать множественный параметр `numbers`, в который мы занесем неограниченное количество чисел. Результатом же функции станет кортеж из двух значений типа `Int`:

```
func getMinMax(numbers: Int...) -> (Int, Int) {  
    var min = numbers[0]  
    var max = numbers[0]  
  
    for number in numbers {  
        if number > max {
```



```

        max = number
    }

    if number < min {
        min = number
    }
}

return (min, max)
}

getMinMax(4, 83, 7, 106, 44) // => (4,106)

```

Как мы видим на последней строчке, функция вернула нам кортеж из значений 4 и 106. Для того чтобы нам было удобнее работать с такими значениями, при объявлении функции мы можем указать имена для элементов кортежа:

```

func getMinMax(numbers: Int...) -> (min: Int, max: Int) {
    var min = numbers[0]
    var max = numbers[0]

    for number in numbers {
        if number > max {
            max = number
        }

        if number < min {
            min = number
        }
    }

    return (min, max)
}

```

Теперь мы можем просто обратиться к этим элементам через точку:

```

getMinMax(4, 83, 7, 106, 44).min // => 4
getMinMax(4, 83, 7, 106, 44).max // => 106

```

То же самое мы могли бы сделать, если бы присвоили результат функции переменной:

```

var value = getMinMax(4, 83, 7, 106, 44)
var minValue = value.min // => 4
var maxValue = value.max // => 106

```

В завершение этого раздела надо отметить, что функции с множественными возвращаемыми значениями предоставляют программисту весьма интересные возможности.

7.4. Функции — объекты первого класса

Функции в Swift являются объектами *первого класса*. Это означает, что мы можем работать с ними как с обычными значениями, — например, присваивать их переменным и константам. Так, давайте создадим новую функцию и присвоим ее переменной:

```
func multiply(first: Int, _ second: Int) -> Int {  
    return first * second  
}  
  
var action = multiply
```

Следует обратить внимание, что на последней строчке мы не поставили скобки в конце имени функции `multiply`. Дело в том, что мы не собираемся присваивать значение, возвращаемое функцией, а хотим присвоить саму функцию. Как мы помним, Swift является языком со статичной типизацией, а это означает, что переменная `action` должна иметь какой-нибудь тип. Так какой же тип может иметь переменная, которой присваивается функция? В Swift такой тип называется *функциональным*. Чтобы было лучше понятно, давайте явно укажем тип для переменной `action`:

```
var action: (Int, Int) -> Int = multiply
```

И переменная `action`, и функция `multiply` имеют функциональный тип `(Int, Int) -> Int`. Как можно видеть, этот тип включает в себя типы параметров и тип возвращаемого значения, но все вместе это и называется функциональным типом. Его иногда еще называют *сигнатурой функции*.

Интересно, что для функционального типа, как и для любого другого типа, мы тоже можем объявить псевдоним:

```
typealias IntAction = (Int, Int) -> Int
```

Иногда это позволяет избавиться от слишком сложного и непонятного объявления функциональных типов.

Присвоив функцию переменной, мы можем работать с переменной `action` так, как будто это функция `multiply`:

```
action(2,3) // => 6
```

Кроме присваивания переменной либо константе, мы можем возвращать функцию из другой функции, а также принимать функцию как параметр другой функции. Давайте сначала рассмотрим функции, принимающие в виде параметра другие функции.

7.4.1. Функции, принимающие параметры в виде функции

Функции могут принимать другие функции в виде своих параметров. Например, передадим предыдущую функцию `multiply` в качестве параметра функции `doubleMultiply`:

```
func multiply(first: Int, second: Int) -> Int {
    return first * second
}

func doubleMultiply(multiply: (Int, Int) -> Int, num1: Int, num2: Int) -> Int {
    return multiply(num1, num2) * 2
}

doubleMultiply(multiply, num1: 2, num2: 3) // => 12
```

Функция `doubleMultiply` просто берет результат `multiply` и умножает его на два. Поскольку функция `multiply` является параметром функции, то при объявлении мы обязательно должны указать его тип: `(Int, Int) -> Int`.

7.4.2. Функции, возвращающие функцию

Функции также могут возвращать другие функции в качестве результата. Для того чтобы это лучше понять, давайте рассмотрим пример. Допустим, у нас есть две функции, которые возвращают нам "Yes" и "No" соответственно:

```
func sayYes() -> String {
    return "Yes"
}

func sayNo() -> String {
    return "No"
}
```

Теперь создадим функцию, которая принимает единственный логический параметр `correctAnswer` и в зависимости от его значения возвращает нам функцию, которая возвращает "Yes" или "No":

```
func quiz(correctAnswer: Bool) -> () -> String {
    return correctAnswer ? sayYes : sayNo
}
```

Теперь, когда мы вызовем функцию с параметром `true`, функция вернет нам функцию `sayYes()`, в противном случае — функцию `sayNo()`:

```
quiz(true) // => () -> String
```

Если мы напишем все это в Playground, то в правой области, где показывается результат выражения, нам просто выведется `() -> String`. Это показывает, что наша функция `quiz` вернула нам функцию, но пока не понятно, какую. Чтобы убедиться, что функция `quiz` возвращает нам правильную функцию, мы должны вызвать эту возвращаемую функцию. Как мы помним из начала главы, функции вызываются с помощью пары скобок в конце названия функции. Руководствуясь этим знанием, мы можем написать такое выражение:

```
quiz(true)() // => "Yes"
```

Здесь мы добавили для функции `quiz` вторую пару скобок, что означает: «вызвать возвращаемую функцию». Аналогично мы могли бы сначала объявить новую переменную, в которую записали бы результат вызова функции `quiz`, а затем вызвать ее:

```
var action = quiz(true)
action() // => "Yes"
```

Как мы видим, результат оказывается тем же.

7.4.3. Вложенные функции

До этого места мы писали глобальные функции, т. е. функции, которые объявлены в глобальной области. Но в Swift есть возможность писать функции внутри других функций. Такие функции называются *вложенными*, или *локальными*, функциями.

Мы можем переписать предыдущий пример и написать объявление функций `sayYes()` и `sayNo()` внутри объявления функции `quiz`:

```
func quiz(correctAnswer: Bool) -> () -> String {

    func sayYes() -> String {
        return "Yes"
    }

    func sayNo() -> String {
        return "No"
    }

    return correctAnswer ? sayYes : sayNo
}
```

При этом все будет работать точно так же — функция `quiz` аналогичным образом будет вызываться и выводить нам другую функцию:

```
quiz(true)() // => "Yes"
```

Что же тогда изменилось? Дело в том, что теперь функции `sayYes()` и `sayNo()` доступны лишь внутри функции, и мы не можем больше вызвать эти функции извне, — теперь они могут вызываться только с помощью родительской функции `quiz`. Следовательно, вложенные функции очень полезны, когда нам нужно скрыть или изолировать какую-либо функцию.

Вложенные функции могут использовать переменные и константы, объявленные в родительской функции. Они полезны, когда нужно изолировать либо скрыть какую-либо функциональность, которую мы не хотим, чтобы можно было вызвать ее извне.

Выводы

- Функции объявляются через ключевое слово `func`.
- Параметры функции по умолчанию являются константами, но при желании можно сделать их переменными с помощью `var`.

- ❑ Названия параметров по умолчанию локальные, т. е. их можно использовать только внутри тела функции. По желанию можно объявить внешние имена для параметров, чтобы они стали доступны вне функции.
- ❑ Если не указаны имена внешних параметров, то по умолчанию внешнее имя для первого параметра опускается, а имена всех остальных параметров становятся внешними.
- ❑ Параметрам можно при объявлении задавать значения по умолчанию.
- ❑ Параметры можно объявлять как сквозные через ключевое слово `inout`. Тогда они могут изменять значения переменных, переданных в эти параметры.
- ❑ Функции могут иметь переменное число параметров. Тогда при объявлении после типа нужно написать троеточие (`Int...`).
- ❑ Функциям можно указать возвращаемое значение.
- ❑ Тип возвращаемого значения указывается при объявлении через «стрелку» (`->`).



ЧАСТЬ II

Углубленное изучение Swift

Вторая часть описывает более сложные структуры языка программирования Swift, большинство которых не встречается в других языках программирования. Эта часть будет наиболее интересна опытному разработчику, а начинающий разработчик сможет перейти к ней после подробного изучения первой части.

- ☐ Глава 8. Опциональные типы.
- ☐ Глава 9. Кортежи.
- ☐ Глава 10. Замыкания.
- ☐ Глава 11. Перечисления.
- ☐ Глава 12. Классы.
- ☐ Глава 13. Наследование.
- ☐ Глава 14. Автоматический подсчет ссылок.
- ☐ Глава 15. Структуры.
- ☐ Глава 16. Проверка типов и приведение типов.
- ☐ Глава 17. Расширения.
- ☐ Глава 18. Протоколы.
- ☐ Глава 19. Обобщенные типы.
- ☐ Глава 20. Обработка ошибок.
- ☐ Глава 21. Расширенные операторы.

ГЛАВА 8



Опциональные типы

В начале первой части, когда мы изучали простые типы данных, мы говорили о том, что в Swift переменные или константы перед использованием всегда должны иметь значение. Но иногда бывают случаи, когда нужно сделать так, чтобы переменная могла не иметь значение вовсе. В Swift отсутствие какого-либо значения задается через `nil`, но мы не можем просто так присвоить переменной `nil`. Давайте рассмотрим пример.

Допустим у нас есть переменная `date`, в которую нам в строковом виде передается текущая дата. Представим, что значение этой переменной передается нам из Интернета, а на следующей строчке кода мы печатаем его значение:

```
var date: String
date = "29.07.2015" // Получено из Интернета
print("Текущая дата: \(date)")
```

Все вроде бы хорошо, и наш кусок кода напечатает нам "Текущая дата: 29.07.2015". Но проблемы начнутся, если по причине недоступности сервера мы не сможем получить значение для переменной `date`. В таком случае компилятор выведет нам ошибку: **Variable 'date' used before being initialized**, что означает «Переменная 'date' была использована до ее инициализации». То есть, перед тем как печатать либо каким-нибудь другим способом использовать значение переменной, нужно убедиться, что в переменной это значение есть.

Чтобы избежать ошибки, мы могли бы вначале присвоить переменной какое-нибудь начальное значение, например: `0`. Но это было бы тоже неправильно, т. к. `0` также является значением, и при сбое получения значения для переменной `date` из Интернета наша программа напечатала бы "Текущая дата: 0".

Мы также могли бы попробовать вначале присвоить переменной `date` значение `nil`:

```
date = nil // Ошибка
```

Но компилятор не дает нам это сделать, поскольку для того чтобы переменная или какой-нибудь другой объект могли принимать значение `nil`, они должны иметь

опциональный тип. И чтобы объявить переменную с опциональным типом, нам нужно при объявлении в конце названия типа написать знак вопроса:

```
var date: String?
```

Теперь эта строка означает, что мы создали опциональную переменную `date`, в которой уже содержится начальное значение `nil`.

Далее нам нужно написать условие, по которому, если значение из Интернета не поступило, и `date` все еще равно `nil`, вывести соответствующее сообщение, а если поступило — вывести текущую дату:

```
var date: String?
date = "29.07.2015" // Получено из Интернета
if date != nil {
    print("Текущая дата: \(date)")
} else {
    print("Текущая дата не была получена")
}
```

Теперь, если мы попробуем симитировать сбой получения значения и закомментируем в этом примере вторую строку, то программа выведет нам сообщение: "Текущая дата не была получена".

Если дата все же получена, и на второй строке переменной `date` присваивается значение, отличное от `nil`, то программа напечатает нам "Текущая дата: Optional("29.07.2015")". Как мы видим, вместо привычной записи "Текущая дата: 29.07.2015" нам печатается значение в скобках и с префиксом `Optional`. Это означает, что тип значения, который содержится теперь в переменной `date`, не простой `String`, а опциональный.

Мы это можем также проверить, если вызовем какой-нибудь стандартный метод для типа `String`, — например, `extend(_:)`, который добавляет в конец строку либо символ:

```
var date: String?
date = "29.07.2015" // Получено из Интернета
date.extend("г.") // Ошибка
```

Мы попытались добавить строку "г." в конец строки "29.07.2015", но компилятор вывел нам ошибку. Дело в том, что он не знает метода `extend(_:)` для типа `String?` (опциональный `String`). Он знает лишь, что такой метод есть у типа `String`. А это означает, что опциональный `String?` и простой `String` кардинально отличаются друг от друга.

Ну, а для того чтобы мы могли работать со значением из переменной `date` как с обычной строкой, нам нужно извлечь это значение из опционального типа. В следующих разделах мы рассмотрим несколько приемов извлечения опциональных типов.

8.1. Опциональная привязка

Одним из способов извлечения опциональных типов является конструкция *опциональной привязки* (optional binding). В ней мы пишем условное выражение, которое извлекает значение из опционального типа и привязывает его к константе. Для того чтобы написать опциональную привязку, нам нужно воспользоваться выражением `if let`:

```
if let unwrappedDate = date {  
    print("Текущая дата \(unwrappedDate)")  
}
```

Эта конструкция проверяет: если в переменной `date` содержится опциональное значение, то оно будет извлечено и присвоено константе `unwrappedDate`. Если же в переменной содержится `nil`, то условие не выполнится.

Применим теперь эту условную конструкцию к нашему примеру:

```
var date: String?  
date = "29.07.2015" // Получено из Интернета  
if let unwrappedDate = date {  
    print("Текущая дата: \(unwrappedDate)")  
} else {  
    print("Текущая дата не была получена")  
}  
// Напечатает "Текущая дата: 29.07.2015"
```

Программа выведет на экран то сообщение, которое мы и хотели: "Текущая дата: 29.07.2015". А если мы прокомментируем вторую строку, то программа выведет "Текущая дата не была получена".

Надо отметить, что при опциональной привязке мы также можем задействовать выражение `if var` вместо `if let`. Тогда извлеченное значение присвоится не константе, а переменной. Однако в официальной документации к языку Swift везде упоминается только `if let`. Тем самым нас очередной раз защищают от случайно изменяющихся значений и рекомендуют использование констант, а не переменных. Мы же, в свою очередь, в книге тоже будем использовать в опциональной привязке только константы.

8.2. Принудительное извлечение

Еще одним способом извлечения опциональных типов является *принудительное извлечение* (forced unwrapping). К нему следует прибегать в тех случаях, когда вы точно знаете, что значение не равно `nil`, иначе будет ошибка. Для того чтобы принудительно извлечь значение из опционального типа, нужно поставить у него в конце восклицательный знак: `!`.

Перепишем наш предыдущий пример и применим принудительное извлечение:

```
var date: String?
date = "29.07.2015"
print("Текущая дата: \(date!)")
// Напечатает "Текущая дата: 29.07.2015"
```

Но, как уже было сказано, принудительное извлечение нужно использовать, только если мы точно уверены, что значение не равно `nil`. Поэтому если мы закомментируем здесь вторую строку, то получим ошибку. И чтобы исключить такую ошибку, нам, все же, придется добавить условное выражение:

```
var date: String?
date = "29.07.2015" // Получено из Интернета
if date != nil {
    print("Текущая дата: \(date!)")
} else {
    print("Текущая дата не была получена")
}
// Напечатает "Текущая дата: 29.07.2015"
```

Как можно видеть, принудительное извлечение — более быстрый способ извлечения опционального типа, но оно не всегда является безошибочным.

8.3. Неявное извлечение

Еще одним способом решения нашей проблемы является *неявное извлечение* (*implicit unwrapping*). Оно отличается от предыдущих способов. С помощью неявного извлечения мы можем сразу указать, что переменная имеет опциональный тип, но ее значение не нужно каждый раз извлекать. Для этого при объявлении переменной после названия типа нам нужно поставить не вопросительный знак `?`, а восклицательный `!`:

```
var date: String!
```

Перепишем наш пример с применением неявного извлечения:

```
var date: String!
date = "29.07.2015" // Получено из Интернета
if date != nil {
    print("Текущая дата: \(date)")
} else {
    print("Текущая дата не была получена")
}
```

Как можно видеть, мы избавились от восклицательного знака при использовании переменной `date`. Но теперь ставим восклицательный знак после названия типа.

8.4. Опциональное сцепление

Следующей важной конструкцией из темы опциональных типов является *опциональное сцепление*. В начале главы мы не смогли воспользоваться для опционального типа методом `extend(_)`. Нам пришлось сначала извлекать значение, прибегнув к опциональной привязке `if let`, чтобы затем иметь возможность добавить в конец извлеченной строки строку или символ. Но для этого есть более удобный способ.

С помощью опционального сцепления мы можем быстро извлечь значение и применить любой метод, который доступен извлеченному типу. Для этого после опциональной переменной мы ставим вопросительный знак `?`, а затем пишем нужный метод:

```
var date: String?  
date = "29.07.2015"  
var fullDate = date?.extend("г.")
```

Здесь на третьей строке мы одновременно и извлекли опциональное значение, и вызвали для него (уже значения типа `String`) метод `extend(_)`, который добавляет строку или символ в конец строки.

Опциональное сцепление очень полезно при передаче значения методу либо любому другому объекту, который будет писаться через точку.

Выводы

- ❑ Опциональные типы помогают нам решить ситуации, когда значение может отсутствовать и равняться `nil`.
- ❑ Чтобы получить значение из опционального типа, его нужно извлечь.
- ❑ Тип? — объявление опционала.
- ❑ Тип! — неявное извлечение.
- ❑ Объект! — принудительное извлечение.
- ❑ Объект? — опциональное сцепление.

ГЛАВА 9



Кортежи

Кортежи в Swift позволяют группировать значения в одно составное значение. В отличие от массивов, элементами кортежа могут быть значения любых типов, даже сами кортежи.

9.1. Объявление кортежа

Для создания кортежа достаточно поместить несколько значений через запятую в круглые скобки:

```
var result = (200, "OK", true)
```

В этом примере `result` — это кортеж, который хранит в себе число 200, строку "OK" и логическое значение `true`. Для того чтобы объявить кортеж без присвоения ему значений, можно просто поместить типы значений в круглые скобки:

```
var result: (Int, String, Bool)
result = (200, "OK", true)
```

9.2. Получение доступа к элементам кортежа

Существуют два способа доступа к элементам кортежа: через указание индекса элемента после имени переменной кортежа и через разложение кортежа. Рассмотрим эти два способа.

9.2.1. Использование индекса элемента

Если мы хотим получить доступ к какому-либо элементу кортежа, то можем написать его индекс через точку после названия переменной или константы (отсчет индекса начинается с нуля):

```
var result = (200, "OK", true)
result.0 // => 200
```

Поскольку кортеж мы объявили как переменную, мы также можем менять значение элементов кортежа следующим способом:

```
var result = (200, "OK", true)
result.0 // => 200
result.0 = 400
result // => (400, "OK", true)
```

Соответственно, если мы бы объявили `result` как константу, то не смогли бы поменять значения его элементов.

9.2.2. Разложение кортежа

Следующий способ получения доступа к элементам кортежа — его разложение:

```
var result = (200, "OK", true)
var (statusCode, statusMessage, hasBody) = result
```

Здесь мы в первой строчке создали кортеж с тремя элементами, а во второй строчке написали, что хотим разложить элементы кортежа `result` по трем переменным: `statusCode`, `statusMessage` и `hasBody`. Теперь переменным `statusCode`, `statusMessage` и `hasBody` будут присвоены значения 200, "OK" и `true` соответственно.

То же самое мы могли бы написать и в виде констант, например:

```
let (statusCode, statusMessage, hasBody) = result
```

Тогда значения `statusCode`, `statusMessage` и `hasBody` нельзя было бы изменить.

Если по какой-либо причине мы не хотим получать доступ к тем или иным элементам, то можем их исключить с помощью знака нижнего подчеркивания `_`:

```
let (_, _, hasBody) = result
```

9.3. Именованние элементов кортежа

Элементам кортежа в Swift можно давать имена. То есть, приведенный ранее пример кортежа мы могли бы написать так:

```
var result = (statusCode: 200, statusMessage: "OK", hasBody: true)
```

В этом случае мы бы могли получать доступ к элементам через указание имени элемента, а не индекса:

```
result.statusCode // => 200
```

Такая конструкция очень похожа на то, что `result` как бы является объектом некоего класса, а `statusCode` — свойство этого класса. Да, действительно, мы обладаем здесь теми же возможностями, что и с использованием свойств классов. Однако Apple рекомендует в таких случаях задумываться о масштабе объекта. Например, если мы хотим создать какой-нибудь временный объект, то лучше всего использовать кортеж с его именованными параметрами. А если нам нужен большой объект

с многочисленными свойствами и методами, то лучше использовать классы или структуры.

9.4. Использование кортежей

9.4.1. Массовое присвоение

Поскольку кортежи созданы для того, чтобы группировать значения, то они полезны в любых местах, где те или иные значения требуется сгруппировать. Первый и самый простой способ их использования — *массовое присвоение*. Например, выражение:

```
var a = 1
var b = 2
var c = 3
```

мы можем записать в одну строку:

```
var (a, b, c) = (1, 2, 3)
```

9.4.2. В циклах *for-in*

Еще одно применение кортежей — в циклах *for-in*. Например, пусть у нас есть словарь `cityTemp`, в котором содержатся значения температуры в нескольких городах:

```
var cityTemp = ["Moscow": 6, "NewYork": 4, "London": 2, "Los Angeles": 22]
```

Мы можем пройтись по элементам этого словаря с помощью цикла *for-in*:

```
var cityTemp = ["Moscow": 6, "NewYork": 4, "London": 2, "Los Angeles": 22]
```

```
for (cityName, temp) in cityTemp {
    print("Температура в городе \(cityName) сейчас равняется \(temp) градусам
        Цельсия")
}
// "Температура в городе Moscow сейчас равняется 6 градусам Цельсия"
// "Температура в городе NewYork сейчас равняется 4 градусам Цельсия"
// "Температура в городе London сейчас равняется 2 градусам Цельсия"
// "Температура в городе Los Angeles сейчас равняется 22 градусам Цельсия"
```

Такой код печатает нам температуру в каждом городе.

9.4.3. В качестве возвращаемого значения для функций

Кортежи можно также использовать как возвращаемое значение для функций. Объявить такую функцию можно так:

```
func getTempInSaratov() -> ( String , Int ) {
    return ("Saratov", 13)
}
```

Здесь мы просто указали, что возвращаемым значением функции будет кортеж с элементами типа `String` и `Int`. Если функция возвращает кортеж, то мы его можем разложить:

```
let (city,temp) = getTempInSaratov()
print("Температура в городе \(city) составляет \(temp) градусов Цельсия")
```

В другом случае, мы можем присвоить результат функции переменной:

```
var tempInSaratov = getTempInSaratov()
```

Теперь давайте сделаем элементы кортежа именованными. Для этого нам просто нужно элементам возвращаемых значений функции приписать имена:

```
func getTempInSaratov() -> (city: String , temp: Int ) {
    return ("Saratov", 13)
}
var tempInSaratov = getTempInSaratov()
print("Температура в городе \(tempInSaratov.city) составляет
\ (tempInSaratov.temp) градусов Цельсия")
// "Температура в городе Saratov составляет 13 градусов Цельсия"
```

9.5. Опциональный кортеж

Если существует вероятность, что вместо кортежа функция может вернуть пустое значение, то лучше всего объявить возвращаемое значение как *опциональный кортеж*. Опциональный кортеж пишется как обычный, только в конце его ставится знак вопроса:

```
(city: String , temp: Int )?
```

Следует заметить, что опциональный кортеж — это не то же самое, что кортеж из опциональных значений: `(city: String? , temp: Int?)`.

Давайте изменим функцию из предыдущего примера и запишем там опциональный кортеж в типе результата функции:

```
func getTempInSaratov() -> (city: String , temp: Int )? {
    return ("Saratov", 13)
}
var tempInSaratov = getTempInSaratov()!
print("Температура в городе \(tempInSaratov.city) составляет
\ (tempInSaratov.temp) градусов Цельсия")
// "Температура в городе Saratov составляет 13 градусов Цельсия"
```

В этом примере не достаточно было указать тип возвращаемого значения функции как опциональный кортеж. Нужно еще извлечь значение из этого кортежа. Мы здесь применили принудительное извлечение результата функции, поставив восклицательный знак после функции `getTempInSaratov() !`.

Выводы

- ❑ Кортежи в Swift позволяют группировать значения в одно составное значение.
- ❑ В отличие от коллекций, элементы кортежа могут иметь любой тип.
- ❑ Получить доступ к элементам кортежа можно либо указав через точку индекс `result.0`, либо с помощью разложения кортежа.
- ❑ Элементы кортежа могут иметь названия, тогда при обращении к ним можно употреблять названия, а не индексы.
- ❑ Кортежи в основном используются для возвращения группы значений функцией.
- ❑ Мы можем объявить кортеж как опциональный, если поставим в его конце знак вопроса: `(city: String , temp: Int)?`.

ГЛАВА 10



Замыкания

Замыкания — это самостоятельные блоки кода с определенной функциональностью. Если вы программировали на других языках программирования, то замыкания в Swift — это то же самое, что и блоки в Objective-C или `lambda` в Python и Ruby.

Попробуем написать простое замыкание:

```
{  
    print("Hello World")  
}
```

Этот блок кода и есть простейшее замыкание. Но само по себе оно нигде применяться не может, т. к. у нее нет ни названия, ни какого-либо обозначения, с помощью которого мы бы смогли на него сослаться. Именно поэтому, просто написав такой блок в Xcode, мы получим ошибку.

Чтобы не было ошибки, мы можем присвоить замыкание какой-нибудь переменной:

```
var sayHello = {  
    print("Hello World")  
}
```

Теперь Xcode не будет выдавать ошибок, и мы можем вызвать это замыкание, просто написав название этой переменной с последующими скобками:

```
sayHello() // -> "Hello World"
```

Замыкание, так же как и функция, может принимать параметры и возвращать значение. Для того чтобы написать параметры, нам нужно использовать особую конструкцию: поскольку границы нашего замыкания ограничиваются строками кода внутри фигурных скобок, то мы не можем писать параметры и тип результата за их пределами, — соответственно, их нужно писать внутри:

```
var sayHello = { () -> Void in  
    print("Hello World")  
}
```

Здесь мы привели пример замыкания без параметров и без результата. Это можно понять из конструкции `() -> Void`, знакомой нам из функций. Но следует обратить внимание, что после этой конструкции нужно писать ключевое слово `in`.

Если мы хотим написать параметры и тип результата, мы делаем это аналогично тому, как мы поступали с функциями:

```
var sayHello = { (name: String) -> String in
    return "Hello \(name)"
}
```

Здесь мы написали замыкание с параметром `name` и типом результата `String`. Нам также пришлось исключить функцию `print()` из тела функции и добавить `return`, т. к. мы уже указали, что замыкание имеет результат, и его тип `String`. Теперь мы сможем вызвать замыкание с параметром:

```
sayHello("David")
// => "Hello David"
```

Так же, как и с простыми типами, мы можем объявление замыкания разбить на две части: сначала указание переменной типа замыкания, а потом присваивание этой переменной блока кода. Для того чтобы указать переменной тип замыкания, необходимо написать тип переменной `() -> Void`:

```
var sayHello: () -> Void
sayHello = {
    print("Hello")
}
```

Эта конструкция говорит, что теперь переменная `sayHello` имеет тип замыкания, не принимает никаких параметров и ничего не возвращает.

Вы, наверное, заметили, что замыкания очень похожи на функции. И это не просто так. Грубо говоря, функции — это частный случай замыканий. Их еще называют безымянными функциями.

Давайте более детально на примере рассмотрим, где нам могут пригодиться замыкания. Одно из частых применений замыканий — это передача их в виде параметра функции. Поэтому напишем функцию-калькулятор, которая будет принимать три параметра: первый и второй — операнды выражения, а третий — выражение, которое мы хотим к ним применить. Допустим, это будут простые выражения сложения, вычитания, умножения и деления. Чтобы мы могли передавать функции определенные действия, мы должны объявить третий параметр функции как замыкание:

```
func calculate( a: Int, _ b: Int, _ calculateOperator: (firstNum:
Int, secondNum: Int) -> (Int) ) -> Int {

}
```

Если абстрагироваться от понятия замыканий, то можно сделать очень интересный вывод. Например, когда мы писали параметры функции в виде простых перемен-

ных, мы передавали телу функции значения и совершали над ними какие-либо действия. А теперь мы передаем функции замыкание. Это означает, что мы как бы передаем функции действие, которое нужно совершить внутри этой функции. Такой подход весьма полезен, и вы убедитесь в этом, когда до конца поймете роль замыканий в языках программирования.

Итак, мы написали первую строку объявления функции и теперь должны написать логику самой функции, т. е. ее тело. Давайте подумаем, как это сделать так, чтобы функция выполняла определенную операцию над двумя числами. Для этого нам нужно вызвать замыкание, которое мы записали в третьем параметре, и передать ему наши операнды выражения в виде параметров:

```
func calculate( a: Int, _ b: Int, _ calculateOperator: (firstNum:
Int, secondNum: Int) -> Int) -> Int {
    return calculateOperator(firstNum: a, secondNum: b)
}
```

Мы изначально не знаем, какие операции будут выполняться над операндами. Эти операции можно будет написать в третьем параметре функции `calculate` при ее вызове. То есть, до вызова функции `calculate` нам нужно сначала иметь замыкание с двумя операторами, которое выполняет какое-либо действие. И лишь затем передать это замыкание нашей функции `calculate`.

Для примера создадим простейшее замыкание, которое складывает два собственных параметра вместе:

```
var addition = { (num1: Int, num2: Int) -> Int in
    return num1 + num2
}
```

А теперь попробуем вызвать нашу функцию, передав ей два определенных числа в виде двух первых параметров и замыкание `addition` в виде третьего:

```
calculate(2,3,addition) // => 5
```

Как мы и предполагали, функция сложила два операнда и выдала результат: 5.

Поскольку третий параметр функции — это замыкание, мы можем написать туда не переменную, в которой содержится замыкание, а сам блок замыкания:

```
calculate(2,3,{ (num1: Int, num2: Int) -> Int in return num1 + num2 }) // => 5
```

Поскольку в объявлении функции мы указали, что третьим параметром функции является замыкание с двумя параметрами типа `Int` и типом результата `Int`, то мы здесь уже можем не писать типы параметров и тип результата, — это за нас сделает вывод типов:

```
calculate(2,3,{ num1, num2 in return num1 + num2 }) // => 5
```

Если замыкание имеет всего лишь одно выражение, то ключевое слово `return` можно опускать:

```
calculate(2,3,{ num1, num2 in num1 + num2 }) // => 5
```

10.1. Сокращенные имена параметров замыкания

В Swift есть возможность не давать параметрам замыкания определенные имена. Это означает, что мы можем при объявлении замыкания указать только тип параметров. Но как же потом обращаться к параметрам этого замыкания в теле функции?

Для обозначения параметров замыканий в теле функции в Swift существуют *сокращенные имена параметров*. Они начинаются от `$0`, следующий параметр `$1` и далее по порядку. Попробуем теперь еще больше сократить наш вызов функции `calculate`:

```
calculate(2,3,{ $0 + $1 }) // => 5
```

Это замыкание просто и лаконично складывает первый параметр со вторым, но при этом здесь работает вывод типов. Исходя из объявления функции, которое мы писали ранее, вывод типов подставляет тип `Int` параметрам `$0` и `$1` и аналогичный тип `Int` результату замыкания.

10.2. Операторы-функции

Забегая вперед, можно сказать, что мы можем еще больше сократить наше замыкание. Дело в том, что некоторые базовые типы в Swift имеют *операторы-функции* — такие как, например, бинарный плюс (бинарный `+`). Так, для двух значений `Int` в операторе-функции бинарный плюс прописано, что нужно сложить вместе эти два числа. Операторы-функции будут рассмотрены подробнее в *главе 20*, а пока нам достаточно знать, что бинарный плюс — это функция, принимающая два параметра типа `Int` и складывающая их друг с другом. Эта функция полностью соответствует третьему параметру функции `calculate`, т. к. он тоже принимает замыкание, принимающее два параметра `Int`. Все это приводит к тому, что третьим параметром в функции `calculate` мы можем писать просто бинарный `+`:

```
calculate(2,3, + ) // => 5
```

10.3. Последующее замыкание

Если функция имеет параметр, который принимает замыкание, и это замыкание пишется как последний параметр, то можно использовать выражение *последующего замыкания* (trailing closure).

Когда мы вызываем функцию, которая принимает как параметр замыкание, то этот блок замыкания мы можем писать не внутри круглых скобок, где обычно и пишутся параметры, а после них:

```
func someFunction(closure: () -> Void ) {  
    // Некоторые выражения функции  
}
```

```
// обычный способ
someFunction({
    // Много строк выражения замыкания
})

// последующее замыкание
someFunction() {
    // Много строк выражения замыкания
}
```

Здесь мы в первом выражении объявили функцию `someFunction`, которая принимает один параметр — замыкание `closure`. Во втором выражении мы вызвали функцию обычным способом — так, как будто мы не знали о последующих замыканиях. В третьем выражении мы использовали последующее замыкание.

Более того, если функция не принимает более никаких параметров, кроме замыкания, то круглые скобки после названия функции можно опустить:

```
// последующее замыкание, функция без круглых скобок
someFunction {
    // Много строк выражения замыкания
}
```

Вызов функции с последующим замыканием на первый взгляд похож на объявление функции. Это сильно путает. Здесь нужно смотреть на ключевое слово `func` в начале выражения. Если это ключевое слово есть, то, значит, — это простое объявление функции, а если нет, то это последующее замыкание.

Применение последующего замыкания полезно, когда замыкание пишется на несколько строк. Тогда код будет смотреться более лаконично.

Рассмотрим последующие замыкания на конкретном примере и создадим функцию, которая станет повторять некоторые действия определенное количество раз. Первым параметром этой функции будет количество повторений (`times`), а вторым — это замыкание (`closure`). Назовем эту функцию `repeatTask`:

```
func repeatTask(times: Int, closure: () -> Void) {
    for _ in 1..times {
        closure()
    }
}
```

Мы использовали цикл от одного до `times`, чтобы выполнить код определенное количество раз. Так как нам не нужно использовать текущее значение внутри тела цикла, то мы теперь можем вызвать нашу функцию и передать ей, например, чтобы она написала три раза "Внимание!".

```
repeatTask(3) {
    print("Внимание!")
}
```

```
// Внимание!  
// Внимание!  
// Внимание!
```

Выводы

- ❑ Замыкания — это самостоятельные блоки кода с определенной функциональностью.
- ❑ Функции — это частный случай замыкания.
- ❑ Замыкание, так же как и функция, может принимать параметры и возвращать значение.
- ❑ Для параметров замыкания можно писать сокращенные имена: \$1, \$2 и т. д.
- ❑ Если функция имеет параметр, который принимает замыкание, и это замыкание пишется как последний параметр, то можно использовать выражение последующего замыкания.

ГЛАВА 11



Перечисления

Давайте представим, что у нас есть игра, в которой мы хотим сделать систему достижений. Например, игрок может выиграть золотую, серебряную или бронзовую медаль. Создадим переменную `playerAchievement` и станем присваивать ей наши медали. Мы можем, например, обозначать эти медали в виде цифр — скажем, золотая медаль — это 1, серебряная — 2, а бронзовая — 3. Но результаты игры, объявленные в таком виде, окажутся нам непонятны — увидев, что переменная `playerAchievement` равна 3, мы должны будем самостоятельно вспомнить, какая это медаль. Либо нам придется искать в своих записях, какая медаль относится к какому числу.

Хорошим решением было бы присваивать значения переменной `playerAchievement` в виде строк. Так, если игрок выиграл золото, то значение переменной `playerAchievement` для него равно "Gold" и т. д.:

```
var playerAchievement = "Gold"
```

Но в этом способе тоже есть подводные камни. Например, кто-то может присвоить переменной `playerAchievement` значение "Platinum", хотя у нас нет платиновых медалей. Мы рассматриваем очень простой пример, но для программы это может закончиться серьезной ошибкой.

В чем же проблема? А заключается она в том, что мы никак не ограничиваем возможные значения. Наше присваивание строк с определенным значением никак не задает границы возможных значений. Так что, нам нужна возможность объявить группу связанных значений и ограничить их количество. Например, для нашего случая, было бы хорошо создать группу `Medal` и включить в нее три элемента "Gold", "Silver" и "Bronse". Именно в этом нам и поможет *перечисления*.

Перечисления создают определенный тип для группы связанных значений и позволяют использовать их в коде более безопасным и ограничивающим способом.

11.1. Объявление перечислений

Перечисления объявляются через ключевое слово `enum`. Далее после названия перечисления открываются квадратные скобки, в которых пишутся элементы перечисления через ключевое слово `case`:

```
enum Medal {  
    case Gold  
    case Silver  
    case Bronze  
}
```

Теперь мы уже можем назначать переменной ограниченные значения: либо `Gold` (Золото), либо `Silver` (Серебро), либо `Bronze` (Бронза). И никакое другое значение ей присвоено быть не может. Но для этого мы сначала должны указать, что переменная `playerAchievement` имеет тип `Medal`:

```
var playerAchievement: Medal
```

Теперь переменной можно присвоить только те три значения, которые мы задали в перечислении. Давайте присвоим ей золото:

```
playerAchievement = Medal.Gold
```

Если же игрок получил серебро, то мы можем присвоить его так:

```
player2Achievement = Medal.Silver
```

Если уже известно, что переменная имеет тип `Medal`, то при присвоении название типа можно опустить и писать только `.Silver`:

```
player2Achievement = .Silver
```

Есть возможность записать это более компактным способом. Перечисления могут применять вывод типов, так что мы можем написать такое присвоение в одну строку — без объявления типа:

```
var playerAchievement = Medal.Bronze
```

Однако при первом присвоении указывать название перечисления `Medal` необходимо. С его помощью компилятор определяет, к какому перечислению относится `.Bronze`. А при присвоении значения во второй раз и в последующие мы можем опустить название перечисления `Medal`.

11.2. Перечисления и оператор *switch*

Одним из наиболее удобных способов сопоставления значения переменной и перечисления является условное выражение с оператором `switch`:

```
var playerAchievement = Medal.Gold
```

```
switch playerAchievement {
```



```
case .Gold:
    print("Поздравляем! Вы заняли первое место и выиграли золотую медаль")
case .Silver:
    print("Поздравляем! Вы заняли второе место и выиграли серебряную медаль")
case .Bronze:
    print("Поздравляем! Вы заняли третье место и выиграли бронзовую медаль")
}
```

Нам следует всегда помнить одно важное свойство оператора `switch` — варианты должны быть исчерпывающими. Для перечислений это означает, что мы должны рассматривать все элементы перечисления как варианты условного выражения `switch`. Если мы не хотим писать все элементы перечисления, то должны указать вариант `default`, под который будут попадать все остальные значения:

```
var playerAchievement = Medal.Gold

switch playerAchievement {
case .Gold:
    print("Поздравляем! Вы заняли первое место и выиграли золотую медаль")
default:
    print("Вы не получили золотую медаль")
}
```

11.3. Связанные значения

В предыдущем примере мы рассмотрели перечисление, с помощью которого можно присваивать медали нашим игрокам. Но иногда нам нужно быть более точными. Например, представим, что мы хотим вместе с видом медали указать еще и количество очков, которые получил игрок. Для этого при объявлении перечисления можно задать *связанные значения*:

```
enum Medal {
    case Gold(Int)
    case Silver(Int)
    case Bronze(Int)
}
```

Здесь мы прописали скобки после элементов перечисления и указали для связанного значения тип `Int`. А поскольку мы хотим в связанном значении писать количество очков, то объявили тип `Int`, хотя можно использовать любой другой тип. Количество связанных значений для каждого элемента перечисления тоже может быть разным.

Теперь давайте попробуем присвоить переменной значение перечисления со связанным значением. Например, пусть это будет золотая медаль и 68 очков:

```
var playerAchievement = Medal.Gold(68)
```

Теперь у нас выдается не просто сухое значение `Gold`, а добавилось количество выигранных очков. Чтобы повысить наглядность примера, давайте для элемента пере-

числения `Gold` добавим еще одно связанное значение типа `String`, в котором мы будем указывать название подарка, который получил обладатель золотой медали:

```
enum Medal {
    case Gold(Int, String)
    case Silver(Int)
    case Bronze(Int)
}

var playerAchievement = Medal.Gold(68, "Автомобиль")
```

Условный оператор `switch` очень хорошо сочетается со связанными значениями перечисления. Мы можем присвоить переменные связанным значениям и использовать их в теле условия:

```
switch playerAchievement {
case .Gold(var score, var present):
    print("Поздравляем! Вы заняли первое место и выиграли золотую медаль. Вы набрали \(score) очков и получите в подарок \(present).")
case .Silver(var score):
    print("Поздравляем! Вы заняли второе место и выиграли серебряную медаль. Вы набрали \(score) очков.")
case .Bronze(var score):
    print("Поздравляем! Вы заняли третье место и выиграли бронзовую медаль. Вы набрали \(score) очков.")
}

// Напечатает "Поздравляем! Вы заняли первое место и выиграли золотую медаль. Вы набрали 68 очков и получите в подарок Автомобиль."
```

Если нужно, чтобы значения переменных не менялись в теле условия, то их можно объявить в виде констант, поменяв ключевое слово `var` на `let`:

```
case .Gold(let score, let present):
    print("Поздравляем! Вы заняли первое место и выиграли золотую медаль. Вы набрали \(score) очков и получите в подарок \(present).")
case .Silver(let score):
    print("Поздравляем! Вы заняли второе место и выиграли серебряную медаль. Вы набрали \(score) очков.")
case .Bronze(let score):
    print("Поздравляем! Вы заняли третье место и выиграли бронзовую медаль. Вы набрали \(score) очков.")
}

// Напечатает "Поздравляем! Вы заняли первое место и выиграли золотую медаль. Вы набрали 68 очков и получите в подарок Автомобиль."
```

11.4. Исходные значения

Кроме связанных значений, перечисления также имеют *исходные значения*. Понятие исходных значений во многом отличается от понятия связанных значений. Исходное значение — это внутреннее значение каждого элемента перечисления.

В отличие от связанного значения, каждый элемент перечисления имеет только одно исходное значение, но тип этого исходного значения одинаков для всех элементов.

Давайте рассмотрим перечисление `DayOfWeek`, в котором пишутся дни недели:

```
enum DayOfWeek: Int
{
    case Monday = 1
    case Tuesday = 2
    case Wednesday = 3
    case Thursday = 4
    case Friday = 5
    case Saturday = 6
    case Sunday = 7
}
```

Обратите внимание на первую строчку. В ней мы указали для перечисления тип `Int`. Это означает, что исходные значения этого перечисления имеют тип `Int`. Далее на следующих строчках мы каждому элементу перечисления присваиваем исходное значение, которое является номером дня недели.

Если исходное значение перечисления имеет тип `Int` как у нас в примере, то это значение можно указывать только для первого элемента перечисления, а всем остальным значения будут присвоены автоматически по возрастающей:

```
enum DayOfWeek: Int
{
    case Monday = 1
    case Tuesday
    case Wednesday
    case Thursday
    case Friday
    case Saturday
    case Sunday
}
```

В этом примере дням недели автоматически присвоены те же исходные значения, что и в предыдущем.

Приведенное правило действует только для исходных значений с типом `Int`. Для всех остальных типов исходные значения обязательно указывать для каждого элемента.

Получить исходное значение из перечисления можно через свойство `rawValue`:

```
var today = DayOfWeek.Tuesday.rawValue // => 3
```

И наоборот, можно получить элемент перечисления, указав только исходное значение:

```
var today = DayOfWeek(rawValue: 3)
// today теперь имеет тип DayOfWeek? и равен dayOfWeek.Tuesday
```

Подобно получению значения из словаря, здесь мы получаем не простое значение, а опциональное, т. к. есть риск, что элемента с таким исходным значением, которое мы запрашиваем, не существует. Давайте попробуем вытащить значение через указание `rawValue` с применением опциональной привязки:

```
let position = 3
if let today = DayOfWeek(rawValue: position) {
    switch today {
        case .Friday:
            print("Наконец-то пятница")
        default:
            print("Обычный день недели")
    }
} else {
    print("Элемента с исходным значением \(position) не существует")
}
// Напечатает "Обычный день недели"
```

11.5. Вложенные перечисления

У перечислений есть одно интересное свойство — перечисления могут содержать в себе перечисления, т. е. перечисления могут быть вложенными:

```
enum DayOfWeek: Int
{
    case Monday
    case Tuesday
    case Wednesday
    case Thursday
    enum Friday {
        case Day
        case Night
    }
    case Saturday
    case Sunday
}
```

Здесь мы вместо элемента `Friday` написали перечисление `Friday`, который имеет свои собственные элементы.

Присвоить элементы вложенного перечисления так же легко, как и присвоить элементы простого перечисления:

```
var today = DayOfWeek.Friday.Day
```

Мы просто сначала пишем название родительского перечисления, потом название вложенного перечисления, а затем только название элемента вложенного перечисления.

Выводы

- ❑ Перечисления создают определенный тип для группы связанных значений.
- ❑ Перечисления объявляются через ключевое слово `enum`, а элементы перечисления — через ключевое слово `case`.
- ❑ Если тип переменной соответствует типу перечисления, то при присвоении значения название перечисления можно опустить: `.Silver`.
- ❑ При объявлении перечисления можно указать связанные значения для каждого элемента. Их тип и количество могут быть разными для каждого элемента.
- ❑ Элементы перечисления могут иметь исходные значения. Тип этих значений общий для всего перечисления, а каждый элемент может иметь только одно исходное значение.
- ❑ Перечисления могут быть вложенными.

ГЛАВА 12



Классы

Большинство языков программирования, разработанных в последние пять лет, являются объектно-ориентированными, и Swift не исключение. Но это не всегда было так. Когда-то давным-давно люди программировали на процедурных языках программирования. Код на этих языках был очень длинным, т. к. он писался как одна большая процедура.

Чтобы этого избежать, придумали объектно-ориентированные языки программирования, которые состоят из самостоятельных объектов — *классов*, похожих на мини-программы. Каждый такой объект представляет собой некую сущность со своими свойствами и логикой, которые могут взаимодействовать между собой.

12.1. Свойства, методы и объекты класса

Классы могут иметь свойства и методы. *Свойства класса* — это его характеристики, которые тем или иным способом описывают объект. Таким образом, если мы создаем класс Кошка, то свойствами этого класса могут быть «цвет», «имя» и т. д. Обычно в свойствах объекта пишут характеристики, с которыми нам нужно будет работать. *Методы класса* — это то, какими объект обладает действиями. Например, для кошки — это «мяукнуть», «прыгнуть» и т. д.

Понятия, которые нам нужно различать, — это классы и *объекты класса*. Класс сам по себе — это каркас, по которому создаются объекты класса. Например, если мы берем пример все того же класса Кошка, то объектом класса может быть определенный кот, например кот Васька. Его свойству «имя» присвоено значение Васька, а цвет у него черный. Он может мяукать и прыгать. Мы также можем создать еще одного кота, т. е. еще один объект класса Кошка. Например, с именем Том и цветом серый.

Чтобы быстрее понять это все, давайте приступим к изучению классов, свойств и методов в Swift.

12.2. Объявление классов

В Swift классы объявляются через ключевое слово `class`:

```
class Person {  
    // описание класса  
}
```

Мы объявили класс `Person`, который описывает некоего человека. Названия классов в Swift тоже пишутся в горбатой нотации, но только первую букву принято писать с заглавной буквы. Это помогает отличать классы от переменных, функций и т. д.

После объявления класса, подобно встроенным типам (`Int`, `String` и т. д.), мы можем указать переменной тип `Person`:

```
var jack: Person
```

Теперь переменная `jack` может хранить только объекты класса `Person`. Чтобы присвоить значение этой переменной, нам нужно сначала научиться создавать объекты для класса `Person`.

Чтобы создать объект определенного класса, нам нужно написать название класса и в конце приписать пустые круглые скобки:

```
Person()
```

Подобно вызову функции, такая конструкция создаст объект класса `Person` и вернет нам его. Чтобы иметь возможность работать с этим объектом, мы должны его присвоить нашей переменной с типом `Person`:

```
jack = Person()
```

Теперь переменная `jack` содержит объект класса `Person`. Подобно простым типам данных, мы могли предварительно не объявлять переменную с типом `Person`. Можно было все написать в одну строку и дать компилятору самому вывести тип из объекта, который мы ему передадим. Компилятор понимает, что мы присваиваем переменной объект определенного класса, и тогда может автоматически назначить переменной тип, соответствующий этому классу. Это означает, что предыдущий пример мы могли записать в одну строку так:

```
var jack = Person()
```

В этом разделе мы объявили пустой класс `Person` без свойств и методов и создали для него объект класса. В следующем разделе мы добавим свойства для нашего класса `Person`.

12.3. Свойства класса

В начале главы мы отмечали, что если класс описывает какой-либо объект из реального мира, то свойства класса — это характеристики описываемого объекта. На самом деле в Swift все проще, чем кажется на первый взгляд. Свойства классов — это просто переменные или константы, созданные внутри объявления класса. Если

мы напишем свойство через ключевое слово `var`, то свойство будет переменным, а если через `let`, то оно станет константой, и его значение изменить будет нельзя.

Для примера мы воспользуемся классом `Person`, который мы создали в предыдущем разделе. Давайте придадим ему свойства: `name` (имя) и `age` (возраст). Мы объявим свойство `name` как константу, т. к. обычно люди не меняют своего имени, а свойство `age` — как переменную, потому что с годами возраст человека меняется.

```
class Person {  
    let name: String  
    var age: Int  
}
```

Свойствам класса, подобно обычным переменным и константам, мы также должны указывать тип хранимого значения. Но, в отличие от переменных и констант, простого указания типа недостаточно. Для приведенного примера компилятор выдаст ошибку. Дело в том, что свойства класса, так же как и обычные константы или переменные, перед применением нуждаются в присвоении начального значения. То есть, для этих свойств нужно инициализировать начальное значение. Но только если с обычными переменными мы могли сначала только указать тип хранимого значения, а значение присвоить потом, то со свойствами класса такое не проходит, поскольку компилятор при создании объекта класса не будет знать, что присвоить его свойствам.

Мы могли поначалу подумать, что компилятор при создании объекта класса создаст пустые свойства, которым потом можно присвоить значение. Но как можно записать отсутствие значения в Swift? Правильно, присвоить переменной `nil`. Но так же как и с простыми переменными и константами, свойства должны быть опциональными, чтобы иметь возможность хранить `nil`.

И если мы укажем нашим свойствам опциональные типы, то ошибка пропадет:

```
class Person {  
    let name: String?  
    var age: Int?  
}
```

В таком случае компилятор не выдает ошибок и при создании объекта класса присвоит этим свойствам `nil`. Но в нашем примере мы рассмотрим более простой способ — чтобы ошибка исчезла, давайте просто присвоим свойствам начальные значения:

```
class Person {  
    let name = "Джек"  
    var age = 23  
}
```

Теперь любой созданный нами объект класса `Person` будет иметь начальное значение имени: "Джек", и возраст: 23. Мы можем проверить это, создав объект класса `Person`:

```
var jack = Person()
```


Если мы хотим получить значение какого-либо свойства объекта класса, то можем просто написать его через точку, после имени переменной, которой мы присвоили объект класса:

```
jack.name // => "Джек"
```

Аналогичным образом мы можем изменять значения свойства, просто присвоив другое значение свойству, написанному через точку:

```
jack.age = 20
```

Теперь значение свойства `age` для объекта класса `jack` равно 20. Обратите внимание, что мы и не пытались изменить значение свойства `name`. Как мы помним, мы объявили его как константу — соответственно значение этого свойства изменить нельзя.

У вас может возникнуть вопрос: а что будет, если мы присвоим объект класса не переменной, а константе? Вопрос интересный. Давайте для этого создадим новый объект класса и присвоим его константе:

```
let constantPerson = Person()
```

Когда мы присваиваем объект класса константе, то мы все равно можем изменять значения его свойств, которые объявлены как переменные. Но если свойство объявлено как константа, мы его изменить не сможем:

```
constantPerson.age = 20  
constantPerson.name = "Джон" // => Ошибка
```

12.3.1. Ленивые свойства

Свойства класса в Swift можно сделать *ленивыми*. Когда мы указываем, что свойство ленивое, то это свойство инициализируется только при обращении к нему.

Давайте посмотрим, как работает ленивое свойство на примере. Для этого создадим новый класс `Profession`, который будет описывать профессию нашего человека:

```
class Profession {  
    var name: String = "Цирюльник"  
}
```

Класс `Profession` имеет одно свойство `name`, в котором указывается название профессии.

Теперь давайте для нашего класса `Person` создадим ленивое свойство `profession`, в котором будет храниться объект класса `Profession`:

```
class Person {  
    let name = "Джек"  
    var age = 23  
    lazy var profession = Profession()  
}
```

Когда мы создадим объект класса `Person`, то произойдет инициализация только двух его первых свойств: `name` и `age`:

```
var jack = Person()  
// объект класса Profession не создан
```

Но как только мы захотим обратиться к свойству `profession`, то произойдет его инициализация, и ему будет присвоен объект класса `Profession`:

```
jack.profession.name // => "Цирюльник"  
// создан объект класса Profession и был присвоен свойству profession класса Person
```

Ленивые свойства очень полезны в таких местах, когда нужно экономить на обращении или получении какого-либо значения. Например, если у нас есть свойство, которое получает данные из Интернета, то было бы разумнее сделать это свойство ленивым, чтобы каждый раз при создании нового объекта класса не срабатывала функция получения данных из Интернета. То есть нам просто нужно, чтобы соединение с Интернетом происходило только, когда мы обращаемся к этому свойству.

12.3.2. Вычисляемые свойства

Иногда бывают случаи, когда нам нужно, чтобы одно из свойств имело не статическое значение, а хранило в себе результат какого-то действия. Например, третье свойство класса могло бы хранить результат сложения значений двух предыдущих свойств. И каждый раз при обращении к третьему свойству оно бы складывало значения двух первых свойств и выдавало бы нам результат. В Swift это можно сделать с помощью *вычисляемых свойств*.

Давайте рассмотрим соответствующий пример. Возьмем наш класс `Person` и добавим для него свойства `firstName` и `lastName`, в которых станем хранить имя и фамилию нашего человека. Затем создадим третье, вычисляемое, свойство `fullName`, которое будет выводить нам его полное имя:

```
class Person {  
    var firstName: String = "Тони"  
    var lastName: String = "Старк"  
    var fullName: String {  
        return firstName + " " + lastName  
    }  
}  
var tony = Person()  
tony.fullName // => "Тони Старк"
```

Здесь мы видим, что вычисляемые свойства пишутся как обычные свойства, только в конце они содержат блок, в котором хранят действия, которые нужно совершить при обращении к ним. Теперь каждый раз при обращении к свойству `fullName` оно будет соединять вместе имя и фамилию человека и передавать нам полученное значение.

Блок кода, который мы записали внутри фигурных скобок после вычисляемого свойства, является в Swift сокращенной записью блока `get` для вычисляемых свойств — он срабатывает, как только происходит обращение к свойству. Такой блок программисты называют «геттер». Мы можем переписать этот блок в более общей форме:

```
class Person {
    var firstName: String = "Тони"
    var lastName: String = "Старк"
    var fullName: String {
        get {
            return firstName + " " + lastName
        }
    }
}

var tony = Person()
tony.fullName // => "Тони Старк"
```

Мы добавили блок `get` внутри вычисляемого свойства `fullName`. И теперь знаем, что этот блок будет срабатывать каждый раз, как только мы захотим получить значение из вычисляемого свойства.

Но кроме блока `get` в вычисляемых свойствах можно писать еще и блок `set`, который будет срабатывать каждый раз, когда мы будем пытаться присвоить вычисляемому свойству значение. Его еще называют «сеттер».

Давайте напишем для нашего вычисляемого свойства `fullName` блок `set`. Представим, что вычисляемому свойству передают готовую строку с именем и фамилией, — например: "Брюс Уэйн". Тогда нам нужно взять эту строку и разбить на две строки: имя и фамилию. А затем присвоить эти две строки нашим свойствам `firstName` и `lastName`.

Чтобы выполнить эти действия, нам не обойтись без сторонней библиотеки. Здесь понадобится функция `componentsSeparatedByString()` из библиотеки `Foundation`, которая делит строку на части по разделителю, который мы ей передаем, и возвращает нам массив с разделенными строками. Функция принимает один параметр — в нем указывается символ, который разделяет строки. В качестве разделителя нам нужен пробел, поэтому в скобках этой функции мы укажем " ":

```
import Foundation

class Person {
    var firstName: String = "Тони"
    var lastName: String = "Старк"

    var fullName: String {
        get {
            return firstName + " " + lastName
        }
    }
}
```

```

        set(newFullName) {
            var separatedNames = newFullName.componentsSeparatedByString(" ")
            firstName = separatedNames[0]
            lastName = separatedNames[1]
        }
    }
}

var jack = Person()
jack.fullName = "Брюс Уэйн"

```

Блок `set` вычисляемого свойства принимает параметр, который отвечает за новое значение. То есть, как только мы присваиваем свойству `fullName` значение "Брюс Уэйн", это значение передается в параметр `newFullName` к блоку `set`. Для этого параметра мы можем использовать любое имя. Есть даже возможность вовсе не использовать в сеттере параметры. Для этого в Swift имеется зарезервированный параметр `newValue`, который можно записать в блоке `set` без указания параметров. Давайте перепишем блок `set` в сокращенной форме:

```

import Foundation

class Person {
    var firstName: String = "Тони"
    var lastName: String = "Старк"

    var fullName: String {
        get {
            return firstName + " " + lastName
        }
        set{
            var separatedNames = newValue.componentsSeparatedByString(" ")
            firstName = separatedNames[0]
            lastName = separatedNames[1]
        }
    }
}

var bruce = Person()
bruce.fullName = "Брюс Уэйн"

```

Здесь мы избавились от лишних скобок для блока `set`.

Теперь каждый раз, когда мы станем передавать нашему свойству `fullName` готовое значение из имени и фамилии, оно будет его разбивать на две части и присваивать свойствам класса `firstName` и `lastName`. Обратите внимание, что мы использовали переменные свойства `firstName` и `lastName`, иначе мы бы не смогли изменить значения этих свойств в сеттере.

Использование сеттера для вычисляемого свойства не обязательно, но геттер должен обязательно присутствовать. Вычисляемые свойства, в которых отсутствует сеттер, называют вычисляемыми свойствами *только для чтения*, потому что их значение можно лишь прочитать, но не изменить.

12.3.3. Наблюдатели свойств

В Swift есть очень полезная возможность слежки за изменением значения параметров. Для этого свойствам можно указать *наблюдатели* `willSet` и `didSet`. Наблюдатель `willSet` — это функция, которая срабатывает *до того*, как значение свойства изменится, а наблюдатель `didSet` — функция, которая срабатывает *после того*, как значение свойства изменилось.

Укажем наблюдателей свойству `lastName` в нашем классе `Person`:

```
class Person {
    var firstName: String = "Брюс"
    var lastName: String = "Уэйн" {
        willSet {
            print("Фамилия собирается измениться на \(newValue)")
        }
        didSet {
            print("Фамилия \(oldValue) изменилась на новую")
        }
    }
}

var bruce = Person()
bruce.lastName = "Ли"
// => "Фамилия собирается измениться на Ли"
// => "Фамилия Уэйн изменилась на новую"
```

Способ записи наблюдателей похож на вычисляемые свойства, только внутри блока свойства мы указываем блоки `willSet` и `didSet` вместо блоков `get` и `set`.

Подобно блоку `set`, у блоков `willSet` и `didSet` имеются свои зарезервированные параметры. Так, для `willSet` есть параметр `newValue`, который хранит новое значение свойства, на которое оно должно измениться. А для `didSet` есть параметр `oldValue`, который хранит старое значение свойства.

Наблюдатели свойства очень полезны в таких ситуациях, когда перед изменением значения или после него нужно выполнить какие-либо действия. Например, проверить значения на корректность, изменить другие свойства и т. д.

12.3.4. Вычисляемые переменные и наблюдатели для переменных

Познакомившись с вычисляемыми свойствами и наблюдателями, пора узнать еще одну интересную вещь. Все, что мы узнали для свойств класса, действует и для

глобальных и локальных переменных. *Локальные переменные* — это переменные, объявленные внутри функции, замыкания и т. д. А *глобальные переменные* — это переменные, объявленные вне каких-либо блоков.

Из сказанного следует, что мы можем создать глобальную вычисляемую переменную:

```
var a = 2
var b = 3

var c: Int {
    return a + b
}
print(c) // => 5
```

И даже назначить наблюдателей для глобальной переменной:

```
var c: Int = 7 {
    didSet {
        print("Значение хочет измениться на \(newValue)")
    }
}
c = 8
// => "Значение хочет измениться на 8"
```

Очень интересная особенность языка Swift, правда? Давайте двигаться дальше.

12.3.5. Свойства типа

Кроме объявления свойств объектов класса, в Swift есть возможность объявить свойства для самого класса, или, по-другому, *свойства типа*. То есть, например, обращение к свойству типа `Person` выглядело бы так:

```
Person.count
```

Обратите внимание, что мы обратились не к свойству объекта класса `Person`, а к самому классу `Person`. Свойства класса действительны и едины для всех объектов класса. С их помощью очень удобно хранить значения, которые должны быть доступны всем объектам класса. Например, в приведенном далее примере мы рассматриваем свойство `count` типа `Person`. С его помощью мы ведем подсчет созданных объектов класса `Person`.

Чтобы объявить свойство типа, нужно перед объявлением свойства написать ключевое слово `static`:

```
class Person {
    static var count = 0
    init() {
        count++
    }
}
```

В этом примере мы увеличиваем значение свойства `count` каждый раз, как только выполняется блок функции `init()`. Забегая вперед, скажем, что функция `init()` выполняется каждый раз, когда создается новый объект класса. То есть, теперь в свойстве `count` класса `Person` мы всегда будем иметь актуальное значение количества созданных объектов класса.

Свойства типа можно также объявить как вычисляемые. Они пишутся как обычные вычисляемые свойства, только перед их объявлением нужно написать ключевое слово `static`:

```
class Person {
    static var someComputedTypeProperty: Int {
        return 22
    }
}
```

Если мы хотим чтобы вычисляемое свойство могло быть переопределено, то вместо `static` следует использовать ключевое слово `class`:

```
class Person {
    static var someOverrideableComputedTypeProperty: Int {
        return 22
    }
}
```

12.4. Инициализаторы

Мы помним, что для классов существует правило, что свойства всегда должны быть инициализированы, — т. е. в них всегда должно присутствовать значение. Соответственно, значения свойствам класса мы присваивали напрямую. Но для этого есть более удобный способ — использование *инициализаторов*.

Инициализаторы — это специальные функции внутри класса, которые вызываются при создании нового объекта класса. Это означает, что мы можем задать определенные действия, которые должны выполняться при создании объекта класса.

Если у вас есть опыт разработки на других языках программирования, то вам ближе будет название *конструктор*. Но в Swift конструкторы называются инициализаторами.

12.4.1. Инициализатор по умолчанию

Присваивать начальные значения для свойств класса удобнее всего через инициализаторы. Давайте присвоим внутри инициализатора начальные значения для наших свойств класса `Person`. Инициализаторы пишутся с помощью ключевого слова `init()`:

```
class Person {
    var firstName: String
    var lastName: String
```

```
init() {  
    firstName = "Тони"  
    lastName = "Старк"  
}  
}
```

С точки зрения эффективности мы ничего особенного не сделали. Просто теперь присвоение новых значений станет происходить, как только будет создаваться новый объект класса.

Поскольку такие инициализаторы могут задавать только фиксированные значения, которые для всех объектов класса будут одинаковыми, их принято называть *инициализаторами по умолчанию*.

Даже когда мы не пишем инициализатор по умолчанию, а присваиваем значения каждому свойству напрямую, он все равно присутствует внутри любого объявления класса. Это следует вспомнить, когда мы станем изучать наследование классов.

12.4.2. Инициализаторы с параметрами

Было бы неплохо иметь возможность сразу создавать уникальные объекты класса со своими уникальными свойствами. Именно поэтому инициализаторы могут принимать параметры. В предыдущем примере мы написали инициализатор с пустыми скобками, т. е. без параметров. Но внутри этих скобок можно писать параметры, которые потом можно будет использовать при создании объекта класса. То есть если до этого при создании объекта класса мы писали `Person()` с пустыми скобками, то теперь мы можем писать там параметры, которые будут передаваться инициализатору `init()` с параметрами.

Давайте рассмотрим это на примере и напишем для инициализатора параметры `firstName` и `lastName`, которые будем присваивать соответствующим свойствам класса:

```
class Person {  
    var firstName: String  
    var lastName: String  
  
    init(firstName: String, lastName: String) {  
        firstName = firstName  
        lastName = lastName  
    }  
}  
// Ошибка
```

Написав таким образом присвоение свойств внутри инициализатора, мы получим ошибку. Проблема здесь в том, что параметры инициализатора имеют такие же названия, как и свойства класса. Компилятор не понимает, где мы указываем параметр инициализатора, а где — свойство класса. Для этого мы можем через свойство `self` явно указать, какой из них является свойством класса:


```
class Person {
    var firstName: String
    var lastName: String

    init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }
}
```

Свойство `self` указывает на сам класс, в котором он написан. То есть, указав `self.firstName`, мы тем самым обратились к собственному свойству `firstName` класса `Person`. Постоянно использовать свойство `self` при обращении к свойству класса не нужно. Его следует применять, только если имена параметров инициализатора совпадают с именами свойств класса.

Теперь, когда у нашего инициализатора есть параметры, мы можем создавать объекты класса с определенными свойствами:

```
var bruce = Person(firstName: "Брюс", lastName: "Уэйн")
```

Согласитесь, что создавать объекты класса таким образом намного удобнее, чем задавать сначала значения внутри объявления класса, а потом изменять их после создания объекта класса.

Мы можем также написать оба инициализатора внутри одного класса:

```
class Person {
    var firstName: String
    var lastName: String

    init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }

    init() {
        firstName = ""
        lastName = ""
    }
}
```

```
var bruce = Person(firstName: "Брюс", lastName: "Уэйн")
var nobody = Person()
```

Здесь следует обратить внимание на то, что внутри инициализатора по умолчанию мы не использовали свойства `self`, т. к. там не было конфликтов имен. Компилятор Swift сам понимает в таких ситуациях, что мы указали свойства класса.

Когда мы пишем несколько инициализаторов для класса, то при создании объекта класса Swift сам определяет, какой из них выбрать, — на основании параметров и их типов, которые мы напомним. Когда мы создаем объект класса без параметров, то

создается объект с теми свойствами, которые мы указали внутри инициализатора по умолчанию. А когда мы создаем объект класса с параметрами, то срабатывает инициализатор с параметрами и присваивает нашим свойствам значения, которые мы ему передадим. Если инициализаторов с параметрами несколько, то Swift выбирает их на основании количества, названия и типа параметров, которые мы указали при создании объекта класса.

12.4.3. Локальные и внешние имена параметров инициализатора

Имена параметров инициализатора, так же как и имена параметров функции, делятся на *локальные* и *внешние*. Различие здесь в том, что в инициализаторах все локальные параметры автоматически становятся внешними, если эти внешние параметры не указаны:

```
class Person {
    var firstName: String
    var lastName: String

    init(first firstName: String, last lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }
}
```

```
var bruce = Person(first: "Брюс", last: "Уэйн")
```

Если мы указали внешние параметры в одном инициализаторе, то и при создании объекта класса мы должны использовать только имена внешних параметров. Так же, если мы в одном из инициализаторов указали внешние параметры, то и для всех остальных инициализаторов с параметрами мы должны указать внешние имена.

При этом, если мы по какой-либо причине не хотим в каком-нибудь инициализаторе использовать внешние имена, мы должны на их месте ставить знак нижнего подчеркивания `_`:

```
class Person {
    var firstName: String
    var lastName: String

    init(first firstName: String, last lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }
    init(_ firstName: String, _ lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }
}
```

Тогда при создании объекта класса мы можем написать параметры без имен:

```
var bruce = Person(first: "Брюс", last: "Уэйн")
var tony = Person("Тони", "Стапк")
```

12.4.4. Проваливающиеся инициализаторы

Если есть вероятность, что инициализация может потерпеть неудачу (например, в случае использования недопустимого параметра при инициализации), то лучше задать инициализатор как *проваливающийся* (failable). Проваливающиеся инициализаторы пишутся со знаком вопроса после ключевого слова `init`:

```
init?() {

}
```

Проваливающийся инициализатор возвращает опциональное значение типа, к которому он относится, или `nil` — если инициализация не прошла успешно. Следует заметить, что в одном классе мы не можем создать проваливающийся инициализатор и простой инициализатор с одним и тем же набором параметров.

Для того чтобы лучше понять смысл проваливающихся инициализаторов, давайте напишем его для нашего класса `Person`. Мы поставим условие, что если при создании объекта класса одному из свойств `firstName` или `lastName` будет присвоено пустое значение (""), то инициализатор вернет `nil`, в противном случае он вернет объект опционального типа `Person`:

```
class Person {
    var firstName: String
    var lastName: String

    init?(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName

        if firstName == "" || lastName == "" {
            return nil
        }
    }
}
```

Теперь, если мы создадим объект класса, он нам вернет опциональный `Person?`:

```
var bruce = Person(firstName: "Брюс", lastName: "Уэйн")
```

То, что нам вернулся опциональный тип, мы можем проверить, попытавшись получить доступ к одному из свойств объекта:

```
bruce.firstName // Ошибка
bruce?.firstName // => Some "Уэйн"
```

То есть, мы должны обязательно использовать опциональное сцепление, чтобы получить доступ к свойству объекта класса `Person?`.

Но если мы создадим объект класса с пустым каким-нибудь свойством, то инициализатор вернет нам `nil`:

```
var somebody = Person(firstName: "", lastName: "Фамилия") // => nil
```

12.4.5. Деинициализаторы

Кроме инициализаторов, для классов мы можем создавать также *деинициализаторы*, которые срабатывают, когда объект класса удаляется из памяти.

В Swift мы не должны каждый раз писать деинициализаторы, потому что все управление памятью в нем происходит без нашего участия. Swift сам следит за тем, используется ли тот или иной объект класса, и если нет, то Swift освобождает память от него.

Зачем же тогда нужны деинициализаторы? Иногда бывают случаи, когда нужно выполнить какие-либо действия после удаления объекта из памяти. Например, если у нас есть объект класса, который при создании соединяется с каким-нибудь внешним ресурсом, то при удалении такого объекта нужно указать программе, чтобы она закрыла это соединение. Сказанное справедливо и при работе с файлами.

Деинициализатор пишется через ключевое слово `deinit`:

```
deinit {  
    // инструкции для деинициализации  
}
```

12.5. Методы

В начале главы мы говорили, что кроме свойств классы могут иметь еще и *методы*. Но только если свойства — это характеристики объекта, то методы — это действия, которые может совершать объект.

12.5.1. Создание методов

Создавать методы в Swift так же легко, как и свойства. В Swift они пишутся как обычные функции — через ключевое слово `func`, но только внутри объявления класса. Все, что мы изучали про функции (см. главу 7), работает и с методами, так что мы быстро рассмотрим те же самые возможности, но только для методов.

Давайте объявим метод `greeting` для нашего класса `Person`, который будет печатать на экране приветствие:

```
class Person {  
    var firstName: String  
    var lastName: String
```

```
init(first firstName: String,last lastName: String) {
    self.firstName = firstName
    self.lastName = lastName
}

func greeting() {
    print("Здравствуй")
}

}

var bruce = Person(first: "Брюс", last: "Уэйн")
bruce.greeting()
// "Здравствуй"
```

Подобно простым функциям, методы тоже могут иметь параметры и возвращающиеся значения:

```
class Person {
    var firstName: String
    var lastName: String

    init(firstName: String,lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }

    func greeting(greetingMessage: String) -> String {
        return "\(greetingMessage)"
    }
}

var bruce = Person(firstName: "Брюс", lastName: "Уэйн")
bruce.greeting("Приветствую")
// "Приветствую"
```

Параметрам методов также можно писать внешние имена:

```
class Person {
    var firstName: String
    var lastName: String

    init(firstName: String,lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }

    func greeting(extMsg greetingMessage: String) -> String {
        return "\(greetingMessage)"
    }
}
```

```
var bruce = Person(firstName: "Брюс", lastName: "Уэйн")
bruce.greeting(extMsg: "Приветствую")
// "Приветствую"
```

Если внешние имена параметров для метода не заданы, то при вызове они ведут себя аналогично функциям — внешнее имя первого параметра опускается, а локальные имена остальных параметров автоматически становятся внешними.

Внутри метода можно писать любые свойства или методы этого класса:

```
class Person {
    var firstName: String
    var lastName: String

    init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }

    func greeting(extMsg greetingMessage: String) -> String {
        return "\(greetingMessage), \(firstName)"
    }
}

var bruce = Person(firstName: "Брюс", lastName: "Уэйн")
bruce.greeting(extMsg: "Приветствую")
// "Приветствую, Брюс"
```

Если присутствуют конфликты имен параметров метода и имен свойств или методов класса, тогда для их исключения следует использовать указатель `self`.

12.5.2. Методы типа

Аналогично свойствам типа, классы могут иметь *методы типа*. Для их объявления следует поместить ключевое слово `static` в начало объявления метода:

```
class SimpleClass {
    static func simpleTypeMethod() {

    }
}

SimpleClass.simpleTypeMethod()
```

Если мы хотим, чтобы наш метод типа имел возможность быть переопределенным, то вместо ключевого слова `static` должны указать `class`:

```
class SimpleClass {
    class func simpleTypeMethod() {

    }
}
```

Что интересно, указатель `self` внутри объявления метода типа указывает на сам тип, а не на объект этого типа.

12.6. Индексаторы

В Swift можно объявлять для объектов класса *индексаторы*. Подобно тому, как мы обращались к элементам массивов и словарей через индекс в квадратных скобках: `array[1]`, мы также можем создать свои классы, чьи объекты будут иметь индексы. При этом для индексов можно задавать любую реализацию, а не только обеспечения доступа к элементам.

12.6.1. Синтаксис индексаторов

Синтаксис объявления индексаторов очень похож на объявление вычисляемых свойств и методов для классов, только перед объявлением индексатора мы пишем ключевое слово `subscript`:

```
subscript(index: Int) -> Int {
    get {
        // инструкции, которые возвращают значение типа Int
    }
    set(newValue) {
        // инструкции, которые задают элементам класса новые значения
    }
}
```

Как мы видим, для индексаторов также можно объявлять геттеры и сеттеры. Ранее был приведен пример индексатора, который может и писать, и читать, но если мы захотим, то можем написать индексатор только для чтения:

```
subscript(index: Int) -> Int {
    // инструкции, которые возвращают определенное значение
}
```

Если в массивах и словарях мы просто хранили значение внутри элементов, то теперь мы можем писать собственные действия, которые нужно совершать при получении или записи нового значения.

Также у нас есть возможность объявлять несколько индексаторов для одного класса. Тогда при обращении компилятор будет смотреть на тип значения, который мы напишем в квадратных скобках. На его основе компилятор решит, к какому индексатору обращаться.

12.6.2. Многомерные индексаторы

Мы можем объявить и *многомерные индексаторы*. Для этого нам нужно записать в `subscript` несколько параметров:

```
subscript(index1: Int, index2: Int) -> Int {  
    // инструкции, которые возвращают определенное значение  
}
```

Давайте рассмотрим простой пример, в котором объект класса будет складывать значения записанных ему индексов:

```
class Addition {  
  
    subscript(a: Int, b: Int) -> Int {  
        return a + b  
    }  
  
}  
  
var result = Addition()  
result[2,3] // => 5
```

В этом примере мы сначала создаем класс, внутри которого объявляем двумерный индексатор, который складывает оба значения индекса. Затем мы создаем объект класса `Addition()` и пишем ему индексы 2 и 3, которые программа складывает вместе и выводит нам результат.

Это был простой пример. Давайте теперь усложним задачу и напишем несколько индексаторов для класса, причем один будет одномерный, а второй — двумерный. Для этого давайте попробуем написать класс, который создает матрицы.

Сначала мы напишем объявление класса и зададим ему инициализатор:

```
class Matrix {  
    let rows: Int  
    let columns: Int  
    var array: [Int]  
    init(rows: Int, columns: Int) {  
        self.rows = rows  
        self.columns = columns  
        array = Array(count: rows * columns, repeatedValue: 0)  
    }  
}
```

Здесь мы создаем для класса `Matrix` два свойства: `rows` и `columns`, которые будут хранить количество строк и столбцов нашей матрицы. После создания новой матрицы с определенным числом строк и столбцов нам не нужно будет менять количество этих самых строк и столбцов, поэтому мы объявили их как константы.

Третьим свойством мы записали массив, который создается на основании введенных строк и столбцов. Чтобы сильно не усложнять задачу, мы воспользуемся одномерным массивом, а не двумерным.

Далее мы пишем инициализатор, который при вводе строк и столбцов создает нам одномерный массив. Количество элементов в нем равно произведению числа столбцов и строк. А значения мы заполнили нулями.

Теперь напишем первый индексатор:

```
class Matrix {
    let rows: Int
    let columns: Int
    var array: [Int]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        array = Array(count: rows * columns, repeatedValue: 0)
    }
    subscript(row: Int, column: Int) -> Int {
        get {

            return array[(row * columns) + column]
        } set {

            array[(row * columns) + column] = newValue
        }
    }
}
```

В первом индексаторе у нас записаны и геттер, и сеттер. Через геттер мы возвращаем значение из матрицы по определенным координатам, а через сеттер задаем это значение.

Второй индексатор мы создадим с одним параметром типа `String`. Если в этом параметре написано слово "сумма", то индексатор будет возвращать нам сумму всех значений нашей матрицы. А если слово будет другое, то нам вернут `nil`. Это означает, что тип возвращаемого значения для этого индексатора мы должны сделать опциональным:

```
class Matrix {
    let rows: Int
    let columns: Int
    var array: [Int]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        array = Array(count: rows * columns, repeatedValue: 0)
    }
    subscript(row: Int, column: Int) -> Int {
        get {

            return array[(row * columns) + column]
        } set {

            array[(row * columns) + column] = newValue
        }
    }
}
```

```

    subscript(word: String) -> Int? {
        if word == "сумма" {
            var sum = 0
            for value in array {
                sum += value
            }
            return sum
        } else {
            return nil
        }
    }
}

```

Давайте постепенно проверять наш код. Сначала создадим матрицу размеров 5 на 5, т. е. создадим объект класса `Matrix`:

```
var myMatrix = Matrix(rows: 5, columns: 5)
```

Теперь по тому, какой тип значения мы используем в индексе, компилятор будет решать, использовать первый индексатор или второй. Если мы используем два числовых значения, то компилятор станет использовать первый `subscript`, а если одно строковое значение, то второй:

```

myMatrix[1,1] = 3
myMatrix[1,2] = 7
myMatrix[1,1] // => 3
myMatrix["сумма"] // => Some 10

```

Заметьте, что на последней строчке программа вернула нам опциональный `Int`. Мы можем принудительно его извлечь, поставив восклицательный знак в конце:

```
myMatrix["сумма"]! // => 10
```

Выводы

- ❑ Классы объявляются через ключевое слово `class`.
- ❑ Чтобы создать объект класса, нам нужно написать название класса и в конце приписать пустые круглые скобки: `jack = Person()`.
- ❑ Свойства классов — это просто переменные или константы, созданные внутри объявления класса.
- ❑ Когда мы указываем, что свойство ленивое: `lazy`, то это свойство инициализируется только при обращении к нему.
- ❑ Для классов можно задавать вычисляемые свойства, чьи значения вычисляются на основе определенного выражения. Для вычисляемых свойств можно писать геттеры и сеттеры.
- ❑ Наблюдатель `willSet` — это функция, которая срабатывает до того, как значение свойства изменится, а наблюдатель `didSet` — функция, которая срабатывает после того, как значение свойства изменилось.

- ❑ Подобно свойствам классов, обычным локальным и глобальным переменным мы также можем назначить наблюдателей и вычисляемые переменные.
- ❑ Для классов мы можем указать свойство типа, которое пишется через ключевое слово `static`. Это свойство может быть и хранимым, и вычисляемым. Для вычисляемых свойств типа мы можем вместо `static` писать ключевое слово `class` — если хотим, чтобы это свойство могло быть переопределено.
- ❑ Инициализаторы — это специальные функции внутри класса, которые вызываются при создании нового объекта класса. Они объявляются через ключевое слово `init()`.
- ❑ Мы можем создать инициализатор по умолчанию, который задает значения по умолчанию для каждого свойства. Либо можем создать инициализатор с параметрами, который устанавливает свойствам определенные значения.
- ❑ Локальные имена параметров инициализаторов автоматически становятся внешними.
- ❑ Если есть вероятность, что инициализация может потерпеть неудачу, то лучше писать инициализатор как проваливающийся: `init?`.
- ❑ Для классов мы можем создавать также деинициализаторы: `deinit`, которые срабатывают, когда объект класса удаляется из памяти.
- ❑ Методы класса являются теми же функциями, только объявленными внутри тела класса.
- ❑ Методы типа указываются через ключевое слово `static`, но если мы хотим, чтобы этот метод имел возможность быть переопределенным, — то пишем вместо `static` ключевое слово `class`.
- ❑ Мы можем создать классы, чьи объекты могут иметь индексы. Они объявляются через ключевое слово `subscript`.

ГЛАВА 13



Наследование

Одной из основ объектно-ориентированных языков программирования является *наследование*. Наследование позволяет создавать дочерние классы, которые наследуют все характеристики родительского класса. Эти характеристики включают в себя свойства, методы, индексаторы и т. д.

Допустим, у нас есть класс `Animal` со свойством `name` и методом `voice()`:

```
class Animal {
    var name: String
    func voice() {
        print("Apppp")
    }
    init() {
        name = ""
    }
    init(name: String) {
        self.name = name
    }
}
```

Создав класс `Animal`, мы можем создать новый дочерний для него класс `Cat`:

```
class Cat: Animal {

}
```

Как можно видеть, чтобы указать, что исходный класс наследует характеристики от другого класса, нужно при объявлении записать название родительского класса через двоеточие.

Теперь все свойства, методы и инициализаторы класса `Animal` унаследованы дочерним классом `Cat`. Для того чтобы это проверить, давайте создадим новый объект класса `Cat` и вызовем свойство `name` родительского класса:

```
var myCat = Cat()
myCat.name // ""
```

А чтобы у нашего кота было имя, мы можем его назначить:

```
myCat.name = "Том"
```

Мы также можем вызвать метод `voice()` родительского класса:

```
myCat.voice() // "Аpppp"
```

13.1. Переопределение

В предыдущем примере наш кот при вызове метода `voice()` произнес "Аpppp". Но если мы хотим, чтобы метод `voice()` имитировал голос кота, то логичнее было бы, чтобы он мяукал. Для этого нам нужно переназначить (переопределить) метод `voice()` для дочернего класса `Cat`. При этом `Animal` при вызове метода `voice()` по-прежнему будет произносить "Аpppp", а `Cat` — мяукать.

Для того чтобы *переопределить* метод родительского класса, нам нужно написать объявление метода внутри дочернего класса с тем же именем, только перед названием этого метода написать ключевое слово `override`:

```
class Cat: Animal {  
    override func voice() {  
        print("Мяу")  
    }  
}
```

Теперь при вызове метода `voice()` для дочернего класса `Cat` мы получим "Мяу":

```
myCat.voice() // "Мяу"
```

Такое переопределение справедливо также и для свойств, индексаторов и инициализаторов. Но если со свойствами и индексаторами все происходит аналогично методам, то с инициализаторами все обстоит немного иначе.

13.2. Наследование инициализаторов

Как мы знаем, все хранимые свойства класса должны иметь начальное значение. Это касается и свойств дочернего класса, наследованных от родительского класса.

Давайте рассмотрим пример и воспользуемся для этого классами `Animal` и `Cat` из предыдущего раздела. Сначала объявим эти два класса:

```
class Animal {  
    var name: String  
    func voice() {  
        print("Аpppp")  
    }  
    init() {  
        name = ""  
    }  
}
```

```

    init(name: String) {
        self.name = name
    }
}

```

```

class Cat: Animal {

}

```

Здесь мы объявили родительский класс `Animal` с одним свойством, с одним методом и двумя инициализаторами, а дочерний класс записали пустым. И сейчас хотим понять, что было унаследовано дочерним классом. В данном случае дочерний класс должен унаследовать оба инициализатора. Давайте проверим, так ли это, и создадим два объекта класса `Cat` с помощью разных инициализаторов:

```

var felix = Cat() // {{name ""}}
var tom = Cat(name: "Tom") // {{name "Tom"}}

```

Как видим, оба инициализатора родительского класса для нашего дочернего класса работают. Здесь мы создали один объект класса через инициализатор по умолчанию, в котором свойству `name` задается пустое значение. А второй объект создали посредством инициализатора с параметром `name`, в котором написали имя "Tom".

На следующем этапе создадим для дочернего класса `Cat` новое свойство `color`:

```

class Cat: Animal {
    var color: String // Ошибка
}

```

Однако, объявив в таком виде новое свойство, мы получили ошибку. Дело в том, что, как мы помним, значение свойства всегда должно быть инициализировано. Поэтому у нас есть два пути. Либо явно присвоить новому свойству пустое значение:

```

class Cat: Animal {
    var color: String = ""
}

```

Тогда всем объектам класса будет присваиваться одно и то же значение для свойства `color = ""`.

Намного лучше записать здесь новый инициализатор со свойством `color`:

```

class Cat: Animal {
    var color: String

    init(color: String) {
        self.color = color
    } // Ошибка
}

```

Но компилятор снова вывел нам ошибку. Сейчас он требует для родительского класса инициализатора по умолчанию, т. е. вызова в нашем инициализаторе функции `super.init()`, где свойство `super` указывает на родительский класс.

Все это происходит потому, что в новом инициализаторе мы не записали, какое значение присвоить свойству `name`, наследуемому от родительского класса. Так как у дочернего класса теперь два свойства (родительское `name` и собственное `color`), то они оба должны обязательно быть инициализированы. Компилятор подсказывает нам вызвать инициализатор по умолчанию родительского класса, потому что там свойству `name` присваивается стандартное пустое значение.

Давайте вызовем этот инициализатор родительского класса и создадим новый объект класса через инициализатор `init(color: String)`:

```
class Cat: Animal {
    var color: String

    init(color: String) {
        self.color = color
        super.init()
    }
}

var barbara = Cat(color: "white") // {{name ""} color "white"}
```

Как мы видим, созданся новый объект класса с пустым свойством `name` и свойством `color`, равным `"white"`.

Теперь, если мы попробуем создать объекты с помощью инициализаторов родительского класса, как в предыдущем примере, то получим ошибку:

```
var felix = Cat() // Ошибка
var tom = Cat(name: "Tom") // Ошибка
```

В первом случае компилятор говорит, что не может найти значение для свойства `color`, а во втором — что не может найти такой инициализатор с параметром `name`. А дело здесь в том, что после объявления хотя бы одного инициализатора в дочернем классе, наследование инициализаторов из родительского класса прекращается — и нам следует инициализировать свои собственные инициализаторы.

Давайте теперь изменим инициализатор дочернего класса и добавим в него параметр `name`. Так мы сможем для объектов класса `Cat` задавать сразу и `color`, и `name`:

```
class Cat: Animal {
    var color: String

    init(color: String, name: String) {
        self.color = color
        self.name = name // Ошибка
    }
}
```

И опять компилятор вывел нам ошибку. Дело в том, что свойства, объявленные в родительском классе, должны быть инициализированы именно там. То есть, мы сначала должны вызвать инициализатор по умолчанию для родительского класса, который присвоит свойству `name` пустое значение, и только потом мы сможем в инициализаторе дочернего класса изменить значение на новое. Соответственно, инициализатор нужно переписать таким образом:

```
class Cat: Animal {
    var color: String

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

Все указанные действия нужно делать именно в предложенном порядке, иначе возникнут ошибки. То есть, для инициализаторов дочернего класса Swift требует: сначала присвоить значения для собственных свойств дочернего класса, затем вызвать инициализатор родительского класса, чтобы он присвоил начальные значения для родительского класса. А после присвоения значения наследованных свойств через родительский инициализатор мы уже можем менять значение унаследованных свойств, — что и делается на третьей строчке инициализатора (`self.name = name`) в приведенном примере.

Чтобы немного сократить эту запись, мы можем через дочерний инициализатор вызвать не родительский инициализатор по умолчанию, а инициализатор с параметром:

```
class Cat: Animal {
    var color: String

    init(color: String, name: String) {
        self.color = color
        super.init(name: name)
    }
}

var barbara = Cat(color: "white", name: "Barbara") // {{name "Barbara"}
color "white"}
```

В этом примере интересно, что в записи `super.init(name: name)` не произошло конфликта имен. Здесь компилятор четко понял, что первый `name` — это параметр родительского инициализатора, а второй `name` — это значение параметра инициализатора, в котором он объявлен.

13.3. Переопределение инициализаторов

В предыдущем примере мы использовали инициализатор для дочернего класса, который не имел того же набора параметров, что и родительский инициализатор. Если бы они имели один и тот же набор параметров, то в дочернем классе нужно было бы переопределить этот инициализатор:

```
class Animal {
    var name: String

    func voice() {
        print("Apppp")
    }
    init() {
        name = ""
    }
    init(name: String) {
        self.name = name
    }
}

class Cat: Animal {

    override init(name: String) {
        self.name = name
    } // Ошибка
}
```

Тем не менее, все равно происходит ошибка. Дело в том, что компилятор опять просит инициализировать свойства родительского класса внутри родительского класса, т. е. нам снова нужно вызвать инициализатор `super.init()`:

```
class Cat: Animal {

    override init(name: String) {
        super.init()
        self.name = name
    }
}
```

Для того чтобы различать функционал дочернего инициализатора и родительского, давайте добавим к свойству имени для дочернего класса префикс "Dr. ":

```
class Cat: Animal {

    override init(name: String) {
        super.init()
        self.name = "Dr. " + name
    }
}
```

Теперь мы можем протестировать эти инициализаторы при создании дочернего и родительского классов:

```
var lion = Animal(name: "Simba") // => {name "Simba"}
var tom = Cat(name: "Tom") // => {{name "Dr. Tom"}}
```

Как мы видим, для родительского класса создан объект со свойством `name` без префикса, а для дочернего с префиксом `"Dr. "`.

13.4. Назначенные и удобные инициализаторы

Для того чтобы решить проблемы инициализации между родительскими и дочерними классами, в Swift есть два вида инициализаторов: назначенные (designated) инициализаторы и удобные (convenience) инициализаторы.

Назначенные инициализаторы — это основные инициализаторы для классов. Они могут инициализировать все свойства исходного класса и должны вызывать инициализатор родительского класса для продолжения процесса инициализации вверх по цепочке. Они пишутся как обычные инициализаторы через ключевое слово `init`:

```
init() {
    // инструкции инициализации
}
```

Следующий вид — *удобные* инициализаторы. С их помощью нужно вызвать инициализаторы, объявленные в том же классе. Но мы не можем с их помощью вызывать инициализаторы родительского класса. Удобные инициализаторы пишутся через ключевое слово `convenience`:

```
convenience init() {
    // инструкции инициализации
}
```

Использовать удобные инициализаторы не обязательно. Они просто призваны облегчить ситуации, когда нужно создать инициализаторы только с одним свойством, а остальные свойства можно вызвать с помощью другого инициализатора того же класса.

Давайте рассмотрим все это на примере, воспользовавшись дочерним и родительским классами из предыдущего примера:

```
class Animal {
    var name: String

    func voice() {
        print("Apppp")
    }
    init() {
        name = ""
    }
}
```

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

```
class Cat: Animal {  
  
}
```

Теперь создадим еще два свойства: `color` (цвет) и `breed` (порода) для дочернего класса и напомним для него инициализатор:

```
class Cat: Animal {  
    var breed: String  
    var color: String  
  
    init(name: String, color: String, breed: String) {  
        self.color = color  
        self.breed = breed  
        super.init(name=name)  
    }  
}
```

Инициализатор, который мы создали, называется назначенным инициализатором и записывается как обычный инициализатор. Ранее мы отметили, что главной особенностью назначенных инициализаторов является то, что они должны обязательно выполнять инициализатор родительского класса. Именно это мы и сделали в последней строке инициализатора `super.init(name=name)`.

Давайте теперь напишем удобный инициализатор для этого класса:

```
class Cat: Animal {  
    var breed: String  
    var color: String  
  
    init(name: String, color: String, breed: String) {  
        self.color = color  
        self.breed = breed  
        super.init(name=name)  
    }  
    convenience init(color: String) {  
        self.init(name: "", color: color, breed: "Persidian")  
    }  
}
```

Теперь с помощью этого удобного инициализатора мы можем создать объект класса только с одним параметром `color`, а все остальные параметры будут присвоены с помощью собственного назначенного инициализатора.

13.5. Необходимые инициализаторы

Если мы хотим, чтобы определенный инициализатор точно унаследовался дочерним классом, мы можем указать, что этот инициализатор *необходимый*. Необходимые инициализаторы пишутся через ключевое слово `required`:

```
class SimpleClass {
    required init() {

    }
}
```

Давайте рассмотрим это на примере классов `Animal` и `Cat`, и создадим внутри родительского класса `Animal` свойство `name` и необходимый инициализатор, а дочерний класс `Cat` пока оставим пустым:

```
class Animal {
    var name: String
    required init(name: String) {
        self.name = name
    }
}

class Cat: Animal {
}
```

Как мы помним, если в дочернем классе не объявлено ни одного инициализатора, то все инициализаторы наследуются от родительского класса. То есть, в нашем случае ошибок никаких не выводится, потому что необходимый инициализатор наследуется от родительского класса.

Но как только мы попробуем объявить инициализатор по умолчанию для дочернего класса, то получим ошибку:

```
class Animal {
    var name: String
    required init(name: String) {
        self.name = name
    }
}

class Cat: Animal {
    init() {

    }
} // Ошибка
```

Здесь компилятор выводит ошибку, потому что реализации необходимого инициализатора для дочернего класса теперь нет.

Выводы

- ❑ Родительский класс указывается через двоеточие: `class Cat: Animal`.
- ❑ Дочерние классы автоматически наследуют все свойства и методы родительского класса.
- ❑ Если для дочернего класса не создано ни одного инициализатора, то дочерний класс наследует все инициализаторы родительского класса.
- ❑ Для переопределения служит ключевое слово `override`.
- ❑ Назначенные инициализаторы должны инициализировать все свойства исходного класса и должны вызвать инициализатор родительского класса для продолжения процесса инициализации вверх по цепочке.
- ❑ Удобные инициализаторы пишутся через ключевое слово `convenience` — с их помощью нужно вызывать инициализаторы, объявленные в том же классе.
- ❑ Если мы хотим, чтобы определенный инициализатор точно унаследовался дочерним классом, то можем указать, что этот инициализатор — необходимый: `required`.

ГЛАВА 14



Автоматический подсчет ссылок

Автоматический подсчет ссылок, или Automatic Reference Counting (ARC), — это технология управления памятью приложений. Она была представлена в 2011 году для языка Objective-C, и вслед за ним Swift тоже унаследовал ее.

До автоматического подсчета ссылок использовался *сборщик мусора*, в котором разработчикам приходилось вручную управлять памятью приложений. По сравнению с ним, ARC имеет множество преимуществ. Например, при использовании ARC не запускаются никакие дополнительные процессы, как в случае со сборщиком мусора, когда запускается отдельный процесс для управления памятью. ARC уже на этапе компиляции сам понимает, от каких объектов и когда нужно освободить память.

В этой главе мы на различных примерах подробно рассмотрим, как правильно использовать автоматический подсчет ссылок в Swift.

14.1. Принципы работы автоматического подсчета ссылок

Каждый раз, когда мы создаем новый объект класса, ARC автоматически выделяет под него ячейку памяти. Ее величина зависит от количества хранимых в ней свойств, методов и их типов.

Приведем пример — создадим класс и три разных объекта класса для него:

```
class Animal {
    var name: String
    init(name: String) {
        self.name = name
        print("\(name) инициализирован")
    }
    deinit {
        print("\(name) деинициализирован")
    }
}
```

```
var monkey1: Animal?  
var monkey2: Animal?  
var monkey3: Animal?
```

Все объекты класса получили значение `nil`, т. к. мы указали для них опциональный тип. Теперь одной из переменных присвоим объект класса `Animal`:

```
monkey1 = Animal(name: "Donkey Kong")  
// напечатает "Donkey Kong инициализирован"
```

Как можно видеть, программа напечатала нам сообщение, что объект класса инициализирован. Теперь в остальные переменные запишем сильные ссылки на этот объект класса:

```
monkey2 = monkey1  
monkey3 = monkey1
```

На следующем шаге мы попытаемся убрать из памяти значения первых двух переменных. Для этого нам нужно присвоить этим переменным `nil` — тогда запустится деинициализатор и выведет нам сообщение, что объект деинициализирован:

```
monkey1 = nil  
monkey2 = nil
```

Следовало бы ожидать, что программа два раза напишет нам, что "Donkey Kong деинициализирован", поскольку это сообщение мы записали в деинициализаторе класса. Но ARC все еще сохраняет в памяти место для этого объекта класса, т. к. по крайней мере один из объектов на него ссылается. А вот если мы уберем из памяти и последний объект — `monkey3`, то тогда ARC полностью освободит память от объекта класса:

```
monkey3 = nil  
// напечатает "Donkey Kong деинициализирован"
```

14.2. Циклы сильных ссылок внутри объектов классов

В предыдущем примере мы рассмотрели, как объекты классов уничтожают сильные ссылки. Но, тем не менее, существует возможность написать код, в котором объекты класса никогда не будут уничтожаться из памяти. Это произойдет, если два объекта класса будут содержать в себе сильные ссылки друг на друга. Такое явление называется *циклом сильных ссылок* между объектами классов и, если его не предотвратить, оно может привести к переполнению памяти.

Рассмотрим, в чем проблема, на примере и создадим два класса: `Person` и `Cat`:

```
class Person {  
    var cat: Cat?  
    deinit {  
        print("Объект класса Person деинициализирован")  
    }  
}
```

```
class Cat {  
    var owner: Person?  
    deinit {  
        print("Объект класса Cat деинициализирован")  
    }  
}
```

Каждый из этих классов обладает опциональным свойством, имеющим тип противоположного класса. И для каждого класса мы написали деинициализатор с выводом информации на экран. Когда объект какого-либо класса уничтожится из памяти, мы узнаем об этом, увидев сообщение, написанное в деинициализаторе.

Теперь давайте создадим для этих классов два объекта. Чтобы иметь возможность потом присвоить этим объектам `nil` и уничтожить объекты класса из памяти, нам нужно объявить эти объекты опциональными:

```
var tom: Cat?  
var jack: Person?  
  
tom = Cat()  
jack = Person()
```

А на следующем шаге свяжем сильными ссылками свойства этих двух объектов классов:

```
tom!.owner = jack  
jack!.cat = tom
```

Здесь мы использовали принудительное извлечение, т. к. объект класса имеет опциональный тип.

Теперь давайте попробуем уничтожить объекты из памяти, присвоив им `nil`:

```
tom = nil  
jack = nil
```

Если бы объекты уничтожились из памяти, выполнялся бы деинициализатор и напечатал бы сообщение об этом. Но ничего не произошло.

Приведенный пример как раз и иллюстрирует проблему цикла сильных ссылок, когда два объекта класса не могут быть освобождены из памяти, поскольку содержат сильные ссылки на друг друга. В следующем разделе мы рассмотрим, как решить эту проблему.

14.3. Решение проблемы циклов сильных ссылок между объектами классов

Swift предоставляет два способа решения проблемы циклов сильных ссылок. Можно указать, что ссылка между двумя объектами слабая (`weak`), либо что ссылка не имеет владельца (`unowned`).

14.3.1. Слабые ссылки

Рассмотрим сначала первый случай. Для того чтобы указать, что ссылка между двумя объектами *слабая*, нам нужно перед свойством, ссылающимся на объект другого класса, написать ключевое слово `weak`. Соответственно, предыдущий пример можно переписать так:

```
class Person {
    weak var cat: Cat?
    deinit {
        print("Объект класса Person деинициализирован")
    }
}

class Cat {
    var owner: Person?
    deinit {
        print("Объект класса Cat деинициализирован")
    }
}

var tom: Cat?
var jack: Person?

tom = Cat()
jack = Person()

tom!.owner = jack
jack!.cat = tom

tom = nil
jack = nil

// Объект класса Cat деинициализирован
// Объект класса Person деинициализирован
```

Как мы видим, сообщения деинициализаторов появились. Когда объект класса уничтожается из памяти, компилятор слаботому свойству, которое ссылается на этот объект, присваивает `nil`. Таким образом, достаточно одной слабой ссылки, чтобы оба объекта класса уничтожились из памяти.

14.3.2. Ссылки без владельца

Следующий способ решения проблемы цикла сильных ссылок — это использование ссылок, не имеющих владельца. Когда мы задействовали слабые ссылки, нам требовалось, чтобы оба свойства обязательно были опционального типа, т. к. именно им компилятор присваивает `nil`. *Ссылки без владельца*, наоборот, всегда имеют

значение, и их нужно объявлять как обычные переменные. Тогда в конце не придется извлекать значение из опционального типа.

Давайте теперь перепишем наш пример с использованием ссылок без владельца:

```
class Person {
    var cat: Cat?
    deinit {
        print("Объект класса Person деинициализирован")
    }
}
```

```
class Cat {
    unowned var owner: Person

    init(owner: Person) {
        self.owner = owner
    }

    deinit {
        print("Объект класса Cat деинициализирован")
    }
}
```

```
var jack: Person?
```

```
jack = Person()
jack!.cat = Cat(owner: jack!)
```

```
jack = nil
```

```
// Объект класса Cat деинициализирован
// Объект класса Person деинициализирован
```

Выводы

- ❑ Автоматический подсчет ссылок — это технология автоматического управления памятью приложений.
- ❑ ARC считает все сильные ссылки на объекты.
- ❑ Если два объекта класса будут содержать в себе сильные ссылки друг на друга, то они никогда не будут убраны из памяти. Это состояние называется циклом сильных ссылок.
- ❑ Для решения цикла сильных ссылок нужно использовать слабые (*weak*) ссылки или ссылки без владельца (*unowned*).

ГЛАВА 15



Структуры

В настоящее время *структуры* существуют во многих языках программирования. Но в разных языках они имеют различное предназначение. В Swift структуры очень похожи на классы. Это не говорит о том, что они полностью идентичны, но у них очень много общего.

В прочих языках программирования структуры являются всего лишь простым контейнером для нескольких значений, и для них нельзя задавать свойства и методы. В Swift, все наоборот — структурам можно назначать свойства, методы, вычисляемые свойства, инициализаторы и т. д. В Swift в виде структур представлены типы `Int`, `Double`, `Float`, `String`, `Bool`, а также массивы и словари.

Объявление структур происходит через ключевое слово `struct`:

```
struct SimpleStruct {  
  
}
```

Одной из особенностей структур является то, что они не могут наследоваться.

15.1. Типы-значения и ссылочные типы

Самое фундаментальное отличие структур от классов заключается в том, что структура является типом-значением, а классы — ссылочным типом. Каждый раз, когда мы присваиваем значение переменной или константе, типы-значения всегда копируют значение, и мы получаем копию этого значения. Но когда мы присваиваем значение из ссылочного типа переменной или константе, значение не дублируется, а ссылается на оригинальное значение. И при смене значения в любой из переменных, ссылающихся на оригинальное значение, произойдет перезапись оригинального значения.

15.2. Оператор идентичности

Так как существует вероятность, что разные переменные и константы могут ссылаться на одни и те же объекты класса, в Swift существуют операторы идентич-

ности: `===` и `!==`. Эти операторы позволяют выяснить, ссылаются ли два их операнда на один и тот же объект класса.

```
class SimpleClass {  
  
}  
  
var a = SimpleClass()  
var b = SimpleClass()  
var c = b  
  
if a === b {  
    print("Переменные ссылаются на один и тот же объект класса")  
} else {  
    print("Переменные не ссылаются на один и тот же объект класса")  
}  
// "Переменные не ссылаются на один и тот же объект класса"  
  
if b === c {  
    print("Переменные ссылаются на один и тот же объект класса")  
} else {  
    print("Переменные не ссылаются на один и тот же объект класса")  
}  
// "Переменные ссылаются на один и тот же объект класса"  
  
if b !== c {  
    print("Переменные не ссылаются на один и тот же объект класса")  
} else {  
    print("Переменные ссылаются на один и тот же объект класса")  
}  
// "Переменные ссылаются на один и тот же объект класса"
```

В этом примере мы создали два разных объекта одного класса `SimpleClass` и записали их в переменные `a` и `b`. Внутри переменной `c` мы записали ссылку на объект класса, хранящийся в `b`. Затем мы стали проверять на идентичность каждую переменную.

15.3. Свойства структур

Внутри структур, как и внутри классов, можно создавать и хранимые, и вычисляемые свойства. Различие заключается только в том, что свойства в структуре не требуют обязательного присвоения значения, — можно просто указать тип, как при обычных переменных:

```
struct Circle {  
  
    var radius: Double
```

```
var area: Double {  
    return radius * radius * 3.14  
}  
  
var myCircle = Circle()  
myCircle.radius    // => 3  
myCircle.area      // => 28.26
```

Различие в свойствах проявляется только при работе с константами. Так, если мы объект структуры присвоили константе, то ни одно из его свойств не может быть изменено. Допустим, у нас есть структура с одним переменным свойством и с другим свойством — константой:

```
struct SimpleStruct {  
    var simpleVar = 2  
    let simpleConst = 4  
}
```

Теперь, если мы попробуем присвоить объект структуры константе, то не сможем изменить ни одно из его свойств, даже если оно записано как переменная:

```
let mySimpleStruct = SimpleStruct()  
mySimpleStruct.simpleConst = 10 // Ошибка  
mySimpleStruct.simpleVar = 10  // Ошибка
```

Эта особенность структур отличает их от классов, в которых в таких случаях можно менять переменные свойства:

```
class SimpleClass {  
    var simpleVar = 2  
    let simpleConst = 4  
}
```

```
let mySimpleClass = SimpleClass()  
mySimpleStruct.simpleConst = 10 // Ошибка  
mySimpleStruct.simpleVar = 10   // Удачное изменение
```

15.4. Свойства типа для структур

Еще одно отличие структур от классов можно найти в свойствах типа. Свойства типа для структур пишутся через ключевое слово `static`:

```
struct SimpleStruct {  
    static var simpleTypeProp = 3  
}
```

И уже они в свою очередь требуют начального значения или инициализации.

15.5. Методы структур

Методы для структур пишутся аналогично методам классов через ключевое слово `func`:

```
struct SimpleStruct {  
  
    func method() -> Int {  
        return 22  
    }  
  
}
```

Одно из отличий методов структур от методов классов проявляется в невозможности изменять у структур собственные свойства. Для того чтобы метод структуры имел возможность изменять свойства структуры, нужно объявить метод как *мутирующий*. Мутирующие методы пишутся через ключевое слово `mutating` перед `func`:

```
struct Circle {  
    var radius: Double  
  
    mutating func changeRadius(radius: Double) {  
        self.radius = radius  
    }  
}  
  
var myCircle = Circle(radius: 3)  
myCircle.changeRadius(4)
```

В этом примере мы объявили мутирующий метод для структуры `Circle`, который меняет значение свойства `radius`:

```
struct Circle {  
    var radius: Double  
  
    mutating func changeRadius(radius: Double) {  
        self.radius = radius  
    }  
}  
  
var myCircle = Circle(radius: 3)  
myCircle.radius  
myCircle.changeRadius(4)  
myCircle.radius
```

15.6. Методы типа для структур

Подобно свойствам типа для структур, методы типа для структур тоже объявляются через ключевое слово `static`:

```
struct SimpleStruct {  
  
    static func simpleTypeMethod() -> Int {  
        return 22  
    }  
  
}  
  
SimpleStruct.simpleTypeMethod() // => 22
```

15.7. Инициализаторы структур

Как мы отмечали ранее, для свойств структуры не обязательно писать начальное значение. И все это потому, что во всех структурах по умолчанию присутствует *поэлементный инициализатор*. Он позволяет без особых проблем создавать объекты класса со всеми его свойствами в параметрах:

```
struct SimpleStruct {  
    var simpleVar: String  
    let simpleConst: Int  
}  
  
var mySimpleStruct = SimpleStruct(simpleVar: "Text", simpleConst = 3)
```

Здесь мы просто объявили новую структуру без инициализаторов и на последней строчке попробовали создать объект структуры с параметрами, соответствующими свойствам класса. Все прошло без ошибок, т. к. во всех структурах автоматически присутствует поэлементный инициализатор.

Выводы

- ❑ В Swift структурам можно назначать свойства, методы, вычисляемые свойства, инициализаторы и т. д.
- ❑ Объявление структур происходит через ключевое слово `struct`.
- ❑ Структуры, в отличие от классов, не могут наследоваться.
- ❑ Структура является типом-значением, а класс — ссылочным типом.
- ❑ Для того чтобы проверить, ссылаются ли две переменные на один и тот же объект класса, существуют операторы идентичности: `===` и `!==`.
- ❑ Если присвоить объект структуры константе, то мы не сможем изменить ни одно из его свойств, даже если оно записано как переменная.
- ❑ Свойства типа и методы типа для структур пишутся через ключевое слово `static`.
- ❑ Для того чтобы методы структуры имели возможность изменять значения свойств, они должны быть объявлены как мутирующие (*mutating*).
- ❑ Во всех структурах по умолчанию присутствует поэлементный инициализатор.

ГЛАВА 16



Проверка типов и приведение типов

С помощью проверки типов можно проверить, принадлежит ли объект соответствующему типу. А приведение типов в Swift позволяет через объект родительского класса обращаться к объекту дочернего класса. Сказанное звучит пока не очень понятно, так что лучше мы посмотрим все это на примерах.

16.1. Проверка типов

Простым примером *проверки типов* может быть условное выражение с использованием оператора проверки типов `is`. Если тип соответствует объекту, то оператор `is` возвращает `true`, иначе — `false`. Чтобы посмотреть, как это работает, сначала создадим новый класс `SimpleClass` и объект этого класса:

```
class SimpleClass {  
  
}  
  
var simpleInst = SimpleClass()
```

Теперь, чтобы проверить, действительно ли объект принадлежит определенному классу, нужно использовать оператор проверки типов:

```
if simpleInst is SimpleClass {  
    print("Объект принадлежит классу SimpleClass")  
}  
// "Объект принадлежит классу SimpleClass"
```

Проверка типов очень полезна, когда мы хотим выполнить действия только для объектов определенного класса. Рассмотрим теперь более сложный пример. Для этого создадим три новых класса: один родительский класс `Parent` и два его дочерних класса: `Child1` и `Child2`:

```
class Parent {  
  
}
```



```
class Child1: Parent {  
  
}  
  
class Child2: Parent {  
  
}
```

Далее создадим для дочерних классов объекты и в произвольном порядке запишем их в массив `myArray`:

```
var child1 = Child1()  
var child2 = Child2()  
  
var myArray = [child1, child2, child2, child1, child2]
```

Так как массивы в Swift строго типизированы, то, скорее всего, мы бы получили здесь ошибку. Массив `myArray` содержит объекты сразу нескольких типов, что, как мы помним, не приемлемо в Swift. Но никакой ошибки не происходит. Дело в том, что компилятор видит, что в массиве содержатся объекты одного родительского класса. Тогда Swift автоматически назначает всему массиву тип `Parent`, т. е. общий для объектов родительский тип.

В таких ситуациях, чтобы отсортировать объекты разных типов в одном массиве, нам помогает проверка типов. Она работает через оператор `is`, который указывается в условном выражении между проверяемым объектом и его типом. Если тип соответствует объекту, то оператор `is` возвращает `true`, иначе — `false`:

```
if item is SomeType {  
    // действия, если объект item имеет тип SomeType  
}
```

С помощью такой конструкции мы можем проверить каждый элемент нашего массива на соответствие определенному типу:

```
for item in myArray {  
    if item is Child1 {  
        print("Child1")  
    } else {  
        print("Child2")  
    }  
}  
// Child1  
// Child2  
// Child2  
// Child1  
// Child2
```

Как можно видеть, используя цикл, мы отсортировали каждый элемент по его дочернему типу.

16.2. Приведение типов

В только что рассмотренном примере не все так идеально, как кажется. Переменная `item` по-прежнему имеет тип `Parent`. И чтобы проверить это утверждение, давайте сначала добавим каждому классу свои свойства:

```
class Parent {
    var parentProp = "Parent Property"
}

class Child1: Parent {
    var child1Prop = "Child1 Property"
}

class Child2: Parent {
    var child2Prop = "Child2 Property"
}
```

Теперь, если мы попробуем получить значение из свойства `child1Prop` дочернего класса `Child1`, то получим ошибку:

```
for item in myArray {
    if item is Child1 {
        print("Child1")
        item.child1Prop // ошибка
    } else {
        print("Child2")
    }
}
```

Дело в том, что конструкция `if item is Child1` только проверяет тип заключенного в ней объекта, но тип `item` остается по-прежнему `Parent`. В этом случае нам поможет *приведение типов*. В Swift используется нисходящее приведение типов (downcasting). Оно позволяет через объект родительского класса присваивать константам и переменным ссылку на объект дочернего класса.

Поскольку есть вероятность, что приведение типов потерпит неудачу, в Swift введены два оператора приведения: оператор опционального приведения типов `as?` и оператор принудительного приведения типов `as!`.

Если приведение типов не потерпело неудачу, оператор `as?` всегда возвращает объект опционального дочернего типа, в противном случае он возвращает нам `nil`. Оператор `as!` принудительного приведения возвращает объект опционального дочернего типа и сразу же извлекает из него значение. Поэтому использовать этот оператор нужно в том случае, когда мы точно уверены, что приведение типов возможно.

Теперь давайте попробуем использовать оператор принудительного приведения, чтобы привести `item` к типу `Child1`. Для этого нужно назначить объект другой переменной или константе и применить оператор `as!`:

```

for item in myArray {
    if item is Child1 {
        print("Child1")
        let newItem = item as! Child1
        print(newItem.child1Prop)
    } else {
        print("Child2")
    }
}
// Child1
// Child1 Property
// Child2
// Child2
// Child1
// Child1 Property
// Child2

```

Обратите внимание, что здесь мы были точно уверены, что оператору принудительного извлечения попадутся только объекты дочернего класса `Child1`, т. к. перед этим, в условном выражении, мы проверяем `item` на соответствие именно этому типу.

Теперь приведем пример с оператором `as?`. Для этого удалим из нашего примера проверку типов и оставим только приведение типа внутри цикла:

```

for item in myArray {
    let newItem = item as Child1 // Ошибка
    print(newItem.child1Prop)
}

```

В этом случае сортировка по типам дочерних классов не происходит, и через приведение типов проходят и объекты с типом `Child1`, и с типом `Child2`. Проблема здесь в том, что Swift не может привести к типу `Child1` объект типа `Child2`. Поэтому нам выводится ошибка. Для того чтобы избежать ошибки, мы должны использовать оператор `as?`:

```

for item in myArray {
    let newItem = item as? Child1 // Ошибка
    print(newItem.child1Prop)
}

```

Теперь, если оператору `as?` попадается объект класса `Child2`, то он возвращает нам `nil`, а если объект класса `Child1`, то этот оператор возвращает объект опционального типа `Child1?`. Это означает, что перед использованием нам нужно извлечь значение из опционального типа:

```

for item in myArray {
    if let newItem = item as? Child1 {
        print(newItem.child1Prop)
    }
}

```

```
// Child1 Property  
// Child1 Property
```

16.3. Проверка типов *Any* и *AnyObject*

В Swift есть два специальных типа, которые могут хранить значения разного типа:

- ❑ тип *AnyObject* может хранить объекты любого класса;
- ❑ тип *Any* может хранить вообще объекты любого типа, в том числе и функционального.

Эти типы нам будут встречаться часто, когда мы станем работать с фреймворками Cocoa. В частности, например, когда мы получаем массивы из Objective-C, то они принимают тип `[AnyObject]`, т. к. массивы в Objective-C не строго типизированы и могут хранить любые значения. Это означает, что мы можем только гадать, значения какого типа содержатся в том или ином массиве. В таких ситуациях мы можем воспользоваться нисходящим приведением типов, чтобы привести значения к нужному нам типу.

16.3.1. Тип *AnyObject*

Создадим новый массив с типом `[AnyObject]` и запишем в него объекты двух произвольных классов. Так мы симулируем получение массива из Objective-C :

```
class SimpleClass {  
  
}  
  
class SimpleClass2 {  
  
}  
  
var simpleVar = SimpleClass()  
var simpleVar2 = SimpleClass2()  
  
var array: [AnyObject] = [simpleVar, simpleVar2,  
simpleVar,simpleVar,simpleVar2]
```

Теперь мы можем применить проверку и нисходящее приведение типов. Поскольку мы будем проверять тип перед его приведением, то можно воспользоваться оператором принудительного приведения `as!:`

```
for item in array {  
    if item is SimpleClass {  
        let object = item as! SimpleClass  
        print("Объект класса SimpleClass")  
    } else {  
        let object = item as! SimpleClass2
```

```
        print("Объект класса SimpleClass2")
    }
}

// "Объект класса SimpleClass"
// "Объект класса SimpleClass2"
// "Объект класса SimpleClass"
// "Объект класса SimpleClass"
// "Объект класса SimpleClass2"
```

16.3.2. Тип *Any*

Как уже было отмечено, если тип `AnyObject` может хранить в себе объекты любого класса, то тип `Any` может хранить абсолютно любые типы, даже функциональные. Для того чтобы проверить это, создадим массив с типом `Any` из различных значений:

```
var array: [Any] = [23,4.5,"String",true,(4,"String in tuple"), {() -> String
in return "String in closure" }]
```

Теперь мы можем использовать оператор `as` вместе с условным выражением `switch`:

```
for item in array {
    switch item {
        case let someInt as Int:
            print("Int")
        case let someDouble as Double:
            print("Double")
        case let someString as String:
            print("String")
        case let someBool as Bool:
            print("Bool")
        case let (myTup1, myTup2) as (Int,String):
            print("Tuple (Int,String)")
        case let myClosure as () -> String:
            print("Closure () -> String")
        default:
            print("other type")
    }
}

// "Int"
// "Double"
// "String"
// "Bool"
// "Tuple (Int,String)"
// "Closure () -> String"
```

Выводы

- ❑ С помощью проверки типов можно проверить, принадлежит ли объект соответствующему типу.
- ❑ Оператор проверки типов: `is`.
- ❑ Приведение типов в Swift позволяет через объект родительского класса обращаться к объекту дочернего класса.
- ❑ Операторы приведения типов: `as?` и `as!`.
- ❑ Тип `AnyObject` может хранить объекты любого класса.
- ❑ Тип `Any` может хранить вообще объекты любого типа, в том числе и функционального.

ГЛАВА 17



Расширения

В объектно-ориентированных языках программирования часто приходится расширять функционал какого-либо существующего класса. Например, добавить те или иные новые методы для встроенного класса `Int`. Для этого во многих языках применяют наследование и дочернему классу создают все те же дополнительные методы и свойства.

В Swift для этого можно использовать *расширения* (extensions). Они позволяют без применения наследования расширить функционал класса или структуры. И для этого нам не нужно иметь доступ к исходному коду класса, который мы расширяем.

Важно понимать, что в расширения мы не можем переопределять существующие методы и свойства. Мы можем только добавлять новые.

Для того чтобы написать расширение, мы пишем ключевое слово `extension`, а затем название типа, который хотим расширить:

```
extension Int {  
    // расширяемый функционал  
}
```

17.1. Расширение свойств

Давайте попробуем расширить класс `Int` новым свойством. В расширениях могут использоваться только вычисляемые свойства. То есть, хранимые свойства внутри расширений недопустимы. Кроме того, внутри расширений также нельзя писать наблюдатели для свойств.

Поэтому мы создадим новое вычисляемое свойство `square`, которое вычисляет квадрат числа:

```
extension Int {  
    var square: Int {  
        return self * self  
    }  
}
```

Теперь попробуем создать новую переменную с типом `Int` и вызвать это свойство:

```
var myNum = 3
myNum.square // => 9
```

Поскольку литералами типа `Int` являются простые числа, то мы можем вызвать это свойство даже для обычных чисел:

```
3.square // => 9
```

Как можно видеть, получился очень красивый метод, вычисляющий квадрат числа.

17.2. Расширение методов

Приведем теперь пример расширения метода для типа `Int`. Когда мы изучали замыкания, то приводили пример с функцией `repeatTask()`, заданное количество раз повторяющей действия, которые мы ей передавали в виде замыкания. Она принимала два аргумента: количество повторений и замыкание, которое нужно повторить.

Создадим сейчас более красивый вариант этой функции. Для этого напомним метод `repeatTask()` к типу `Int`, который также станет принимать замыкание, но количество повторений будет зависеть от `Int` числа, к которому мы пишем этот метод. То есть если мы напишем `3.repeatTask({ инструкции })`, то метод выполнит инструкции внутри фигурных скобок три раза.

```
extension Int {
    func repeatTask(closure: ()->() ) {
        for i in 0..
```

Здесь мы организовали цикл от 0 до `self` — т. к. `self` является указателем на объект класса `Int`, к которому мы будем применять этот метод. Теперь можно попробовать наш новый метод:

```
3.repeatTask({
    print("Hello")
})
// "Hello"
// "Hello"
// "Hello"
```

Все прекрасно работает. А чтобы стало еще красивее, мы можем убрать круглые скобки, поскольку замыкание `closure` является единственным аргументом функции, и его можно записать как последующее замыкание:

```
3.repeatTask {
    print("Hello")
}
```



```
// "Hello"  
// "Hello"  
// "Hello"
```

В расширениях мы также можем писать мутирующие методы, которые изменяют значения самого объекта или его свойств. Для структур и перечислений мы обязательно должны указывать ключевое слово `mutating` перед объявлением функции, которую хотим сделать мутирующей.

Напишем метод типа `Int`, который увеличивает значение на единицу:

```
extension Int {  
    mutating func inc() {  
        self = self + 1  
    }  
}
```

Метод `inc()`, увеличивающий значения на единицу, мы можем теперь применять для значений типа `Int`. Но следует помнить, что для литеральных значений и констант мутирующие методы мы применять не можем. Поэтому мы будем использовать их для переменных:

```
var myVar = 3  
myVar.inc()  
print(myVar)    // "4"
```

17.3. Расширение инициализаторов

Кроме классов и методов, мы также можем расширять инициализаторы. Но существуют некоторые ограничения для инициализаторов внутри расширений:

- ❑ для классов в расширениях мы можем писать только удобные инициализаторы. Назначенные инициализаторы или деинициализаторы мы для них писать не можем;
- ❑ для структур и перечислений — если мы хотим, чтобы определенная структура имела кроме инициализатора по умолчанию и поэлементного инициализатора еще один инициализатор, то его следует записать внутри расширения.

Выводы

- ❑ Расширения позволяют без применения наследования расширить функционал класса или структуры.
- ❑ Расширения объявляются через ключевое слово `extension`.
- ❑ В расширениях могут использоваться только вычисляемые свойства.
- ❑ В расширениях можно писать и обычные, и мутирующие методы для структур.
- ❑ Для инициализаторов внутри расширений существуют ограничения, отмеченные в разд. 17.3.

ГЛАВА 18



Протоколы

Протоколы в Swift позволяют создавать требования для типов. Они представляют собой простые наборы из методов свойств и других требований, которыми должен обладать любой тип, соответствующий тому или иному протоколу. Для этих требований не нужно писать реализацию — в протоколах пишутся только названия и типы, которых нужно придерживаться.

18.1. Объявление протокола

Чтобы создать простой протокол, нужно использовать ключевое слово `protocol`, а затем записать название протокола с большой буквы:

```
protocol SimpleProtocol {  
    // требования  
}
```

Чтобы указать, что класс соответствует определенному протоколу, нужно написать название этого протокола после названия класса через двоеточие:

```
class SimpleClass: SimpleProtocol {  
    // реализация класса  
}
```

Если класс соответствует нескольким протоколам, то их названия нужно записать через запятую. Если у класса есть родительский класс, сначала пишется его название, а затем указываются протоколы.

18.2. Требования для свойств

Внутри протокола свойства всегда указываются как переменные через ключевое слово `var`. Кроме того, внутри протокола не указывается, должно ли свойство быть хранимым или вычисляемым. Указывается лишь, является ли свойство читаемым и записываемым (имеет блоки `get` и `set`) либо определено только для чтения (имеет

лишь блок `get`). Напомним, что все хранимые свойства являются и читаемыми, и записываемыми. И только вычисляемые свойства могут быть либо только для чтения, либо для чтения и записи.

Создадим сейчас новый протокол `HumanProtocol`, в котором станем проверять различные типы на соответствие тем или иным данным о каком-либо человеке, — например, проверим на наличие три таких свойства: `firstName`, `lastName`, `fullName`:

```
protocol HumanProtocol {
    var firstName: String { get set }
    var lastName: String { get set }
    var fullName: String { get set }
}
```

Все указанные свойства мы записали и как для чтения (`get`), и как для записи (`set`). Теперь все типы, имеющие свойства с такими же именами и возможностью чтения и записи, будут соответствовать протоколу `HumanProtocol`. Проверить работу протокола можно на классе `Person`, который мы создавали в предыдущих главах:

```
class Person: HumanProtocol {
    var firstName: String
    var lastName: String
    var fullName: String {
        get {
            return firstName + " " + lastName
        }
        set {
            var separatedNames = newValue.componentsSeparatedByString(" ")
            firstName = separatedNames[0]
            lastName = separatedNames[1]
        }
    }

    init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }
}
```

Если компилятор не выводит ошибок, значит, класс `Person` соответствует протоколу. Но как только мы прокомментируем блок `set` из свойства `fullName`, то сразу получим ошибку:

```
import Foundation

class Person: HumanProtocol {
    var firstName: String
    var lastName: String
    var fullName: String {
```

```

        get {
            return firstName + " " + lastName
        }
        //      set {
        //          var separatedNames = newValue.componentsSeparatedByString(" ")
        //          firstName = separatedNames[0]
        //          lastName = separatedNames[1]
        //      }
    }

    init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }
}
// Ошибка: Type 'Person' does not conform to protocol 'HumanProtocol'

```

Ошибка происходит потому, что мы в протоколе для свойства `fullName` указали и `get` и `set`, а в классе блок `set` убрали в комментарии.

Кроме требования для свойств объектов класса, мы можем также написать требования для свойств типа. Для этого мы пишем ключевое слово `class` перед объявлением свойства:

```

protocol SimpleProtocol {
    class var classVariable
}

```

Это требование будет также справедливо и для свойств, объявленных через ключевое слово `static` внутри перечислений и структур.

18.3. Требования для методов

Кроме требований для свойств, внутри протоколов мы можем также писать требования и для методов. Требования для методов пишутся аналогично простому объявлению, но без фигурных скобок и тела метода. В протоколах поддерживаются множественные параметры, но не поддерживаются значения по умолчанию для параметров метода.

Для примера мы можем добавить нашему протоколу требование для метода `greeting()`, который будет просто печатать приветствие:

```

protocol HumanProtocol {
    var firstName: String { get set }
    var lastName: String { get set }
    var fullName: String { get set }

    func greeting() -> String
}

```

Обратите внимание, что мы записали тип возвращаемого значения `String`, т. к. наш метод будет возвращать нам строковое значение с приветствием. Теперь, чтобы класс `Person` соответствовал протоколу `HumanProtocol`, нужно в нем написать объявление нового метода `greeting()` с типом возвращаемого значения `String`:

```
import Foundation

class Person: HumanProtocol {
    var firstName: String
    var lastName: String
    var fullName: String {
        get {
            return firstName + " " + lastName
        }
        set {
            var separatedNames = newValue.componentsSeparatedByString(" ")
            firstName = separatedNames[0]
            lastName = separatedNames[1]
        }
    }

    func greeting() -> String {
        return "Привет, меня зовут \(firstName)"
    }

    init(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }
}
```

Никаких ошибок нет — значит, метод написан правильно.

Кроме требования для обычных методов в протоколах можно также писать мутирующие методы. Для этого перед ключевым словом `func` для мутирующего метода нужно записать ключевое слово `mutating`:

```
protocol SimpleProtocol {
    mutating func simpleMutatingFunc() -> Int
}
```

Указав в протоколе, что функция должна быть мутирующей, при ее объявлении нам нужно вписывать ключевое слово `mutating` только для структур. Для классов `mutating` указывать не нужно.

Кроме мутирующих методов, внутри протоколов можно также писать требования для методов типа. Для этого перед объявлением метода нужно записать ключевое слово `class`:

```
protocol SimpleProtocol {  
    class func simpleClassFunc() -> String  
}
```

Это требование будет также справедливо и для методов, объявленных через ключевое слово `static` внутри перечислений и структур.

18.4. Требования для инициализаторов

Внутри протоколов можно писать требования и для инициализаторов. Их следует писать аналогично простому объявлению, только без фигурных скобок и тела инициализатора:

```
protocol SimpleProtocol {  
    init(SimpleParameter: Int)  
}
```

18.5. Протоколы как типы

Протоколы можно указывать как тип переменной или константы. Тогда от них требуется, чтобы они хранили объекты, которые соответствуют этому протоколу.

Для примера объявим два класса: `Animal` со свойством `name` и `Person` со свойством `firstName`:

```
class Person {  
    var name: String  
  
    init() {  
        firstName = ""  
    }  
}  
  
class Animal {  
    var name: String  
  
    init() {  
        name = ""  
    }  
}
```

Объявим также протокол, который требует иметь свойство `firstName: String`:

```
protocol Named {  
    var firstName: String { get set }  
}
```

Теперь мы можем создать переменную с типом `Named`, которая является протоколом:

```
var myVar: Named
```

Если мы попробуем присвоить объект класса `Animal`, то получим ошибку:

```
myVar = Animal() // Ошибка
```

Ошибка говорит о том, что объект класса `Animal` не соответствует протоколу `Named`. Но если мы присвоим объект класса `Person`, то ошибка исчезнет:

```
myVar = Person()
```

18.6. Соответствие протоколу через расширение

Соответствовать протоколу можно и через расширение. Тогда в конце объявления расширения для определенного типа пишется имя протокола, а внутри тела расширения нужно объявить все свойства и методы для соответствия протоколу:

```
protocol SimpleProtocol {  
  
}  
  
extension Int: SimpleProtocol {  
  
}
```

18.7. Наследование протоколов

Протоколы могут наследоваться друг другом. Тогда при объявлении дочернего протокола мы пишем двоеточие и указываем родительский протокол:

```
protocol SimpleParentProtocol {  
  
}  
  
protocol SimpleChildProtocol : SimpleParentProtocol {  
  
}
```

В этом случае все требования из родительского протокола передадутся в дочерний протокол.

18.8. Протоколы только для классов

При разработке приложений иногда будут встречаться случаи, когда протокол требуется проверять на соответствие только для классов, т. к. в нем содержатся требования лишь для ссылочных типов. В этом случае мы можем объявить протокол, который будет работать только с классами. Тогда после двоеточия нам нужно указать ключевое слово `class`:

```
protocol SimpleProtocol: class {  
  
}
```

Если же этот протокол имеет еще и родительский протокол, то их названия указываются через запятую:

```
protocol SimpleChildProtocol : class, SimpleParentProtocol {  
  
}
```

18.9. Сочетание протоколов

Если мы хотим для одного объекта иметь поддержку двух протоколов, то можем записать их через конструкцию сочетания протоколов:

```
protocol SimpleProtocol1 {  
  
}  
  
protocol SimpleProtocol2 {  
  
}  
  
var myVar: protocol<SimpleProtocol1, SimpleProtocol2>
```

В таком случае переменная `myVar` может хранить лишь объекты, удовлетворяющие двум этим протоколам: `SimpleProtocol1` и `SimpleProtocol2`.

18.10. Проверка объекта на соответствие протоколу

С помощью операторов `is` и `as` мы можем проверять объекты на соответствие протоколу и приводить их к определенному типу протокола:

- ❑ оператор `is` возвращает `true`, если объект класса соответствует протоколу, и `false` — если не соответствует;
- ❑ оператор `as?` возвращает опциональное значение типа протокола и возвращает `nil`, если объект класса не соответствует протоколу;
- ❑ оператор `as!` выполняет нисходящее приведение к типу протокола и выводит ошибку, если нисходящее приведение не произошло.

18.11. Расширения протоколов

С обновлением до версии 2.0 в Swift появилась возможность писать расширения для протоколов. С их помощью можно дополнять протоколы новыми свойствами или методами. Внутри расширений протоколов можно писать реализацию для методов, а все объекты, которые соответствуют этому протоколу, получают возможность использовать этот метод:


```
extension SomeProtocol {  
    func newMethodForProtocol() -> Bool {  
        return true  
    }  
}
```

Выводы

- ❑ Протоколы в Swift позволяют создавать требования для типов. Они представляют собой простые наборы из методов свойств и других требований, которыми должен обладать любой тип, соответствующий этому протоколу.
- ❑ Для того чтобы создать протокол, нужно использовать ключевое слово `protocol`.
- ❑ Внутри протокола свойства всегда указываются через `var`. Кроме того, свойствам не задается тип вычисляемых или хранимых, поскольку это определяется наличием у них только геттера или и геттера, и сеттера.
- ❑ Требования для методов пишутся аналогично простому объявлению, но без фигурных скобок и тела метода. В протоколах поддерживаются множественные параметры, но не поддерживаются значения по умолчанию для параметров метода.
- ❑ Внутри протоколов можно также писать требования и для инициализаторов. Их нужно записывать аналогично простому объявлению, только без фигурных скобок и тела инициализатора.
- ❑ Протоколы можно указывать как тип переменной или константы. Тогда от них требуется, чтобы они хранили объекты, которые соответствуют этому протоколу.
- ❑ Соответствовать протоколу можно и через расширение.
- ❑ Протоколы могут наследоваться друг другом. Тогда при объявлении дочернего протокола мы пишем двоеточие и указываем родительский протокол.
- ❑ Если мы хотим для одного объекта иметь поддержку двух протоколов, то можем записать их через конструкцию сочетания протоколов.
- ❑ Мы можем объявить протокол, который будет работать только с классами.
- ❑ Оператор `is` возвращает `true`, если объект класса соответствует протоколу, и `false` — если не соответствует.
- ❑ Оператор `as?` возвращает опциональное значение типа протокола и возвращает `nil`, если объект класса не соответствует протоколу.
- ❑ Оператор `as!` выполняет нисходящее приведение к типу протокола и выводит ошибку, если нисходящее приведение не произошло.
- ❑ С помощью расширений протоколов можно создавать для протоколов собственные методы и свойства.

ГЛАВА 19



Обобщенные типы

В строго типизированных языках программирования существует распространенная проблема с дублированием одинаковых функций, отличающихся только типом, с которым они работают. Такова, например, функция `printArray`, которая печатает значения из массива `Int`:

```
func printArray(array: [Int]){
    for item in array {
        print(item , appendNewLine: false)
        print(" | " , appendNewLine: false)
    }
}
```

Согласитесь, что функция, которая печатает значения из массива `String`, будет выглядеть аналогично:

```
func printArray(array: [String]){
    for item in array {
        print(item , appendNewLine: false)
        print(" | " , appendNewLine: false)
    }
}
```

Различие здесь определяется только типом массива, который будет приниматься в виде параметра. Это означает, что мы должны были бы писать для каждого типа массива свою функцию с таким же алгоритмом. Чтобы избежать подобных ситуаций, в Swift есть очень важная особенность — *обобщенные типы*.

Сами того не зная, мы уже сталкивались в Swift с обобщенными типами. Массивы и словари тоже являются обобщенными типами коллекций — мы хранили в них значения одного какого-либо типа.

19.1. Обобщенные функции

Обобщенные функции могут работать с любым типом. Для примера напомним обобщенную версию функции `printArray()`, рассмотренную ранее:

```
func printArray<T>(array: [T]){
    for item in array {
        print(item , appendNewLine: false)
        print(" | " , appendNewLine: false)
    }
}
```

Чтобы написать обобщенную функцию, мы сначала в угловых скобках после названия функции пишем указатель типа. Название этого указателя принято писать как простую букву `T`, но оно может быть и другим. Затем мы просто ставим этот указатель в тех местах, где у нас был написан тип. Когда функция уже будет работать со значениями определенного типа, компилятор заполнит места с указателем `T` нужным типом.

Теперь можно попробовать создать два массива из значений типа `Int` и из значений типа `String`, а затем применить нашу функцию к этим массивам:

```
var arrayOfInts = [2,4,2,5,125,2,52,2,3]
var arrayOfStrings = ["one","two","three","four"]

printArray(arrayOfInts)
printArray(arrayOfStrings)

// 2 | 4 | 2 | 5 | 125 | 2 | 52 | 2 | 3 | one | two | three | four |
```

Все правильно — обе функции напечатали нам значения своих элементов. Так как мы использовали функцию `print(_:appendNewLine:)` с параметром `appendNewLine`, равным `false`, то функции напечатали значения в ряд.

19.2. Обобщенные типы

В начале главы мы отметили, что встроенные типы словарей и массивов тоже являются обобщенными типами. Подобно им, мы можем создавать свои собственные обобщенные типы, которые могут работать с любыми типами.

Давайте создадим свой собственный класс `NameValue`, который будет хранить пару: имя и значение любого типа:

```
class NameValue<T,U> {
    var name: T
    var value: U

    init(name: T, value: U) {
        self.name = name
        self.value = value
    }
}
```

Теперь мы можем создавать объекты этого класса с любым типом параметров:

```
var broomstick = NameValue(name: "Nimbus", value: 3000)
var adress = NameValue(name: "Maine", value: "Toluca Lake")
```

19.3. Ограничения типов

В предыдущем примере мы рассмотрели обобщенный тип, который может хранить в себе значения любого одного типа. Но иногда полезно ограничивать типы значений для обобщенных типов. Мы можем это сделать, указав тип через двоеточие:

```
func simpleFunction<T: SimpleClass, P: SimpleProtocol>(simpleT: T, simpleP: P)
{
    // тело функции
}
```

Здесь мы ограничили тип для указателя `T` только типом `SimpleClass`, а для типа `P` — только протоколом `SimpleProtocol`.

Выводы

- ❑ Обобщенные типы могут работать с любыми типами.
- ❑ Для того чтобы функция или тип были обобщенными, достаточно в угловых скобках (`< >`) написать указатель на тип и заполнить в теле функции или типа все места, где должен быть написан тип.
- ❑ Для обобщенных типов мы можем писать ограничения.

ГЛАВА 20



Обработка ошибок

Поскольку наши функции могут выполняться с ошибками, начиная со Swift 2.0 мы можем обрабатывать эти ошибки. Если у нас есть подозрение, что функция может вернуть ошибку, мы можем написать перед открывающейся фигурной скобкой ключевое слово `throws`:

```
func someFunction() throws -> String
```

Для того чтобы удобнее было обрабатывать ошибки, мы можем создавать свои *указатели ошибок*. Указатели ошибок в Swift являются простыми перечислениями, которые должны соответствовать протоколу `ErrorType`:

```
enum WindowsMachineError: ErrorType {  
    case OutOfMemory  
    case RegistryError  
    case BlueScreen  
}
```

Теперь мы можем написать функцию, которая бы могла вернуть ошибку:

```
func windowsStart(registryClean: Bool) throws {  
    guard registryClean == true else {  
        throw WindowsMachineError.RegistryError  
    }  
}
```

В этой функции, если параметр `registryClean` равняется `true`, ошибки не происходит, а если `false` — происходит ошибка `WindowsMachineError.RegistryError`. Указатель `throw` останавливает поток выполнения и выводит ошибку. Чтобы мы могли обрабатывать ошибки и писать сообщения об ошибках, нам нужно воспользоваться конструкцией `do-try-catch`:

```
do {  
    try windowsStart(false)  
} catch WindowsMachineError.OutOfMemory {  
    print("Недостаточно памяти")  
}
```

```
} catch WindowsMachineError.RegistryError {  
    print("Ошибка реестра")  
}  
} catch WindowsMachineError.BlueScreen {  
    print("Синий экран смерти")  
}  
}  
// => "Ошибка реестра"
```

Оператор `try` проверяет функцию на ошибки. Если функция выполнялась с ошибкой, то оператор `try` начинает сопоставлять полученную ошибку с имеющимися вариантами `catch`. Как только он находит нужный вариант `catch`, он выполняет инструкции, записанные в этом блоке `catch`.

Если мы уверены, что функция точно не вернет ошибку, то можно воспользоваться принудительным оператором `try`, который пишется с восклицательным знаком в конце:

```
try! windowsStart(true)
```

Здесь мы уверены, что функция не вернет ошибку, так как оператор `registryClean` равняется `true`.

Выводы

- ❑ Если мы хотим проверить функцию на ошибки, то пишем после названия функции ключевое слово `throws`.
- ❑ Указатели ошибок в Swift являются простыми перечислениями, которые должны соответствовать протоколу `ErrorType`.
- ❑ Указатель `throw` останавливает поток выполнения и выводит ошибку.
- ❑ Чтобы мы могли обрабатывать ошибки и писать сообщения об ошибках, нам нужно воспользоваться конструкцией `do-try-catch`.

ГЛАВА 21



Расширенные операторы

Кроме операторов, рассмотренных нами в *главе 4*, в Swift существуют еще несколько групп операторов, которые относятся к *расширенным операторам*. В этой главе, кроме них, мы рассмотрим еще и объявление собственных реализаций для существующих операторов, или *перегрузку операторов*.

21.1. Оператор объединения по нулевому указателю

Для того чтобы облегчить работу с опциональными типами, существует оператор объединения по нулевому указателю:

```
a ?? b
```

Этот оператор извлекает значение из опционального типа *a*, если он содержит значение, и возвращает *b*, если *a* содержит *nil*. В этом выражении *a* всегда должно быть опциональным, а *b* должно иметь тип, который хранит в себе *a* при извлечении.

Оператор объединения по нулевому указателю является сокращенной записью следующего выражения:

```
a != nil ? a! : b
```

Здесь мы использовали тернарный оператор для извлечения опционального значения, если значение *a* не *nil*, и возвращения *b*, если *a* равно *nil*.

21.2. Операторы с переполнением

Если мы попробуем прибавить значение числу, которое не может хранить значений больше, чем в нем уже есть, то получим ошибку — по умолчанию Swift не позволяет этого делать. Но для того чтобы это было возможно, существуют *операторы переполнения*, которые позволяют записать в переменную неправильное значение:

- ❑ сложение с переполнением: `&+`;
- ❑ вычитание с переполнением: `&-`;
- ❑ умножение с переполнением: `&*`.

21.2.1. Переполнение значения

Приведем пример, когда значение может быть переполнено:

```
var a = UInt8.max
// a равен 255, которое является наибольшим значением, которое может хранить UInt8
a = a &+ 1
// a теперь равен 0
```

21.2.2. Потеря значения

Кроме переполнения значения, мы также можем вычесть число из минимального значения. Тогда это приведет к потере значения:

```
var a = UInt8.min
// a равен 0, которое является наименьшим значением, которое может хранить UInt8
a = a &- 1
// a теперь равен 255
```

А если мы будем использовать знаковый тип, то значение опустится до отрицательного:

```
var a = Int8.min
// a равен -128, которое является наименьшим значением, которое может хранить Int8
a = a &- 1
// a теперь равен 127
```

21.3. Перегрузка операторов

Операторы для классов и структур могут иметь свои собственные реализации. Эта техника известна как *перегрузка операторов*. По умолчанию вновь созданные классы и структуры не умеют работать со стандартными операторами, поэтому для каждого оператора мы можем написать свою собственную реализацию.

Для примера создадим структуру `Color`, описывающую цвет и имеющую три свойства: `red`, `green` и `blue`:

```
struct Color {
    var red: UInt8
    var green: UInt8
    var blue: UInt8
}
```


Теперь напишем собственную реализацию оператора сложения для двух структур `Color`. Для этого нам нужно воспользоваться ключевым словом `func`:

```
func + (left: Color, right: Color) -> Color {  
    return Color(red: left.red + right.red, green: left.green + right.green,  
blue: left.blue + right.blue)  
}
```

Нужно отметить, что операторы в Swift являются теми же функциями, только записанными символами, и называются они *операторами-функциями*. Эту тему мы затрагивали, когда изучали замыкания. Поэтому мы здесь объявили глобальную функцию, которая принимает два параметра: `left` и `right`, описывающих левый операнд и правый операнд. Внутри тела функции мы складываем свойства структур вместе.

Теперь создадим два объекта этой структуры и попытаемся сложить их вместе:

```
var red = Color(red: 255, green: 0, blue: 0)  
var green = Color(red: 0, green: 255, blue: 0)  
var yellow = red + green // => {red 255, green 255, blue 0}
```

Как мы видим, оператор-функция вернула нам новую структуру со свойствами, являющимися суммами свойств двух предыдущих структур.

Рассмотрим еще один пример перегрузки оператора сложения с присвоением: `+=`. Для него нам нужно использовать сквозные параметры:

```
func += (inout left: Color, right: Color) {  
    left = left + right  
}
```

21.4. Побитовые операторы

Для облегчения работы с бинарными значениями Swift предоставляет *побитовые операторы*.

21.4.1. Побитовый оператор *NOT*

Побитовый оператор `NOT` (`~`) инвертирует все биты в числе:

```
var a: UInt8 = 0b00001111  
var b = ~a // равно 11110000
```

21.4.2. Побитовый оператор *AND*

Побитовый оператор `AND` (`&`) объединяет биты двух чисел:

```
var a: UInt8 = 0b11111100  
var b: UInt8 = 0b00111111  
var c = a & b // равно 00111100
```

21.4.3. Побитовый оператор OR

Побитовый оператор OR (`|`) сравнивает биты двух чисел:

```
var a: UInt8 = 0b10110010
var b: UInt8 = 0b01011110
var c = a | b // равно 11111110
```

21.4.4. Побитовый оператор XOR

Побитовый оператор XOR (`^`), называемый еще *исключающим* побитовым оператором OR, также сравнивает биты двух чисел, только исключает несовпадения:

```
var a: UInt8 = 0b00010100
var b: UInt8 = 0b000000101
var c = a ^ b // равно 00010001
```

21.4.5. Побитовые операторы левого и правого сдвига

Побитовые операторы левого (`<<`) и правого сдвига (`>>`) сдвигают все биты числа влево или вправо:

```
var a: UInt8 = 4 // 00000100 в двоичном формате
a << 1 // 00001000
a << 2 // 00010000
a << 5 // 10000000
a << 6 // 00000000
a >> 2 // 00000001
```

Выводы

- ☐ Оператор объединения по нулевому указателю (`a ?? b`) извлекает значение из опционального типа `a`, если он содержит значение, и возвращает `b`, если `a` содержит `nil`.
- ☐ Сложение с переполнением: `&+`.
- ☐ Вычитание с переполнением: `&-`.
- ☐ Умножение с переполнением: `&*`.
- ☐ Мы можем создавать собственные реализации для существующих операторов. Это называется перегрузкой операторов.
- ☐ Побитовый оператор NOT: `~`.
- ☐ Побитовый оператор AND: `&`.
- ☐ Побитовый оператор OR: `|`.
- ☐ Побитовый оператор XOR: `^`.
- ☐ Побитовые операторы левого `<<` и правого сдвига `>>`.

Заключение

Поздравляем! Вы добрались до конца этой книги. Теперь можно смело сказать, что вы достаточно хорошо разбираетесь в Swift. Завершение изучения материала книги ляжет в основу ваших новых проектов на языке программирования Swift.

Но перед тем как с вами попрощаться, мы бы хотели порекомендовать вам, куда двигаться дальше.

Изучайте фреймворки Apple

Вторым этапом на пути к разработке приложений на iOS и OS X является изучение фреймворков Apple. Cocoa и Cocoa Touch сочетают в себе множество фреймворков и библиотек для полноценной разработки приложений на iOS и OS X. Их изучение поможет вам получить доступ к API операционной системы, к библиотекам работы с анимацией, 3D-сценами, сетевыми функциями.

Вступайте в Apple's Developer Program

Если вы планируете писать приложения для публикации их в AppStore, то вам следует вступить в программу для разработчиков Apple. Кроме возможности публиковать свои приложения в AppStore, вам также будут доступны все последние бета-релизы программного обеспечения и инструментов для разработчика.

Вперед, к новым высотам!

Теперь, когда вы знаете, что изучать дальше, мы можем с вами прощаться. Надеемся, эта книга была вам полезна для изучения Swift.

Удачи в покорении новых высот со Swift!