Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Информационные технологии»

Т. С. Семенченя

РАЗРАБОТКА ПРИЛОЖЕНИЙ ДЛЯ IPHONE И IPAD

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ
для студентов специальности
1-40 05 01 «Информационные системы и технологии
(по направлениям)» направления специальности
1-40 05 01-01 «Информационные системы и технологии
(в проектировании и производстве)»
дневной и заочной форм обучения

Гомель 2022

УДК 004.438(075.8)
ББК 32.973я73
С30

Рекомендовано научно-методическим советом
факультета автоматизированных и информационных систем
ГГТУ им. П. О. Сухого
(*протокол № 10 от 25.06.2021 г.*)

Рецензент: зав. каф. «Промышленная электроника» ГГТУ им. П. О. Сухого
канд. техн. наук, доц. *Ю. В. Крышнев*

Семенченя, Т. С.

С30 Разработка приложений для iPhone и iPad : учеб.-метод. пособие для студентов специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)» направления специальности 1-40 05 01-01 «Информационные системы и технологии (в проектировании и производстве)» днев. и заоч. форм обучения / Т. С. Семенченя. – Гомель : ГГТУ им. П. О. Сухого, 2022. – 132 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

Представлен теоретический материал по учебной дисциплине «Разработка приложений для iPhone и iPad», который знакомит студентов с языками программирования для iPhone и iPad, структурой приложения для iPhone, а также с основами языка Swift.

Для студентов специальности 1-40 05 01 «Информационные системы и технологии (по направлениям)» направления специальности 1-40 05 01-01 «Информационные системы и технологии (в проектировании и производстве)» дневной и заочной форм обучения.

УДК 004.438(075.8)
ББК 32.973я73

ВВЕДЕНИЕ

Лекционный материал знакомит студентов с основами разработки приложений для устройств iOS – iPhone и iPad. Система iOS представляет собой перспективную платформу, демонстрирующую бурный рост, начиная с ее появления в 2007 года. Распространение мобильных устройств означает, что люди могут использовать программное обеспечение везде, где они захотят, с помощью как телефонов, так и других аксессуаров. С появлением системы iOS 10, среды Xcode 8, языка Swift 3 и последней версии комплекта для разработки программного обеспечения (SDK) возможности стали еще более захватывающими и простыми.

Целью дисциплины «Разработка приложений для iPhone и iPad» является создание у студентов обучаемых компетенций в области разработки и распространений мобильных приложений для устройств, работающих под управлением операционной системы iOS.

Задачами дисциплины «Разработка приложений для iPhone и iPad» являются:

– знакомство с современными языками программирования мобильных приложений для iOS;

– получение практических навыков разработки, тестирования и публикации мобильных приложений.

1. ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ ДЛЯ ПЛАТФОРМЫ IPHONE

Сегодня использование современных смартфонов для решения возникающих задач стало нормой. В связи с этим многие компании обращают все более пристальное внимание на обеспечение функционального доступа к предлагаемым ими услугам посредством мобильных приложений.

На сегодняшний день существует три самые популярные операционные системы для мобильных телефонов и планшетов: Android, IOS и Windows.

Среди мобильных телефонов лидером является Android – 70,43%, мобильная операционная система iOS – 29,06% – практически каждый третий смартфон. На планшетах лидирующей является iOS – 59,97%, а у Android 39,79% всех планшетов. Исходя из статистики можно сделать выводы, что на сегодняшний день iOS является одной из популярнейших мобильных операционных систем в мире, и в такой ситуации спрос на мобильное программирование под данную операционную систему растет небывалыми темпами.

Для разработки под iOS обязательно нужна операционная система MacOS, а также существуют две самые популярные среды разработки приложений на iOS – это Xcode от Apple и AppCode от JetBrains.

Главный инструмент разработчика под iOS – среда программирования Xcode.

Xcode – это интегрированный программный продукт компании Apple для разработки программного обеспечения для платформ: iOS, macOS, WatchOS и tvOS. Xcode уникален и поддерживает целый ряд технологий, он содержит все, что необходимо разработчику: интуитивно понятный редактор кода с подсветкой синтаксиса, расширенные возможности отладки программ, простой, но многофункциональный интерфейс и многое другое.

Interface Builder – это интерфейсный редактор, который позволяет проектировать и создавать полный пользовательский интерфейс мобильных приложений. Является составной частью IDE Xcode. Компания Apple позаботилась о том, чтобы интерфейс Xcode был дружелюбен и понятен (рисунок 1).

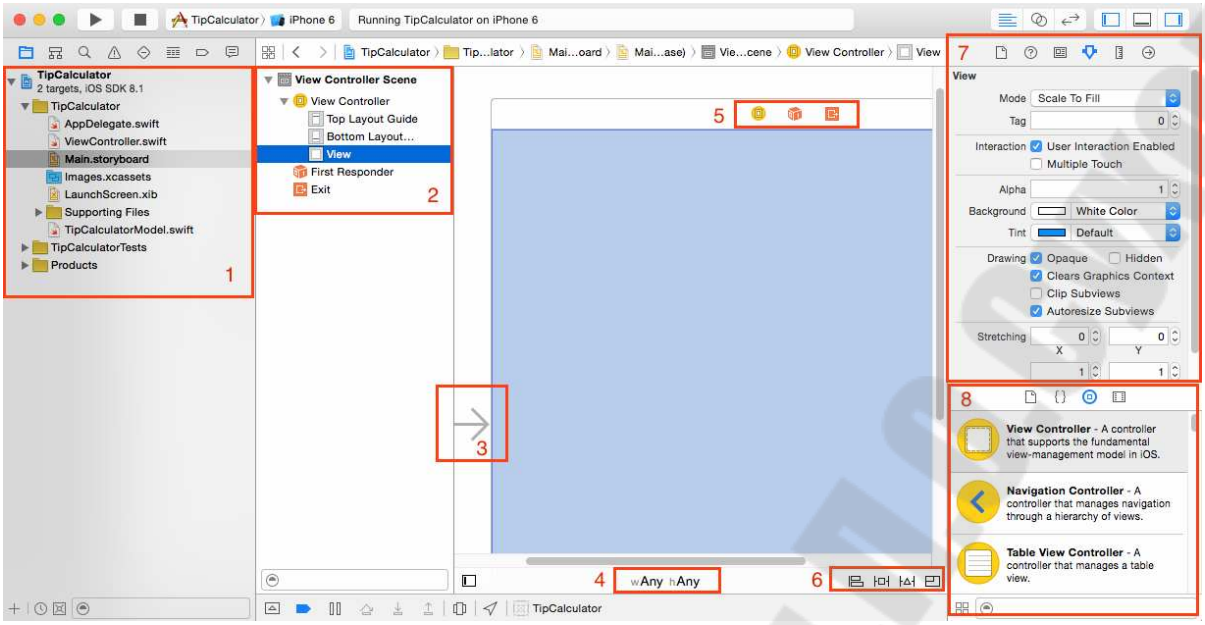


Рисунок 1 – Интерфейс

На рисунке 1 цифрами обозначены отдельные элементы программы:

– В левом краю расположен Project Navigator (навигатор проекта), где отображаются все файлы проекты, которые могут содержать программный код или какие-либо настройки.

– Слева в Interface Builder располагается схема документа, где можно посмотреть все визуальные элементы, добавленные в окно приложения, которые будут отображаться на экране устройств. Нажав на стрелочку рядом с «View Controller», можно увидеть иерархию элементов на текущем «экране». На рисунке 1 всего один View Controller с одним пустым View.

– Данная стрелка расположена слева от главного окна View Controller и указывает на то, что данный лист или «экран» является входным, то есть появляется при первоначальной загрузке приложения на устройстве или эмуляторе.

– Внизу Interface Builder видны надписи «w Any», «h Any». Эти надписи значат, что в данный момент редактируется внешний вид приложения, которое будет работать в интерфейсе любого размера.

– Наверху маленькие иконки отображают, соответственно, окно самого View Controller'a, First Responder, Exit.

– Внизу справа Interface Builder'a четыре иконки для Auto Layout.

– Справа вверху Interface Builder'а располагается Inspectors (инспекторы) для выбранного вами элемента. Здесь легко можно настраивать атрибуты элементов интерфейса, таких как: шрифт, размер текста, цвет текста, тени и т.д.

– Внизу в панели справа находится библиотека объектов (Libraries), в которой отображаются различные элементы интерфейса, которые можно использовать в проекте.

Левую, правую и нижние панели легко можно скрыть кнопками, расположенными в правом верхнем углу, чтобы сосредоточиться на процессе написания программы. Левая панель может отображать разную информацию: когда активна первая иконка, то панель отображает все файлы, которые относятся к текущему проекту.

Настройки проекта Xcode и файлы проекта

Если кликнуть по названию проекта, то можно перейти к основным настройкам проекта (рисунок 2). В настройках проекта всегда можно изменять некоторые параметры, такие как: версия приложения, разработчик, версия iOS, для которой ведется разработка, устройства, для которых ведется разработка и многое другое.

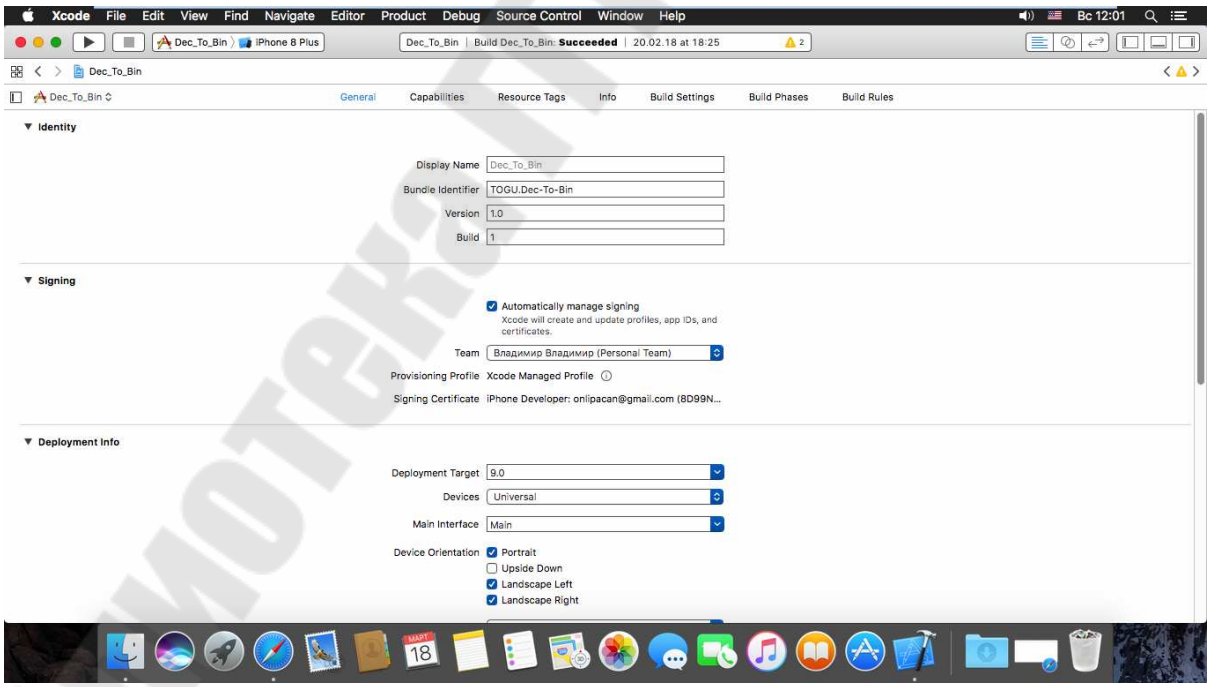


Рисунок 2 – Меню настроек проекта

Файл проекта «AppDelegate.swift» содержит в себе специальный программный код с функциями. Этот файл является важным и содержит следующие некоторые методы:

– методы, необходимые для конфигурации и загрузки приложения в память;

– методы, которые должны вызываться при сворачивании приложения или, например, при входящем звонке;

– методы, которые срабатывают при переходе в фоновый режим работы приложения;

– методы обновления информации приложения при переходе в активный режим работы;

– методы удаления приложения из памяти при закрытии его пользователем.

Файл проекта «ViewController.swift» содержит в себе основной программный код любого приложения, написанного в Xcode.

Файл проекта «Main.storyboard» является основным при создании интерфейса приложения, ведь именно в нём содержится информация о расположении различных элементов (текстовых полей, кнопок, количества окон и др.). Из библиотеки объектов легко можно перетаскивать элементы будущего интерфейса (label, button, text field и многие др.) и помещать на экран приложения.

Например, на рисунке 3 можно увидеть, что на главном экране приложения добавлены из библиотеки: два TextField, три Lable, Button и Switch. К этим элементам можно привязывать код или выводить какую-либо информацию, а также изменять их размеры, перемещать и менять различные свойства: цвет, шрифт и т.д.

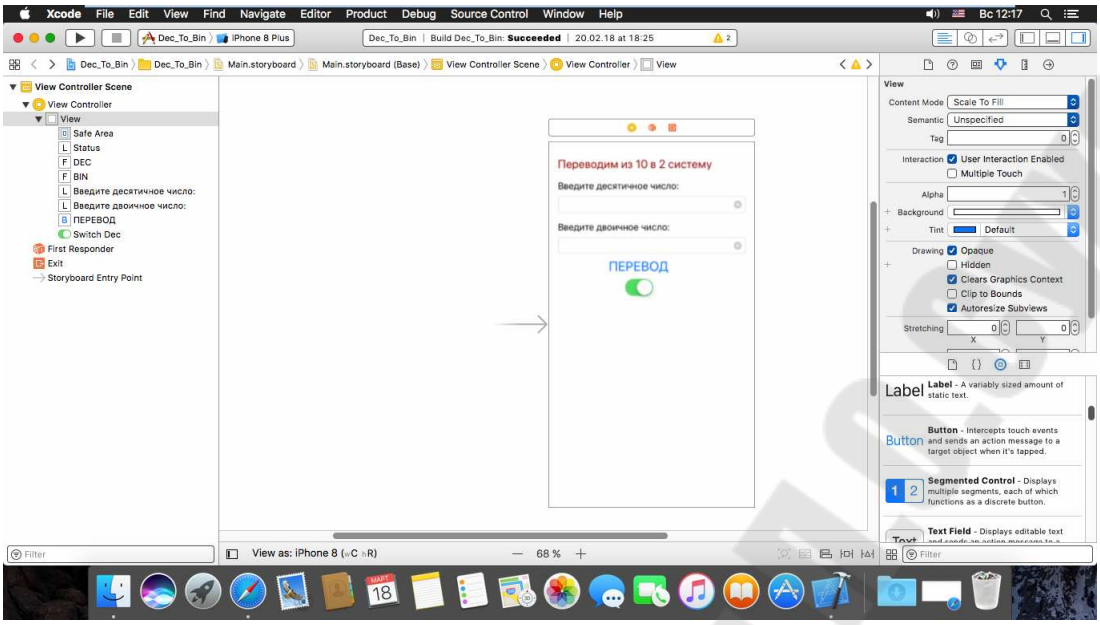


Рисунок 3 – Пример экрана приложения

Так как в Xcode можно создать универсальное приложение для устройств с разным размером экрана, то существуют и инструменты, которые помогают настроить адаптацию под различные размеры экрана и ориентации устройства (альбомные или портретные).

Файл проекта «images. xcassets» содержит в себе иконки и изображения, которые в дальнейшем будут использоваться в проекте.

Файл проекта «LaunchScreen.xib» – это то, что показывается пользователю при загрузке приложения. Здесь можно добавлять любые элементы из библиотеки объектов, которые будут создавать общий вид, являющийся, по сути, загрузочной заставкой приложений.

Создание проекта и шаблоны в Xcode

Создать проект в среде Xcode несложно. Чтобы это сделать, необходимо запустить Xcode и выбрать «Create a new Xcode project» на первоначальном экране приветствия.

Apple предоставляет по умолчанию множество различных шаблонов проектов для iOS, каждый из которых полезен для начала работы над различными проектами:

– Шаблон Single View Applications самый простой и является отправной точкой для многих приложений. Создаётся совершенно пустой интерфейс, который может изменяться разработчиком по своему усмотрению.

– Шаблон Master-Detail App подходит, если планируется, что приложение будет активно использовать элемент UITableView. По умолчанию Xcode создаёт проект, который имеет вид таблицы, куда пользователь может добавить строку, используя в правом углу панель навигации.

– Шаблон Page-Based Application создаёт страницу-книгу, позволяя пользователю перелистывать страницы. Такой шаблон подходит, если планируется создать приложение, которое отображает информацию в книжном формате, т.е. пользователь будет просматривать страницы свайпом влево или вправо.

– Шаблон Tabbed Applications создаёт панель вкладок и две готовые вкладки. При переключении вкладок происходит переход к разным частям приложения.

– Шаблон Games позволяет создавать игры, используя технологии Apple OpenGL ES, GLKit, Scene Kit и Metal.

Для создания базового приложений подойдёт шаблон Single View. После того, как шаблон выбран, необходимо нажать кнопку «Next» для перехода на следующее окно (рисунок 4).

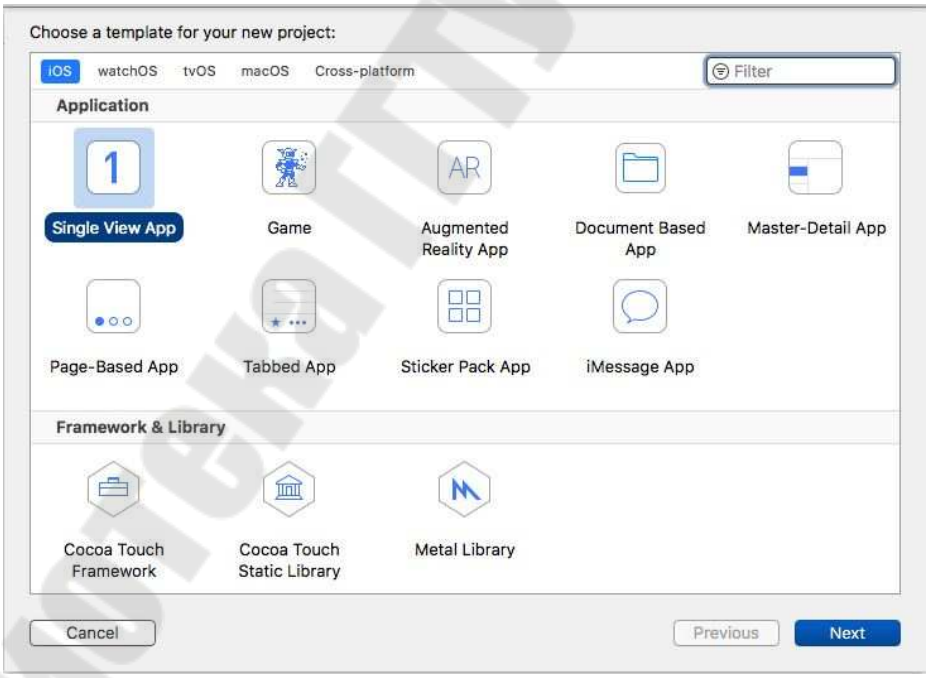


Рисунок 4 – Создание базового приложения

Далее необходимо заполнить поля (рисунок 5).

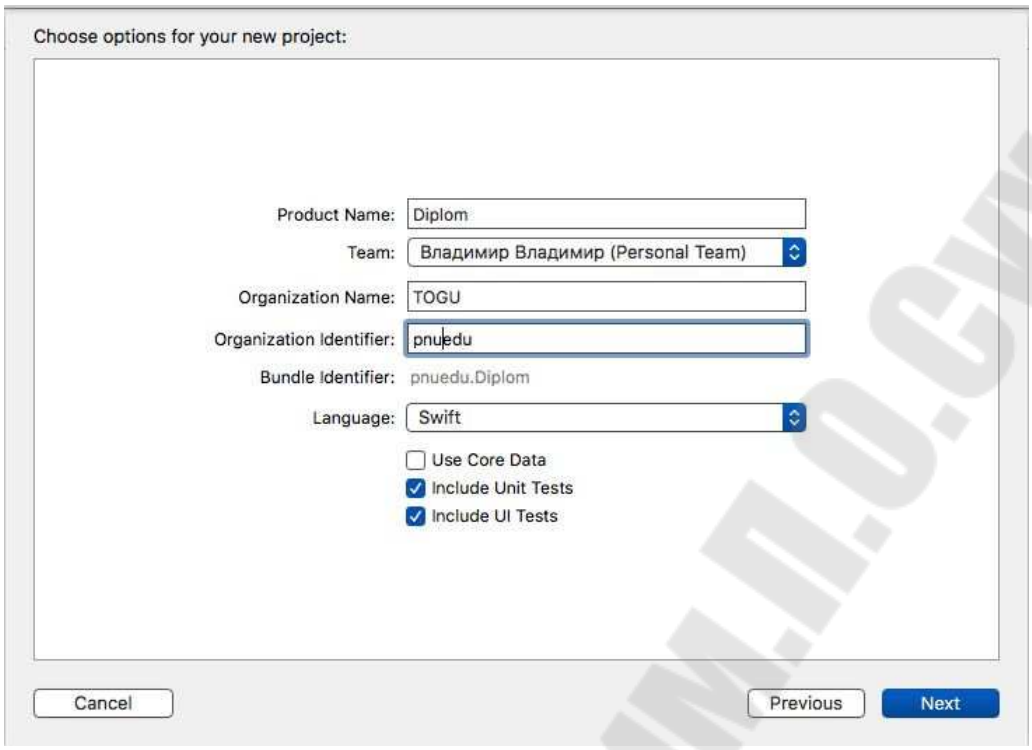


Рисунок 5 – Окно при создании приложения

Product Name (Название продукта) – название будущего приложения. Название проекта в будущем можно изменить во время разработки.

Team (Команда) – Название команды разработчиков, занимающихся разработкой приложений. В данном случае выбран сертификат одного человека «персональной команды».

Organization Name (Название организации) и Organization Identifier (Идентификатор организации) – поля требуются для указания компании, которая разрабатывает программное обеспечение. Необходимо для того, чтобы подать заявку в App Store для публикации приложения. Идентификатор организации в стиле обозначения обратного имени домена. Например, если название организации «togu», то идентификатор организации будет «com.togu».

Language (Язык) – поле со списком, состоящее из двух языков, на одном из которых будет происходить разработка мобильного приложения: Swift или Objective C.

2. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ SWIFT

1.1 Константы и переменные

Константы и переменные связывают имя со значением определенного типа. Значение константы не может быть изменено после его установки, тогда как переменной может быть установлено другое значение в будущем.

Объявление констант и переменных

Константы и переменные должны быть объявлены, перед тем как их использовать. Константы объявляются с помощью ключевого слова let, а переменные с помощью var. Вот пример того, как константы и переменные могут быть использованы для отслеживания количества попыток входа, которые совершил пользователь:

```
let maximumNumberOfLoginAttempts = 10
var currentLoginAttempt = 0
```

Этот код можно прочесть как:

«Объяви новую константу с именем maximumNumberOfLogin Attempts, и задай ей значение 10. Потом, объяви новую переменную с именем currentLoginAttempt, и задай ей начальное значение 0.»

В этом примере максимальное количество доступных попыток входа объявлено как константа, потому что максимальное значение никогда не меняется. Счетчик текущего количества попыток входа объявлен как переменная, потому что это значение должно увеличиваться после каждой неудачной попытки входа.

Вы можете объявить несколько констант или несколько переменных на одной строке, разделяя их запятыми:

```
var x = 0.0, y = 0.0, z = 0.0
```

Аннотация типов

Вы можете добавить обозначение типа, когда объявляете константу или переменную, чтобы иметь четкое представление о типах значений, которые могут хранить константы или переменные. Написать обозначение типа, можно поместив двоеточие после имени

константы или переменной, затем пробел, за которым следует название используемого типа.

Этот пример добавляет обозначение типа для переменной с именем welcomeMessage, чтобы обозначить, что переменная может хранить String:

```
var welcomeMessage: String
```

Вы можете создать несколько переменных одного типа в одной строке, разделенной запятыми, с одной аннотацией типа после последнего имени переменной:

```
var red, green, blue: Double
```

Название констант и переменных

Вы можете использовать почти любые символы для названий констант и переменных, включая Unicode-символы:

```
let π = 3.14159
let 你好 = "你好世界"
let ?? = "dogcow"
```

Имена констант и переменных не могут содержать пробелы, математические символы, стрелки, приватные (или невалидные) кодовые точки Unicode, а так же символы отрисовки линий или прямоугольников. Так же имена не могут начинаться с цифр, хотя цифры могут быть включены в имя в любом другом месте. Если вы объявили константу или переменную определенного типа, то вы не можете объявить ее заново с тем же именем или заставить хранить внутри себя значение другого типа. Также вы не можете изменить константу на переменную, а переменную на константу.

Вы можете изменить значение переменной на другое значение совместимого типа. В примере ниже значение friendlyWelcome изменено с “Hello!” на “Bonjour!”:

```
var friendlyWelcome = “Hello!”
friendlyWelcome = “Bonjour!”
// теперь friendlyWelcome имеет значение “Bonjour!”
```

В отличие от переменных, значение константы не может быть изменено, после того, как было установлено. Если вы попытаетесь его изменить, то будет выведена ошибка компиляции:

```
let languageName = "Swift"
languageName = "Swift++"
// Это ошибка компилляции: languageName cannot be changed (значение languageName не может быть изменено).
```

Печать констант и переменных

Вы можете напечатать текущее значение константы или переменной при помощи функции print(_:separator:terminator:):

```
print(friendlyWelcome)
// Выведет "Bonjour!"
```

Функция print(_:separator:terminator:) является глобальной, которая выводит одно или более значений в подходящем виде. В Xcode, например, функция print(_:separator:terminator:) выводит значения в консоль. Параметры separator и terminator имеют дефолтные значения, так что при использовании функции их можно просто пропустить. По умолчанию функция заканчивает вывод символом переноса строки. Чтобы вывести в консоль значения без переноса на новую строку, вам нужно указать пустую строку в параметре terminator, например, print(someValue, terminator: "").

Комментарии

Используйте комментарии, чтобы добавить неисполняемый текст в коде, как примечание или напоминание самому себе. Комментарии игнорируются компилятором Swift во время компиляции кода.

Комментарии в Swift очень похожи на комментарии в C. Однострочные комментарии начинаются с двух слешей (//):

```
// это комментарий
```

Вы также можете написать многострочные комментарии, которые начинаются со слеша и звездочки (/*) и заканчиваются звездочкой, за которой следует слеш (*/):

```
/* это тоже комментарий,
```

но написанный на двух строках */

В отличие от многострочных комментариев в C, многострочные комментарии в Swift могут быть вложены в другие многострочные комментарии. Вы можете написать вложенные комментарии, начав многострочный блок комментариев, а затем, начать второй многострочный комментарий внутри первого блока. Затем второй блок закрывается, а за ним закрывается первый блок:

```
/* это начало первого многострочного комментария
/* это второго, вложенного многострочного комментария */
это конец первого многострочного комментария */
```

Вложенные многострочные комментарии позволяют закомментировать большие блоки кода быстро и легко, даже если код уже содержит многострочные комментарии.

Точки с запятой

В отличие от многих других языков, Swift не требует писать точку с запятой (;) после каждого выражения в коде, хотя вы можете делать это, если хотите. Однако точки с запятой требуются, если вы хотите написать несколько отдельных выражений на одной строке:

```
let cat = "?"; print(cat)
// Выведет "?"
```

2.2 Типы данных

Каждая переменная или константа хранит в себе значение определенного типа. Например, переменная name хранит строку:

```
var name = "Tom"
```

В языке Swift имеются следующие типы данных:

– Int8: представляет целые числа со знаком размером не более 8 бит (от -128 до 127)

– UInt8: представляет целые положительные числа размером не более 8 бит (от 0 до 255)

– Int16: представляет целые числа со знаком размером не более 16 бит (от -32768 до 32767)

– UInt16: представляет целые положительные числа размером не более 16 бит (от 0 до 65535)

– Int32: представляет целые числа со знаком размером не более 32 бита (от -2147483648 до 2147483647)

– UInt32: представляет целые положительные числа размером не более 32 бита (от 0 до 4294967295)

– Int64: представляет целые числа со знаком размером не более 64 бита (от -9223372036854775808 до 9223372036854775807)

– UInt64: представляет целые положительные числа размером не более 64 бита (от 0 до 18446744073709551615)

– Int: представляет целые числа со знаком, например, 1, -30, 458. На 32-разрядных платформах эквивалентен Int32, а на 64-разрядных - Int64

– UInt: представляет целые положительные числа, например, 1, 30, 458. На 32-разрядных платформах эквивалентен UInt32, а на 64-разрядных - UInt64

– Float: 32-битное число с плавающей точкой, содержит до 6 чисел в дробной части

– Double: 64-битное число с плавающей точкой, содержит до 15 чисел в дробной части

– Float80: 80-битное число с плавающей точкой

– Bool: представляет логическое значение true или false

– String: представляет строку

– Character: представляет отдельный символ

Целые числа

Integer (целое число) – это число, не содержащее дробной части, например, как 42 и -23. Целые числа могут быть либо знаковыми (положительными, ноль или отрицательными) либо беззнаковыми (положительными или ноль).

Swift предусматривает знаковые и беззнаковые целые числа в 8, 16, 32 и 64 битном форматах.

Границы целых чисел

Вы можете получить доступ к минимальному и максимальному значению каждого типа целого числа с помощью его свойств min и max:

```
let minValue = UInt8.min // minValue равен 0, а его тип UInt8
let maxValue = UInt8.max // maxValue равен 255, а его тип UInt8
```

Тип значения этих свойств соответствует размеру числа (в примере выше этот тип UInt8) и поэтому может быть использован в выражениях наряду с другими значениями того же типа.

Int

В большинстве случаев вам не нужно будет указывать конкретный размер целого числа для использования в коде. В Swift есть дополнительный тип целого числа – Int, который имеет тот же размер что и разрядность системы:

- На 32-битной платформе, Int того же размера что и Int32
- На 64-битной платформе, Int того же размера что и Int64

Если вам не нужно работать с конкретным размером целого числа, всегда используйте в своем коде Int для целых чисел.

UInt

Swift также предусматривает беззнаковый тип целого числа – UInt, который имеет тот же размер что и разрядность системы:

- На 32-битной платформе, UInt того же размера что и UInt32
- На 64-битной платформе, UInt того же размера что и UInt64

Числа с плавающей точкой

Число с плавающей точкой – это число с дробной частью, например как 3.14159, 0.1, и -273.15.

Типы с плавающей точкой могут представлять гораздо более широкий спектр значений, чем типы целых значений, и могут хранить числа намного больше (или меньше) чем может хранить Int. Swift предоставляет два знаковых типа с плавающей точкой:

– Double – представляет собой 64-битное число с плавающей точкой. Используйте его когда число с плавающей точкой должно быть очень большим или чрезвычайно точным

– Float – представляет собой 32-битное число с плавающей точкой. Используйте его, когда значение не нуждается в 64-битной точности.

Логические типы

В Swift есть простой логический тип Bool. Этот тип называют логическим, потому что он может быть только true или false. Swift предусматривает две логические константы, true и false соответственно:

```
let orangesAreOrange = true
let turnipsAreDelicious = false
```

Текстовые типы

Для работы с текстом применяются два типа данных: Character и String. Character представляет отдельный символ, а String – строку из нескольких символов. При этом надо отметить, что String – это не просто набор объектов Character, это отдельный тип, который по функциональности отличается.

Самый простой способ определения строки и символов представляет использование строковых литералов - значений в двойных кавычках:

```
var a: Character = "a"
var hello: String = "hello"
```

В отличие от строки в переменную типа Character мы не можем засунуть больше одного символа, то есть в следующем случае мы столкнемся с ошибкой:

```
var a: Character = "abc"
```

Строки могут быть пустыми, то есть по сути не содержать ничего, но при этом быть инициализированными:

```
var str1: String = ""
var str2: String = String()
```

Пустая строка создается при помощи присвоения "" или при помощи инициализатора String()

Явная типизация

Тип переменных и констант может определен явно или неявно. Выше тип определялся неявно. Но мы также можем явным образом определить тип:

```
var age: Int = 32
var name: String = "Tom"
```

Определение переменной с типом происходит по шаблону:

var название_переменной: тип_переменной = значение_переменной

Также мы можем сначала определить переменную, а потом присвоить ей значение:

```
var name: String
name = "Tom"
```

Также можно определить сразу набор однотипных переменных:

```
var height, weight: Double
```

Неявная типизация

Если мы явным образом не указываем тип переменных и констант, то он автоматически выводится системой на основании хранящегося значения. Например:

```
var name = "Tom"
```

Здесь явным образом не указан тип переменной name, однако поскольку она хранит строку, то система будет рассматривать эту переменную как объект типа String. Или, например:

```
var age = 32
```

Здесь также явно не указан тип переменной, поэтому система будет рассматривать эту переменную как объект Int, то есть целое число.

Однако при таком подходе следует учитывать, что Swift не всегда выводит те типы, которые нам могут быть нужны. Например, все целые числа Swift воспринимает как объекты типа Int, а дробные числа - как объекты типа Double. Это надо учитывать, чтобы не попасть в некорректные ситуации. Например:

```
var d = 3.4          // тип Double
var f : Float = 1.2
```

```
d = f                // ! Ошибка - разные типы
```

В данном случае на основании присвоенного значения переменная d будет представлять тип Double, а переменная f представляет тип Float, поэтому при присвоении переменной d значения f мы получим ошибку.

Типобезопасность

Swift является типобезопасным языком со строгой типизацией, поэтому после того, как для переменной будет установлен тип, мы его уже изменить не сможем. Так, в следующем случае мы столкнемся с ошибкой:

```
var age: Int
age = "Tom"
```

Ошибка возникает, так как переменная age ожидает число, а строка "Том" не является числом и не соответствует переменной age по типу.

Мы можем присваивать значение переменной или константы другой переменной:

```
var age: Int = 22
var years = age
```

Базовые операторы

Оператор – это специальный символ или выражение для проверки, изменения или сложения величин. Например, оператор сложения (+) суммирует два числа let i = 1 + 2

Операторы делятся на унарные, бинарные и тернарные:

- Унарные операторы применяются к одной величине (например, -a). Унарные префиксные операторы ставятся непосредственно перед величиной (например, !b), а унарные постфиксные операторы – сразу за ней (например, c!).
- Бинарные операторы применяются к двум величинам (например, 2 + 3) и являются инфиксными, так как ставятся между этими величинами.

– Тернарные операторы применяются к трем величинам. Как и в языке C, в Swift есть только один такой оператор, а именно – тернарный условный оператор (a ? b : c).

Величины, к которым применяются операторы, называются операндами. В выражении 1 + 2 символ + является бинарным оператором, а его операндами служат 1 и 2.

Оператор присваивания

Оператор присваивания (a = b) инициализирует или изменяет значение переменной a на значение b:

```
let b = 10
var a = 5
a = b
// теперь a равно 10
```

Если левая часть выражения является кортежем с несколькими значениями, его элементам можно присвоить сразу несколько констант или переменных:

```
let (x, y) = (1, 2)
// x равно 1, а y равно 2
```

В отличие от C и Objective-C оператор присваивания в Swift не может возвращать значение. К примеру, следующее выражение недопустимо:

```
if x = y {
 // это неверно, так как x = y не возвращает никакого значения
}
```

Эта особенность не позволяет разработчику спутать оператор присваивания (=) с оператором проверки на равенство (==). Благодаря тому, что выражения типа if x = y некорректны, подобные ошибки при программировании на Swift не произойдут.

Арифметические операторы

Язык Swift поддерживает четыре стандартных арифметических оператора для всех числовых типов:

- сложение (+)
- вычитание (-)
- умножение (*)
- деление (/)

```
1 + 2 // равно 3
5 - 3 // равно 2
2 * 3 // равно 6
10.0 / 2.5 // равно 4.0
```

Оператор сложения служит также для конкатенации, или же склейки, строковых значений (тип String):

```
"hello, " + "world"  // равно "hello, world"
```

Оператор целочисленного деления

Оператор целочисленного деления (a % b) показывает, какое количество b помещается внутри a, и возвращает остаток деления a на b.

Оператор целочисленного деления работает следующим образом. Для вычисления выражения 9 % 4 сначала определяется, сколько четверок содержится в девятке (рисунок 6).

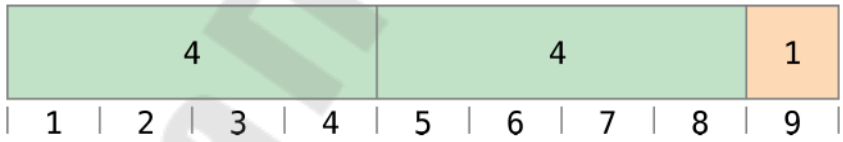


Рисунок 6 – Пример работы оператора целочисленного деления

В одной девятке содержатся две четверки, а остатком будет 1 (выделено оранжевым цветом).

На языке Swift это записывается так:

```
9 % 4    // равно 1
```

Оператор унарного минуса

Для изменения знака числового значения служит префиксный минус -, который называется оператором унарного минуса:

```
let three = 3
```

```
let minusThree = -three // minusThree равно -3
let plusThree = -minusThree // plusThree равно 3, т. е. "минус минус три"
```

Оператор унарного минуса (-) ставится непосредственно перед значением, без пробела.

Оператор унарного плюса

Оператор унарного плюса (+) просто возвращает исходное значение без каких-либо изменений:

```
let minusSix = -6
let alsoMinusSix = +minusSix // alsoMinusSix равно -6
```

Хотя оператор унарного плюса не выполняет никаких действий, он придает коду единообразие, позволяя зрительно отличать положительные значения от отрицательных.

Составные операторы присваивания

Как и в языке C, в Swift имеются составные операторы присваивания, совмещающие простое присваивание (=) с другой операцией. Одним из примеров может служить оператор присваивания со сложением (+=):

```
var a = 1
a += 2
// теперь a равно 3
```

Выражение a += 2 является краткой формой записи a = a + 2. Таким образом, один и тот же оператор выполняет одновременно операцию сложения и присваивания.

Операторы сравнения

Язык Swift поддерживает все стандартные операторы сравнения из C:

- равно (a == b)
- не равно (a != b)
- больше (a > b)
- меньше (a < b)

- больше или равно (a >= b)
- меньше или равно (a <= b)

В языке Swift есть также два оператора проверки на идентичность/тождественность (=== и !==), определяющие, ссылаются ли два указателя на один и тот же экземпляр объекта.

Каждый оператор сравнения возвращает значение типа Bool, указывающее, является ли выражение истинным:

```
1 == 1 // истина, так как 1 равно 1
2 != 1 // истина, так как 2 не равно 1
2 > 1 // истина, так как 2 больше чем 1
1 < 2 // истина, так как 1 меньше 2
1 >= 1 // истина, так как 1 больше либо равно 1
2  <= 1 // ложь, так как 2 не меньше либо равно 1
```

Операторы сравнения часто используются в условных выражениях, включая инструкцию if:

```
let name = "world"
if name == "world" {
 print("hello, world")
} else {
 print("Мне жаль, \(name), но я тебя не узнаю")
}
// напечатает "hello, world", так как name очевидно равно "world"
```

Тернарный условный оператор

Тернарный условный оператор – это специальный оператор из трех частей, имеющий следующий синтаксис: выражение ? действие1 : действие2. Он выполняет одно из двух действий в зависимости от того, является ли выражение true или false. Если выражение равно true, оператор выполняет действие1 и возвращает его результат; в противном случае оператор выполняет действие 2 и возвращает его результат.

Тернарный условный оператор является краткой записью следующего кода:

```
if выражение {
 действие1
```

```
} else {
 действие2
}
```

Ниже приведен пример расчета высоты строки в таблице. Если у строки есть заголовок, то она должна быть выше своего содержимого на 50 точек, а если заголовка нет, то на 20 точек:

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
// rowHeight равно 90
```

В развернутом виде этот код можно записать так:

```
let contentHeight = 40
let hasHeader = true
var rowHeight = contentHeight
if hasHeader {
   rowHeight = rowHeight + 50
} else {
   rowHeight = rowHeight + 20
}
// rowHeight равно 90
```

Логические операторы

Логические операторы изменяют или комбинируют логические значения типа Boolean (булево) – true и false. Язык Swift, как и другие C-подобные языки, поддерживает три стандартных логических оператора:

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

Оператор логического НЕ

Оператор логического НЕ (!a) инвертирует булево значение – true меняется на false, а false становится true.

Оператор логического НЕ является префиксным и ставится непосредственно перед значением, без пробела. Как видно из следующего примера, его можно воспринимать как "не a":

```
let allowedEntry = false
if !allowedEntry {
 print("ACCESS DENIED")
}
// Выведет "ACCESS DENIED"
```

Конструкция if !allowedEntry означает «если не allowedEntry». Идущая за ней строка будет выполнена, только если «не allowedEntry» является истиной, т. е. если allowedEntry равно false.

Как видно из этого примера, удачный выбор булевой константы и имен переменных делает код коротким и понятным, без двойных отрицаний и громоздких логических выражений.

Оператор логического И

Оператор логического И (a && b) дает на выходе true тогда и только тогда, когда оба его операнда также равны true.

Если хотя бы один из них равен false, результатом всего выражения тоже будет false. На самом деле, если первое значение равно false, то второе даже не будет анализироваться, так как оно все равно не изменит общий результат на true. Такой подход называется краткой проверкой условия (short-circuit evaluation).

В следующем примере проверяются два значения типа Bool, и если они оба равны true, программа разрешает доступ:

```
let enteredDoorCode = true
let passedRetinaScan = false
if enteredDoorCode && passedRetinaScan {
 print("Welcome!")
} else {
 print("ACCESS DENIED")
}
// Выведет "ACCESS DENIED"
```

Оператор логического ИЛИ

Оператор логического ИЛИ (a || b) является инфиксным и записывается в виде двух вертикальных палочек без пробела. С его помощью можно создавать логические выражения, которые будут давать true, если хотя бы один из операндов равен true.

Как и описанный выше оператор логического И, оператор логического ИЛИ использует краткую проверку условия. Если левая часть выражения с логическим ИЛИ равна true, то правая не анализируется, так как ее значение не повлияет на общий результат.

В приведенном ниже примере первое значение типа Bool (hasDoorKey) равно false, а второе (knowsOverridePassword) равно true. Поскольку одно из значений равно true, результат всего выражения тоже становится true и доступ разрешается:

```
let hasDoorKey = false
let knowsOverridePassword = true
if hasDoorKey || knowsOverridePassword {
 print("Welcome!")
} else {
 print("ACCESS DENIED")
}
// Выведет "Welcome!"
```

2.3 Управляющие конструкции

В Swift есть все знакомые нам операторы управления потоком из C-подобных языков. К ним относятся: циклы for-in и while для многократного выполнения задач, инструкции if, guard и switch для выполнения различных ветвлений кода в зависимости от определенных условий, а также такие инструкции, как break и continue для перемещения потока выполнения в другую точку вашего кода.

Циклы For-in

Цикл for-in используется для итерации по коллекциям элементов, таких как диапазоны чисел, элементы массива или символы в строке.

Можно использовать цикл for-in вместе с массивом для итерации по его элементам:

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
    print("Hello, \(name)!")
}
```

```
// Hello, Anna!
// Hello, Alex!
// Hello, Brian!
// Hello, Jack!
```

Таким же образом вы можете производить итерацию по словарю, чтобы получить доступ к его паре ключ-значение. Когда происходит итерация по словарю, каждый его элемент возвращается как кортеж (ключ, значение). Вы можете разложить члены кортежа на отдельные константы для того, чтобы использовать их в теле цикла for-in. Здесь ключи словаря распадаются в константу animalName, а его значения - в константу legCount:

```
let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in numberOfLegs {
   print("\(animalName)s have \(legCount) legs")
}
// ants have 6 legs
// cats have 4 legs
// spiders have 8 legs
```

Содержимое словаря по сути своей не является упорядоченным, поэтому и извлекаемые из него значения во время итерации тоже могут быть не упорядочены.

Вы так же можете использовать for-in с числовыми диапазонами. Следующий пример напечатает несколько первых значений таблицы умножения на 5:

```
for index in 1...5 {
   print("\(index) умножить на 5 будет \(index * 5)")
}
// 1 умножить на 5 будет 5
// 2 умножить на 5 будет 10
// 3 умножить на 5 будет 15
// 4 умножить на 5 будет 20
// 5 умножить на 5 будет 25
```

Коллекция элементов, по которой происходит итерация, является закрытым диапазоном чисел от 1 до 5 включительно, так как

используется оператор закрытого диапазона(...). Значение index устанавливается в первое число из диапазона (1), и выражение внутри цикла выполняются. В данном случае, цикл содержит только одно выражение, которое печатает запись из таблицы умножения на пять для текущего значения index. После того как выражение выполнено, значение index обновляется до следующего значения диапазона (2), и функция print(_:separator: terminator:) снова вызывается. Этот процесс будет продолжаться до тех пор, пока не будет достигнут конец диапазона.

В примере выше index является константой, значение которой автоматически устанавливается в начале каждой итерации цикла. Как таковую, ее не нужно объявлять перед использованием. Ее объявление неявно происходит в объявлении цикла, без необходимости использования зарезервированного слова let.

Если Вам не нужно каждое значение из диапазона, то вы можете игнорировать их, используя символ подчёркивания вместо имени переменной:

```
let base = 3
let power = 10
var answer = 1
for _ in 1...power {
   answer *= base
}
print("\(base) в степени \(power) равно \(answer)")
// Выведет "3 в степени 10 равно 59049"
```

В этом примере вычисляется значение одного числа возведенное в степень другим (в данном случае 3 в степени 10). Начальное значение 1 (то есть 3 в степени 0) умножается на 3 десять раз, используя закрытый диапазон значений, который начинается с 1, и заканчивается 10. В данном случае нет необходимости знать значения счётчика во время каждой итерации цикла – он просто должен выполниться необходимое количество раз. Символ подчёркивания "_" (который используется вместо переменной цикла) игнорирует ее отдельные значения и не предоставляет доступ к текущему значению во время каждой итерации цикла.

В некоторых случаях вы можете не захотеть использовать замкнутый диапазон, который включает в себя оба конечных

значения диапазона. Предположим, что вы хотите отрисовать минутные значения в виде черточек на часах. Вы будете рисовать 60 таких отметок, начиная с 0 минуты. Используйте полузамкнутый диапазон (..<), чтобы включить нижнюю границу, но не верхнюю.

```
let minutes = 60
for tickMark in 0..<minutes {
   // render the tick mark each minute (60 times)
}
```

Некоторые пользователи, возможно, захотят иметь поменьше минутных делений, и, предположим, они захотят иметь отметки на циферблате только на каждые 5 минут. Для того, чтобы у нас была возможность пропустить ненужные временные отметки используйте функцию stride(from:to:by:).

```
let minuteInterval = 5
for tickMark in stride(from: 0, to: minutes, by: minuteInterval) {
   // render the tick mark every 5 minutes (0, 5, 10, 15 ... 45, 50, 55)
}
```

Так же вы можете работать и с закрытыми диапазонами, но уже при помощи метода stride(from:through:by:):

```
let hours = 12
let hourInterval = 3
for tickMark in stride(from: 3, through: hours, by: hourInterval) {
   // render the tick mark every 3 hours (3, 6, 9, 12)
}
```

Циклы While

Цикл while выполняет набор инструкций до тех пор, пока его условие не станет false. Этот вид циклов лучше всего использовать в тех случаях, когда количество итераций до первого входа в цикл неизвестно. Swift предлагает два вида циклов while:

- while – вычисляет условие выполнения в начале каждой итерации цикла.
- repeat-while – вычисляет условие выполнения в конце каждой итерации цикла.

While

Цикл while начинается с вычисления условия. Если условие истинно, то инструкции в теле цикла будут выполняться до тех пор, пока оно не станет ложным.

Общий вид цикла while выглядит следующим образом:

```
while условие {
инструкции
}
```

На рисунке 7 показана игра Змеи и Лестницы.

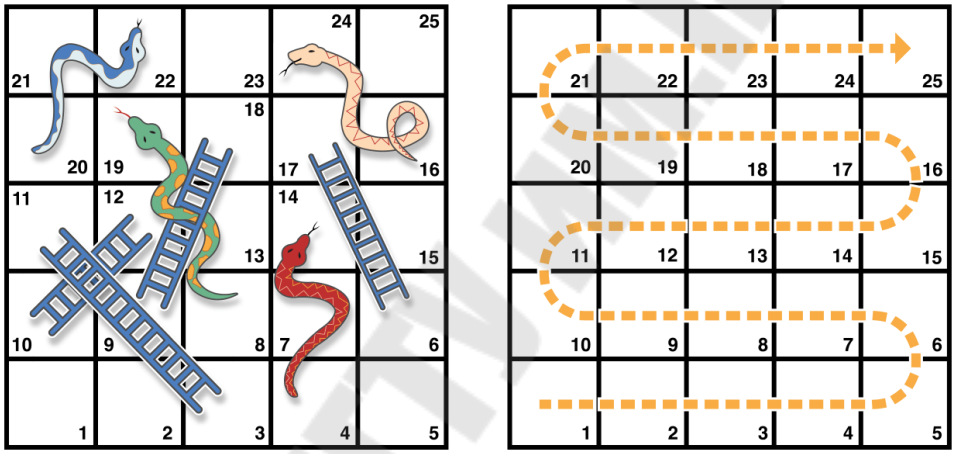


Рисунок 7 – Игра змеи и лестницы

Игра проходит по следующим правилам:

– Доска разделена на 25 квадратов и цель состоит в том, чтобы стать на 25-ый квадрат или за его пределами.

– Игрок начинает с "нулевого квадрата", который расположен в самом левом нижнем углу доски.

– В начале каждого хода вы бросаете игральную кость и перемещаетесь на то число шагов, которое выпало после броска, в направлении, которое указывает пунктирная стрелка.

– Если ваш ход заканчивается на основании лестницы, то вы поднимаетесь по ней вверх.

– Если ваш ход заканчивается на голове змеи, то вы спускаетесь вниз по этой змее.

Игровая доска в примере представлена массивом значений типа Int. Его размер хранится в константе finalSquare, которая

используется как для инициализации массива, так и для проверки условия победы. Игровое поле инициализируется 26-ю, а не 25-ю целочисленными нулевыми значениями (каждое с индексом от 0 до 25 включительно):

```
let finalSquare = 25
var board = [Int](repeating: 0, count: finalSquare + 1)
```

Затем, для обозначения лестниц и змей, некоторым квадратам присваиваются специальные значения. Квадраты с основанием лестницы, перемещающие вас вверх по доске, имеют положительные значения, тогда как квадраты с головой змеи, спускающие вас вниз – отрицательное.

```
board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
```

Квадрат 3 с основанием лестницы перемещает вас вверх на 11 квадрат. Чтобы это сделать, элементу массива board[03] присваивается +08, что эквивалентно значению 8 типа Int (разница между 3 и 11). Для того чтобы уточнить формулировку игрового поля, оператор унарного плюса (+i) уравновешивает оператор унарного минуса (-i), а числам ниже 10 приписаны нули. (В этих двух стилистических надстройках нет прямой необходимости, но они делают код более читаемым).

```
var square = 0
var diceRoll = 0
while square < finalSquare {
  // бросок кубика
  diceRoll += 1
  if diceRoll == 7 { diceRoll = 1 }
  // начать ходить на выпавшее количество шагов
  square += diceRoll
  if square < board.count {
    // если мы все еще на поле, идти вверх или вниз по змеям или лестницам
    square += board[square]
  }
```

```
}
print("Game over!")
```

Данный пример использует самый простой подход к реализации броска кубика. Вместо использования генератора случайных чисел, значение diceRoll начинается с 0. Каждую итерацию цикла переменная diceRoll увеличивается на 1 с помощью инфиксного оператора (+= 1), после чего проверяется не стало ли её значение слишком большим. Возвращаемое значение += diceRoll равно значению переменной diceRoll после её инкрементирования. Когда это значение становится равным 7, оно сбрасывается на 1. В итоге мы получаем последовательность значений diceRoll, которая всегда будет выглядеть следующим образом: 1, 2, 3, 4, 5, 6, 1, 2 и так далее.

После броска кубика игрок перемещается вперед на количество клеток, равное значению переменной diceRoll. Возможен случай, когда бросок кубика может переместить игрока за пределы квадрата 25. В таком случае игра заканчивается. Для того чтобы справиться с таким сценарием, код проверяет что значение square меньше чем свойство count массива board перед прибавлением значения, хранящегося в board[square] к текущему значению square для перемещения игрока вверх или вниз по змеям или лестницам.

Текущая итерация цикла заканчивается, после чего проверяется условие цикла, для того чтобы понять нужно ли переходить к следующей итерации. Если игрок переместился на квадрат 25 или за его пределы, значение условия будет вычислено как false и игра закончится.

В данном случае использование while является наиболее подходящим, так как продолжительность игры неизвестна перед началом цикла. Цикл просто исполняется до тех пор, пока не будет выполнено конкретное условие.

Цикл repeat-while

Другой вариант цикла while, известный как цикл repeat-while, выполняет одну итерацию до того, как происходит проверка условия. Затем цикл продолжает повторяться до тех пор, пока условие не станет false.

Общий вид цикла repeat-while выглядит следующим образом:

```
repeat {
```

```
инструкции
} while условие
```

Ниже снова представлен пример игры Змеи и Лестницы, написанный с использованием цикла repeat-while. Значения переменных finalSquare, board, square и diceRoll инициализированы точно таким же образом, как и в случае с циклом while:

```
let finalSquare = 25
var board = [Int](repeating: 0, count: finalSquare + 1)
board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
var square = 0
var diceRoll = 0
```

В этой версии игры в начале цикла происходит проверка на наличие змей или лестниц на квадрате. Ни одна лестница на поле не приведет игрока на квадрат 25. Таким образом невозможно победить в игре, переместившись вверх по лестнице. Следовательно, такая проверка в самом начале цикла является абсолютно безопасной.

В начале игры игрок находится на квадрате 0. board[0] всегда равняется 0 и не оказывает никакого влияния:

```
repeat {
  // идти вверх или вниз по змеям или лестницам
  square += board[square]
  // бросить кубик
  diceRoll += 1
  if diceRoll == 7 { diceRoll = 1 }
  // начать ходить на выпавшее количество шагов
  square += diceRoll
} while square < finalSquare
print("Game over!")
```

После проверки на наличие змей и лестниц происходит бросок кубика и игрок продвигается вперед на количество квадратов, равное diceRoll. После этого текущая итерация цикла заканчивается.

Условие цикла (while square < finalSquare) такое же, как раньше, но в этот раз оно не вычисляется до окончания первого запуска цикла.

Структура цикла repeat-while лучше подходит для этой игры, чем цикл while в предыдущем примере. В цикле repeat-while выше square += board[square] всегда выполняется сразу, в то время как в цикле while происходит проверка того, что square все еще находится на поле. Такой принцип работы цикла repeat-while снимает необходимость проверки выхода за границы массива, которую мы видели в предыдущей версии игры.

Условные инструкции

Иногда бывает полезным исполнять различные куски кода в зависимости от условий.

Swift предоставляет нам два варианта добавить условные ответвления кода - это при помощи инструкции if и при помощи инструкции switch. Обычно мы используем инструкцию if, если наше условие достаточно простое и предусматривает всего несколько вариантов. А вот инструкция switch подходит для более сложных условий, с многими вариантами выбора, и очень полезна в ситуациях, где по найденному совпадению с условием и выбирается соответствующая ветка кода для исполнения.

Инструкция if

В самой простой своей форме инструкция if имеет всего одно условие if. Эта инструкция выполняет установленные инструкции только в случае, когда условие true:

```
var temperatureInFahrenheit = 30
if temperatureInFahrenheit <= 32 {
    print ("It's very cold. Consider wearing a scarf.")
}
// Выведет "It's very cold. Consider wearing a scarf.
```

В приведенном примере проверяется значение температуры, которая может быть ниже или 32 (0 по Цельсию) градусов по Фаренгейту либо равна или выше. Если она ниже, то выведется сообщение. В противном случае никакого сообщения не будет, и код продолжит свое выполнение после закрывающей фигурной скобки инструкции if.

Инструкция if может предусматривать еще один дополнительный набор инструкций в ветке известной как

оговорка else, которая нужна в случае, если условие false. Эти инструкции указываются через ключевое слово else:

```
temperatureInFahrenheit = 40
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else {
    print("It's not that cold. Wear a t-shirt.")
}
// Выведет "It's not that cold. Wear a t-shirt."
```

В этом коде всегда будет выполняться код либо в первом, либо во втором ответвлении. Из-за того что температура выросла до 40 градусов Фаренгейта, значит больше не обязательно носить шарф, таким образом ответвление else выполняется.

Вы можете соединять инструкции if между собой, чтобы создать более сложные условия:

```
temperatureInFahrenheit = 90
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear sunscreen.")
} else {
    print("It's not that cold. Wear a t-shirt.")
}
// Выведет "It's really warm. Don't forget to wear sunscreen.
```

В приведенном коде была добавлена дополнительная инструкция if, для соответствия определенным температурам. Конечное условие else соответствует всем температурам, не соответствующим первым двум условиям.

Последняя else опциональна и может быть удалена, если в ней нет необходимости:

```
temperatureInFahrenheit = 72
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
```

```
    print("It's really warm. Don't forget to wear sunscreen.")
}
```

В этом примере температура ни высокая, ни низкая, и вообще она не соответствует ни одному условию, так что никакого сообщения мы не увидим.

Инструкция switch

Инструкция switch подразумевает наличие какого-то значения, которое сравнивается с несколькими возможными шаблонами. После того как значение совпало с каким-либо шаблоном, выполняется код, соответствующий ответвлению этого шаблона, и больше сравнений уже не происходит. Switch представляет собой альтернативу инструкции if, отвечающей нескольким потенциальным значениям.

В самой простой форме в инструкции switch значение сравнивается с одним или более значений того же типа:

```
switch значение для сопоставления {
    case значение 1:
        инструкция для значения 1
    case значение 2, значение 3:
        инструкция для значения 2 или значения 3
    default:
        инструкция, если совпадений с шаблонами не найдено
}
```

Каждая инструкция switch состоит из нескольких возможных случаев или cases, каждый из которых начинается с ключевого слова case. Помимо сравнения с конкретными значениями, Swift предлагает еще несколько опций для каждого случая для создания более сложных шаблонных сравнений. Об этих опциях мы поговорим далее в этой главе.

Тела каждого отдельного блока case в switch - это отдельная ветка исполнительного кода, что делает switch похожим на инструкцию if. Инструкция switch определяет какое ответвление должно быть выбрано. Это известно как переключение на значение, которое в настоящее время рассматривается.

Каждая инструкция switch должна быть исчерпывающей. То есть это значит, что каждое значение обязательно должно

находить совпадение с шаблоном в каком-либо случае (case). Если неудобно вписывать все возможные варианты случаев, то вы можете определить случай по умолчанию, который включает в себя все значения, которые не были включены в остальные случаи. Такой случай по умолчанию называется default, и он всегда идет после всех остальных случаев.

В следующем примере switch рассматривает единичный символ в нижнем регистре, который называется someCharacter:

```
let someCharacter: Character = "z"
switch someCharacter {
case "a":
   print("The first letter of the alphabet")
case "z":
   print("The last letter of the alphabet")
default:
   print("Some other character")
}
// Выведет "The last letter of the alphabet"
```

Первый кейс инструкции switch соответствует первой букве английского алфавита - a, второй кейс соответствует последней букве - z. Так как switch должна иметь кейс для каждого возможного символа, а не просто для каждой буквы алфавита, то в инструкции switch предусмотрен дефолтный кейс, который звучит как default, в который входят все символы кроме a и z. Как раз это условие гарантирует, что инструкция switch будет исчерпывающей.

Отсутствие case-провалов

Большое отличие инструкции switch в языке Swift от инструкции switch в C и Objective-C составляет отсутствие провалов через условия. Вместо этого инструкция switch прекращает выполнение после нахождения первого соответствия с case и выполнения соответствующего кода в ветке, без необходимости явного вызова break. Это делает инструкцию switch более безопасным и простым для использования, чем в C, и исключает исполнение кода более чем одного случая.

Тело каждого случая *должно* включать в себя хотя бы одно исполняемое выражение. Код не будет исполнен и выдаст ошибку компиляции, если написать его следующим образом:

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
case "a":  // ошибка, так как кейс имеет пустое тело
case "A":
   print("The letter A")
default:
   print("Not the letter A")
}
// ошибка компиляции
```

В отличии от инструкции switch в языке C, switch в Swift не соответствует ни «a», ни «A». Но зато вы получите ошибку компиляции о том, что case «a»: не содержит ни одного исполняемого выражения. Такой подход исключает случайные "проваливания" из одного случая в другой, что делает код безопаснее и чище своей краткостью.

Для того, чтобы switch с одним кейсом подходил под «a» и «A»,объедините два значения в один составной кейс, разделив значения запятыми.

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
case "a", "A":
   print("The letter A")
default:
   print("Not the letter A")
}
// Выведет "The letter A"
```

Соответствие диапазону

Значения в кейсах switch могут быть проверены на их вхождение в диапазон. Пример ниже использует целочисленные диапазоны для описания любых значений художественным языком:

```
let approximateCount = 62
```

```
let countedThings = "moons orbiting Saturn"
var naturalCount: String
switch approximateCount {
case 0:
   naturalCount = "no"
case 1..<5:
   naturalCount = "a few"
case 5..<12:
   naturalCount = "several"
case 12..<100:
   naturalCount = "dozens of"
case 100..<1000:
   naturalCount = "hundreds of"
default:
   naturalCount = "many"
}
print("There are \(naturalCount) \(countedThings).")
// Выводит "There are dozens of moons orbiting Saturn."
```

В приведенном выше примере, approximateCount оценивается в инструкции switch. Каждый кейс сравнивает это значение с числом или интервалом. Поскольку значение approximateCount попадает на диапазон от 12 до 100, naturalCount присваивается значение "dozens of", и исполнение перемещается из инструкции switch.

Операторы передачи управления

Операторы передачи управления меняют последовательность исполнения вашего кода, передавая управление от одного фрагмента кода другому. В Swift есть пять операторов передачи управления:

- continue
- break
- fallthrough
- return
- throw

Оператор Continue

Оператор continue говорит циклу прекратить текущую итерацию и начать новую. Он как бы говорит: "Я закончил с текущей итерацией", но полностью из цикла не выходит.

Следующий пример убирает все пробелы и гласные в нижнем регистре из строки, для того чтобы создать загадочную фразу-головоломку:

```
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
let charactersToRemove: [Character] = ["a", "e", "i", "o", "u", " "]
for character in puzzleInput {
   if charactersToRemove.contains(character) {
      continue
   } else {
      puzzleOutput.append(character)
   }
}
print(puzzleOutput)
// Выведет "grtmndsthnklk"
```

Пример выше вызывает оператор continue, когда он находит соответствие с гласными звуками или пробелом, вызывая тем самым прекращение текущей итерации и начало новой.

Оператор Break

Оператор break останавливает выполнение всей инструкции управления потоком. Оператор break может быть использован внутри инструкции switch или внутри цикла, когда вы хотите остановить дальнейшее выполнение switch или цикла раньше, чем он должен закончиться сам по себе.

Оператор Break в цикле

Когда оператор break используется внутри цикла, то он немедленно прекращает работу цикла, и выполнение кода продолжается с первой строки после закрывающей скобки цикла (}). Никакой последующий код из текущей итерации цикла выполняться не будет, и никакие дальнейшие итерации цикла не будут запускаться.

Оператор Fallthrough

Инструкция switch в Swift не проваливается из каждого кейса в следующий. Напротив, как только находится соответствие с первым кейсом, так сразу и прекращается работа всей инструкции. А в языке C, работа инструкции switch немного сложнее, так как требует явного прекращения работы при нахождении соответствия словом break в конце кейса, в противном случае при соответствии мы провалимся в следующий случай и так далее пока не встретим слово break. Избежание провалов значит что инструкция switch в Swift более краткая и предсказуемая, чем она же в C, так как она предотвращает срабатывание нескольких кейсов по ошибке.

Если вам по какой-то причине нужно аналогичное проваливание как в C, то вы можете использовать оператор fallthrough в конкретном кейсе. Пример ниже использует fallthrough для текстового описания целого числа:

```
let integerToDescribe = 5
var description = "The number \(integerToDescribe) is"
switch integerToDescribe {
case 2, 3, 5, 7, 11, 13, 17, 19:
   description += " a prime number, and also"
   fallthrough
default:
   description += " an integer."
}
print(description)
// Выведет "The number 5 is a prime number, and also an integer."
```

В примере мы объявляем новую переменную типа String, названную description и присваиваем ей исходное значение. Потом мы определяем величину integerToDescribe, используя инструкцию switch. Если значение integerToDescribe одно из значений списка кейса, то мы получаем текстовое описание значения, которое дополняется значением, которое находится в default, так как на уровень выше в сработавшем кейсе стоит ключевое слово fallthrough, после чего завершается работа инструкции switch.

Если значение integerToDescribe не принадлежит списку значений нашего единственного кейса, то срабатывает кейс по умолчанию, который имеет все оставшиеся значения, невошедшие в

первый кейс, и integerToDescribe получает значение только то, что есть в default.

После того как сработала инструкция switch, мы получаем описание числа, используя функцию print(_:separator:terminator:). В нашем примере мы получаем 5, что корректно определено как простое число.

2.4 Кортежи

Кортежи группируют несколько значений в одно составное значение. Значения внутри кортежа могут быть любого типа, то есть, нет необходимости, чтобы они были одного и того же типа.

В данном примере (404, "Not Found") это кортеж, который описывает код HTTP статуса. Код HTTP статуса – особое значение, возвращаемое веб-сервером каждый раз, когда вы запрашиваете веб-страницу. Код статуса 404 Not Found возвращается, когда вы запрашиваете страницу, которая не существует.

```
let http404Error = (404, "Not Found")
// http404Error имеет тип (Int, String), и равен (404, "Not Found")
```

Чтобы передать код статуса, кортеж (404, "Not Found") группирует вместе два отдельных значения Int и String: число и понятное человеку описание. Это может быть описано как "кортеж типа (Int, String)".

Вы можете создать кортеж с любой расстановкой типов, и они могут содержать сколько угодно нужных вам типов. Ничто вам не мешает иметь кортеж типа (Int, Int, Int), или типа (String, Bool), или же с любой другой расстановкой типов по вашему желанию.

Вы можете разложить содержимое кортежа на отдельные константы и переменные, к которым можно получить доступ привычным способом:

```
let (statusCode, statusMessage) = http404Error
print("The status code is \(statusCode)")
// Выведет "The status code is 404"
print("The status message is \(statusMessage)")
// Выведет "The status message is Not Found"
```

Если вам нужны только некоторые из значений кортежа, вы можете игнорировать части кортежа во время разложения с помощью символа подчеркивания (_):

```
let (justTheStatusCode, _) = http404Error
print("The status code is \(justTheStatusCode)")
// Выведет "The status code is 404"
```

В качестве альтернативы можно получать доступ к отдельным частям кортежа, используя числовые индексы, начинающиеся с нуля:

```
print("The status code is \(http404Error.0)")
// Выведет "The status code is 404"
print("The status message is \(http404Error.1)")
// Выведет "The status message is Not Found"
```

Вы можете давать имена отдельным элементам кортежа во время объявления:

```
let http200Status = (statusCode: 200, description: "OK")
```

Когда вы присвоили имя элементу кортежа, вы можете обратиться к нему по имени:

```
print("The status code is \(http200Status.statusCode)")
// Выведет "The status code is 200"
print("The status message is \(http200Status.description)")
// Выведет "The status message is OK"
```

Кортежи особенно полезны в качестве возвращаемых значений функций. Функция, которая пытается получить веб-страницу, может вернуть кортеж типа (Int, String), чтобы описать успех или неудачу в поиске страницы. Возвращая кортеж с двумя отдельными значениями разного типа, функция дает более полезную информацию о ее результате, чем, если бы, возвращала единственное значение одного типа, возвращаемое функцией.

2.5 Опциональные типы

Опциональные типы используются в тех случаях, когда значение может отсутствовать. Опциональный тип подразумевает, что возможны два варианта: или значение есть, и его можно извлечь из опционала, либо его вообще нет.

Приведем пример, который покажет, как опционалы могут справиться с отсутствием значения. У типа Int в Swift есть инициализатор, который пытается преобразовать значение String в значение типа Int. Тем не менее, не каждая строка может быть преобразована в целое число. Строка "123" может быть преобразована в числовое значение 123, но строка "hello, world" не имеет очевидного числового значения для преобразования.

В приведенном ниже примере используется метод Int() для попытки преобразовать String в Int:

```
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)
// для convertedNumber выведен тип "Int?", или "опциональный Int"
```

Поскольку метод Int() может иметь недопустимый аргумент, он возвращает опциональный Int, вместо Int. Опциональный Int записывается как Int?, а не Int. Знак вопроса означает, что содержащееся в ней значение является опциональным, что означает, что он может содержать некое Int значение, или он может вообще не содержать никакого значения. (Он не может содержать ничего другого, например, Bool значение или значение String. Он либо Int, либо вообще ничто)

nil

Мы можем установить опциональную переменную в состояние отсутствия значения, путем присвоения ему специального значения nil

```
var serverResponseCode: Int? = 404
// serverResponseCode содержит реальное Int значение 404
serverResponseCode = nil
// serverResponseCode теперь не содержит значения
```

Если объявить опциональную переменную без присвоения значения по умолчанию, то переменная автоматически установится в nil для вас:

```
var surveyAnswer: String?
// surveyAnswer автоматически установится в nil
```

3 ФУНКЦИИ

Функции – это самостоятельные фрагменты кода, решающие определенную задачу. Каждой функции присваивается уникальное имя, по которому ее можно идентифицировать и "вызвать" в нужный момент.

Язык Swift предлагает достаточно гибкий единый синтаксис функций – от простых C-подобных функций без параметров до сложных методов в стиле Objective-C с локальными и внешними параметрами. Параметры могут служить как для простой инициализации значений внутри функции, так и для изменения внешних переменных после выполнения функции.

Каждая функция в Swift имеет тип, описывающий тип параметров функции и тип возвращаемого значения. Тип функции можно использовать аналогично любым другим типам в Swift, т. е. одна функция может быть параметром другой функции либо ее результирующим значением. Функции также могут вкладываться друг в друга, что позволяет инкапсулировать определенный алгоритм внутри локального контекста.

Объявление и вызов функций

При объявлении функции можно задать одно или несколько именованных типизированных значений, которые будут ее входными данными (или параметрами), а также тип значения, которое функция будет возвращать в качестве результата (или возвращаемый тип).

У каждой функции должно быть имя, которое отражает решаемую задачу. Чтобы воспользоваться функцией, ее нужно "вызвать", указав имя и входные значения (аргументы), соответствующие типам параметров этой функции. Аргументы функции всегда должны идти в том же порядке, в каком они были указаны при объявлении функции.

В приведенном ниже примере функция называется greet(person:), потому что это отражает ее задачу – получить имя пользователя и вежливо поздороваться. Для этого задается один входной параметр типа String под названием person, а возвращается тоже значение типа String, но уже содержащее приветствие:

```
func greet(person: String) -> String {
```

```
    let greeting = "Hello, " + person + "!"
    return greeting
}
```

Вся эта информация указана в объявлении функции, перед которым стоит ключевое слово func. Тип возвращаемого значения функции ставится после результирующей стрелки -> (это дефис и правая угловая скобка).

Из объявления функции можно узнать, что она делает, какие у нее входные данные и какой результат она возвращает. Объявленную функцию можно однозначно вызывать из любого участка кода:

```
print(greet(person: "Anna"))
// Выведет "Hello, Anna!"
print(greet(person: "Brian"))
// Выведет "Hello, Brian!"
```

Функция greet(person:) вызывается, принимая значение типа String, которое стоит после имени person, например вот так - greet(person: "Anna"). Поскольку функция возвращает значение типа String, вызов функции greet(person:) может быть завернут в вызов для функции print(_:separator:terminator:), чтобы напечатать полученную строку и увидеть возвращаемое значение.

Тело функции greet(person:) начинается с объявления новой константы типа String под названием greeting, и устанавливается простое сообщение-приветствие. Затем это приветствие возвращается в точку вызова функции с помощью ключевого слова return. После выполнения оператора return greeting функция завершает свою работу и возвращает текущее значение greeting.

Функцию greet(person:) можно вызывать многократно и с разными входными значениями. В примере выше показано, что будет, если функцию вызвать с аргументом "Anna" и со значением "Brian". В каждом случае функция возвратит персональное приветствие.

Чтобы упростить код этой функции, можно записать создание сообщения и его возврат в одну строку:

```
func greetAgain(person: String) -> String {
    return "Hello again, " + person + "!"
}
print(greetAgain(person: "Anna"))
```

```
// Выведет "Hello again, Anna!"
```

Параметры функции и возвращаемые значения

В языке Swift параметры функций и возвращаемые значения реализованы очень гибко. Разработчик может объявлять любые функции – от простейших, с одним безымянным параметром, до сложных, со множеством параметров и составными именами.

Функции без параметров

В некоторых случаях функции могут не иметь входных параметров. Вот пример функции без входных параметров, которая при вызове всегда возвращает одно и то же значение типа String:

```
func sayHelloWorld() -> String {
   return "hello, world"
}
print(sayHelloWorld())
// Выведет "hello, world"
```

Обратите внимание, что несмотря на отсутствие параметров, в объявлении функции все равно нужно ставить скобки после имени. При вызове после имени функции также указываются пустые скобки.

Функции с несколькими входными параметрами

У функции может быть несколько параметров, которые указываются через запятую в скобках.

Эта функция принимает два параметра: имя человека и булево значение, приветствовали ли его уже, и возвращает соответствующее приветствие для этого человека:

```
func greet(person: String, alreadyGreeted: Bool) -> String {
   if alreadyGreeted {
      return greetAgain(person: person)
   } else {
      return greet(person: person)
   }
}
print(greet(person: "Tim", alreadyGreeted: true))
// Выведет "Hello again, Tim!"
```

Вы вызываете функцию greet(person:alreadyGreeted:), передавая значение типа String параметру с ярлыком person и булево значение с ярлыком alreadyGreeted, взятое в скобки через запятую. Обратите внимание, что эта функция отличается от функции greet(person:), которую вы видели в предыдущем разделе. Хотя имена обеих функций начинаются с greet, функция greet(person:alreadyGreeted:) принимает два аргумента, а greet(person:) принимает только один.

Функции, не возвращающие значения

В некоторых случаях функции могут не иметь возвращаемого типа. Вот другая реализация функции greet(person:), которая выводит свое собственное значение типа String, но не возвращает его:

```
func greet(person: String) {
    print("Hello, \(person)!")
}
greet(person: "Dave")
// Выведет "Hello, Dave!"
```

Так как у функции нет выходного значения, в ее объявлении отсутствует результирующая стрелка (->) и возвращаемый тип.

Выходное значение функции может быть игнорировано:

```
func printAndCount(string: String) -> Int {
    print(string)
    return string.count
}
func printWithoutCounting(string: String) {
    let _ = printAndCount(string: string)
}
printAndCount(string: "hello, world")
// Выведет "hello, world" и возвращает значение 12
printWithoutCounting(string: "hello, world")
// Выведет "hello, world", но не возвращает значения
```

Первая функция, printAndCount(string:) выводит строку, а затем возвращает подсчет символов в виде целого (Int). Вторая функция, printWithoutCounting(string:) вызывает первую, но игнорирует ее возвращаемое значение. При вызове второй функции первая функция

по-прежнему печатает сообщение, но ее возвращаемое значение не используется.

Функции, возвращающие несколько значений

Вы можете использовать кортежный тип в качестве возвращаемого типа для функции для возврата нескольких значений в виде составного параметра.

В следующем примере объявлена функция minMax(array:), которая ищет минимальный и максимальный элементы в массиве типа Int:

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

Функция minMax(array:) возвращает кортеж из двух значений типа Int. Этим значениям присвоены имена min и max, чтобы к ним можно было обращаться при запросе возвращаемого типа функции.

Тело функции minMax(array:) начинается с инициализации двух рабочих переменных currentMin и currentMax значением первого целого элемента в массиве. Затем функция последовательно проходит по всем остальным значениям в массиве и сравнивает их со значениями currentMin и currentMax соответственно. И наконец, самое маленькое и самое большое значения возвращаются внутри кортежа типа Int.

Так как имена элементов кортежа указаны в возвращаемом типе функции, к ним можно обращаться через точку и считывать значения:

```
let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \(bounds.max)")
```

```
// Выведет "min is -6 and max is 109"
```

Обратите внимание, что элементам кортежа не нужно давать название в момент возвращения кортежа из функции, так как их имена уже указаны как часть возвращаемого типа функции.

Опциональный кортеж как возвращаемый тип

Если возвращаемый из функции кортеж может иметь "пустое значение", то его следует объявить как опциональный кортеж, т. е. кортеж, который может равняться nil. Чтобы сделать возвращаемый кортеж опциональным, нужно поставить вопросительный знак после закрывающей скобки:(Int, Int)? или (String, Int, Bool)?.

Функция minMax(array:) выше возвращает кортеж из двух значений типа Int, однако не проверяет корректность передаваемого массива. Если аргумент array содержит пустой массив, для которого count равно 0, функция minMax в том виде, в каком она приведена выше, выдаст ошибку выполнения, когда попытается обратиться к элементу array[0].

Для устранения этого недочета перепишем функцию minMax(array:) так, чтобы она возвращала кортеж-опционал, который в случае пустого массива примет значение nil:

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty { return nil }
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

Чтобы проверить, возвращает ли эта версия функции minMax(array:) фактическое значение кортежа или nil, можно использовать привязку опционала:

```
if let bounds = minMax(array: [8, -6, 2, 109, 3, 71]) {
   print("min is \(bounds.min) and max is \(bounds.max)")
}
// Выведет "min is -6 and max is 109"
```

Ярлыки аргументов и имена параметров функций

Каждый параметр функции имеет ярлык аргумента и имя параметра. Ярлык аргумента используется при вызове функции. Каждый параметр при вызове функции записывается с ярлыком аргумента, стоящим перед ним. Имя параметра используется при реализации функции. По умолчанию параметры используют имена их параметров в качестве ярлыка аргумента.

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {
    // Внутри тела функции firstParameterName и secondParameterName
    // ссылаются на значения аргументов, первого и второго параметров.
}
someFunction(firstParameterName: 1, secondParameterName: 2)
```

Все параметры должны иметь уникальные имена. Несмотря на то, что несколько параметров могут иметь один ярлык аргумента, уникальные ярлыки аргумента метки помогают сделать ваш код более читабельным.

Указываем ярлыки аргументов

Вы пишете ярлык аргумента перед именем параметра через пробел:

```
func someFunction(argumentLabel parameterName: Int) {
   // В теле функции parameterName относится к значению аргумента
   // для этого параметра.
}
```

Вот вариант функции greet(person:), которая принимает имя человека и его родной город, затем возвращает приветствие:

```
func greet(person: String, from hometown: String) -> String {
    return "Hello \(person)!  Glad you could visit from \(hometown)."
}
print(greet(person: "Bill", from: "Cupertino"))
// Выводит "Hello Bill!  Glad you could visit from Cupertino."
```

Использование ярлыков аргументов позволяет функции вызываться в более выразительной манере, в виде предложения, при этом все же предоставляя тело функции в более читаемом виде и с более понятыми намерениями.

Пропуск ярлыков аргумента

Если вы не хотите использовать ярлык аргумента в качестве параметра, используйте подчеркивание (_) вместо явного ярлыка аргумента для этого параметра.

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {
    // В теле функции firstParameterName и secondParameterName
    // ссылаются на значения аргументов для первого и второго параметров.
}
someFunction(1, secondParameterName: 2)
```

Если у параметра есть ярлык аргумента, то аргумент должен иметь ярлык при вызове функции.

Значения по умолчанию для параметров

При объявлении функции любому из ее параметров можно присвоить значение по умолчанию. Если у параметра есть значение по умолчанию, то при вызове функции этот параметр можно опустить.

```
func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int = 12) {
    // Если вы пропускаете второй аргумент при вызове функции, то
    // значение parameterWithDefault будет равняться 12 внутри тела функции.
```

```
}
someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6) // parameterWithDefault равен 6
someFunction(parameterWithoutDefault: 4) // parameterWithDefault равен 12
```

Расположите параметры, у которых нет дефолтных значений в начале списка параметров функции до параметров с дефолтными значениями. Параметры, не имеющие значения по умолчанию, как правило, более важны для значения функции - их запись в первую очередь облегчает распознавание функции уже вызванной ранее, независимо от того, опущены ли какие-то параметры по умолчанию.

Вариативные параметры

Вариативным называют параметр, который может иметь сразу несколько значений или не иметь ни одного. С помощью вариативного параметра можно передать в функцию произвольное число входных значений. Чтобы объявить параметр как вариативный, нужно поставить три точки (...) после его типа.

Значения, переданные через вариативный параметр, доступны внутри функции в виде массива соответствующего типа. Например, вариативный параметр numbers типа Double... доступен внутри функции в виде массива-константы numbers типа [Double].

В приведенном ниже примере вычисляется среднее арифметическое (или же среднее) последовательности чисел, имеющей произвольную длину:

```
func arithmeticMean(_ numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)
// возвращает 3.0, что является средним арифметическим этих пяти чисел
arithmeticMean(3, 8.25, 18.75)
```

// возвращает 10.0, что является средним арифметическим этих трех чисел

Сквозные параметры

Параметры функции по умолчанию являются константами. Попытка изменить значение параметра функции из тела этой функции приводит к ошибке компиляции. Это означает, что вы не сможете изменить значение параметра по ошибке. Если вы хотите, чтобы функция изменила значение параметра, и вы хотите, чтобы эти изменения сохранились после того, как закончился вызов функции, определите этот параметр в качестве сквозного параметра.

Для создания сквозного параметра нужно поставить ключевое слово inout перед типом объявлением параметра. Сквозной параметр передает значение в функцию, которое затем изменяется в ней и возвращается из функции, заменяя исходное значение.

Вы можете передать только переменную в качестве аргумента для сквозного параметра. Вы не можете передать константу или значения литерала в качестве аргумента, так как константы и литералы не могут быть изменены. Вы ставите амперсанд (&) непосредственно перед именем переменной, когда передаете ее в качестве аргумента сквозного параметра, чтобы указать, что он может быть изменен с помощью функции.

Вот пример функции под названием swapTwoInts(_:_:), у которой есть два сквозных целочисленных параметра – a и b:

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
   let temporaryA = a
   a = b
   b = temporaryA
}
```

Функция swapTwoInts(_:_:) просто меняет значение переменной b на значение a, а значение a – на значение b. Для этого функция сохраняет значение a в локальной константе temporaryA, присваивает значение b переменной a, а затем присваивает значение temporaryA переменной b.

Вы можете вызвать функцию swapTwoInts (_: _:) с двумя переменными типа Int, чтобы поменять их значения. Обратите

внимание, что имена someInt и anotherInt начинаются с амперсанда, когда они передаются в swapTwoInts (_: _:) функции:

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// Выведет "someInt is now 107, and anotherInt is now 3"
```

В вышеприведенном примере видно, что исходные значения переменных someInt и anotherInt изменены функцией swapTwoInts (_: _:), несмотря на то, что изначально они были объявлены за ее пределами.

Функциональные типы

У каждой функции есть специальный функциональный тип, состоящий из типов параметров и типа возвращаемого значения.

Пример:

```
func addTwoInts(a: Int, _ b: Int) -> Int {
  return a + b
}
func multiplyTwoInts(a: Int, _ b: Int) -> Int {
  return a * b
}
```

В данном примере объявлены две простые математические функции – addTwoInts и multiplyTwoInts. Каждая из этих функций принимает два значения типа Int и возвращает одно значение типа Int, содержащее результат математической операции.

Обе функции имеют тип (Int, Int) -> Int. Эта запись означает следующее:

"функция с двумя параметрами типа Int, возвращающая значение типа Int".

Вот еще один пример, но уже функции без параметров и возвращаемого значения:

```
func printHelloWorld() {
```

```
  print("hello, world")
}
```

Эта функция имеет тип () -> Void, т. е. "функция без параметров, которая возвращает Void".

Использование функциональных типов

В Swift с функциональными типами можно работать так же, как и с другими типами. Например, можно объявить константу или переменную функционального типа и присвоить ей функцию соответствующего типа:

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

Эта запись означает следующее:

"Объявить переменную mathFunction, имеющую тип "функция, принимающая два значения типа Int, и возвращающая одно значение типа Int". Присвоить этой новой переменной указатель на функцию addTwoInts".

Функция addTwoInts имеет тот же тип, что и переменная mathFunction, поэтому с точки зрения языка Swift такое присваивание корректно.

Теперь функцию можно вызывать с помощью переменной mathFunction:

```
print("Result: \(mathFunction(2, 3))")
// Выведет "Result: 5"
```

Той же переменной можно присвоить и другую функцию такого же типа – аналогично нефункциональным типам:

```
mathFunction = multiplyTwoInts
print("Result: \(mathFunction(2, 3))")
// Выведет "Result: 6"
```

Как и в случае с любым другим типом, вы можете не указывать тип явно, а предоставить Swift самостоятельно вывести функциональный тип при присваивании функции константе или переменной:

```
let anotherMathFunction = addTwoInts
// для константы anotherMathFunction выведен тип (Int, Int) -> Int
```

Функциональные типы как типы параметров

Функциональные типы наподобие (Int, Int) -> Int могут быть типами параметров другой функции. Это позволяет определять некоторые аспекты реализации функции непосредственно во время ее вызова.

Следующий код печатает на экране результаты работы приведенных выше математических функций:

```
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    print("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
// Выведет "Result: 8"
```

В этом примере объявлена функция printMathResult(_:_:_:), у которой есть три параметра. Первый параметр под названием mathFunction имеет тип (Int, Int) -> Int. Соответственно, аргументом этого параметра может быть любая функция такого же типа. Второй и третий параметры называются a и b и относятся к типу Int. Они служат для передачи двух входных значений для математической функции.

При вызове printMathResult(_:_:_:) получает в качестве входных данных функцию addTwoInts(_:_:) и два целочисленных значения 3 и 5. Затем она вызывает переданную функцию со значениями 3 и 5, а также выводит на экран результат 8.

Задача функции printMathResult(_:_:_:) заключается в том, чтобы печатать результат работы математической функции соответствующего типа. При этом конкретные детали этой математической функции не имеют значения – главное, чтобы она была подходящего типа. Все это позволяет безопасно управлять работой функции printMathResult(_:_:_:) непосредственно во время вызова.

Функциональные типы как возвращаемые типы

Функциональный тип можно сделать возвращаемым типом другой функции. Для этого нужно записать полный функциональный тип сразу же после возвратной стрелки (->) в возвращаемой функции.

В следующем примере объявлены две простые функции – stepForward(_:) и stepBackward(_:). Функция stepForward(_:) возвращает входное значение, увеличенное на единицу, а функция stepBackward(_:) – уменьшенное на единицу. Обе функции имеют тип (Int) -> Int:

```
func stepForward(_ input: Int) -> Int {
   return input + 1
}
func stepBackward(_ input: Int) -> Int {
   return input - 1
}
```

Следующая функция под названием chooseStepFunction(backward:) имеет возвращаемый тип (Int) -> Int. Функция chooseStepFunction(backward:) возвращает функцию stepForward(_:) или функцию stepBackward(_:) в зависимости от значения логического параметра backward:

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
   return backward ? stepBackward : stepForward
}
```

Теперь с помощью chooseStepFunction(backward:) можно получать функцию, которая будет сдвигать значение влево или вправо:

```
var currentValue = 3
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
// moveNearerToZero ссылается на функцию stepBackward()
```

В предыдущем примере мы определяли, нужно ли прибавить или отнять единицу, чтобы последовательно приблизить переменную currentValue к нулю. Изначально currentValue имеет значение 3, т. е. сравнение currentValue > 0 даст true, а функция

chooseStepFunction(backward:), соответственно, возвратит функцию stepBackward(_:). Указатель на возвращаемую функцию хранится в константе moveNearerToZero.

Так как moveNearerToZero теперь ссылается на нужную функцию, можно использовать эту константу для отсчета до нуля:

```
print("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
   print("\(currentValue)... ")
   currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// 3...
// 2...
// 1...
// zero!
```

Вложенные функции

Все ранее рассмотренные в этом разделе функции являются глобальными, т. е. определенными в глобальном контексте. Но помимо глобальных можно объявлять и функции, находящиеся внутри других функций, или же вложенные.

Вложенные функции по умолчанию недоступны извне, а вызываются и используются только заключающей функцией. Заключающая функция может также возвращать одну из вложенных, чтобы вложенную функцию можно было использовать за ее пределами.

Приведенный выше пример с функцией chooseStepFunction-(backward:) можно переписать со вложенными функциями:

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
   func stepForward(input: Int) -> Int { return input + 1 }
   func stepBackward(input: Int) -> Int { return input - 1 }
   return backward ? stepBackward : stepForward
}
var currentValue = -4
let moveNearerToZero = chooseStepFunction(backward:
currentValue > 0)
```

```
// moveNearerToZero теперь ссылается на вложенную функцию stepForward()
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// -4...
// -3...
// -2...
// -1...
// zero!
```

4 НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

4.1 Классы в Swift

Swift является объектно-ориентированным языком, а значит позволяет представить программу как набор взаимодействующих между собой объектов.

Ключевым моментом для построения объектно-ориентированного дизайна является возможность абстрагироваться от конкретных взаимодействий в программе, от конкретных предметов и явлений с помощью абстрактных конструкций. В языке Swift такими абстрактными конструкциями являются классы, структуры и перечисления.

Класс является описанием объекта, а объект представляет экземпляр этого класса.

Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке - наличие двух рук, двух ног, головы, пищеварительной, нервной системы, головного мозга и т.д. Есть некоторый шаблон - этот шаблон можно назвать классом. Реально же существующий человек (фактически экземпляр данного класса) является объектом этого класса.

Для определения класса используется ключевое слово class, после которого идет название класса.

```
class User {
    // здесь пишется определение класса
}
```

В данном случае класс называется User. Все тело класса заключается в фигурные скобки.

Классы и объекты

Класс может содержать переменные и константы, которые хранят состояние объекта. Переменные и константы, определенные в классе, еще называют хранимые свойства класса.

```
class User {
    var name: String = "John"
```

```
    var age: Int = 18
}
```

Здесь класс User определяет две переменных name и age, которые хранят соответственно имя и возраст пользователя.

Важно отметить, что здесь не просто определяются переменные, кроме того, им присваиваются начальные значения. Все переменные и константы в классе должны быть инициализированы к моменту использования класса.

Кроме переменных и констант в классе могут определены методы. Методы представляют собой функции, ассоциированные с определенным типом - классом, перечислением или структурой.

```
class User {
    var name: String = "John"
    var age: Int = 18

    func move() {
        print("\(name) is moving")
    }
}
```

В данном примере в классе User объявлен метод move.

Инициализаторы

После определения класса мы можем использовать его в программе, в частности, создавать его объекты. Чтобы создать объект класса используется инициализатор.

```
название_класса()
```

Каждый класс имеет инициализатор по умолчанию. Выражение User() представляет вызов инициализатора.

Важно, что хранимые свойства класса (то есть переменные и константы) должны быть инициализированы и иметь определенное значение ко времени создания объекта класса. В данном случае свойствам класса name и age напрямую присваиваются значения.

```
class User {
```

```
    var name: String = "John"
    var age: Int = 18
}

var john: User = User()
    // User() - вызов инициализатора
```

Также для инициализации свойств может использоваться собственный инициализатор. Для переопределения инициализатора в классе нам надо использвать ключевое слово init. Стоит отметить, что так как переменным name и age не присваиваются начальные значения, а их инициализация производится в инициализаторе, то можно определить эти переменные как константы.

```
class User {
    let age: Int
    let name: String

    init() {
        age = 22
        name = "Tom"
    }
}

var tom: User = User()
```

Фактически инициализатор представляет функцию, которая выполняет начальную инициализацию объекта.

При необходимости мы можем определять дополнительные инициализаторы.

Второй инициализатор принимает два параметра name и age для установки свойств класса.

Так как параметры и свойства класса называются одинаково, то для их разграничения вместе с названиями свойств используется ключевое слово self.

Давайте посмотрим пример вызова данного инициализатора.

```
class User {
    var age: Int
```

```
    var name: String

    init() {/*Инициализация свойств*/}

init(name: String, age: Int){
        self.age = age
        self.name = name
    }
}
```

Пример использования инициализатора с произвольными значениями для полей name и age.

```
var bob: User = User(name: "Bob", age: 34)
```

Инициализаторы могут определять значения по умолчанию для параметров.

```
class User {
    var age: Int
    var name: String

init(name: String = "Tom", age: Int = 21){
        self.age = age
        self.name = name
    }
}
```

Одни инициализаторы могут вызывать другие. Вызывающие инициализаторы должны быть определены с ключевым словом convenience.

```
class User {
    //Объявление свойств name и age

convenience init(){
        self.init(name: "Tom", age: 22)
    }
    init(name: String, age: Int){
```

```
        self.age = age
        self.name = name
    }
}
```

Существует специальная разновидность инициализаторов, которая позволяет возвратить значение nil, если в процессе инициализации объекта произошла какая-нибудь ошибка. Данный тип инициализоторов называется отказоустойчивый инициализатор (Failable Initializer).

Далее представлен пример объявления отказоустойчивого инициализатора.

```
class User{
   //Объявление свойств name и age

  init?(name: String, age: Int) {
        self.name = name
        self.age = age
        if(age < 0) {
           return nil
        }
    }
}
```

Поскольку пользователь, представленный классом User, в принципе не может иметь возраст меньше нуля. Поэтому ситуация, когда для возраста передается число меньше нуля, может рассматриваться как ошибочная. И в этом случае мы как раз можем использовать отказоустойчивый инициализатор.

Для определения такого инициализатора после слова init ставится знак вопроса, а в самом инициализаторе можно предусмотреть ситуацию, при которой он возвращает значение nil.

Возвращая nil, мы тем самым указываем, что мы не можем создать объект User по тем данным, которые переданы в инициализатор.

Важно учитывать, что объект, создаваемый отказоустойчивым инициализатором, будет представлять не тип User, а опциональный тип User?. Поэтому для получения значения нам надо еще использовать операцию ! (восклицательный знак).

var bob: User = User(name: "Bob", age: 34)!

Свойства

Свойства предназначены для хранения состояния объекта. Свойства бывают двух типов:

Хранимые свойства – переменные или константы, определенные на уровне класса или структуры.

Вычисляемые свойства – конструкции, динамически вычисляющие значения. Могут применяться в классе, перечислении или структуре.

Хранимые свойства представляют простейшую форму хранения значений в виде констант или переменных.

```
class User {
    let name: String = "John"
    var age: Int = 18
}
```

При определении хранимых свойств следует предоставить им значения по умолчанию либо напрямую, либо в одном из инициализаторов класса.

```
class User {
    var age: Int
    let name: String

    init() {
        age = 22
        name = "Tom"
    }
}

var tom: User = User()
```

Важно учитывать, что объект, создаваемый этим инициализатором, будет представлять не тип User, а опциональный тип User?. Поэтому для получения значения нам надо еще использовать операцию ! (восклицательный знак).

```
var user: User = User()
print(user.age)     // 22
print(user.name)    // Tom
```

Существуют «ленивые» хранимые свойства, значение которых устанавливается при первом обращении к ним. Использование подобных свойств позволяет более эффективно использовать память, не загромождая ее ненужными объектами, которые могут не потребоваться.

Ленивые свойства определяются с помощью ключевого слова lazy. Модификатор lazy может использоваться только для свойств, которые определяются с помощью var.

```
class User {
    lazy var age: Int = 22
    lazy var name: String = "Tom"
}
```

Вычисляемые свойства не хранят значения, а динамически вычисляют его, используя блок get. Также они могут содержать вспомогательный блок set, который может применяться для установки значения.

```
var имя_свойства: тип {
    get {
        //вычисление значения
    }
    set (параметр) {
        // установка значения
    }
}
```

Рассмотрим следующий пример. Допустим, у нас есть программа, которая вычисляет приыль при вложеннии определенной суммы на определенный период. Свойство profit представляет вычисляемое свойство.

Его блок get возвращает результат арифметических операций.

Блок set позволяет реализовать обратную связь между суммой прибыли и суммой вклада: мы вводим ожидаемую прибыль и получим сумму вклада, необходимую для получения этой прибыли.

Параметр newProfit в блоке set это и присваиваемое значение. newProfit – это случайное название параметра, которое может быть любым, важно понимать, что оно передает значение типа, которое представляет свойство – типа Double.

```
class Account {
    var capital: Double = 0 // сумма вклада
    var rate: Double = 0.01 // процентная ставки

    var profit: Double {
        get {
            return capital + capital * rate
        }
        set(newProfit) {
            self.capital = newProfit / (1 + rate)
        }
    }
}
```

В данном случае когда мы обращаемся к свойству profit срабатывает блок get, а при установке значение – блок set.

```
var account: Account = Account (capital: 1000, rate: 0.1)
print(account.profit) // 1100

// Ожидаемая прибыль
account.profit = 1210
print(account.capital)// 1100 - необходимая сумма вклада
```

Также мы можем использовать сокращенную форму блока set. Переданное значение теперь передается через ключевое слово newValue.

```
class Account {
    var capital: Double = 0
    var rate: Double = 0.01
```

```
    var profit: Double {
                        …
        set{
                                //сокращенная форма
  self.capital = newValue / (1 + rate)
}
    }
}
```

Не всегда в вычисляемых свойствах необходим блок set. Иногда нам не нужно устанавливать новое значение свойства, а требуется только возвратить его. В этом случае мы можем опустить блок set и создать свойство только для чтения.

```
var profit: Double {
        return capital + capital * rate
}
```

В языке Swift существуют наблюдатели свойств, которые следят за изменением значений свойств и при необходимости могут реагировать на эти изменения. Обозреватели свойств вызываются каждый раз при установке нового значения свойства, даже если новое значение не отличается от старого.

Наблюдатели свойств могут быть двух типов:

– willSet: вызывается перед установкой нового значения.
– didSet: вызывается после установки нового значения.

Общий синтаксис наблюдателей свойств можно выразить следующим образом:

```
var свойство: тип {
    willSet (параметр){
        // выражения
    }
    didSet (параметр){
        // выражения
    }
}
```

В языке Swift существуют наблюдатели свойств, которые следят за изменением значений свойств и при необходимости могут реагировать на эти изменения. Обозреватели свойств вызываются каждый раз при установке нового значения свойства, даже если новое значение не отличается от старого.

Статические свойства предваряются ключевым словом static.

Далее представлен пример использования статических свойств.

Важно, что в самом классе мы не можем обратиться к свойству типа просто по имени свойства или через ключевое слово self.

Для обращения к свойству типа требуется писать полное название типа: Greeting.english.

```
class Greeting {
   static let english = "hello"
   static let french = "salut"
   static let german = "halo"
}

print(Greeting.english) // hello
```

Важно, что в самом классе мы не можем обратиться к свойству типа просто по имени свойства или через ключевое слово self.

Для обращения к свойству типа требуется писать полное название типа.

Далее представлен пример использования статических свойств.

```
class Greeting {
   static let english = "hello"
   static let french = "salut"
   static let german = "halo"

   func hiFrench() -> String{
      return Greeting.french;
   }
}
```

В отличии от свойств хранения экземпляра, необходимо всегда присваивать значение по умолчанию свойствам типа. Это потому, что

тип сам по себе не имеет инициализатора, который мог бы присвоить значение хранимому свойству типа.

Хранимые свойства типа отложено инициализируются при первом обращении к ним. Они гарантировано инициализируются только один раз, даже если они доступны сразу для нескольких потоков. Эти свойства не нуждаются в маркировке lazy.

4.2 Наследование

Одной из главных целей объектно-ориентированного подхода является многократное использование кода.

Объединять код для его многократного использования позволяют замыкания и объектные типы данных. В методологии ООП, помимо создания экземпляров различных перечислений, структур и классов, существует возможность создания нового класса на основе уже существующего с автоматическим включением в него всех свойств, методов и сабскриптов класса-родителя. Данный подход называется наследованием.

В рамках наследования «старый» класс называется суперклассом (или базовым классом), а «новый» – подклассом (или субклассом, или производным классом).

Синтаксис наследования

Для наследования одного класса другим необходимо указать имя суперкласса через двоеточие после имени объявляемого класса.

```
СИНТАКСИС
class SomeSubClass: SomeSuperClass {
// тело подкласса
}
```

Для создания производного класса SomeSubClass, для которого базовым является SomeSuperClass, необходимо указать имя суперкласса через двоеточие после имени подкласса.

В результате все свойства и методы, определенные в классе SomeSuperClass, становятся доступными в классе SomeSubClass без их непосредственного объявления в производном типе.

Рассмотрим пример, в котором создается базовый класс Quadruped с набором свойств и методов. Данный класс описывает

сущность «четвероногое животное». Дополнительно объявляется субкласс Dog, описывающий сущность «собака». Все характеристики класса Quadruped применимы и к классу Dog, поэтому их можно наследовать.

```
// суперкласс
class Quadruped {
var type = ""
var name = ""
func walk(){
print("walk")
}
}
// подкласс
class Dog: Quadruped {
func bark(){
print("woof")
}
}
var dog = Dog()
dog.type = "dog"
dog.walk() // выводит walk
dog.bark() // выводит woof
```

Экземпляр myDog позволяет получить доступ к свойствам и методам родительского класса Quadruped. Кроме того, класс Dog расширяет собственные возможности, реализуя в своем теле дополнительный метод bark().

Доступ к наследуемым характеристикам

Доступ к наследуемым элементам родительского класса в производном классе реализуется так же, как к собственным элементам данного производного класса, то есть с использованием ключевого слова self.

В качестве примера в класс Dog добавим метод, выводящий на консоль кличку собаки. Кличка хранится в свойстве name, которое наследуется от класса Quadruped.

```
// подкласс
class Dog: Quadruped {
```

```
func bark(){
print("woof")
}
func printName(){
print(self.name)
}
}
var dog = Dog()
dog.name = "Dragon Wan Helsing"
dog.printName() // выведет Dragon Wan Helsing
```

Для класса безразлично, с какими характеристиками он взаимодействует, собственными или наследуемыми. Данное утверждение справедливо до тех пор, пока не меняется реализация наследуемых характеристик.

Переопределение наследуемых элементов

Субкласс может создавать собственные реализации свойств, методов и сабскриптов, наследуемых от суперкласса. Такие реализации называются переопределением. Для переопределения параметров суперкласса в Swift необходимо указать ключевое слово *override* перед определением элемента.

Переопределение методов

Довольно часто реализация метода, который «достался в наследство» от суперкласса, не соответствует требованиям разработчика. В таком случае в субклассе нужно переписать данный метод, обеспечив к нему доступ по прежнему имени. Объявим новый класс NoisyDog, который описывает сущность «беспокойная собака». Класс Dog является суперклассом для NoisyDog. В описываемый класс необходимо внедрить собственную реализацию метода bark(), но так как одноименный метод уже существует в родительском классе Dog, мы воспользуемся механизмом переопределения.

```
class NoisyDog: Dog{
override func bark(){
print ("woof")
print ("woof")
print ("woof")
```

```
}
}
var badDog = NoisyDog()
badDog.bark() // выводит woof woof woof
```

С помощью ключевого слова override мы сообщаем Swift, что метод bark() в классе NoisyDog имеет собственную реализацию.

Переопределенный метод не знает деталей реализации метода родительского класса. Он знает лишь имя и перечень входных параметров родительского метода.

Доступ к переопределенным элементам суперкласса

Несмотря на то что переопределение изменяет реализацию свойств, методов и сабскриптов, Swift позволяет осуществлять доступ внутри производного класса к переопределенным элементам суперкласса. Для этого в качестве префикса имени элемента вместо ***self*** используется ключевое слово ***super***.

В предыдущем примере в методе bark() класса NoisyDog происходит дублирование кода. В нем используется функция вывода на консоль литерала «woof», хотя данный функционал уже реализован в одноименном родительском методе. Перепишем реализацию метода bark() таким образом, чтобы избежать дублирования кода.

```
class NoisyDog: Dog{
override func bark(){
for _ in 1...3 {
super.bark()
}
}
}
var badDog = NoisyDog()
badDog.bark() // выводит woof woof woof
```

Вывод на консоль соответствует выводу реализации класса из предыдущего примера.

Доступ к переопределенным элементам осуществляется по следующим правилам:

– Переопределенный метод с именем someMethod() может вызвать одноименный метод суперкласса, используя конструкцию super. someMethod() внутри своей реализации (в коде переопределенного метода).

– Переопределенное свойство someProperty может получить доступ к свойству суперкласса с таким же именем, используя конструкцию super.someProperty внутри реализации своего геттера или сеттера.

– Переопределенный сабскрипт someIndex может обратиться к сабскрипту суперкласса с таким же форматом индекса, используя конструкцию super[someIndex] внутри реализации сабскрипта.

Переопределение инициализаторов

Инициализаторы являются такими же наследуемыми элементами, как и методы. Если в подклассе набор свойств, требующих установки значений, не отличается, то наследуемый инициализатор может быть использован для создания экземпляра подкласса.

Тем не менее вы можете создать собственную реализацию наследуемого инициализатора. Запомните, что если вы определяете инициализатор с уникальным для суперкласса и подкласса набором входных аргументов, то вы не переопределяете инициализатор, а объявляете новый.

Если подкласс имеет хотя бы один собственный инициализатор, то инициализаторы родительского класса не наследуются.

В качестве примера переопределим наследуемый от суперкласса Quadruped пустой инициализатор. В классе Dog значение наследуемого свойства type всегда должно быть равно «dog». В связи с этим перепишем реализацию инициализатора таким образом, чтобы в нем устанавливалось значение данного свойства.

```
class Dog: Quadruped {
override init(){
super.init()
self.type = "dog"
}
}
func bark(){
print("woof")
```

```
}
func printName(){
print(self.name)
}
```

Прежде чем получать доступ к наследуемым свойствам в переопределенном инициализаторе, необходимо вызвать инициализатор родителя. Он выполняет инициализацию всех наследуемых свойств.

Если в подклассе есть собственные свойства, которых нет в суперклассе, то их значения в инициализаторе необходимо указать до вызова инициализатора родительского класса.

Переопределение наследуемых свойств

Как отмечалось ранее, вы можете переопределять любые наследуемые элементы. Наследуемые свойства иногда ограничивают функциональные возможности субкласса. В таком случае можно переписать геттер или сеттер данного свойства или при необходимости добавить наблюдатель.

С помощью механизма переопределения вы можете расширить наследуемое «только для чтения» свойство до «чтение–запись», реализовав в нем и геттер и сеттер. Но обратное невозможно: если у наследуемого свойства реализованы и геттер и сеттер, вы не сможете сделать из него свойство «только для чтения».

Субкласс не знает деталей реализации наследуемого свойства в суперклассе, он знает лишь имя и тип наследуемого свойства. Поэтому необходимо всегда указывать и имя и тип переопределяемого свойства.

Превентивный модификатор final

Swift позволяет защитить реализацию класса целиком или его отдельных элементов. Для этого необходимо использовать превентивный модификатор ***final***, который указывается перед объявлением класса или его отдельных элементов:

- final class для классов;
- final var для свойств;
- final func для методов;
- final subscript для сабскриптов.

При защите реализации класса его наследование в другие классы становится невозможным. Для элементов класса их наследование происходит, но переопределение становится недоступным.

Подмена экземпляров классов

Наследование, помимо всех перечисленных возможностей, позволяет заменять требуемые экземпляры определенного класса экземплярами одного из подклассов.

Рассмотрим пример. В нем объявим массив элементов типа Quadruped и добавим в него несколько элементов.

```
var animalsArray: [Quadruped] = []
var someAnimal = Quadruped()
var myDog = Dog()
var badDog = NoisyDog()
animalsArray.append(someAnimal)
animalsArray.append(myDog)
animalsArray.append(badDog)
```

В результате в массив animalsArray добавляются элементы типов Dog и NoisyDog. Это происходит несмотря на то, что в качестве типа массива указан класс Quadruped.

Приведение типов

Ранее нами были созданы три класса: Quadruped, Dog и NoisyDog, а также определен массив animalsArray, содержащий элементы всех трех типов данных. Данный набор типов представляет собой иерархию классов, поскольку между всеми классами можно указать четкие зависимости (кто кого наследует). Для анализа классов в единой иерархии существует специальный механизм, называемый приведением типов.

Путем приведения типов вы можете выполнить следующие операции:

– проверить тип конкретного экземпляра класса на соответствие некоторому типу или протоколу;

– преобразовать тип конкретного экземпляра в другой тип той же иерархии классов.

Проверка типа

Проверка типа экземпляра класса производится с помощью оператора ***is***. Данный оператор возвращает true в случае, когда тип проверяемого экземпляра является указанным после оператора классом или наследует его. Для анализа возьмем определенный и заполненный ранее массив animalsArray.

```
for item in animalsArray {
if item is Dog {
print("Yap")
}
}
// Yap выводится 2 раза
```

Данный код перебирает все элементы массива animalsArray и проверяет их на соответствие классу Dog. В результате выясняется, что ему соответствуют только два элемента массива: экземпляр класса Dog и экземпляр класса NoisyDog.

Преобразование типа

Как неоднократно отмечалось ранее, объявленный и наполненный массив animalsArray имеет элементы разных типов данных из одной иерархической структуры. Несмотря на это, при получении очередного элемента вы будете работать исключительно с использованием методов класса, указанного в типе массива (в данном случае Quadruped).

То есть, получив элемент типа Dog, вы не увидите определенный в нем метод bark(), поскольку Swift подразумевает, что вы работаете именно с экземпляром типа Quadruped.

Для того чтобы преобразовать тип и сообщить Swift, что данный элемент является экземпляром определенного типа, используется оператор ***as***, точнее, две его вариации: ***as?*** и ***as!***. Данный оператор ставится после имени экземпляра, а после него указывает имя класса, в который преобразуется экземпляр.

Между обеими формами оператора существует разница:

– as? ИмяКласса возвращает либо экземпляр типа ИмяКласса? (опционал), либо nil в случае неудачного преобразования;

– вариант as! ИмяКласса производит принудительное извлечение значения и возвращает экземпляр типа ИмяКласса или вызывает ошибку в случае неудачи.

Снова приступим к перебору массива animalsArray. На этот раз будем вызывать метод bark(), который не существует в суперклассе Quadruped, но присутствует в подклассах Dog и NoisyDog.

```
for item in animalsArray {
if var animal = item as? NoisyDog {
animal.bark()
}else if var animal = item as? Dog {
animal.bark()
}else{
item.walk()
}
}
```

Каждый элемент массива animalArray записывается в параметр item.

Далее в теле цикла данный параметр с использованием оператора as? пытается преобразоваться в каждый из типов данных нашей структуры классов. Если item преобразуется в тип NoisyDog или Dog , то у него становится доступным метод bark() .

4.3 Полиморфизм

Полиморфизм представляет возможность взаимозаменяемости типов, которые находятся в одной иерархии классов. Например, возьмем следующую иерархию классов:

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

    init(name: String, age: Int){
        self.name = name
        self.age = age
    }
    func display(){
```

```
        print("Имя: \(name)  Возраст: \(age)")
    }
}

class Employee : Person{
    var company: String
    init(name: String, age: Int, company: String) {
        self.company = company
        super.init(name:name, age: age)
    }
    override func display(){
        print("Имя: \(name)  Возраст: \(age)  Сотрудник компании: \(company)")
```

В данном случае класс Manager (менеджер компании) наследуется от класса Employee (сотрудник компании), а класс Employee – от класса Person (человек). Тем самым класс Manager ненапрямую тоже наследуется от Person.

Поскольку и сотрудник компании и менеджер компании в то же время являются людьми, то есть объектами класса Person, то мы можем написать следующим образом:

```
let tom: Person = Person(name:"Tom", age: 23)
let bob: Person = Employee(name: "Bob", age: 28, company: "Apple")
let alice: Person = Manager(name: "Alice", age: 31, company: "Microsoft")
```

Все три константы представляют тип Person, однако первая хранит ссылку на объект Person, вторая – на объект Employee, а третья – на объект Manager. Таким образом, переменная или константа одного типа может принимать многообразные формы в зависимости от конкретного объекта, на который она указывает.

Но что будет, если мы вызовем метод display() для всех трех объектов:

```
let tom: Person = Person(name:"Tom", age: 23)
let bob: Person = Employee(name: "Bob", age: 28, company: "Apple")
```

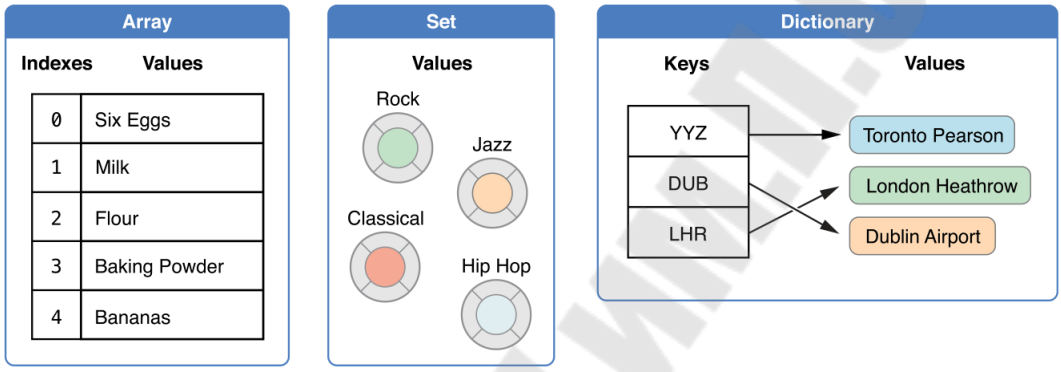
```
let alice: Person = Manager(name: "Alice", age: 31, company: "Microsoft")

tom.display()       // Имя: Tom  Возраст: 23
bob.display()       // Имя: Bob  Возраст: 28  Сотрудник компании: Apple
alice.display()     // Имя: Alice  Возраст: 31  Менеджер компании: Microsoft
```

Несмотря на то, что все три константы представляют тип Person, при вызове метода display будет вызываться реализация метода именно того класса, ссылку на объект которого хранит константа. Данный примем называется динамической диспетчеризацией – во время выполнения программы на основании типа объекта система решает, какую именно реализацию метода вызывать. В одной стороны, это нам дает ряд преимуществ – мы можем работать с объектом производного типа как с объектом базового типа и использовать его везде, где требуется объект базового типа. Но с другой стороны, поскольку решение о выбере реализации принимается во время выполнения, то это несколько замедляет общий ход работы программы.

5 КОЛЛЕКЦИИ

Swift обеспечивает три основных типа коллекций – это Массивы, Множества и Словари для хранения коллекций значений. Массивы – это упорядоченные коллекции значений. Множества – это неупорядоченные коллекции уникальных значений. Словари – это неупорядоченные коллекции, хранящие пары "ключ-значение".



Массивам, множествам и словарям в Swift всегда понятно, какие типы значений и ключи они могут хранить. Это означает, что вы не можете по ошибке вставить неверное значение в коллекцию. Это также означает, что вы можете быть уверены в типах значений, которые вы получите из коллекции.

Изменчивость коллекций

Когда вы создаете массив или словарь и присваиваете его переменной, то созданная коллекция будет изменяемой. Это означает, что вы можете изменить коллекцию после ее создания путем добавления, удаления, или изменения элементов этой коллекции. И наоборот, когда вы присвоите массив или словарь константе, то он будет неизменяемым, а его размер и содержимое не может быть изменено.

5.1 Массив

Сокращённый синтаксис массивов

Полная форма записи массива в Swift пишется Array<Element>, где Element это тип значения, который может хранить массив.

Вы можете также написать массив в сокращенной форме как [Element].

Создание пустого массива

Вы можете создать пустой массив определенного типа с помощью синтаксиса инициализатора:

```
var someInts = [Int]()
print("someInts is of type [Int] with \(someInts.count) items.")
// Выведет "someInts is of type [Int] with 0 items."
```

В качестве альтернативы, если контекст уже содержит информацию о типе, например, аргумент функции или уже типизированную переменную или константу, вы можете создать пустой массив с помощью пустого литерала массива, который записывается в виде [] (пустой пары квадратных скобок):

```
someInts.append(3)
// массив someInts теперь содержит одно значение типа Int
someInts = []
// массив someInts теперь пуст, но все равно имеет тип [Int]
```

Создание массива с дефолтным значением

Тип массива в Swift также обеспечивает инициализатор для создания массива определенного размера со всеми его значениями, установленными на одно и тоже дефолтное значение. Вы передаете этому инициализатору дефолтное значение соответствующего типа (называемое repeating): и сколько раз это значение повторяется в новом массиве (так называемый count):

```
var threeDoubles = Array(repeating: 0.0, count: 3)
// threeDoubles имеет тип [Double] и равен [0.0, 0.0, 0.0]
```

Создание массива, путем объединения двух массивов

Вы можете создать новый массив, объединив два существующих массива с совместимыми типами с оператором сложения (+). Новый тип массива выводится из типа двух массивов, которые вы объединяете вместе:

```
var anotherThreeDoubles = Array(repeating: 2.5, count: 3)
// anotherThreeDoubles имеет тип [Double] и равен [2.5, 2.5, 2.5]
```

```
var sixDoubles = threeDoubles + anotherThreeDoubles
// тип sixDoubles выведен как [Double] и равен [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]
```

Создание массива через литералы массива

Вы можете инициализировать массив с помощью литерала массива, который является быстрым способом писать одно или несколько значений как набор значений массива.

Литерал массива пишется в виде списка значений, разделенных запятыми и окруженными парой (квадратных) скобок:

```
[значение 1, значение 2, значение 3]
```

В приведенном ниже примере создается массив под названием shoppingList для хранения String значений:

```
var shoppingList: [String] = ["Eggs", "Milk"]
// shoppingList был инициализирован с двумя элементами
```

Переменная shoppingList объявлена как "массив из String значений", который записывается как [String]. Поскольку для этого массива указан тип значения String, ему разрешено хранить только String значения. Здесь, массив shoppingList инициализирован двумя String значениями ("Eggs" и "Milk"), написанными внутри литерала массива.

Доступ и изменение массива

Вы можете получить доступ к массиву и изменять его либо через его методы и свойства, либо используя синтаксис сабскриптов.

Чтобы узнать количество элементов в массиве, проверьте его read-only свойство count:

```
print("The shopping list contains \(shoppingList.count) items.")
// Выведет "The shopping list contains 2 items."
```

Логическое свойство isEmpty можно использовать в качестве быстрого способа узнать, является ли свойство count равным 0:

```
if shoppingList.isEmpty {
```

```
print("The shopping list is empty.")
} else {
print("The shopping list is not empty.")
}
// Выведет "The shopping list is not empty."
```

Вы можете добавить новый элемент в конец массива через вызов метода append:

```
shoppingList.append("Flour")
// shoppingList теперь содержит 3 элемента, а кое-кто делает блины
```

Кроме того, добавить массив с одним или несколькими совместимыми элементами можно с помощью оператора сложения и присвоения (+=):

```
shoppingList += ["Baking Powder"]
// shoppingList теперь хранит 4 элемента
shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
// shoppingList теперь хранит 7 элементов
```

Можно извлечь значение из массива с помощью синтаксиса сабскриптов, поместив индекс значения, который вы хотите получить, внутри квадратных скобок сразу после имени массива.

```
var firstItem = shoppingList[0]
// firstItem равен "Eggs"
```

Для вставки элемента по заданному индексу внутрь массива, вызовите его метод insert(_:at:) :

```
shoppingList.insert("Maple Syrup", at: 0)
// shoppingList теперь содержит 7 элементов
// "Maple Syrup" теперь первый элемент списка
```

Вызвав этот insert(_:at:) метод, мы вставили новый элемент со значением "Maple Syrup" в самое начало списка покупок, то есть в элемент с индексом 0.

Аналогичным образом можно удалить элемент из массива с помощью метода remove(at:). Этот метод удаляет элемент с указанным индексом и возвращает удалённый элемент (хотя вы можете игнорировать возвращаемое значение если оно вам не нужно):

```
let mapleSyrup = shoppingList.remove(at: 0)
// элемент который имел индекс 0 был удален
// shoppingList теперь содержит 6 элементов, и нет Maple Syrup
// константа mapleSyrup теперь равна удаленной строке "Maple Syrup"
```

Если вы хотите удалить последний элемент из массива, то можно использовать метод removeLast() вместо remove(at:), чтобы избежать необходимости запроса свойства count для массива. Также как и метод remove(at:), removeLast() возвращает удаленный элемент:

```
let apples = shoppingList.removeLast()
// последний элемент массива был удален
// shoppingList теперь содержит 5 элементов, и нет яблок
// константа apples теперь равна удаленной строке "Apples"
```

5.2 Множества

Множество хранит различные значения одного типа в виде коллекции в неупорядоченной форме. Вы можете использовать множества как альтернативы массиву, когда порядок для вас значения не имеет или когда вам нужно быть уверенным в том, что значения внутри коллекции не повторяются.

Синтаксис типа множества

Тип множества Swift записывается как Set<Element>, Element является типом, который хранится в множестве. В отличие от массивов множества не имеют сокращенной формы записи.

Создание и инициализация пустого множества

Вы можете создать пустое множество конкретного типа, используя синтаксис инициализатора:

```
var letters = Set<Character>()
```

```
print("letters имеет тип Set<Character> с \(letters.count) элементами.")
// Выведет "letters имеет тип Set<Character> с 0 элементами."
```

Доступ и изменение множества

Получить доступ и модифицировать множества можно через свойства и методы.

Для того, чтобы выяснить количество элементов в множестве вам нужно использовать свойство count:

```
print("У меня есть \(favoriteGenres.count) любимых музыкальных жанра.")
// Выведет "У меня есть 3 любимых музыкальных жанра."
```

Используйте булево свойство isEmpty в качестве сокращенной проверки наличия элементов во множестве или другими словами равно ли свойство count 0:

```
if favoriteGenres.isEmpty {
   print("Мне все равно какая музыка играет. Я не придирчив.")
} else {
   print("У меня есть свои музыкальные предпочтения.")
}
// Выведет "У меня есть свои музыкальные предпочтения."
```

Вы можете добавить новый элемент во множество, используя метод insert(_:):

```
favoriteGenres.insert("Jazz")
// теперь в favoriteGenres находится 4 элемента
```

Вы так же можете удалить элемент из множества, используя метод remove(_:), который удаляет элемент, который является членом множества и возвращает удаленное значение или nil, если удаляемого элемента нет. Так же все объекты множества могут быть удалены единовременно при помощи метода removeAll().

```
if let removedGenre = favoriteGenres.remove("Rock") {
   print("\(removedGenre)? С меня хватит.")
```

```
} else {
   print("Меня это не сильно заботит.")
}
// Выведет "Rock? С меня хватит."
```

Можно проверить наличие определенного элемента во множестве, используя метод contains(_:):

```
if favoriteGenres.contains("Funk") {
   print("О! Да я встал с правильной ноги!")
} else {
   print("Слишком много Funk'a тут.")
}
// Выведет "Слишком много Funk'a тут."
```

Выполнение операций с множествами

Вы можете очень эффективно использовать базовые операции множеств, например, комбинирование двух множеств, определение общих значений двух множеств, определять содержат ли множества несколько, все или ни одного одинаковых значения.

Базовые операции множеств

Иллюстрации внизу изображают два множества a и b в результате применения различных методов (рисунок 8).

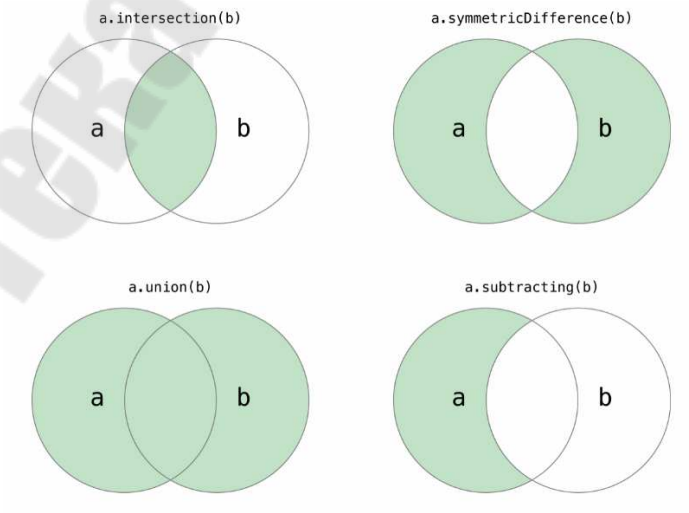


Рисунок 8 – Базовые операции множеств

– Используйте метод union (_:) для создания нового множества состоящего из всех значений обоих множеств.

– Используйте метод intersection(_:) для создания нового множества из общих значений двух входных множеств.

– Используйте метод subtracting (_:) для создания множества со значениями не принадлежащих указанному множеству из двух входных.

– Используйте метод symmetricDifference(_:) для создания нового множества из значений, которые не повторяются в двух входных множествах.

```
let oddDigits: Set = [1, 3, 5, 7, 9]
let evenDigits: Set = [0, 2, 4, 6, 8]
let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]

oddDigits.union(evenDigits).sorted()
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
oddDigits.intersection(evenDigits).sorted()
// []
oddDigits.subtracting(singleDigitPrimeNumbers).sorted()
// [1, 9]
oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()
// [1, 2, 9]
```

Взаимосвязь и равенство множеств

Иллюстрация ниже отображает три множества a, b и c (рисунок 9).

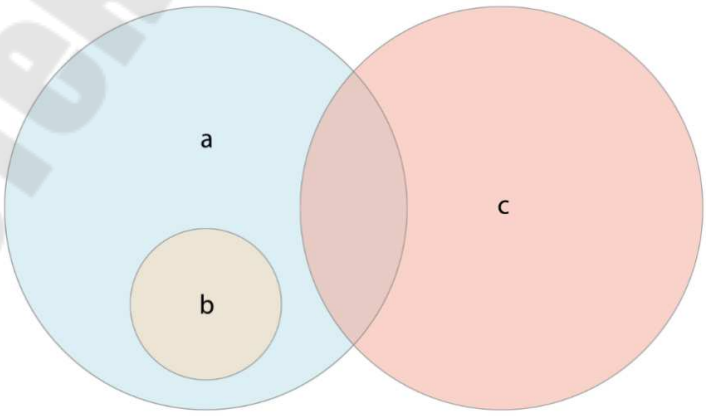


Рисунок 9 – Взаимосвязь и равенство множеств

Множество a является надмножеством множества b, так как содержит все его элементы, соответственно множество b является подмножеством множества a, опять таки потому, что все его элементы находятся в a. Множества b и c называются разделенными, так как у них нет общих элементов.

– Используйте оператор равенства (==) для определения все ли значения двух множеств одинаковы.

– Используйте метод isSubset(of:) для определения все ли значения множества содержатся в указанном множестве.

– Используйте метод isSuperset(of:), чтобы определить содержит ли множество все значения указанного множества.

– Используйте
методы isStrictSubset(of:) или isStrictSuperset(of:) для определения является ли множество подмножеством или надмножеством, но не равным указанному сету.

– Используйте метод isDisjoint(with:) для определения того, отсутствуют ли общие значения в двух множествах или нет.

```
let houseAnimals: Set = ["собака", "кошка"]
let farmAnimals: Set = ["корова", "курица", "баран", "собака", "кошка"]
let cityAnimals: Set = ["ворона", "мышь"]
houseAnimals.isSubset(of: farmAnimals)
// true
farmAnimals.isSuperset(of: houseAnimals)
// true
farmAnimals.isDisjoint(with: cityAnimals)
// true
```

5.3 Словари

Словарь представляет собой контейнер, который хранит несколько значений одного и того же типа. Каждое значение связано с уникальным ключом, который выступает в качестве идентификатора этого значения внутри словаря. В отличие от элементов в массиве, элементы в словаре не имеют определенного порядка. Используйте словарь, когда вам нужно искать значения на основе их идентификатора, так же как в реальном мире словарь используется для поиска определения конкретного слова.

Сокращенный синтаксис словаря

В Swift тип словаря в полной форме пишется как Dictionary<Key, Value>, где Key это тип значения который используется как ключ словаря, а Value это тип значения который словарь может хранить для этих ключей.

Вы можете также написать словарь в сокращенной форме как [Key: Value].

Создание пустого словаря

Подобно массивам вы можете создать пустой словарь определенного типа с помощью синтаксиса инициализатора:

```
var namesOfIntegers = [Int: String]()
// namesOfIntegers является пустым [Int: String] словарем
```

В этом примере создается пустой словарь с типом [Int: String] для хранения удобных для восприятия имен числовых значений. Его ключи имеют тип Int, а значения - String.

Если контекст уже предоставляет информацию о типе, вы можете создать пустой словарь с помощью литерала пустого словаря, который пишется как [:] (двоеточие внутри пары квадратных скобок):

```
namesOfIntegers[16] = "sixteen"
// namesOfIntegers теперь содержит 1 пару ключ-значение
namesOfIntegers = [:]
// namesOfIntegers теперь опять пустой словарь с типом [Int: String]
```

Создание словаря с литералом словаря

Литерал словаря это краткий способ написать одну или несколько пар ключ-значение в виде коллекций словаря.

Пара ключ-значение является комбинацией ключа и значения. В литерале словаря, ключ и значение в каждой паре ключ-значение разделено двоеточием. Пары ключ-значение написаны как список, разделенный запятыми и окруженный парой квадратных скобок:

```
[ключ 1: значение 1, ключ 2: значение 2, ключ 3: значение 3]
```

В примере ниже создается словарь, который хранит имена международных аэропортов. В этом словаре ключи являются трехбуквенным кодом международной ассоциации воздушного транспорта, а значения - названия аэропортов:

```
var airports: [String: String] = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]
```

Словарь airports объявлен с типом [String: String], что означает "словарь ключи которого имеют тип String и значения которого также имеют тип String".

Словарь airports инициализирован с помощью литерала словаря, содержащего две пары ключ-значение. Первая пара имеет ключ "YYZ" и значение "Toronto Pearson". Вторая пара имеет ключ "DUB" и значение "Dublin".

Этот словарь содержит две пары String: String. Этот тип ключ-значение подходит типу который мы присвоили переменной airports (словарь содержащий только String ключи, и только String значения), и поэтому присвоение литерала словаря допустимо в качестве способа инициализации словаря airports двумя начальным элементами.

Подобно массивам, вы можете не писать тип словаря если вы инициализируете его с помощью литерала словаря, чьи ключи и значения имеют соответствующие типы. Инициализация airports может быть записана в более краткой форме:

```
var airports = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]
```

Поскольку все ключи в литерале имеют одинаковый тип, и точно так же все значения имеют одинаковый тип, то Swift может вывести, что [String: String] является правильным типом для использования в словаре airports.

Доступ и изменение словаря

Вы можете получить доступ к словарю и изменять его либо через его методы и свойства, либо используя синтаксис индексов. Подобно массивам, вы можете узнать количество элементов в словаре через его read-only свойство count:

```
print("The airports dictionary contains \(airports.count) items.")
```

```
// Выведет "The airports dictionary contains 2 items."
```

Логическое свойство isEmpty можно использовать в качестве быстрого способа узнать, является ли свойство count равным 0:

```
if airports.isEmpty {
  print("The airports dictionary is empty.")
} else {
  print("The airports dictionary is not empty.")
}
// Выведет "The airports dictionary is not empty."
```

Вы можете добавить новый элемент в словарь с помощью синтаксиса индексов. Используйте новый ключ соответствующего типа в качестве индекса, а затем присвойте новое значение соответствующего типа:

```
airports["LHR"] = "London"
// словарь airports теперь содержит 3 элемента
```

Вы также можете использовать синтаксис индексов для изменения значения связанного с определенным ключом:

```
airports["LHR"] = "London Heathrow"
// значение для "LHR" поменялось на "London Heathrow"
```

В качестве альтернативы индексам, можно использовать метод словаря updateValue(forKey:), чтобы установить или обновить значение для определенного ключа. Подобно примерам с индексами вверху, метод updateValue(forKey:) устанавливает значение для ключа если оно не существует, или обновляет значение, если этот ключ уже существует. Однако, в отличие от индексов, метод updateValue(forKey:) возвращает старое значение после выполнения обновления. Это позволяет вам проверить состоялось ли обновление или нет.

Метод updateValue(forKey:) возвращает опциональное значение соответствующее типу значения словаря. Например, для словаря, который хранит String значения, метод возвратит String? тип, или "опциональный String". Это опциональное значение содержит старое значение для этого ключа, если оно существовало до обновления, либо nil если значение не существовало.

```
if let oldValue = airports.updateValue("Dublin Airport", forKey: "DUB") {
  print("The old value for DUB was \(oldValue).")
}
// Выведет "The old value for DUB was Dublin."
```

Вы также можете использовать синтаксис индексов чтобы получить значение из словаря для конкретного ключа. Поскольку есть вероятность запросить ключ для несуществующего значения, индекс словаря возвращает опциональное значение, соответствующее типу значений словаря. Если словарь содержит значение для запрошенного ключа, индекс возвращает опциональное значение, содержащее существующее значение для этого ключа. В противном случае индекс возвращает nil:

```
if let airportName = airports["DUB"] {
  print("The name of the airport is \(airportName).")
} else {
  print("That airport is not in the airports dictionary.")
}
// Выведет "The name of the airport is Dublin Airport."
```

Вы можете использовать синтаксис индексов для удаления пары ключ-значение из словаря путем присвоения nil значению для этого ключа:

```
airports["APL"] = "Apple International"
// "Apple International" несуществующий аэропорт для APL, так что удалим его
airports["APL"] = nil
// APL теперь был удален из словаря"
```

Кроме того, можно удалить пару ключ-значение из словаря с помощью метода removeValue(forKey:). Этот метод удаляет пару ключ-значение если она существует и затем возвращает значение, либо возвращает nil если значения не существует:

```
if let removedValue = airports.removeValue(forKey: "DUB") {
  print("The removed airport's name is \(removedValue).")
```

```
} else {
  print("The airports dictionary does not contain a value for DUB.")
}
// Выведет "The removed airport's name is Dublin Airport."
```

Итерация по словарю

Вы можете сделать итерацию по парам ключ-значение в словаре с помощью for-in цикла. Каждое значение в словаре возвращается как кортеж (ключ, значение),и вы можете разложить части кортежа по временным константам или переменным в рамках итерации:

```
for (airportCode, airportName) in airports {
  print("\(airportCode): \(airportName)")
}
// LHR: London Heathrow
// YYZ: Toronto Pearson
```

Вы также можете получить коллекцию ключей или значений словаря через обращение к его свойствам keys и values:

```
for airportCode in airports.keys {
  print("Airport code: \(airportCode)")
}
// Airport code: LHR
// Airport code: YYZ

for airportName in airports.values {
  print("Airport name: \(airportName)")
}
// Airport name: London Heathrow
// Airport name: Toronto Pearson
```

Если вам нужно использовать ключи или значения словаря вместе с каким-либо API, которое принимает объект Array, то можно инициализировать новый массив с помощью свойств keys и values:

```
let airportCodes = [String](airports.keys)
// airportCodes теперь ["YYZ", "LHR"]
```

```
let airportNames = [String](airports.values)
// airportNames теперь ["Toronto Pearson", "London Heathrow"]
```

Тип словаря в Swift является неупорядоченной коллекцией. Для итерации по ключам или значениям словаря в определенной последовательности, используйте метод sorted() для свойств keys или values словаря.

6 ОБОБЩЕНИЯ

Обобщения (универсальные шаблоны, generics) позволяют вам писать гибкие, общего назначения функции и типы, которые могут работать с любыми другими типами, с учетом требований, которые вы определили. Вы можете написать код, который не повторяется и выражает свой контент в ясной абстрактной форме.

Проблема, которую решают обобщения

Приведем обычную, стандартную, неуниверсальную функцию swapTwoInts(_:_:), которая меняет два Int местами:

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
   let temporaryA = a
   a = b
   b = temporaryA
}
```

Эта функция использует сквозные параметры для замещения значения a и b.

Функция swapTwoInts(_:_:) обменивает начальные значения переменных a и b местами. Вы можете использовать эту функцию для замещения двух значений типа Int:

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// Выведет "someInt is now 107, and anotherInt is now 3"
```

Функция swapTwoInts(_:_:) полезная, но она применима только для значений типа Int. Если вы хотите поменять местами два значения типа String или два значения Double, то вам придется написать больше функций, к примеру, swapTwoStrings(_:_:) или swapTwoDoubles(_:_:), которые показаны ниже:

```
func swapTwoStrings(_ a: inout String, _ b: inout String) {
   let temporaryA = a
   a = b
   b = temporaryA
```

```
}

func swapTwoDoubles(_ a: inout Double, _ b: inout Double) {
   let temporaryA = a
   a = b
   b = temporaryA
}
```

Вы может быть заметили, что тела функций swapTwoInts(_:_:), swapTwoStrings(_:_:), и swapTwoDouble(_:_:) идентичны. Единственное отличие в том, что они поддерживают значения различных типов (Int, String, и Double).

Было бы намного удобнее написать одну более гибкую функцию, которая бы могла заменить значения двух переменных любого типа. Универсальный код позволяет вам написать такую функцию.

Универсальные функции

Универсальные функции могут работать с любыми типами. Ниже приведена универсальная версия функции swapTwoInts(_:_:), которая теперь называется swapTwoValues(_:_:):

```
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
   let temporaryA = a
   a = b
   b = temporaryA
}
```

Тело функции swapTwoValues(_:_:) идентично телу функции swapTwoInts(_:_:). Однако первая строка функции swapTwoValues(_:_:) немного отличается от аналогичной строки функции swapTwoInts(_:_:). Вот как можно сравнить первые строки этих функций:

```
func swapTwoInts(_ a: inout Int, _ b: inout Int)
func swapTwoValues<T>(_ a: inout T, _ b: inout T)
```

Универсальная версия использует заполнитель имени типа (называется T в нашем случае) вместо текущего имени типа (Int, String, Double…). Заполнитель имени типа ничего не говорит о том, чем должно являться T, но он говорит о том, что и a и b должны быть одного типа T, независимо от того, что такое T. Текущий тип T будет определяться каждый раз, как вызывается функция swapTwoValues(_:_:).

Другое отличие в том, что за именем универсальной функции (swapTwoValues(_:_:)) идет заполнитель имени типа (T) в угловых скобках (<T>). Угловые скобки говорят Swift, что T является заполнителем имени типа внутри определения функции swapTwoValues(_:_:). Так как T является заполнителем, то Swift не смотрит на текущее значение T.

Функция swapTwoValues(_:_:) теперь может быть вызвана точно так же как и функция swapTwoInts, за исключением того, что в нее можно передавать значения любого типа, до тех пор пока они одного типа. Каждый раз при вызове функции swapTwoValues(_:_:), тип T выводится из типов, которые передаются в эту функцию.

В двух примерах ниже T имеет значение типа Int и String соответственно:

```
var someInt = 3
var anotherInt = 107
swapTwoValues(&someInt, &anotherInt)
// someInt равен 107, a anotherInt равен 3

var someString = "hello"
var anotherString = "world"
swapTwoValues(&someString, &anotherString)
// someString равна "world", a anotherString равна "hello"
```

Параметры типа

В примере выше в функции swapTwoValues(_:_:) заполнитель имени типа T пример параметра типа. Параметры типа определяют и называют тип наполнителя, и пишутся сразу после имени функции, между угловыми скобками (например, <T>).

Как только вы определили параметр типа, то вы можете использовать его в качестве типа параметра функции (как например, параметры a и b в функции swapTwoValues(_:_:)) или как возвращаемый функциональный тип, или как аннотация типа внутри тела функции. В каждом случае заполнитель типа отображается параметром типа, который заменяется на актуальное значение типа при вызове функции. (В swapTwoValues(_:_:) в нашем примере выше произошло замещение T на Int в момент, когда функция была вызвана в первый раз, и на String при повторном вызове.)

Вы можете использовать несколько параметров типа, просто вписав их в угловых скобках через запятую.

Именование параметров типа

В большинстве случаев параметры типа имеют описательные имена, такие как Key и Value в Dictionary<Key, Value> и Element в Array<Element>, которые помогут читающему код определить взаимосвязь между параметром типа и универсальным типом или функцией, в которых он используется. Тем не менее, когда между ними нет значимых отношений, то по традиции именами становятся отдельные буквы, такие как T, U, V, как например T в функции swapTwoValues(_:_:).

Универсальные типы

В дополнение к универсальным функциям, Swift позволяет вам определять ваши универсальные типы. Это к примеру универсальные классы, структуры и перечисления, которые могут работать с любыми типами, наподобие тому, как работают Array или Dictionary.

Эта секция покажет вам как создать вашу универсальную коллекцию типа Stack. Stack - упорядоченная коллекция значений, аналогичная массиву, но с более строгим набором операций, чем имеет тип Array языка Swift. Массив позволяет вам вставлять и удалять элементы с любой позиции массива. Однако, Stack позволяет добавлять новые элементы только в конец коллекции. Аналогично стек позволяет удалять элементы только с конца коллекции.

Ниже приведена иллюстрация поведения добавления и удаления элемента из стека (рисунок 10).

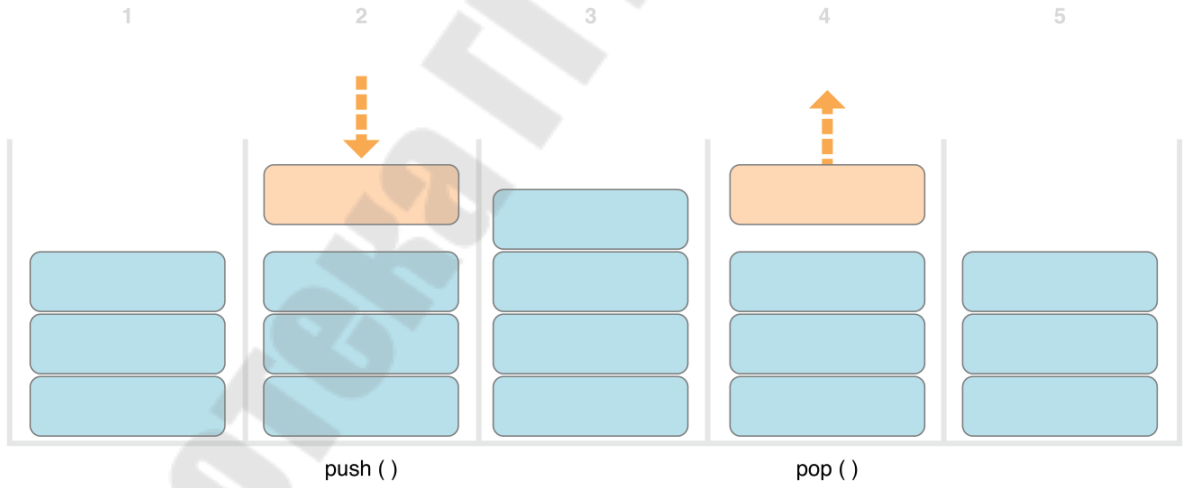


Рисунок 10 – Добавление и удаление элементов из стека

1. На данный момент у нас три значения в стеке.
2. Четвертое значение “затолкнули” на самый верх стека.
3. На этот момент в стеке находится четыре значения, самое свежее значение находится наверху.
4. Последнее значение удалено или “выстреляно” из стека.

5. После удаления значения, стек снова имеет три значения.

Вот как написать неуниверсальную версию стека, в этом случае мы используем стек для хранения Int значений:

```
struct IntStack {
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

Эта структура использует свойство items типа Array для хранения значений в стеке. Stack предоставляет нам два метода, push и pop для добавления последнего элемента в стек и для удаления последнего элемента из стека. Эти методы отмечены как mutating, потому как они вынуждены менять массив items.

Тип IntStack, показанный выше, может быть использован только со значениями Int. Но он будет куда полезнее, если мы определим его как универсальный класс Stack, который может управлять стеком любого типа.

Вот универсальная версия структуры:

```
struct Stack<Element> {
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
}
```

Обратите внимание как универсальная версия Stack похожа на не универсальную, вообще отличаясь только тем, что мы используем заполнитель типа, вместо указания конкретного типа Int. Этот параметр типа написан внутри угловых скобок (<Element>), сразу после имени структуры.

Element определяет заполнитель имени типа для “какого-то типа Element”, который будет предоставлен позже. Этот будущий тип

может ссылаться на Element в любом месте определения структуры. В этом случае наш некоторый тип Element используется в трех местах:

– Для создания свойства items, которое инициализируется пустым массивом типа Element.

– Для указания того, что метод push(_:) имеет единственный параметр item, который должен быть типа Element.

– Для указания типа возвращаемого значения методом pop(), которое должно быть типом Element.

Из-за того, что это является универсальным типом, то Stack может быть использован для создания стека любых корректных типов в Swift, аналогичным образом как это осуществляют типы Array или Dictionary.

Вы создаете новый экземпляр Stack, вписав тип хранимых значений стека в угловые скобки. Например, создадим новый стек строк, вы напишите Stack<String>():

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
stackOfStrings.push("cuatro")
// stack содержит 4 строки
```

Теперь stackOfStrings выглядит вот так после добавления последних четырех значений (рисунок 11).

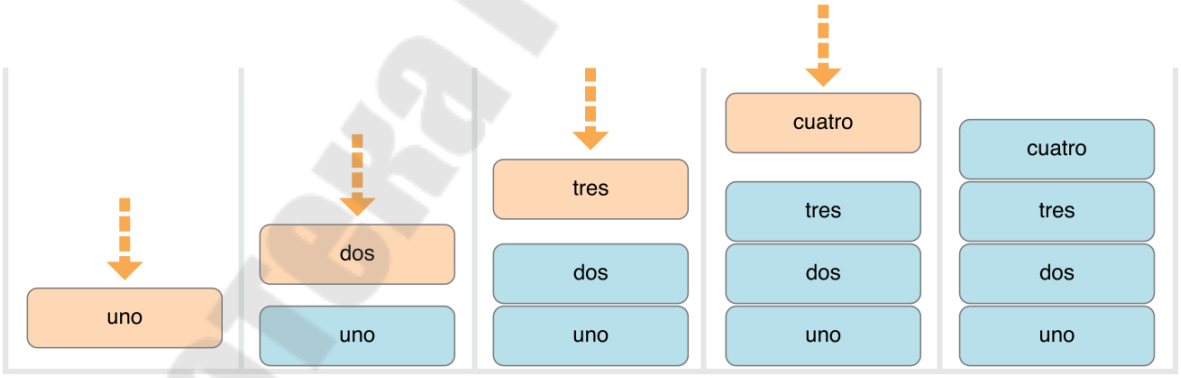


Рисунок 11 – Стек после добавления четырех значений

Удаляя последнее значение, он возвращает его и удаляет его из стека “cuatro”:

```
let fromTheTop = stackOfStrings.pop()
// fromTheTop равен "cuatro", а stack содержит 3 строки
```

После удаления верхней величины, стек выглядит так:

Расширяем универсальный тип

Когда вы расширяете универсальный тип, вы не обеспечиваете список параметров в качестве определения расширения. Вместо этого, список параметров типа, из исходного определения типа, доступен внутри тела расширения, а имена исходных параметров типа используются для ссылки на параметры типа из исходного определения.

Следующий пример расширяет универсальный тип Stack, для добавления вычисляемого свойства только для чтения topItem, которое возвращает верхний элемент стека, без “выстреливания” его из этого стека:

```
extension Stack {
    var topItem: Element? {
        return items.isEmpty ? nil : items[items.count - 1]
    }
}
```

Свойство topItem возвращает опциональное значение типа Element. Если стек пустой, то topItem возвращает nil. Если стек не пустой, то topItem возвращает последний элемент массива items.

Обратите внимание, что расширение не определяет списка параметров типа. Вместо этого, имя существующего параметра типа Stack - Element, используется внутри расширения для отображения опционального типа вычисляемого свойства topItem.

Вычисляемое свойство topItem теперь может быть использовано внутри экземпляра Stack для доступа к значению и для запроса к последнему элемента стека, без дальнейшего его удаления:

```
if let topItem = stackOfStrings.topItem {
    print("The top item on the stack is \(topItem).")
}
// Выведет "The top item on the stack is tres."
```

Расширения общего типа могут также включать требования, которые должны удовлетворять экземпляры расширенного типа, чтобы получить новые функциональные возможности

Ограничения типа

Функция swapTwoValues(_:_:) и тип Stack могут работать с любыми типами. Однако иногда бывает нужно внедрить определенные ограничения типа на типы, которые могут быть использованы вместе с универсальными функциями или универсальными типами. Ограничения типа указывают на то, что параметры типа должны наследовать от определенного класса или соответствовать определенному протоколу или композиции протоколов.

Для примера возьмем тип Dictionary, который имеет некоторые ограничения типов, которые могут быть использованы в качестве ключей. Тип ключа словаря должен быть хешируемым. Таким образом он должен предоставить способ представить себя уникальным. Dictionary нужно, чтобы его ключи были хешируемыми, таким образом он может проверить, содержит ли конкретный ключ какое-либо значение. Без этого требования, Dictionary не в состоянии понять, должен ли он заменить или вставить значение для конкретного ключа, и не в состоянии найти значение для конкретного ключа, которое уже есть в словаре.

Такое требование внедряется ограничениями типа для типа ключа словаря, которое определяет, что каждый ключ должен соответствовать протоколу Hashable, специальному протоколу, который определен в стандартной библиотеке Swift. Все базовые типы в Swift (String, Int, Double, Bool) по умолчанию являются хешируемыми типами.

Вы можете определить свои собственные ограничения типа, когда создаете пользовательские универсальные классы, и эти ограничения предоставляют еще больше возможностей универсальному программированию. Абстрактные понятия, как Hashable, характеризуют типы с точки зрения их концептуальных характеристик, а не их явного типа.

Синтаксис ограничения типа

Вы пишете ограничения типа, поместив ограничение единственного класса или протокола после имени параметра типа, и разделив их между собой запятыми, обозначая их в качестве части списка параметров. Базовый синтаксис для ограничений типа универсальной функции показан ниже:

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {
    // тело функции…
}
```

Выше описанная гипотетическая функция имеет два параметра типа. Первый параметр типа – T, имеет ограничение типа, которое требует, чтобы T, было подклассом класса SomeClass. Второй параметр типа – U,

имеет ограничение типа, которое требует, чтобы U соответствовал протоколу SomeProtocol.

Ограничение типа в действии

Ниже приведена неуниверсальная функция findIndex(ofString:in:), которая получает значение типа String для того, чтобы его найти, и массив значений типа String, внутри которого и будет происходить поиск. Функция findIndex(ofString:in:) возвращает опциональное значение Int, которое является индексом первого совпадения строки с элементом внутри массива или nil, которое означает отсутствие совпадения строки с каким-либо элементом массива:

```
func findIndex(ofString valueToFind: String, in array: [String]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

Функция findIndex(ofString:in:) может быть использована для поиска строкового значения в массиве строк:

```
let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]
if let foundIndex = findIndex(ofString: "llama", in: strings) {
    print("The index of llama is \(foundIndex)")
}
// Выведет "The index of llama is 2"
```

Однако нахождение индекса совпадения значения в массиве бывает полезным не только для строк. Вы можете написать ту же функцию, но только в универсальной форме. Давайте напишем такую функцию и назовем ее findIndex, а все упоминания типа String заменим на тип T.

Вот как будет выглядеть версия функции findIndex(ofString:in:) в универсальной форме findIndex(of:in:). Обратите внимание, что возвращаемый функцией тип все еще равен Int?, потому что функция возвращает опциональное значение индекса, а не опциональное значение элемента массива. Но будьте осторожны, так как эта функция не компилируется, по причинам, указанным после примера:

```
func findIndex<T>(of valueToFind: T, in array:[T]) -> Int? {
```

```
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

Как мы и сказали, эта функция не компилируется. Проблема находится в строке “if value == valueToFind”. Не каждый тип в Swift может быть сравнен оператором равенства (==). Если вы создаете свой класс или структуру для отображения сложной модели данных, например, то смысл выражения “равен чему-то” для этого класса или структуры Swift не может додумать за вас. Из-за этого нет никакой гарантии того, что этот код будет работать для любого возможного класса T, и соответствующая ошибка компиляции выскакивает, когда вы пытаетесь скомпилировать код.

Но не все еще потеряно. Стандартная библиотека Swift определяет протокол Equatable, который требует любой соответствующий ей тип реализовывать равенство оператору равенства (==) и реализовывать неравенство оператору неравенства (!=), для того, чтобы значения этих типов можно было сравнивать между собой. Все стандартные типы Swift автоматически поддерживают протокол Equatable.

Любой тип, который удовлетворяет протоколу Equatable, может быть безопасно использован в функции findIndex(of:in:), потому что гарантирована поддержка оператора равенства и неравенства. Для отображения этого факта, вы пишете ограничение типа Equatable, как часть определения параметра типа, когда вы определяете функцию:

```
func findIndex<T: Equatable>(of valueToFind: T, in array:[T]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

Единственный параметр типа для функции findIndex(of:in:) записывается как T: Equatable, что означает “любой тип T, который соответствует протоколу Equatable”.

Теперь функция findIndex(of:in:) благополучно компилируется и может быть использована с любыми типами Equatable, например, String, Double:

```
let doubleIndex = findIndex(of: 9.3, in: [3.14159, 0.1, 0.25])
// doubleIndex опциональный Int не имеющий значения, потому что значения 9.3 нет в массиве
let stringIndex = findIndex(of: "Andrea", in: ["Mike", "Malcolm", "Andrea"])
// stringIndex опциональный Int равный 2
```

Связанные типы

При определении протокола бывает нужно определить еще один или более связанных типов в качестве части определения протокола. Связанный тип дает плейсхолдер имени типу, который используется как часть протокола. Фактический тип, который будет использоваться связанным типом не указывается до тех пор, пока не будет принят протокол. Связанные типы указываются при помощи ключевого слова associatedtype.

Связанные типы в действии

Ниже приведен пример протокола Container, который объявляет связанный тип Item:

```
protocol Container {
    associatedtype Item
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}
```

Протокол Container определяет три требуемых возможности, которые должен иметь любой контейнер:

– Должна быть возможность добавлять новый элемент в контейнер при помощи метода append(_:).

– Должна быть возможность получить доступ к количеству элементов в контейнере через свойство count, которое возвращает значение типа Int.

– Должна быть возможность получить значение через индекс элемента, который принимает значение типа Int.

Этот протокол не указывает количество и способ хранения элементов в контейнере или какого типа они должны быть. Протокол

только лишь указывает три “кусочка” функциональности, которые должны быть предоставлены контейнером, чтобы он считался Container. Соответствующий тип может предоставлять дополнительную функциональность, пока он удовлетворяет этим трем требованиям.

Любой тип, который удовлетворяет протоколу Container должен иметь возможность указывать на тип хранящихся элементов. Конкретно, он должен гарантировать, что только элементы правильного типа будут добавлены в контейнер, и должно быть ясно какой тип элементов будет возвращаться сабскриптом.

Для определения этих требований, протокол Container должен иметь способ ссылаться на тип элементов, которые он будет хранить, без указания типа элементов, которые может хранить конкретный контейнер. Протокол Container должен указать, что любое значение переданное в метод append(_:) должно иметь тот же тип, что и тип элементов контейнера, и что значение, возвращаемое сабскриптом контейнера, должно быть того же типа, что и элементы контейнера.

Чтобы добиться этого, протокол Container объявляет связанный тип Item, который записывается как associatedtype Item. Протокол не определяет для чего конкретно нужен алиас Item, потому что эта информация остается для любого соответствующего класса протоколу. Тем не менее, алиас Item предоставляет способ сослаться на тип элементов в Container и определить тип для использования метода append(_:) и сабскрипта, для того, чтобы гарантировать, что желаемое поведение любого Container имеет силу.

Ниже приведена версия неуниверсального типа IntStack, который адаптирован под протокол Container:

```
struct IntStack: Container {
    // исходная реализация IntStack
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
    // удовлетворение требований протокола Container
    typealias Item = Int
    mutating func append(_ item: Int) {
        self.push(item)
    }
    var count: Int {
```

```
        return items.count
    }
    subscript(i: Int) -> Int {
        return items[i]
    }
}
```

Тип IntStack реализует все три требования протокола Container, и в каждом случае оборачивает часть существующей функциональности типа IntStack для удовлетворения этих требований.

Более того, IntStack указывает, что для этой реализации контейнера, подходящий тип Item будет Int. Определение typealias Item = Int преобразует абстрактный тип Item в конкретный тип Int для этой реализации протокола Container.

Благодаря выводу типов Swift, вам фактически не нужно указывать конкретный тип Int для Item как часть определения IntStack. Так как IntStack соответствует протоколу Container, Swift может вывести соответствующий тип для Item, просто посмотрев на тип параметра item метода append(_:) и на тип возвращаемого значения сабскрипта. И на самом деле, если удалить строку кода typealias Item = Int, все будет продолжать работать, потому что все еще ясно какой тип должен быть использован для Item.

Вы так же можете создать универсальный тип Stack, который соответствует протоколу Container:

```
struct Stack<Element>: Container {
    // исходная реализация Stack<Element>
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
    // удовлетворение требований протокола Container
    mutating func append(_ item: Element) {
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Element {
```

```
        return items[i]
    }
}
```

В этот раз тип параметра Element использован в качестве параметра item метода append(_:) и в качестве возвращаемого типа сабскрипта. Таким образом Swift может вывести, что Element подходящий тип для использования его в качестве типа Item для этого конкретного контейнера.

7 ЗАМЫКАНИЯ

Замыкания – это самодостаточные блоки с определенным функционалом, которые могут быть переданы и использованы в вашем коде. Замыкания в Swift похожи на блоки в C и Objective-C, и лямбды в других языках программирования.

Замыкания могут захватывать и хранить ссылки на любые константы и переменные из контекста, в котором они объявлены. Эта процедура известна как заключение этих констант и переменных, отсюда и название "замыкание". Swift выполняет всю работу с управлением памятью при захвате за вас.

Глобальные и вложенные функции, которые были представлены в главе Функции, являются частным случаем замыканий. Замыкания принимают одну из трех форм:

– Глобальные функции являются замыканиями, у которых есть имя и которые не захватывают никакие значения.

– Вложенные функции являются замыканиями, у которых есть имя и которые могут захватывать значения из включающей их функции.

– Выражения замыкания являются безымянными замыканиями, написанные в облегченном синтаксисе, которые могут захватывать значения из их окружающего контекста.

Выражения замыкания в Swift имеют четкий, ясный, оптимизированный синтаксис в распространенных сценариях. Эти оптимизации включают:

– Вывод типа параметра и возврат типа значения из контекста
– Неявные возвращающиеся значения однострочных замыканий
– Сокращенные имена параметров
– Синтаксис последующих замыканий

Замыкающие выражения, являются способом написания встроенных замыканий через краткий и специализированный синтаксис. Замыкающие выражения обеспечивают несколько синтаксических оптимизаций для написания замыканий в краткой форме, без потери ясности и намерений. Примеры замыкающих выражений ниже, показывают эти оптимизации путем рассмотрения метода sorted(by:) при нескольких итерациях, каждая из которых изображает ту же функциональность в более сжатой форме.

Метод sorted

В стандартной библиотеке Swift есть метод sorted(by:), который сортирует массив значений определенного типа, основываясь на результате сортирующего замыкания, которые вы ему передадите. После завершения процесса сортировки, метод sorted(by:) возвращает новый массив того же типа и размера как старый, с элементами в правильном порядке сортировки. Исходный массив не изменяется методом sorted(by:).

Примеры замыкающих выражений ниже используют метод sorted(by:)для сортировки массива из String значений в обратном алфавитном порядке. Вот исходный массив для сортировки:

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

Замыкание метода sorted(by:) принимает два аргумента одного и того же типа, что и содержимое массива, и возвращает Bool значение, которое решает поставить ли первое значение перед вторым, или после второго. Замыкание сортировки должно вернуть true, если первое значение должно быть *до* второго значения, и false в противном случае.

Этот пример сортирует массив из String значений, так что сортирующее замыкание должно быть функцией с типом (String, String) -> Bool.

Один из способов обеспечить сортирующее замыкание, это написать нормальную функцию нужного типа, и передать ее в качестве аргумента метода sorted(by:):

```
func backward(_ s1: String, _ s2: String) -> Bool {
  return s1 > s2
}
var reversedNames = names.sorted(by: backward)
// reversedNames равен ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

Если первая строка (s1) больше чем вторая строка (s2), функция backward(_:_:) возвращает true, что указывает, что s1 должна быть перед s2 в сортированном массиве. Для символов в строках, "больше чем" означает "появляется в алфавите позже, чем". Это означает что

буква "B" "больше чем" буква "A", а строка "Tom" больше чем строка "Tim". Это делает обратную алфавитную сортировку, с "Barry" поставленным перед "Alex", и так далее.

Тем не менее, это довольно скучный способ написать то, что по сути, является функцией с одним выражением (a > b). В этом примере, было бы предпочтительнее написать сортирующее замыкание в одну строку, используя синтаксис замыкающего выражения.

Синтаксис замыкающего выражения

Синтаксис замыкающего выражения имеет следующую общую форму:

```
{ (параметры) -> тип результата in
выражения
}
```

Синтаксис замыкающего выражения может использовать сквозные параметры. Значения по умолчанию не могут быть переданы. Вариативные параметры могут быть использованы в любом месте в списке параметров. Кортежи также могут быть использованы как типы параметров и как типы возвращаемого значения.

Пример ниже показывает версию функции backward(_:_:) с использованием замыкающего выражения:

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool
in
    return s1 > s2
})
```

Обратите внимание, что объявление типов параметров и типа возвращаемого значения для этого однострочного замыкания идентично объявлению из функции backward(_:_:). В обоих случаях, оно пишется в виде (s1: String, s2: String) -> Bool. Тем не менее, для однострочных замыкающих выражений, параметры и тип возвращаемого значения пишутся внутри фигурных скобок, а не вне их.

Начало тела замыкания содержит ключевое слово in. Это ключевое слово указывает, что объявление параметров и возвращаемого значения замыкания закончено, и тело замыкания вот-вот начнется.

Поскольку тело замыкания настолько короткое, оно может быть записано в одну строку:

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in return s1 > s2 })
```

Это показывает, что общий вызов метода sorted остался прежним. Пара скобок по-прежнему обособляют весь набор параметров метода.

Определение типа из контекста

Поскольку сортирующее замыкание передается как аргумент метода, Swift может вывести типы его параметров и тип возвращаемого значения, через тип параметра метода sorted(by:). Этот параметр ожидает функцию имеющую тип (String, String) -> Bool. Это означает что типы (String, String) и Bool не нужно писать в объявлении замыкающего выражения. Поскольку все типы могут быть выведены, стрелка результата (->) и скобки вокруг имен параметров также могут быть опущены:

```
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 })
```

Всегда можно определить типы параметров и тип возвращаемого значения, когда мы передаем замыкание функции в виде однострочного замыкающего выражения. В результате, когда замыкание используется как параметр метода, вам никогда не нужно писать однострочное замыкание в его полном виде.

Тем не менее, вы всё равно можете явно указать типы, если хотите. И делать это предполагается, если это поможет избежать двусмысленности для читателей вашего кода. В случае с методом sorted(by:), цель замыкания понятна из того факта, что сортировка происходит, и она безопасна для читателя, который может предположить, что замыкание, вероятно, будет работать со значениями String, поскольку оно помогает сортировать массив из строк.

Неявные возвращаемые значения из замыканий с одним выражением

Замыкания с одним выражением могут неявно возвращать результат своего выражения через опускание ключевого слова return из их объявления, как показано в этой версии предыдущего примера:

```
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 })
```

Здесь, функциональный тип аргумента метода sorted(by:)дает понять, что замыкание вернет Bool значение. Поскольку тело замыкания содержит одно выражение (s1 > s2), которое возвращает Bool значение, то нет никакой двусмысленности, и ключевое слово return можно опустить.

Сокращенные имена параметров

Swift автоматически предоставляет сокращённые имена для однострочных замыканий, которые могут быть использованы для обращения к значениям параметров замыкания через имена $0, $1, $2, и так далее.

Если вы используете эти сокращенные имена параметров с вашим замыкающим выражением, вы можете пропустить список параметров замыкания из его объявления, а количество и тип сокращенных имен параметров будет выведено из ожидаемого типа метода. Ключевое слово in также может быть опущено, поскольку замыкающее выражение полностью состоит из его тела:

```
reversedNames = names.sorted(by: { $0 > $1 })
```

Здесь, $0 и $1 обращаются к первому и второму String параметру замыкания.

Операторные функции

Здесь есть на самом деле более короткий способ написать замыкающее выражение выше. Тип String в Swift определяет свою специфичную для строк реализацию оператора больше (>) как функции, имеющей два строковых параметра и возвращающей значение типа Bool. Это точно соответствует типу метода, для параметра метода sorted(by:). Таким образом, вы можете просто

написать оператор больше, а Swift будет считать, что вы хотите использовать специфичную для строк реализацию:

```
reversedNames = names.sorted(by: >)
```

Последующее замыкание

Если вам нужно передать выражение замыкания функции в качестве последнего аргумента функции и само выражение замыкания длинное, то оно может быть записано в виде последующего замыкания. Последующее замыкание – замыкание, которое записано в виде замыкающего выражения вне (и после) круглых скобок вызова функции, даже несмотря на то, что оно все еще является аргументом функции. Когда вы используете синтаксис последующего замыкания, то вы не должны писать ярлык аргумента замыкания в качестве части вызова самой функции. Функция может включать в себя несколько последующих замыканий, однако, первые несколько примеров используют по одному последующему замыканию:

```
func someFunctionThatTakesAClosure(closure: () -> Void) {
  // тело функции
}
// Вот как вы вызываете эту функцию без использования последующего замыкания:

someFunctionThatTakesAClosure(closure: {
  // тело замыкания
})
// Вот как вы вызываете эту функцию с использованием последующего замыкания:

someFunctionThatTakesAClosure() {
  // тело последующего замыкания
}
```

Сортирующее строки замыкание может быть записано вне круглых скобок функции sorted(by:), как последующее замыкание:

```
reversedNames = names.sorted() { $0 > $1 }
```

Если выражение замыкания является единственным аргументом функции, и вы пишете его используя синтаксис последующего замыкания, то вы можете опустить написание круглых скобок вызова самой функции после ее имени.

```
reversedNames = names.sorted { $0 > $1 }
```

Последующие замыкания полезны в случаях, когда само замыкание достаточно длинное, и его невозможно записать в одну строку. В качестве примера приведем вам метод map(_:) типа Array в языке Swift, который принимает выражение замыкания как его единственный аргумент. Замыкание вызывается по одному разу для каждого элемента массива и возвращает альтернативную отображаемую величину (возможно другого типа) для этого элемента. Природа отображения и тип возвращаемого значения определяется замыканием.

После применения замыкания к каждому элементу массива, метод map(_:) возвращает новый массив, содержащий новые преобразованные величины, в том же порядке, что и в исходном массиве.

Вот как вы можете использовать метод map(_:) вместе с последующим замыканием для превращения массива значений типа Int в массив типа String. Массив [16, 58, 510] используется для создания нового массива ["OneSix", "FiveEight", "FiveOneZero"] :

```
let digitNames = [0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four", 5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"]
let numbers = [16, 58, 510]
```

Код выше создает словарь отображающий цифры и их английскую версию имен. Так же он объявляет массив целых значений для преобразования в массив строк.

Вы можете использовать массив numbers для создания значений типа String, передав замыкающее выражение в метод map(_:) массива в качестве последующего замыкания. Обратите внимание, что вызов numbers.map не включает в себя скобки после map, потому что метод map(_:) имеет только один параметр, который мы имеем в виде последующего замыкания:

```
let strings = numbers.map { (number) -> String in
    var number = number
    var output = ""
    repeat {
        output = digitNames[number % 10]! + output
        number /= 10
    } while number > 0
    return output
}
//тип строк был выведен как [String]//значения ["OneSix", "FiveEight", "FiveOneZero"]
```

Метод map(_:) вызывает замыкание один раз для каждого элемента массива. Вам не нужно указывать тип входного параметра замыкания, number, так как тип может быть выведен из значений массива, который применяет метод map.

В этом примере переменная number инициализирована при помощи значения параметра замыкания number, так что значение может быть изменено внутри тела замыкания. (Параметры функций и замыкания всегда являются константами.) Выражение замыкания так же определяет возвращаемый тип String для указания типа, который будет храниться в массиве на выходе из метода map(_:).

Замыкающее выражение строит строку, названную output, каждый раз, когда оно вызывается. Оно рассчитывает последнюю цифру number, используя оператор деления с остатком (number % 10) и использует затем эту получившуюся цифру, чтобы найти соответствующую строку в словаре digitNames. Это замыкание может быть использовано для создания строкового представления любого целого числа, большего чем 0.

Строка, полученная из словаря digitNames, добавляется в начало переменной output, путем правильного формирования строковой версии числа наоборот. (Выражение number % 10 дает нам 6 для 16, 8 для 58 и 0 для 510).

Переменная number после вычисления остатка делится на 10. Так как тип значения Int, то наше число округляется вниз, таким образом 16 превращается в 1, 58 в 5, 510 в 51.

Процесс повторяется пока number /= 10 не станет равным 0, после чего строка output возвращается замыканием и добавляется к выходному массиву функции map(_:).

Использование синтаксиса последующих замыканий в примере выше аккуратно инкапсулирует функциональность замыкания сразу после функции map(_:), которой замыкание помогает, без необходимости заворачивания всего замыкания внутрь внешних круглых скобок функции map(_:).

Если функция принимает несколько последующих замыканий, вы можете пропустить ярлык параметра для первого из них, а для остальных уже указать нужно. Например, функция ниже загружает изображение в фотогалерею:

```
func loadPicture(from server: Server, completion: (Picture) -> Void, onFailure: () -> Void) {
    if let picture = download("photo.jpg", from: server) {
        completion(picture)
    } else {
        onFailure()
    }
}
```

Когда вы вызываете эту функцию для загрузки изображений, вы используете два замыкания. Первое замыкание – это обработчик, который отображает изображение после успешной загрузки. Второе замыкание – обработчик, который отображает ошибку пользователю, если произошла ошибка во время загрузки.

```
loadPicture(from: someServer) { picture in
    someView.currentPicture = picture
} onFailure: {
    print("Couldn't download the next picture.")
}
```

В этом примере метод loadPicture(from:completion:onFailure:) передает свою сетевую задачу в фоновый поток и вызывает одно из замыканий, когда сетевая задача выполнена. Написание функции таким способом позволяет вам разделять код, который ответственен за обработку ошибки во время сетевой задачи от кода, который отвечает за успешную загрузку.

Захват значений

Замыкания могут захватывать константы и переменные из окружающего контекста, в котором оно объявлено. После захвата замыкание может ссылаться или модифицировать значения этих констант и переменных внутри своего тела, даже если область, в которой были объявлены эти константы и переменные уже больше не существует.

В Swift самая простая форма замыкания может захватывать значения из вложенных функций, написанных внутри тела других функций. Вложенная функция может захватить любые значения из аргументов окружающей ее функции, а так же константы и переменные, объявленные внутри тела внешней функции.

Вот пример функции makeIncrementer, которая содержит вложенную функцию incrementer. Вложенная функция incrementer() захватывает два значения runningTotal и amount из окружающего контекста. После захвата этих значений incrementer возвращается функцией makeIncrementer как замыкание, которое увеличивает runningTotal на amount каждый раз как вызывается.

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
  var runningTotal = 0
  func incrementer() -> Int {
    runningTotal += amount
    return runningTotal
  }
  return incrementer
}
```

Возвращаемый тип makeIncrementer Void -> Int. Это значит, что он возвращает функцию, а не простое значение. Возвращенная функция не имеет параметров и возвращает Int каждый раз как ее вызывают.

Функция makeIncrementer(forIncrement:) объявляет целочисленную переменную runningTotal, для хранения текущего значения инкрементора, которое будет возвращено. Переменная инициализируется значением 0.

Функция makeIncrementer(forIncrement:) имеет единственный параметр Int с внешним именем forIncrement и локальным именем amount. Значение аргумента передается этому параметру, определяя

на сколько должно быть увеличено значение runningTotal каждый раз при вызове функции.

Функция makeIncrementer объявляет вложенную функцию incrementer, которая непосредственно и занимается увеличением значения. Эта функция просто добавляет amount к runningTotal и возвращает результат.

Если рассматривать функцию incrementer() отдельно, то она может показаться необычной:

```
func incrementer() -> Int {
   runningTotal += amount
   return runningTotal
}
```

Функция incrementer() не имеет ни одного параметра и она ссылается на runningTotal и amount внутри тела функции. Она делает это, захватывая существующие значения от runningTotal и amount из окружающей функции и используя их внутри. Захват ссылки дает гарантию того, что runningTotal не исчезнет при окончании вызова makeIncrementer и гарантирует, что runningTotal останется переменной в следующий раз, когда будет вызвана функция incrementer().

Приведем пример makeIncrementer в действии:

```
let incrementByTen = makeIncrementer(forIncrement: 10)
```

Этот пример заставляет константу incrementByTen ссылаться на функцию инкрементора, которая добавляет 10 к значению переменной runningTotal каждый раз как вызывается. Многократный вызов функции показывает ее в действии:

```
incrementByTen()// возвращает 10
incrementByTen()// возвращает 20
incrementByTen()// возвращает 30
```

Если вы создаете второй инкрементор, он будет иметь свою собственную ссылку на новую отдельную переменную runningTotal :

```
let incrementBySeven = makeIncrementer(forIncrement: 7)
```

incrementBySeven()//возвращает значение 7

Повторный вызов первоначального инкрементора (incrementByTen) заставит увеличиваться его собственную переменную runningTotal и никак не повлияет на переменную, захваченную в incrementBySeven :

incrementByTen()//возвращает 40

Замыкания – ссылочный тип

В примере выше incrementBySeven и incrementByTen константы, но замыкания, на которые ссылаются эти константы имеют возможность увеличивать значение переменных runningTotal, которые они захватили. Это из-за того, что функции и замыкания являются *ссылочными типами*.

Когда бы вы ни присваивали функцию или замыкание константе или переменной, вы фактически присваиваете ссылку этой константе или переменной на эту функцию или замыкание. В примере выше выбор замыкания, на которое ссылается incrementByTen, константа, но не содержимое самого замыкания.

Это так же значит, что если вы присвоите замыкание двум разным константам или переменным, то оба они будут ссылаться на одно и то же замыкание:

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()//возвращает 50
incrementByTen()//возвращает 60
```

Пример выше показывает, что вызов alsoIncrementByTen то же самое, что и вызов incrementByTen. Потому что и та и другая функция ссылаются на одно и то же замыкание: и то, и другое замыкание возвращают один и тот же runningTotal.

Сбегающие замыкания

Когда говорят, что замыкание *сбегает* из функции, то это значит, что это замыкание было передано в функцию в качестве аргумента и вызывается уже после того, как функция вернула значение. Когда вы объявляете функцию, которая имеет замыкание в

качестве одного из параметров, то вы пишете @escaping до типа параметра, для того чтобы указать, что замыкание может *сбежать*.

Если замыкание хранится в переменной, которая была объявлена вне функции, а затем эта переменная была передана в качестве аргумента в функцию, то получается, что замыкание, которое посредством переменной передается в функцию, сбегающее. В качестве примера можно рассмотреть функции, которые выполняют асинхронные операции в завершающем обработчике, который является замыканием. То есть получается, что функция завершает свою работу, после чего вызывается завершающий обработчик. Или другими словами обработчик не вызывается, пока не завершится работа функции, таким образом получается, что данному замыканию нужно сбежать из области работы функции, чтобы отработать позже. Например:

```
var completionHandlers: [() -> Void] = []func someFunctionWithEscapingClosure(completionHandler: @escaping () -> Void) {
  completionHandlers.append(completionHandler)
}
```

Функция someFunctionWithEscapingClosure(_:) принимает и добавляет в массив замыкание, объявленное за пределами функции. Если вы не поставите маркировку @escaping, то получите ошибку компиляции.

Сбегающее замыкание, которое имеет ссылку на self требует отдельного рассмотрения, если self ссылается на экземпляр класса. Захватывая self в сбегающем замыкании, вы можете случайно создать зацикленность сильных ссылок.

Обычно замыкание захватывает переменные неявно, просто используя их внутри тела, но в случае с self вам нужно делать это явно. Если вы хотите захватить self, напишете self явно, когда используете его, или включите self в лист захвата замыкания. Когда вы пишете self явно, вы явно указываете свое намерение, а так же помогаете сами себе тем, что напоминаете проверить наличие цикла сильных ссылок. Например, в коде ниже замыкание переданное в метод someFunctionWithEscapingClosure(_:) ссылается на self явно. А вот замыкание, переданное в метод someFunctionWithNonescapingClosure(_:) является несбегающим, что значит, что оно может ссылаться на self неявно.

```
func someFunctionWithNonescapingClosure(closure: () -> Void) {
   closure()
}
 class SomeClass {
   var x = 10
   func doSomething() {
      someFunctionWithEscapingClosure { self.x = 100 }
      someFunctionWithNonescapingClosure { x = 200 }
   }
}
 let instance = SomeClass()
instance.doSomething()print(instance.x)// Выведет "200"
 completionHandlers.first?()print(instance.x)// Выведет "100"
```

Ниже приведена версия doSomething(), которая захватывает self, включая его в лист захвата замыкания, а затем неявно ссылается на него:

```
class SomeOtherClass {
   var x = 10
   func doSomething() {
      someFunctionWithEscapingClosure { [self] in x = 100 }
      someFunctionWithNonescapingClosure { x = 200 }
   }
}
```

Если self является экземпляром структуры или перечисления, то вы можете всегда ссылаться на self неявно. Однако, сбегающие замыкания не могут захватить изменяемую ссылку на self, когда self является экземпляром структуры или перечисления. Структуры и перечисления не допускают общей изменчивости.

```
struct SomeStruct {
   var x = 10
   mutating func doSomething() {
      someFunctionWithNonescapingClosure { x = 200 }  // Ok
      someFunctionWithEscapingClosure { x = 100 }     // Error
   }
```

}

Вызов функции someFunctionWithEscapitngClosure в примере выше вызовет ошибку, так как находится замыкание внутри mutable метода, таким образом self так же получается изменяемым (*mutable*). Ошибка получается из-за того, что мы нарушаем правило, которое гласит, что в структурах сбегающие замыкания не могут захватывать изменяемую ссылку на self.

Автозамыкания (autoclosures)

Автозамыкания – замыкания, которые автоматически создаются для заключения выражения, которое было передано в качестве аргумента функции. Такие замыкания не принимают никаких аргументов при вызове и возвращают значение выражения, которое заключено внутри нее. Синтаксически вы можете опустить круглые скобки функции вокруг параметров функции, просто записав обычное выражение вместо явного замыкания.

Нет ничего необычного в вызове функций, которые принимают автозамыкания, но необычным является реализовывать такие функции. Например, функция assert(condition:message:file:line:) принимает автозамыкания на место condition и message параметров. Ее параметр condition вычисляется только в сборке дебаггера, а параметр message вычисляется, если только condition равен false.

Автозамыкания позволяют вам откладывать вычисления, потому как код внутри них не исполняется, пока вы сами его не запустите. Это полезно для кода, который может иметь сторонние эффекты или просто является дорогим в вычислительном отношении, потому что вы можете контролировать время исполнения этого кода. Пример ниже отображает как замыкания откладывают вычисления:

var customersInLine = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]print(customersInLine.count)// Выведет "5"
let customerProvider = { customersInLine.remove(at: 0) }print(customersInLine.count)// Выведет "5"
print("Now serving \(customerProvider())!")// Выведет "Now serving Chris!"print(customersInLine.count)// Выведет "4"

Даже если первый элемент массива customersInLine удаляется кодом внутри замыкания, элемент массива фактически не

удаляется до тех пор, пока само замыкание не будет вызвано. Если замыкание так и не вызывается, то выражение внутри него никогда не выполнится и, соответственно, элемент не будет удален из массива. Обратите внимание, что customerProvider является не String, а () -> String, то есть функция не принимает аргументов, но возвращает строку. Вы получите то же самое поведение, когда сделаете это внутри функции:

```
// customersInLine равен ["Alex", "Ewa", "Barry", "Daniella"]func serve(customer customerProvider: () -> String) {
    print("Now serving \(customerProvider())!")
}
serve(customer: { customersInLine.remove(at: 0) } )// Выведет "Now serving Alex!"
```

Функция serve(customer:) описанная выше принимает явное замыкание, которое возвращает имя клиента. Версия функции serve(customer:) ниже выполняет ту же самую операцию, но вместо использования явного замыкания, она использует автозамыкание, поставив маркировку при помощи атрибута @autoclosure. Теперь вы можете вызывать функцию, как будто бы она принимает аргумент String вместо замыкания. Аргумент автоматически преобразуется в замыкание, потому что тип параметра customerProvider имеет атрибут @autoclosure.

```
// customersInLine равен ["Ewa", "Barry", "Daniella"]func serve(customer customerProvider: @autoclosure () -> String) {
    print("Now serving \(customerProvider())!")
}
serve(customer: customersInLine.remove(at: 0))// Выведет "Now serving Ewa!"
```

Если вы хотите чтобы автозамыкание могло сбежать, то вам нужно использовать оба атрибута и @autoclosure, и @escaping.

```
// customersInLine равен ["Barry", "Daniella"]var customerProviders: [() -> String] = []func collectCustomerProviders(_ customerProvider: @autoclosure @escaping () -> String) {
    customerProviders.append(customerProvider)
```

```
}
collectCustomerProviders(customersInLine.remove(at: 0))
collectCustomerProviders(customersInLine.remove(at: 0))
 print("Collected \(customerProviders.count) closures.")// Выведет "Collected 2 closures."for customerProvider in customerProviders {
    print("Now serving \(customerProvider())!")
}// Выведет "Now serving Barry!"// Выведет "Now serving Daniella!"
```

В коде выше, вместо того, чтобы вызывать переданное замыкание в качестве аргумента customer, функция collectCustomerProviders(_:) добавляет замыкание к массиву customerProviders. Массив объявлен за пределами функции, что означает, что замыкание в массиве может быть исполнено после того, как функция вернет значение. В результате значение аргумента customerProvider должен иметь “разрешение” на “побег” из зоны видимости функции.

8 ИСПОЛЬЗОВАНИЕ ШАБЛОНА ПРОЕКТИРОВАНИЯ MVC В РАЗРАБОТКЕ IPHONE И IPAD ПРИЛОЖЕНИЙ

Концепция «модель – контроллер – представление» (Model-View-Controller – MVC) представляет собой очень логичный способ разделения кода, лежащего в основе приложений с графическим пользовательским интерфейсом. В настоящее время практически все объектно-ориентированные среды разработки в той или иной степени используют концепцию MVC, но лишь некоторые из них действительно полностью воплощают парадигму MVC.

Шаблон MVC разделяется по функциональным возможностям на три категории.

– Модель. Состоит из классов, в которых хранятся данные приложения.

– Представление. Создает окна, элементы управления и другие элементы, которые пользователь видит и с которыми взаимодействует.

– Контроллер. Связывает модель и представление, реализует логику приложения, в соответствии с которой оно обрабатывает данные, введенные пользователем.

Каждый объект, который вы создаете в своей программе, может быть легко отнесен к одной из категорий, но при этом не должен реализовать какие-либо функции, присущие двум другим. Например, экземпляр класса UIButton, обеспечивающий отображение кнопки, не должен содержать код, выполняемый при нажатии на нее, а код, производящий работу с базой аккаунтов, не должен рисовать таблицу на экране смартфона.

MVC позволяет достичь максимального разделения трех основных категорий, что впоследствии позволяет удобно обновлять и перерабатывать программу, а также повторно использовать отдельные компоненты. Так, например класс, который обеспечивает отображение кнопки, без труда может быть дополнен, расширен и многократно использован в любых приложениях.

Все возможности по разработке iOS-приложений предоставляет iOS SDK (software development kit, комплект средств разработки), который входит в состав Xcode. Данный SDK предоставляет огромное число ресурсов, благодаря которым вы можете строить UI, организовывать мультитач управление, хранение данных в БД, работу

с мультимедиа, передачу данных по сети, использование функций устройств (акселерометр и т. д.) и многое-многое другое. В состав iOS SDK входит фреймворк Cocoa Touch, который как раз построен на принципе MVC.

Во время разработки приложений в Xcode вы работали с категорией «Отображения» с помощью Interface Builder, но при этом не обеспечивали решения каких-либо бизнес-процессов с помощью графических элементов.

Категория «Контроллер» включает в себя специфические классы, обеспечивающие функциональность ваших приложений, например UIViewController, а точнее, его потомок ViewController.

Литература

1. Swift: разработка приложений в среде Xcode для iPhone и iPad с использованием iOS SDK / Дэвид Марк [и др.]. – М. [и др.] : Вильямс, 2017. – 808 с.

2. Усов, В. Swift. Основы разработки приложений под iOS и OS X / Василий У. – 4-е изд. – СПб [и др.] : Питер, 2019. – 444 с.

3. Харазян, А. А. Язык Swift / А. А. Харазян. – СПб: БХВ-Петербург, 2016. – 172 с.

4. Уинквист, Г. Swift для детей. Самоучитель по созданию приложений для iOS / Г. Уинквист, Мэтт Маккарти Ж ; пер. с англ. П. Миронова. – М. : Манн, Иванов и Фербер, 2018. – 368 с.

5. Грей, Э. Карманный справочник: программирование в среде iOS и OS X / Э. Грей. – М. : Вильямс, 2016. – 288 с.

6. Xcode уроки. – Режим доступа: https://2compa.ru/programmirovanie/xcode-and-swift/interfeis-ide-xcode-i-shablony/. – Дата доступа: 05.05.2021.

7. Руководство по языку программирования Swift. – Режим доступа: https://metanit.com/swift/tutorial/. – Дата доступа: 01.05.2021.

8. Руководство Swift: SwiftBook. – Режим доступа: https://swiftbook.ru/content/languageguide/. – Дата доступа: 15.04.2021.

9. Swift. Язык программирования с открытым кодом. Мощь, простота и потрясающие приложения. – Режим доступа: https://www.apple.com/ru/swift/. – Дата доступа: 05.05.2021.

СОДЕРЖАНИЕ

Семенченя Татьяна Сергеевна

РАЗРАБОТКА ПРИЛОЖЕНИЙ ДЛЯ IPHONE И IPAD

**Учебно-методическое пособие
для студентов специальности
1-40 05 01 «Информационные системы и технологии
(по направлениям)» направления специальности
1-40 05 01-01 «Информационные системы и технологии
(в проектировании и производстве)»
дневной и заочной форм обучения**

Подписано к размещению в электронную библиотеку
ГГТУ им. П. О. Сухого в качестве электронного
учебно-методического документа 11.11.22.
Рег. № 64Е.
http://www.gstu.by