

Программа iOS-разработчик

Конспект

A smartphone screen displays a course outline for iOS development. The top bar shows the time as 9:41 and signal strength. The main content area has a light gray background with a red gradient at the bottom. A blue circular icon next to the first item indicates it is selected.

- Введение в iOS-разработку** Часть 1
- Пользовательский интерфейс
- Многопоточность
- Работа с сетью
- Хранение данных
- Мультимедиа и другие фреймворки

Оглавление

1 НЕДЕЛЯ 1	3
1.1 Знакомство со специализацией	3
1.1.1 IDE и инструменты	3
1.1.2 Создание UI	4
1.1.3 Сетевой слой	5
1.1.4 Многопоточность	6
1.1.5 Отладка приложений	6
1.1.6 Хранение данных	7
1.2 Знакомство с курсом	8
1.2.1 Знакомство со средствами разработки	8
1.2.2 Типы и Control Flow	8
1.2.3 Функции, методы, замыкания	9
1.2.4 Жизненный цикл объектов	9
1.2.5 Типизация в Swift	9
1.2.6 Курсовое задание	10
1.3 История и предпосылки Swift	10
1.3.1 LLVM и традиционные компиляторы	11
1.4 В чем преимущества Swift над Objective-C	13
1.4.1 Расширения	14
1.4.2 Протоколы	15
1.4.3 Замыкания	15
1.4.4 ARC	16
1.4.5 Динамическая диспетчеризация	16
1.4.6 Указатели	17
1.4.7 Нулевые указатели	18
1.4.8 Автоопределение типов	19
1.4.9 Функциональное программирование	19
1.4.10 Перечисления	20
1.4.11 Unicode	21

1.4.12 Будущее Swiftt	21
---------------------------------	----

Глава 1

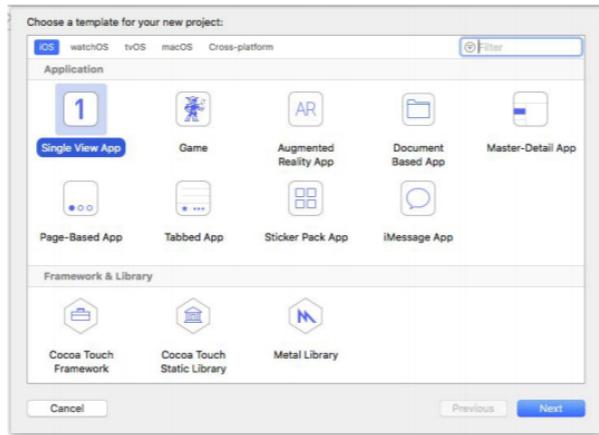
НЕДЕЛЯ 1

1.1. Знакомство со специализацией

Добро пожаловать на специализацию, посвященную созданию приложений на Swift. Она объединяет в себе несколько курсов, охватывающих большинство аспектов разработки мобильных приложений. Хочу вкратце рассказать вам о том, какие тематики будут рассматриваться во время прохождения специализации. Начнем, конечно же, с основ.

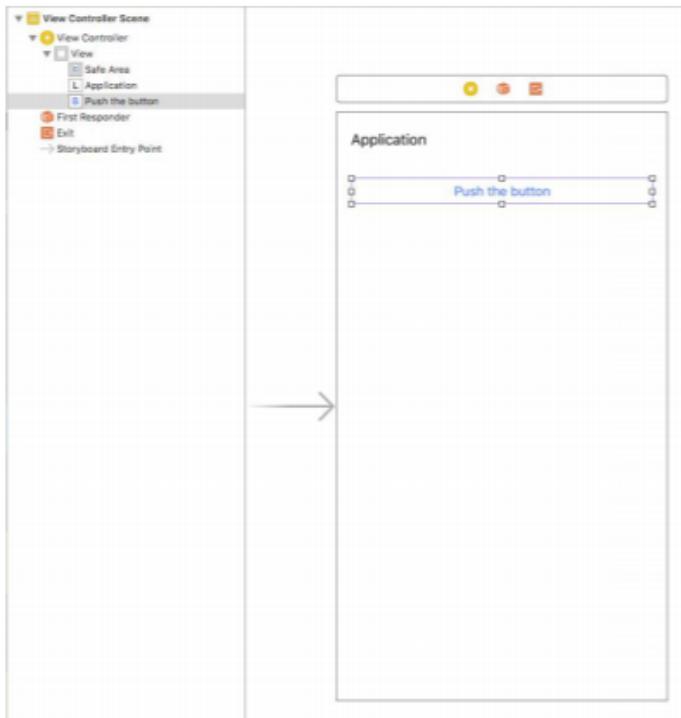
1.1.1. IDE и инструменты

Знакомство с *IDE* и инструментами разработки. Познакомимся с базовыми строительными блоками языка Swift, классами, структурами, объектами. Научимся выстраивать взаимодействие между ними с помощью функций и замыканий. Узнаем, каким образом мы можем хранить объекты внутри приложений и организовывать их в коллекции. Мы расскажем, как добавлять функционал к уже имеющимся классам, и о том, как правильно организовывать код внутри приложения. После знакомства с основами языка, начнем работу над курсовым проектом. Он будет разрабатываться на протяжении всей специализации, становясь все сложнее и интереснее.



1.1.2. Создание UI

Большое количество материала будет посвящено созданию интерфейса приложения, и снова мы начнем с элементарных вещей (создание экранов, кнопок, таблиц, полей ввода), постепенно переходя к более продвинутым темам: организации навигации внутри приложения, поддержки нескольких устройств и способов создания универсального представления, которое будет корректно отображаться и на iPhone, и на iPad. Мы научимся распознавать жесты пользователей и реагировать на них, создавая обратную связь с помощью анимации. На этом этапе в нашем приложении будет готова не только модель для хранения данных, но и графическое оформление, соответствующее гайдлайнам Apple.



1.1.3. Сетевой слой

Следующим этапом будет создание сетевого слоя для взаимодействия с сервером. Не все так просто — необходимо будет выбрать способ общения с сервером и позаботиться о безопасности передачи данных. При выполнении сетевых запросов мы будем получать данные, которые необходимо будет обрабатывать и представлять пользователю. И снова предстоит нелегкий выбор: написать собственный код или воспользоваться готовой сторонней библиотекой. А что если пользователь свернул приложение, но мы загрузили еще не все данные? Как научить приложение работать в фоновом режиме и какие возможности для этого есть? Об этом мы расскажем на одном из курсов.

```
1 func response(queue: DispatchQueue?, completionHandler: @escaping
    ↳ (DefaultDataResponse) -> Void) -> Self
2 func responseData(queue: DispatchQueue?, completionHandler: @escaping
    ↳ (DataResponse<Data>) -> Void) -> Self
```

```
3 func responseString(queue: DispatchQueue?, encoding: String.Encoding?,
→ completionHandler: @escaping (DataResponse<String>) -> Void) -> Self
```

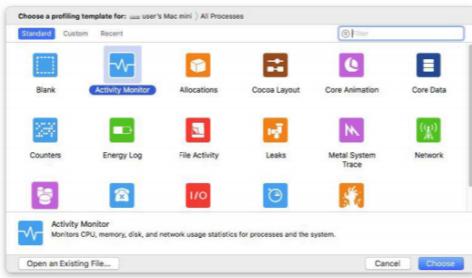
1.1.4. Многопоточность

После создания сетевого слоя мы научимся использовать мощность мобильного устройства по полной. Даже самый современный смартфон может тормозить и выдавать неприемлемую производительность, если неправильно организовать работу с данными и отрисовкой изображения. Какие-то задачи необходимо выполнять в основном потоке, а для каких-то выбрать более низкий приоритет. Тогда интерфейс не будет лагать, а данные будут обновляться незаметно для пользователя. Часть проблем выявляется сразу: вы запускаете приложение и видите некорректные результаты и ошибки, а о некоторых вы будете узнавать только из отзывов в Apple Store. Но их тоже нужно будет решать. Как?

```
1 import UIKit
2 DispatchQueue.global(qos: .userInitiated).async {
3     // Загружаем файлы
4     DispatchQueue.main.async {
5         // Обновляем UI
6     }
7 }
```

1.1.5. Отладка приложений

Есть целый набор инструментов для работы с памятью, интерфейсом, объектами, базами данных и так далее. Вам, как профессиональному разработчику они будут спасать жизнь и не раз. Хороший уровень владения инструментами поможет сэкономить огромное количество времени при отладке. Но зачем исправлять ошибки, если их можно избежать? Мы научимся писать код, тестирование которого можно автоматизировать. Задача нетривиальная, но интересная.



1.1.6. Хранение данных

Далее в приложении, которое мы разрабатываем, будет добавлена возможность работы с большими объемами данных. Часть из них мы сможем хранить в облаке, а часть — на мобильном устройстве, в локальной базе данных. Выполнение запросов, работа с контекстами, синхронизация данных, оптимизация производительности — это то, чему вы научитесь. И, конечно же, мы не обойдем стороной возможности самых современных устройств и мобильных операционных систем Apple: Force Touch, Face ID, виджеты, отправка push-уведомлений, работа с камерой. Возможно, это не самые важные вещи, однако при работе с ними могут возникать трудности и мы постараемся помочь вам избежать их. Мы рады, что вы присоединились к обучению. Добро пожаловать и желаем успехов!

Хранение данных

Запуск приложения

ENTITIES
Post
User

FETCH REQUESTS

CONFIGURATIONS
Default

Attributes

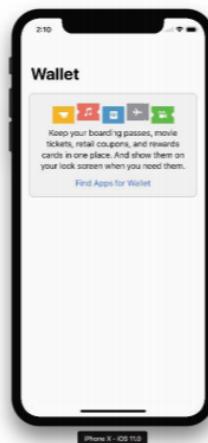
Attribute	Type
S secondName	String
S name	String

Relationships

Relationship	Destination	Inverse
+ -		

Fetched Properties

Fetched Property	Predicate
+ -	



1.2. Знакомство с курсом

Добро пожаловать на курс по обучению программированию на языке Swift. Программа курса посвящена изучению основных принципов программирования на этом языке. Мы познакомим вас с основными составляющими языка: классами, структурами и перечислениями. Расскажем, как с ними можно взаимодействовать, используя функции и замыкания, о том, как организовать хранение данных при помощи словарей, массивов и свойств. Как сделать ваш код более универсальным, используя протоколы и дженерики. Каким образом избежать утечек памяти и написать код, который будет легко читаться и выполняться именно так, как вы ожидаете. При подготовке материалов мы постарались рассказать как о базовых вещах при взаимодействии с языком, так и о более интересных и сложных подходах при разработке.

1.2.1. Знакомство со средствами разработки

Первая неделя посвящена знакомству с языком и средой разработки Xcode. Вы научитесь создавать проекты, выполнять базовые настройки и работать с playground.



1.2.2. Типы и Control Flow

```
1 var value: Int
2 let name = "Hello World!"
3
4 repeat {
5     statements
6 } while condition
```

Во время второй недели вы узнаете об основных типах, используемых в Swift, и в чем заключается различие между ними. Также мы вас познакомим с основными способами управления потоком выполнения кода.

1.2.3. Функции, методы, замыкания

Третья неделя нацелена на работу с функциями, методами и замыканиями. Мы расскажем о протоколах и о том, как строятся коллекции на их основе.

```
1 struct Helloer {
2     func sayHello() {
3         print("Hello world!")
4     }
5 }
6 let helloer = Helloer().sayHello()
```

1.2.4. Жизненный цикл объектов

Следующий блок материала позволит познакомиться вам с жизненным циклом объектов, их инициализацией и деинициализацией, и о том, как это связано с управлением памятью в приложении. Также мы затронем вопросы наследования и расширения функциональности классов, поговорим об атрибутах, ограничивающих доступ к нам.

```
1 class MyClass {
2     var value: Int
3
4     init() {
5         value = 42
6     }
7     deinit {
8         value = 0
9
10    }
11 }
```

1.2.5. Типизация в Swift

Пятая неделя даст более подробную информацию о типизации в Swift. Вы узнаете об использовании дженериков, проверки типов и об их кастинге. Также мы объединим весь изученный за курс материал в одной приложении.

```
1 class Device {
2     var name: String
3
4     init(name: String) {
5         self.name = name
6     }
7 }
8
9 func printDescription(device: Device) {
10    print("Device: \(device.name)")
11 }
```

1.2.6. Курсовое задание

Заключительная неделя будет вашим первым шагом на пути к построению приложения. Желаю вам приятного просмотра и успешного обучения!

```
1 struct Post {
2     typealias Identifier = GenericIdentifier<Post>
3     let id: Identifier
4     let author: User.Identifier
5     let description: String
6     let imageURL: NSURL
7     let userLikes: Bool
8     let likedByCount: Int
9     let createdTime: Date
10 }
```

1.3. История и предпосылки Swift

Знакомство с новым языком программирования от Apple мы начнем с того, что разберемся, для чего он был создан и на чем писали приложения под операционные системы от яблочной компании до его появления. В начале 80-х, вдохновившись языком Smalltalk, Том Лав и Брэд Кокс разрабатывают свой язык программирования. Изначально он назывался ООРС, что расшифровывается как Object Oriented Precompiler, так как являлся препроцессором для C, добавляющим в него поддержку объектно-ориентированного программирования. В 1983 году они основывают

компанию Productivity Products International, сокращенно PPI, и дают своему языку новое имя — Objective-C.

В 88-м году компания NeXT приобретает у PPI лицензию на использования Objective-C для разработки своей операционной системы. Они добавляют в компилятор GCC поддержку Objective-C, а также создают фреймворки AppKit и Foundation Kit. И хотя компьютеры NeXT продавались не очень, сама операционная система привлекла к себе много внимания.

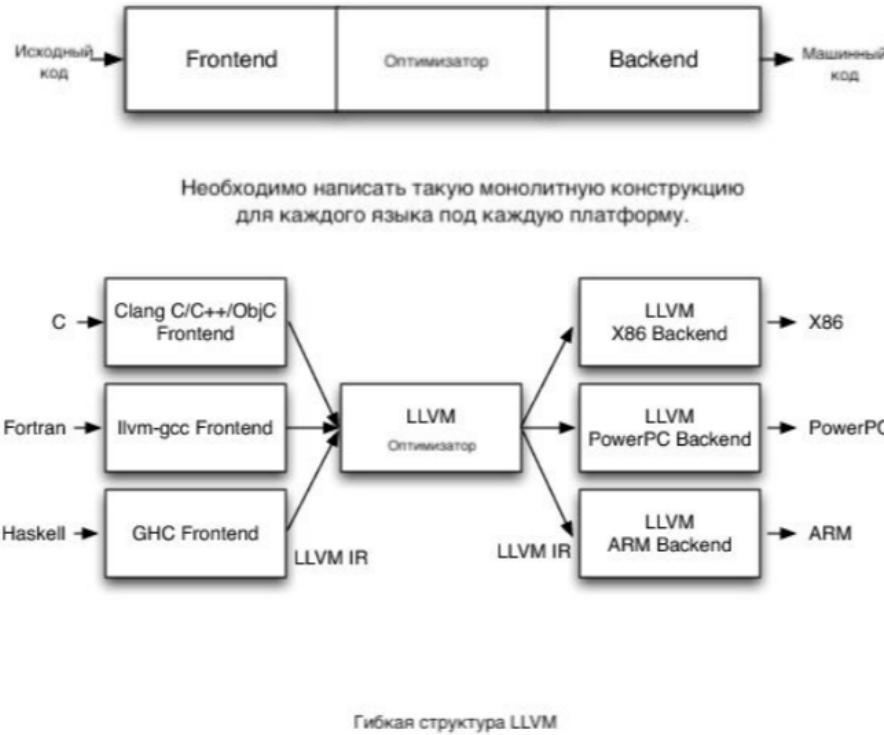
В 1995 году NeXT выкупает права на Objective-C полностью, а год спустя вместе с языком программирования и операционной системой ее выкупает Apple. Они используют наработки NeXT для создания новой операционной системы Mac OS X, которая была выпущена в 2001 году и заменила классическую Mac OS. Позже на ее основе были созданы iOS, watchOS и tvOS. Таким образом, основным языком для Apple стал Objective-C, а AppKit и Foundation Kit стали неотъемлемой частью операционных систем купертиновской компании.

1.3.1. LLVM и традиционные компиляторы

Тем временем в 2000 году Крис Латтнер в рамках своей магистерской работы начинает разработку Low Level Virtual Machine, сокращенно LLVM. Изначально он был написан как часть стека GCC в качестве замены существующего кодогенератора.

1980e	1988
В начале 1980х Tom Love и Brad Cox разрабатывают OOPC	В 1988 компания NeXT приобретает лицензию на использование Objective-C
1995	2000
В 1995 году NeXT выкупает права на Objective-C, а вскоре ее саму приобретает Apple	В 2000 году Chris Lattner начинает разработку LLVM

LLVM — это большой проект, включающий в себя множество технологий. Основная его идея заключается в том, что компилятор транслирует исходный код в универсальное промежуточное представление. Дальше LLVM оптимизирует его и превращает в ассемблерные команды, соответствующие целевой операционной системе и архитектуре. Одним из преимуществ такого подхода является то, что оптимизацию можно написать только один раз для промежуточного представления, а не для каждого языка, как это было раньше. Плюс это позволяет уменьшить зависимость между компонентами. Можно разработать компилятор для любого языка независимо от архитектуры, а для поддержки всех этих языков на новой системе достаточно реализовать для нее трансляцию промежуточного представления в ассемблерный код. На слайдах вы видите высокоуровневую схему этих подходов. В данном примере три компилятора, три транслятора промежуточного представления под каждую из платформ и один LLVM-оптимизатор. Чтобы поддержать такое же количество языков и платформ в традиционном стиле, пришлось бы создать девять монолитных и сложных в поддержке компиляторов.



Первая версия LLVM была выпущена в 2003 году, а в 2005 Apple наняла Латтнера, для того чтобы использовать эти технологии в своих продуктах. Спустя два года LLVM-GCC вместе с XCode 3.1 входят в состав Mac OS X Leopard.

Однако у GCC было несколько недостатков. Основные из них — это огромная кодовая база и проблемы с лицензированием. Поэтому параллельно с этим Крис Латтнер и его команда занимаются разработкой нового компилятора — Clang. Также этот проект входит в статический анализатор и несколько других утилит для анализа кода. И уже с версией XCode 3.2 Apple дает возможность разработчикам пользоваться новым компилятором, а в версии 4.2 он становится компилятором по умолчанию. Также появляется новый отладчик LLDB. Таким образом, Apple имеет современный стек утилит для компиляции C, C++ и Objective-C кода. Однако кое-что в этой системе остается старым, а именно язык — Apple сильно связан с Objective-C. На нем написано много кода, в том числе и сторонними разработчиками приложения. Но он имеет множество недостатков. Мы не будем сейчас подробно на них останавливаться. Тему сравнения Swift и Objective-C мы решили вынести в отдельное видео.

В 2010 году Крис Латтнер начинает работу над новым языком программирования — он должен был сохранить основные концепции предшественника, то есть динамическую диспетчеризацию, позднее связывание, расширение, но при этом быть более безопасным и предоставлять больше

синтаксического сахара, чтобы облегчить разработку.

2007

LLVM был выпущен в 2003, а в 2007 LLVM-GCC вошел в состав Mac OS X Leopard

Недостатки GCC

- Большой объем кодовой базы
- Лицензия GPL

2010

В 2010 году Крис Латтнер начинает работу над новым языком программирования

Через 4 года на WWDC 2014 Apple презентует первую версию Swift и добавляет его в XCode 6.0. Новый язык привлекает к себе много внимания — он получает звание любимого языка программирования по результатам опроса разработчиков на Stack Overflow в 2015 году. Первые версии Swift не имели обратной совместимости. После выпуска новых версий приходилось переписывать код под новую версию. Несмотря на наличие мигратора многое приходилось делать вручную. Учитывая ежегодные релизы и общую сырость языка, многие были не готовы использовать новый язык в реальных проектах.

Но команда разработчиков Swift сделала своей основной целью совместимость будущих версий языка. Это означает не только менее кардинальные изменения в синтаксисе, но и совместимость на уровне библиотек и исполняемых файлов, собранных разными версиями Swift. Так, например, в XCode 9.0 появилась возможность совмещать в одном проекте код, написанный на версии языка 3.2 и 4.0. Пройдя наш курс, вы убедитесь в том, что Swift прост в изучении, безопасен и выразителен. при этом в Swift имеются конструкции и для продвинутых разработчиков, а именно дженерики, указатели, замыкания, а также некоторые приемы функционального программирования. На этом наше видео об истории появления Swift закончено. Не пропустите сравнение Swift и Objective-C. В нем мы расскажем, почему нужно выбирать для разработки именно Swift.

Цель

2014

Совместимость версий – основная цель разработчиков Swift

На WWDC 2014 Apple презентует первую версию Swift

1.4. В чем преимущества Swift над Objective-C

Теперь мы сравним Swift и Objective-C, рассмотрим недостатки последнего, которые и подтолкнули Apple к созданию нового языка. Если что-то из перечисленного далее вам не понятно,

переживать не стоит — в нашем курсе мы объясним значение всех терминов, и вы сможете позже вернуться к этому видео и понять, чего же мы были лишены до выхода нового языка от Apple. Swift похож на С-подобные языки программирования. Например, вместо квадратных скобок для вызова методов используется более привычная dot-нотация. Это делает язык более привычным и понятным для многих разработчиков. Однако по предоставляемым возможностям — он, скорее, Objective-C без С-составляющей. Благодаря этому он хорошо подходит для начинающих. Им не придется разбираться с указателям, импортом заголовочных файлов и прочими особенностями С. При этом Swift не только перенял полезные возможности своего предка, но и добавил много нового. Он стал намного большим, чем просто Objective-C без С. Для начала давайте рассмотрим, что Swift перенял у своего предшественника.

1.4.1. Расширения

Пожалуй, самое заметное — это расширения. Swift тоже позволяет добавлять новые методы существующим сущностям, в том числе и тем, чьи исходники вам недоступны.

```
1  Swift
2
3  extension UIColor {
4      var sw_hex: String? {
5          // ...
6      }
7  }
```

```
1  Objective-C
2
3  @interface UIColor (ELNUtils)
4  - (NSString *)eln_hex;
5  @end
6
7  @implementation UIColor (ELNUtils)
8  - (NSString *)eln_hex {
9      // ...
10 }
11 @end
```

1.4.2. Протоколы

С расширениями тесно связаны протоколы. Apple выводит их на новый уровень, называя Swift протоколоориентированным языком программирования.

```
1  Swift
2
3  protocol MyProtocol {
4      var i: Int {get set}
5      func function()
6 }
```

```
1  Objective-C
2
3  @protocol MyProtocol
4  @property (assign, nonatomic) int i;
5  - (void)function;
6  @end
```

1.4.3. Замыкания

Замыкания представляют из себя блоки кода, которые можно передавать из функции в функцию. При этом они захватывают переменные и не дают им удалиться из памяти.

```
1  Swift
2  func doSomething(completion: (Double) -> Int) {}
3
4  doSomething {
5      (closureInput: Double) -> Int in
6
7      // ...
8      return 0
9 }
```

```
1 Objective-C
2 - (void)doSomethingWithCompletion:(int (^)(double))completion;
3
4 [self doSomethingWithCompletion:^int(double blockInput) {
5     // ...
6     return 0;
7 }];
```

1.4.4. ARC

Так же, как и в Objective-C, в Swift используется автоматический подсчет ссылок на объект — Automatic reference counting (ARC). В большинстве случаев вам не придется задумываться о том, когда удалить объект из памяти.

1.4.5. Динамическая диспетчеризация

Менее заметная на первый взгляд возможность — это динамическая диспетчеризация. Она позволяет выбирать конкретную реализацию метода во время выполнения программы, а не на этапе компиляции. Также в Swift сохранены многие синтаксические конструкции из Objective-C: квадратные скобки для создания массива, обращение к экземпляру из метода через self, основные типы данных, а также многие операторы и ключевые слова.

```
1 Swift
2 class SomeClass {
3     func doSomething() {
4         print("SomeClass")
5     }
6 }
7
8 class SomeSubClass: SomeClass {
9     override func doSomething() {
10         print("SomeSubClass")
11     }
12 }
13 let object: SomeClass = SomeSubClass()
14 object.doSomething()
15 // Выведем "SomeSubClass"
```

```
1  Objective-C
2  @interface SomeClass: NSObject
3  - (void)doSomething;
4  @end
5
6  @implementation SomeClass
7  - (void)doSomething {
8      NSLog(@"%@", @"SomeClass");
9  }
10 @end
11
12 @interface SomeSubClass: SomeClass
13 - (void)doSomething;
14 @end
15
16 @implementation SomeSubClass
17 - (void)doSomething {
18     NSLog(@"%@", @"SomeSubClass");
19 }
20 @end
21
22 SomeClass *object = [[SomeSubClass alloc] init];
23 [object doSomething];
24 // \text{Выведем} SomeSubClass
```

1.4.6. Указатели

Перейдем к различиям. Swift был создан с целью упростить создание и поддержку программного обеспечения. Для этого он должен быть безопасным, быстрым и выразительным. Исходя из этих принципов, были внесены изменения в существующие конструкции С и Objective-C при реализации их аналогов в Swift. Например, указатели в привычном нам виде отсутствуют. Классические указатели не являются безопасными. Арифметика указателей, и типизированный void *, и возможность произвольного доступа к неопределенному участку памяти — это потенциальные источники багов, которые будет очень сложно найти. В Swift на начальном этапе вам, скорее всего, вовсе не понадобятся указатели. Для продвинутых разработчиков имеются несколько видов указателей, отличающихся поведением, но спроектированных так, чтобы быть максимально безопасными.

```
1  Swift
2  var value = 5
3  withUnsafePointer(to: &value) {
4      pointer in
5
6      let doubleValue = Double(pointer.pointee)
7      pointer++
8  }
9  // Безопасная
10 // конвертация типов
11 pointer++ // Невозможно
```

```
1  Objective-C
2  int value = 5;
3  void *pointer = &value;
4  double doubleValue = *(int *)pointer;
5  void *someValue = pointer++;
```

1.4.7. Нулевые указатели

Следующее важное различие — это отсутствие нулевых указателей. Термин, придуманный Энтони Хоаром, был назван ошибкой на миллиард долларов. Нулевые указатели — это причина огромного количества ошибок и уязвимостей. Конечно, нам необходима возможность как-то выражать отсутствие значений, и она у нас есть. Для этого нужно использовать optionalные значения. Они имеют строгую типизацию, и вы обязаны в своем коде обработать ситуации, в которых вы получаете null. К тому же optionalными могут и примитивные типы данных. В Objective-C для этого приходится использовать специальные значения — например, NSNotFound, являющаяся просто константой, равной NSIntegerMax. Однако это небезопасно. Вам нужно следить, чтобы допустимые значения случайно не пересеклись со специальными. Помимо безопасности optionalные значения делают код более понятным. Разработчик, читающий ваш код или использующий ваш интерфейс, всегда будет знать, что функция может вернуть пустое значение. Для этого даже не требуется читать документацию. Вообще, строгая типизация — это одно из важнейших преимуществ Swift. Благодаря ей у компилятора есть возможность подсказать вам, если вы ошиблись с типами.

```
1  Swift
2  let result = [].index(of: "")  
3  if let result = result {  
4      // ...  
5 }
```

```
1  Objective-C
2  NSUInteger result = @[] indexOfObject:@""];
3  if (result != NSNotFound) {
4      // ...
5 }
```

1.4.8. Автоопределение типов

Однако несмотря на это нам не всегда необходимо указывать тип явно. В большинстве случаев компилятор сам выберет корректный тип переменной или константы. Благодаря этому код становится более лаконичными, но остается понятным для разработчиков.

```
1  Swift
2  let string = "string"
```

```
1  Objective-C
2  NSString *string = @"string";
```

1.4.9. Функциональное программирование

Также в Swift есть некоторые базовые элементы функционального программирования. Трансформация коллекций осуществляется вызовом цепочки соответствующих методов и передачи в них функций. Они содержат логику, которая будет применяться для каждого элемента в коллекции.

```
1  Swift
2  let result = [1, 2, 3, 4].map {$0 * 7 }.reduce(1) {$0 * $1 }
3
4  result // 57624
```

```
1  Objective-C
2  NSMutableArray<NSNumber *> *array = @[@1, @2, @3, @4];
3
4  [array enumerateObjectsUsingBlock:^(NSNumber *obj, NSUInteger idx, BOOL *stop) {
5      obj = @(obj.integerValue * 7);
6  }];
7  NSUInteger result = 1;
8  for (NSNumber *value in array) {
9      result *= value.integerValue;
10 }
```

1.4.10. Перечисления

Возможности перечислений были значительно расширены — они могут хранить любые типы данных, а также иметь свои методы и реализовывать протоколы.

```
1  protocol StringRepresentable {
2      func toString() -> String
3  }
4
5  enum UniversalPoint: StringRepresentable {
6      case cartesian(x: Double, y: Double)
7      case polar(r: Double, t: Double)
8      func toString() -> String {
9          switch self {
10             case .cartesian(let x, let y):
11                 return "Cartesian system x:@(x) y:@(y)"
12             case .polar(let r, let t):
13                 return "Polar system r:@(r) y:@(t)"
14         }
15     }
```

```
16 }
17 let point = UniversalPoint.polar(r: 10.5, t: 180.0)
18 print(point.toString()) // "Polar system r:10.5 y:180.0"
```

1.4.11. Unicode

Еще одна важная особенность Swift — это поддержка Unicode-строк. Работа со строками проектировалась таким образом, чтобы обеспечить максимально корректную работу составных графических символов. В отличие от обычного массива ASCII-символов графемы Unicode могут занимать различный объем памяти. Поэтому поддержка Unicode — это очень нетривиальная задача. Но Swift все сделает за вас. В Swift есть множество других мелких особенностей. Нет необходимости ставить точку с запятой в конце каждой строки. Есть возможность переопределять операторы или создавать новые. Оператор присваивания не возвращает значения, чтобы его нельзя было случайно перепутать с оператором сравнения и многими другими. О них мы будем рассказывать в соответствующих лекциях. Конечно, мы перечислили далеко не все различия между языками, однако, надеемся, что вы поняли, в чем была задумка создателей нового языка от Apple. В будущем, разобравшись с ними получше, вы поймете, почему он был реализован именно так, а не иначе.

```
1 let string = "\u{E9}\u{65}\u{301} R U"
2
3 string // "ééR U"
4
5 string.count // 3
6
7 string.utf8.count // 13
```

1.4.12. Будущее Swift

С будущими и прошлыми изменениями языка можно ознакомиться на сайте Swift Evolution, а подписавшись на рассылки, вы можете следить за обсуждением разработчиков о будущем Swift и даже принять в них участие.

Swift evolution:

<https://apple.github.io/swift-evolution/>

Mailing lists:

<https://lists.swift.org/mailman/listinfo>

На этом мы закончим обсуждение языка Swift и перейдем к установке и настройке среды программирования Xcode.

О проекте

Академия e-Legion — это образовательная платформа для повышения квалификации в мобильной разработке. Слушайте лекции топовых разработчиков, выполняйте практические задания и прокачивайте свои скиллы. Получите высокооплачиваемую профессию — разработчик мобильных приложений.

Программа “iOS-разработчик”

Блок 1. Введение в разработку Swift

- Знакомство со средой разработки Xcode
- Основы Swift
- Обобщённое программирование, замыкания и другие продвинутые возможности языка

Блок 2. Пользовательский интерфейс

- Особенности разработки приложений под iOS
- UIView и UIViewController
- Создание адаптивного интерфейса
- Анимации и переходы
- Основы отладки приложений

Блок 3. Многопоточность

- Способы организации многопоточности
- Синхронизация потоков
- Управление памятью
- Основы оптимизации приложений

Блок 4. Работа с сетью

- Использование сторонних библиотек
- Основы сетевого взаимодействия
- Сокеты
- Парсинг данных

- Основы безопасности

Блок 5. Хранение данных

- Способы хранения данных
- Core Data
- Accessibility

Блок 6. Мультимедиа и другие фреймворки

- Работа с аудио и видео
- Интернационализация и локализация
- Геолокация
- Уведомления
- Тестирование приложений