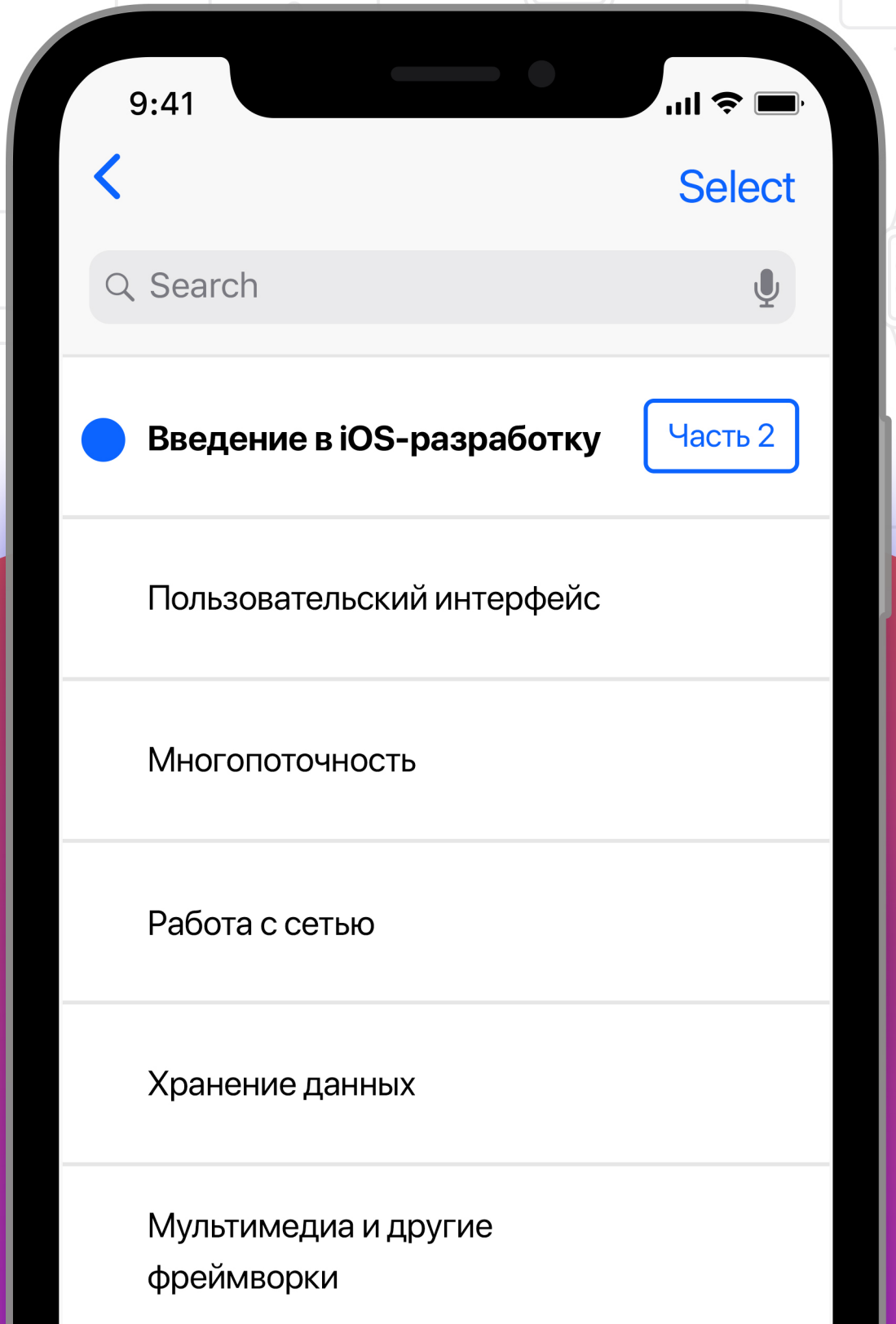
фонд развития
онлайн образования
eldf.ru

e·legion

academy.e-legion.com

Программа iOS-разработчик

Конспект



Оглавление

Глава 2

НЕДЕЛЯ 2

2.1. Создание переменных и констант

2.1.1. Область памяти

Привет! Теперь мы поговорим об основных типах языка Swift и о том, как с ними работать. Но перед тем как начать эту работу, необходимо объявить переменную или константу, в которой будет содержаться значение какого-то типа. Для этого используются ключевые слова var или let для объявления переменных или констант соответственно.

```
var  // переменная, значение можно менять
let  // константа, значение задается единожды, доступ после присвоения значения
```

```
var variableName: Type
let variableName: Type
```

Чтобы выделить память и начать работу с некой переменной или константой, необходимо указать ключевое слово, var или let, имя этой переменной, опционально указать ее тип и можно присвоить изначальное значение. Нужно заметить, что Swift является строго типизированным языком. Поэтому, однажды объявив переменную с целочисленным типом, присвоить строку к ней у нас уже никогда не получится. Это позволяет избежать очень многих ошибок еще на этапе программирования и компиляции. Также стоит заметить, что константе не обязательно указывать изначальное значение сразу же. Его можно объявить и позже. Главное, чтобы оно было там в момент первого использования и указывалась всего лишь один раз при любом истечении обстоятельств.

2.1.2. Объявление типа

Явное объявление типа

```
var variableName: Type = value
let constantName: Type = value
```

Неявное объявление типа

```
var variableName = value
let constantName = value
```

Строгая типизация в Swift

```
var variableName: Type = value
variableName = otherTypeValue // error
var x, y, z: Float
```

2.1.3. Вложенные типы

Swift позволяет работать нам с вложенными типами. Можно объявить переменную или константу внутреннего типа без создания инстансов внешнего типа. Для доступа к вложенным типам используется дот нотации, или нотация через точку

Внутри классов и структур объявлять собственные типы.

Можно объявить переменную или константу внутреннего типа, без создания инстанса внешнего типа.

Для доступа к вложенным типам используется дот нотация.

```
class OuterClass {
    class InnerClass {

    }
}

let a: OuterClass.InnerClass
```

2.1.4. Именование констант и переменных

Для именования констант и переменных можно использовать практически любые Unicode-символы. Нельзя лишь начинать это имя с цифры, оно не должно в себе содержать непечатаемые символы, математические символы, стрелки, а также приватные unicode-символы.

- Можно использовать Unicode символы в том числе эмодзи.
- Нельзя начинать именование с цифр, не должны содержать непечатаемые символы (whitespaces), математические символов, стрелки, а также приватные юникод символов.

Хорошие переменные

```
let isPrivate = true
let availableColors: [UIColor]
```

Плохие переменные

```
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = "dogcow"
```

Хоть у вас и есть возможность объявить имя переменной, которая состоит только из эмодзи, делать мы этого очень не рекомендуем. Это поможет как вам при дальнейшей работе с вашим кодом, так и другим людям, которым придется с ним работать. Рекомендуется называть все переменные и константы английскими именами в lowerCamelCase.

2.1.5. Целочисленные значения

Для работы с целочисленными значениями в Swift предусмотренные типы int и Uint для работы со знаковыми и беззнаковыми целочисленными значениями соответственно. Они будут иметь 32 или 64 бита в зависимости от платформы, на которой исполняется код. Если же вам необходимо точно знать, с какой длины значением вы работаете, то можно использовать типы вроде int8, int16, int32, int64 и их аналоги для беззнакового целочисленного значения. Рекомендуется их использовать, только если мы работаем с каким-то специфичным сетевым протоколом или же на низком уровне, близко к железу. Предпочтительно всегда использовать типы int и Uint. Это позволит избежать многих ошибок и улучшит конвертацию.

```
1  Int, Int8, Int16, Int32, Int64  // знаковые
2  UInt, UInt8, UInt16, UInt32, UInt64 // беззнаковые
```

Int и UInt будут использовать 32 или 64 длину в зависимости от платформы.

Точно указываем длину при работе с данными из сети или при работе с железом на низком уровне, если это необходимо.

Int и UInt предпочтительнее. Меньше ошибок. Легче конвертирование при работе с разными платформами.

2.1.6. Значения с плавающей точкой

Для работы со значениями с плавающей точкой предусмотрены типы float и double. Float является 32-разрядным значением, обеспечивающим точность в шесть знаков после точки. Double является значением с плавающей точкой двойной точности, которая позволяет обеспечить точность в 15 знаков после точки. Эти типы отлично подходят для хранения чисел, точность представления которых не является критичной, например, координаты некоторого объекта. Но если вы работаете с такими важными данными, как цены, рекомендуется использовать другой тип, decimal, о котором мы будем говорить в дальнейших видео.

Для представления чисел вида 3.1415926, 0.1, 273.5

- *Float* 32 — разрядные значения с плавающей точкой, минимум 6 знаков после точки
- *Double* 64 — разрядные значения с плавающей точкой, минимум 15 знаков после точки

Подходит, например, для задания координат.

2.1.7. Неточности значений с плавающей точкой и их сравнение

Как и во множестве других языков программирования, операции со значениями с плавающей точкой не являются точными. Можно рассмотреть простейший пример, сравнив литералы 0.3 и сумму литералов 0.1 и 0.2. Эти значения не будут равны. О том, как правильно сравнивать значения с плавающей точкой, вы ознакомитесь в материалах для самостоятельного изучения.

```
Float  // 32 bit.
Double // 64 bit

0.1 + 0.2 = 0.30000000000000004
0.3 != 0.30000000000000004
FLT_EPSILON // равно 2 в степени -23

x = 0.3
y = (0.1 + 0.2)
```

Вычисляем разность z = (x - y)

Берем модуль от z

Если z <= FLT_EPSILON, значит числа равны

2.1.8. Booleans

Для хранения логического типа используется тип bool. В отличие от C, objective C и других языков, логический false не является эквивалентом 0, а любое другое значение эквивалентом true. Соответственно, мы не можем передать целочисленное значение как условие оператора if.

Ключевое слово Bool

true или false

В отличие от C или Objective-C логическое false не является эквивалентом 0, а любое другое значение — эквивалентом true

```
//Некорректное использование
let i = 1
if i {
  // will not compile with error
}
```

2.1.9. Tuple (Кортеж)

Далее мы рассмотрим Tuple, или Кортеж. Ранее мы использовали три перемены, чтобы объединить какие-то схожие данные, однако куда удобнее использовать для этого Tuple. Мы можем объявить координаты объекта, объявив Tuple с тремя элементами и именуя каждый из них координатами X, Y и Z соответственно. Теперь мы можем объявить переменную, всего лишь передав эти три значения, а обращаться к каждой из них по имени. Именовать каждый из элементов Кортежа совершенно необязательно, можно обращаться к ним по индексу. Также Кортеж может содержать элементы абсолютно разных типов.

```
var coordinate = (x: 0.0, y: 0.0, z: 0.0)
let http404Error = (404, "Not Found")
```

В кортеже можно группировать разные типы.

```
var someTuple = (code: 404, description: "Not Found")
```

Теперь можно не указывать имена при обращении

```
someTuple = (200, "Ok")
```

2.1.10. Optional

Для работы со значениями, которых в каких-то случая может не быть, используется optional-тип, мы легко можем конвертировать строку с символами 1, 2, 3 в целочисленное значение 123, но такое не пройдет со строкой, в которой содержится фраза Hello world. Именно поэтому конструктор целочисленного значения со строкой возвращает нам optional-тип, в котором может содержаться либо число, которое нам удалось распарсить из данной строки, либо же nil.

Optional используется в тех случаях, когда у переменной может отсутствовать значение

Конвертируем строку в число

```
let correct = Int("123")
let incorrect = Int("Hello World!")
```

Конструктор возвращает Optional<Int> или Int?

Int? равен Int или nil

2.1.11. Nil

Nil является ключевым словом, которое обозначает отсутствие значения для некоторой переменной. nil нельзя присвоить любой переменной. Его можно присвоить только той переменной или константе, тип которой объявлен как optional.

- nil — обозначает отсутствующее значение для переменной.
- nil нельзя присвоить обычной переменной, только Optional<Type> тип может принимать значение nil.

2.1.12. Использование If с Nil

Для работы с optional-значениями необходимо проверять, находится ли в них какое-то значение. Для этого можно сравнить эту переменную или константу с nil. Если же проверка не удалась, значит, значение присутствует, и мы можем безопасно к нему обращаться. Для распаковки переменной, unwrapping, используется оператор восклицательный знак, указываемый после переменной или константы. Если же в переменной в данный момент содержится nil, то произойдет exception, и выполнение нашей программы будет закончено. Использование оператором восклицательного знака, или Force unwrap, крайне не рекомендуется. Это небезопасно, затрудняет чтение и зачастую приводит к ошибкам. Для этого лучше использовать эквивалентный оператор if let, где мы сразу же в операторе ветвления можем создать новую переменную, в которой будет уже содержаться не optional-значение. Если же в переменной или константе содержался nil, то тело условия выполнено не будет. В этой лекции мы узнали, как объявлять константы и переменные, как поступать в тех случаях, когда переменная может не иметь никакого значения, а также узнали об основных типах, с которыми мы будем работать в течение всей разработки на языке Swift.

```
var x: Int?
```

После сравнения используем force unwrap (!) для работы со значением.

Затрудняет чтение и небезопасно.

```
if x != nil {
   return x! + 5
}
```

Лучше if let — эквивалентно проверке с присвоением значения.

```
if let nonOptionalX = x {
  return nonOptionalX + 5
}
```

2.1.13. Операторы

В этой лекции вы познакомитесь с операторами и особенностями работы с ними в Swift. Операторы — это специальные символы или фразы, которые вы используете для комбинирования, проверки или изменения значений.

2.1.14. Типы операторов

В Swift используется большая часть стандартных C операторов. Некоторые из них были пере-

работаны для уменьшения количества ошибок при разработке. Например, оператор присвоения не возвращает значения, чтобы избежать путаницы с оператором эквивалентности. При использовании арифметических операндов плюс, минус, умножить, делить на литералах выполняется проверка на переполнение. Помимо этого, в Swift добавлены операторы диапазона, на которых мы остановимся подробнее. Операторы можно разделить на три типа по количеству операндов: унарные, бинарные и тернарный, который в Swift'е только один и своего вы создать не можете.

Операторы — это специальные символыили фразы, которые вы используете для комбинирования, проверки или изменения значений. Большая часть операторов — стандартные операторы языка C.

Унарный

```
-variable; !variable
```

Бинарный

```
variable + variable
```

Тернарный

```
boolValue ? value1 : value2
```

https://developer.apple.com/documentation/swift/operator_declarations

2.1.15. Оператор присвоения (=)

Оператор присваивания предназначен для инициализации или обновления переменной или константы. Ещё раз хочу обратить внимание, что в отличие от C оператор присваивания не возвращает значение присвоения. То есть пример, показанный на слайде, будет не верен.

```
//Для объявление или обновление значения
let b = 10
var a = 5
a = b
let (x, y) = (1, 2)
```

В отличие от C, подобный код приведет к ошибке

```
let x = 1
if x = 5 {
```

```
3
4  }
```

2.1.16. Арифметические операторы

Большая часть арифметических операторов работает с двумя операндами. Исключениями являются операторы плюс и минус, которые могут быть и унарными. Минус для инверсии значения, плюс же никакого смысла в себе не несёт, но может служить для лучшего форматирования кода. При использовании арифметических операторов на литералах выполняется проверка на переполнение. Оператор плюс также можно использовать для конкатенации строк и массивов.

```
1  //Бинарные операторы:
2  let a = 5
3  let b = 3
4  var c = 0
5  c = a + b
6  c = a / b
7
8  //Унарные операторы:
9  c = -a
10
11 //Конкатенация строк:
12 let helloWorld = "Hello" + "world!"
```

2.1.17. Составные операторыприсваивания

Составные арифметические операторы позволяют немного сократить запись. Они сочетают в себе оператор присваивания и оператор для выполнения арифметической операции. На слайде пример одних и тех же операций, записанных по разному.

```
1   var a = 1
2   a += 2 // значение a равно 3, a += 2 // эквивалентно a = a + 2
```

//Также используются

```
a -= 2
a *= 2
a /= 2
```

2.1.18. Операторы сравнения

Swift поддерживает все основные операторы сравнения из языка C. Также используются операторы эквивалентности === и !==, оператор неэквивалентности. Операторы сравнения возвращают в качестве результата boolean. Операторы сравнения могут быть использованы для сравнения кортежей в том случае, если количество элементов в кортежах совпадает, а типы, используемые в кортежах, поддерживают сравнение. Стандартная реализация ограничена сравнением кортежей до семи элементов. Для сравнения большего количества элементов в кортеже вам понадобится написать собственные реализации операторов сравнения.

```
// Все основные C операторы сравнения (== , <, >).
// Операторы эквивалентности (=== и !==) возвращают Bool.

// Для кортежей до 7 элементов
(1, "zebra") < (2, "apple")
(4, "dog") == (4, "dog")
// Типы и количество элементов должны совпадать.
```

2.1.19. Тернарный условный оператор

Тернарный оператор условия не имеет особенностей в реализации. На слайде показана запись в виде данного оператора и та же самая запись, но с использованием выражения if... else.

```
// Используется для сокращения записи if-else.
question ? answer1 : answer2
if question {
  answer1
} else {
  answer2
}
```

2.2. Nil-coalescing оператор (Бинарный оператор (??))

Nil-coalescing operator служит для работы с optional значениями в Swift. Это бинарный оператор, который разворачивает значение левого операнда, а если оно равно nil, то вовзращается значение правого операнда. В качестве операнда справа не может использоваться optional значение.

```
var a: Int?
let b: Int = 5
var c: Int = 0

c = (a ?? b) // c = 5

if a != nil {
  c = a!
} else {
  c = b
}
```

2.2.1. Операторы диапазона

В Swift используется оператор диапазона, который является сокращённой записью для инициализации ряда значений. В качестве операндов на данный момент могут использоваться только Int значения. Значение левого операнда должно быть меньше значения правого операнда. Соответственно, вы можете создавать только восходящие последовательности. Для создания нисходящей последовательности можно воспользоваться функцией reversed.

- Сокращенная форма для представления ряда значений.
- Поддерживается только тип Int.
- Обязательно a < b.
- Восходящий диапазон от a до b (a...b).
- Для нисходящего диапазона используем функцию reversed() (a...b).reversed().

2.2.2. Типы диапазонов

Диапазоны бывают закрытые, полуоткрытые и односторонние. Закрытый диапазаон предоставляет последовательность значений, начиная от левого операнда и заканчивая правым. Удоб-

но использовать для организации циклов в указанном вами диапазоне. Полуоткрытый диапазон отличается от закрытого лишь тем, что значение правого операнда не включается в последовательность. Применение находит при работе с массивами. Полуоткрытый диапазон позволяет исключать значения только правого операнда.

Три типа: закрытые, полуоткрытые и односторонние.

- Закрытый диапазон (a...b) включает оба значения 1...5
- Полуоткрытый (a..<b) в отличие от закрытого, b не входит. 1..<5

2.2.3. Операторы диапазона (односторонний диапазон)

Односторонний диапазон создаёт последовательность либо от минимального возможного значения до значения, указанного вами, либо же от значения, указанно вами, до максимального возможного значения. Так же, как и полуоткрытый диапазон, чаще всего находит свое применение при работе с массивами.

- От минимально возможного значения, до значения указанного вами (...a)
- От значения указанного вами до максимально возможного значения (a...)

```
let letters = ["a", "b", "c", "d", "e"]
for letter in letters[2...] {
  print(letter) // c, d, e
}
for letter in letters[...2] {
  print(letter) // a, b, c
}
```

2.2.4. Операторы диапазона (без ограничений)

Вы можете создавать диапазон без привязки к коллекции. При попытке итерации подобного диапазона вы либо получите ошибку, если левая граница не будет указана, либо получите диапазон, максимальным значением которого будет максимальное значение Int'a же.

```
//Исключительные ситуации
let rangeLeft = ...5
for index in rangeLeft {
  print(index)
```

```
} // Приведет к ошибке

let rangeRight = 5...
for index in rangeRight {
  print(index)
} // Правая граница равна максимальному значению Int
```

2.2.5. Логические операторы

Swift поддерживает три стандартных логических оператора: НЕ, И, ИЛИ. Для них задано стандартное поведение, операторы можно комбинировать, например, для проверки выполнения нескольких условий. Если вам не приходилось использовать их ранее, то в материалах для самостоятельной подготовки вы найдёте ссылку для ознакомления с материалами.

Стандартная реализация операторов Модифицируют или комбинируют булевы значения

- Логическое НЕ

```
(!a)
```

- Логическое И

```
(a && b)
```

- Логическое ИЛИ

```
(a || b)
```

В этом главе мы познакомились с основными операторами в Swift.

2.3. Строки и символы

2.3.1. Создание строки

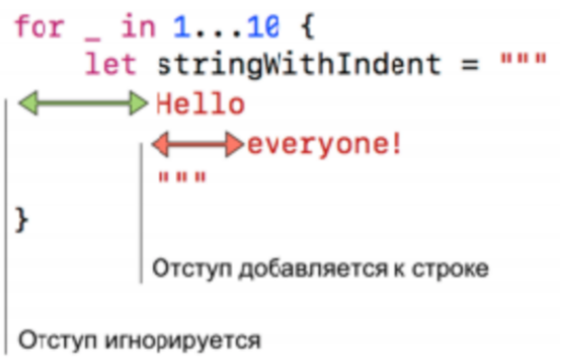
Данная лекция посвящена работе со строками и символами в Swift. Строка — это последовательность символов. В Swift для хранения строк используется тип String. На слайде представлен синтаксис для создания строк.

```swift
let immutableGreeting = "Hello everyone!"

var mutableGreeting = "Hello everyone!"

let emptyString = ""
```

2.3.2. Многострочный текст

Многострочный текст создается с помощью трех двойных кавычек. Отступы и переносы, которые находятся внутри кавычек, будут сохранены в String. Если вам необходимо добавить отступы у текста для лучшей читаемости, то необходимо перед закрывающими кавычками поставить пробелы или табы, которые будут игнорироваться для предыдущих строк.

```swift
let multiLine = """
This
is
string
"""
```



2.3.3. Escape последовательности

Внутри строк можно использовать escape-последовательности с помощью обратного слэша, например, для переноса каретки, либо $\backslash u$ для отображения *Unicode*-символов. Пустая строка создается с помощью двух двойных кавычек. В отличие от Objective-C мутабельность строк определяется не классом NSMutableString и NSString, а ключевым словом при определении новой строки: var для изменяемой строки и let — для неизменяемой.

```
/*

\\ - //обратный слэш
\'' - //двойная кавычка
\n - //перенос строки
\u{00000000} - //unicode символ

*/
```

```
print("\u{1F40E}") // выводит 🐎
```

2.3.4. String — это коллекция

Начиная со Swift 4, со строками можно работать как с массивами. Тип String поддерживает протокол Collection. Например, используя цикл for in, получаем каждый символ, который относится к типу Character.

```
for char in "Кот" {
    print(char)
}

let catCharacters: [Character] = ["К", "о", "т"]
let catString = String(catCharacters)
```

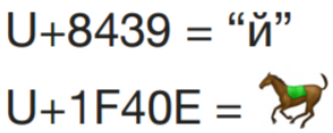
2.3.5. Конкатенация строк

Строку можно собрать из массива символов Character. Отдельный же символ можно создать как переменную или константу того же типа. Строки можно объединять при помощи операторов + и +=. Также к строке можно добавлять символы при помощи функции append.

```
let exclamation: Character = "!"
var cat: String = "Кот"
var result: String = ""
cat.append(exclamation) // Кот!
result = cat + " " + cat // Кот! Кот!
result += " " + cat // Кот! Кот! Кот!
```

2.3.6. Unicode

Строки можно конструировать из литералов, выражений и значений с помощью добавления их в строковый литерал. Для включения значений используется выражение обратный слэш и две скобки, внутри которых помещается выражение. Часто подобное выражение находит применение при использовании функции Print. Типы Character и String поддерживает Unicode. Это значит, что вы можете хранить практически любой символ, из любого языка в стандартизированном виде. Одно Unicode-значение хранит информацию об одном Unicode-символе. Например, может храниться значение символа «й» или эмодзи лошади.




```
"\u{0435}" // e
"\u{0308}" //
let yoLetter = "\u{0435}\u{0308}" // ë
```

2.3.7. Сложение символов

Каждый экземпляр типа Character может хранить графему. Графема — это одно или несколько Unicode-значений, объединенных в один символ. Помещаем в одну константу e, в другую — точки от «ё». Складываем, получаем новый символ с совершенно другим кодом. Таким простым способом удобно собирать символы, используемые в языках, или комбинировать эмодзи, например из двух региональных индикаторов R и U можно собрать флаг. Для определения количества символов у типа String используется свойство count. Возьмем для примера слово «белье» и прибавим к нему символ в виде точек от «ё». Получаем «бельё». При этом количество символов осталось то же, что и было.

```
let r = "\u{1F1F7}" // " R " : String
let u = "\u{1F1FA}" // " U " : String

let ruFlag = r + u // "🇷🇺" : String
ruFlag.count // 1

let ruFlagCharacter: Character = ruFlag.first!
                              // "🇷🇺"
```

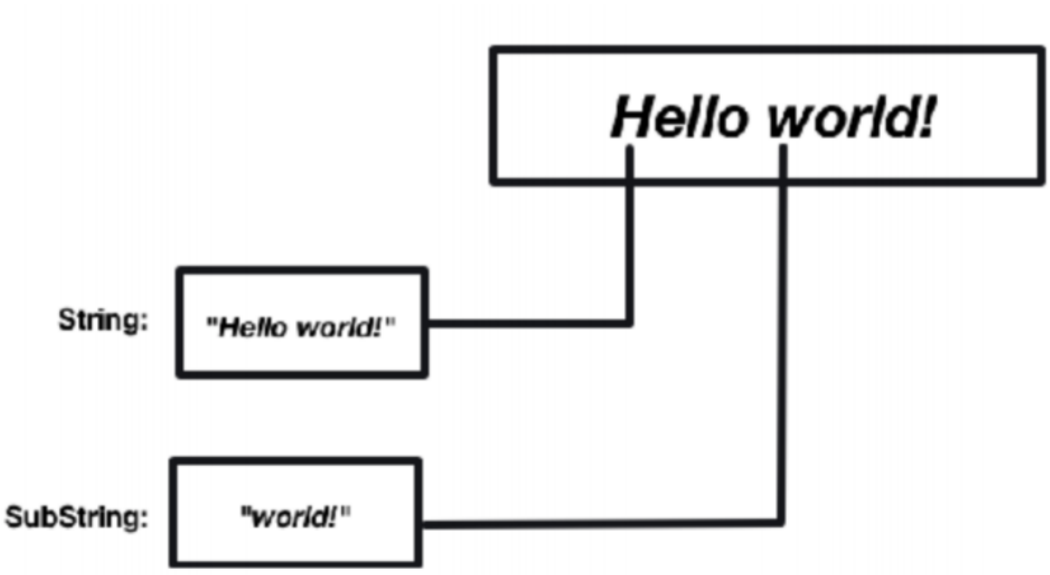
2.3.8. Работа со строками как с коллекцией

Так как String поддерживает протокол Collection, мы можем использовать методы для работы с коллекциями. Здесь показаны примеры получения индекса, удаления символа с определенным индексом или вставка нового символа в слово. Более подробно можно посмотреть в документации по соответствующим протоколам.

```swift
var word = "Какое-то слово!"
print(word[word.startIndex]) // К
print(word[word.index(word.startIndex, offsetBy: 3)]) // о

word.remove(at: word.startIndex)
print(word) // акое-то слово!

word.insert("К", at: word.startIndex)
print(word) // Какое-то слово!
```

2.3.9. SubString

Когда мы получаем подстроку из строки при помощи Subscript, то есть обращения к элементам по индексу или методов наподобие Prefix, мы получаем экземпляр типа SubString.



2.3.10. Создание строки из SubString

Для SubString реализованы те же интерфейсы, что и для String, то есть работа осуществляется в большинстве случаев таким же образом. Зачем тогда нужен сам SubString? В целях экономии памяти и улучшения производительности. SubString использует для хранилища ту же память, что и оригинальная строка. То есть в большинстве случаев вам не нужно обращать внимание на издержки производительности, пока вы не модифицируете SubString. SubString не следует использовать для долговременного хранения строк, даже если в ней содержится всего один символ. Указатель на оригинальную строку будет сохранен, и та строка, на которую мы ссылаемся, не сможет быть удалена из памяти.

```
let string = "Hello word!"
let subString = string.suffix(5)
let stringFromSubString = String(subString)
```

2.3.11. Сравнение строк

Сравнение строк выполняется при помощи оператора равенства и неравенства. При сравнении строк выполняется сравнение графем. Рассмотрим на примере: создаем две строки, одна состав-

ная, а вторая была представлена изначально одним словом. В результате сравнения получаем, что обе строки равны.

```
var firstWord = "бельe"
firstWord += "\u{0308}"
let secondWord = "бельё"


if firstWord == secondWord {
    print("Строки равны")
} // Строки равны
```

Начало и конец строки можно проверить на совпадение с определенным значением при помощи hasPrefix и hasSuffix.

```
let run = "прибежать"
let prefix = "при"
let suffix = "ать"

if run.hasPrefix(prefix) && run.hasSuffix(suffix) {
    print("Начинается на \"при\", заканчивается на \"ать\"")
} // Начинается на "при", оканчивается на "ать"
```

Данной информации вам хватит для работы со строками практически в любом приложении. С дополнительной информацией можно ознакомиться в официальной документации от Apple.

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