Table of Contents

Swift

Виды пропертей в swift.

- Stored – являются полями класса, хранят внутри себя значение. Могут содержать наблюдатели свойства. willSet(newValue) didSet(oldValue). Наблюдатели есть только у stored properties.
- Computed – вычисляются на лету, не хранят непосредственного значения.

Ключевое слово lazy может быть применено к stored свойству, таким образом свойства будет неинициализировано, до попытки первого обращения к нему
Всегда объявляйте lazy свойства как переменные (с помощью ключевого слова var), потому что ее значение может быть не получено до окончания инициализации. Свойства-константы всегда должны иметь значение до того, как закончится инициализация, следовательно они не могут быть объявлены как lazy. Lazy *потоконебезопасно*. Инициализация может произойти одновременно с разных потоков, правда в итоге, останется только 1 переменная.

Квалификаторы доступа

Public classes and class members can only be subclassed and overridden *ONLY IN* the defining module (target).
Open classes and class members can be subclassed and overridden *both inside and outside* the defining module (target).
Internal – visibility in the module (default)
Fileprivate restricts the use of an entity to its defining source file. You typically use fileprivate access to hide the implementation details of a specific piece of functionality when those details are used within an entire file

Private – as for other lanugages

Важный момент по поводу Extensions. Если в классе есть приватное поле, и екстеншен объявлен внутри того же файла, что и сам класс, то экстеншен будет иметь доступ к приватному полю. Если естеншен объявлен в другом файле, то не будет.

Closures

Замыкания – это ссылочный тип
{ (параметры) -> тип результата in
выражения
}

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in return s1 > s2 })
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in return s1 > s2 })
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 })
reversedNames = names.sorted{$0 > $1 }
```

Если замыкание идет последним параметром, можно вынести его за скобки. Так же есть доступ к сокращенным именам параметров $0, $1

Замыкания способны захватывать внешние переменные.

```
var a = 1
var b = 2
let closure: () -> Int = {
    return a+b
}
var result = closure() // 3
a = 3
b = 5
result = closure() //5
```

В качестве оптимизации Swift может захватить и хранить копию значения, если это значение не изменяется самим замыканием, а так же не изменяется после того, как замыкание было создано. Swift также берет на себя управление памятью по утилизации переменных, когда они более не нужны.

@escaping – обозначает “сбегающие” замыкания. Грубо говорят, это значит, что замыкание, которое было передано в функцию. Может быть вызвано потом, вне функции в которую передано. Т.е. замыкание не принадлежит самой функции.

```
var completionHandlers: [() -> Void] = []
func someFunctionWithEscapingClosure(completionHandler: @escaping () -> Void) {
        completionHandlers.append(completionHandler)
}

func someFunctionWithNonescapingClosure(closure: () -> Void) { closure() }

class SomeClass { var x = 10 func doSomething() {
someFunctionWithEscapingClosure { self.x = 100 }// добавление {x = 100} в массив
completionHandlers

someFunctionWithNonescapingClosure { x = 200 } }// захват переменной   исполнение выражения,
но не передача его куда-то вне.
}
```

В данном примере есть массив замыкание, и функция, в которую мы передаем замыкание, добавляет замыкание во внешний по отношению к функции массив.
Определение замыкания через @escaping означает, что вы должны сослаться на self явно внутри самого замыкания.

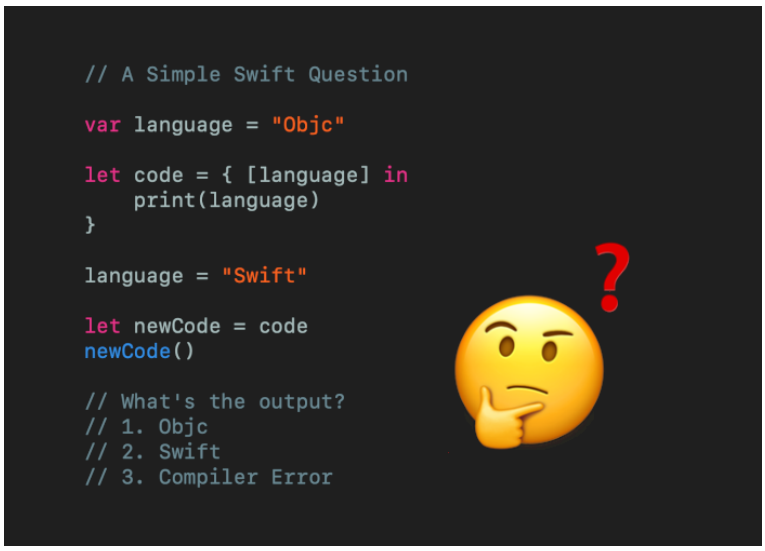
Способы захвата переменных замыканиями

Есть два способа захвата,

1. через [], захватывая переменную как параметр
2. без захвата через[], тогда захват сработает по ссылке.

В данном примере правильным ответом будет *Objc*.

Связано это с особенностью захвата переменной из контекста замыканием.
Если убрать скобки, то выведет *swift*.



Поскольку language – строка, это значит что она value type, и передается *копированием*. Когда мы делаем захват через [], как параметр, мы захватываем переменную как параметр, те создаем ***копию*** строки.

А если сделать захват без скобок, то будет создана ссылка на language переменную. Из за особенностей захвата контекста и получается такая разница в работе. Если бы переменная language была классом, то разницы как захватывать, через [] или нет, не было бы, т.к классы – ссылочные типы, в любом случае получили бы ссылку.

Autoclosures

Автозамыкания - замыкания, которые автоматически создаются для заключения выражения, которое было передано в качестве аргумента функции. Автозамыкания позволяют вам откладывать вычисления, потому как код внутри них не исполняется, пока вы сами его не запустите.

```
let summ = { return 3 + 5 }
```

Наследование

- Наследоваться можно от классов.
- Структуры и перечисления наследование не поддерживают.

Наследование статичных методов

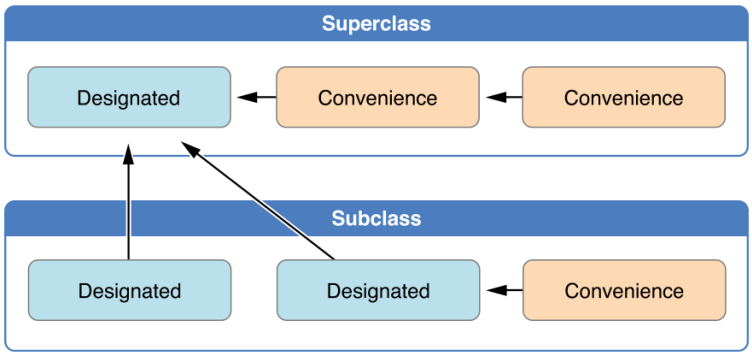
Статичные методы способны наследоваться.
Вызываем через ChildClass.metod().

- Статику можно наследовать, но нельзя оверрайдить.
- Добавлять вычисляемые свойства и вычисляемые свойства типа
- Определять методы экземпляра и методы типа
- Предоставлять новые инициализаторы
- Определять сабскрипты (индексы)
- Определять новые вложенные типы
- Обеспечить соответствие существующего типа протоколу
- Запрещено хранить stored properties, оверрайдить методы.

Инициализаторы

- У структур создаются инициализаторы по умолчанию, с возможность инициализации всех полей структуры
- Если у класса нет вообще инициализаторов, то создается инициализатор по умолчанию
- Неопциональные свойства должны быть обязательно проинициализированы до создания класса
- Инициализаторы наследуются, их можно переопределять
- Инициализаторы могут вызывать другие инициализаторы для инициализации части экземпляра. Этот процесс называется - делегирование инициализатора.
- Назначенные (designated) инициализаторы в основном инициализаторы класса. Они предназначены для того, чтобы полностью инициализировать все свойства представленные классом и чтобы вызвать соответствующий инициализатор суперкласса для продолжения процесса инициализации цепочки наследований суперклассов.

- До тех пор пока не вызван инициализатор суперкласса, я не могу настраивать унаследованные проперти и не имею доступа к self. Те **я обязан(!)** инициализировать сначала все проперти текущего класса. Потом позвать super.init() и только после этого могу настраивать унаследованные проперти
- Если я хочу обязать класс создавать какой-то инициализатор, я помечаю его словом **required,** это значит, что все классы наследники, обязаны реализовывать этот инициализатор
- Вспомогательные инициализаторы пишутся точно так же, но только дополнительно используется вспомогательное слово convince. Вспомогательный инициализатор в итоге **всегда(!)** зовет назначенный.



Двухфазная инициализация

Фаза первая

- Назначенный или вспомогательный инициализатор вызывается в классе init()
- Память под новый экземпляр этого класса выделяется. Но она еще не инициализирована.
- Назначенный инициализатор для этого класса подтверждает, что все свойства, представленные этим классом, имеют значения. Память под эти свойства теперь инициализирована.
- Назначенный инициализатор передает инициализатору суперкласса, что пора проводить те же действия, только для его собственных свойств super.init()
- Так продолжается по цепочке до самого верхнего суперкласса.
- После того как верхушка этой цепочки достигнута и последний класс в цепочке убедился в том, что все его свойства имеют значение, только тогда считается, что память для этого экземпляра полностью инициализирована. На этом первая фаза кончается.

Фаза вторая

1. Двигаясь вниз по цепочке, каждый назначенный инициализатор в этой цепочке имеет такую возможность, как настраивать экземпляр. Теперь инициализаторы получают доступ к self и могут изменять свои свойства и устанавливать значения для унаследованных свойств суперкласса, создавать экземпляры и вызывать методы.
2. Наконец, каждый вспомогательный инициализатор в цепочки имеет возможность настраивать экземпляр и работать с self.

Опционалы

Технически представляют из себя дженерик енам с associated value

```
public enum Optional<Wrapped> : ExpressibleByNilLiteral {

    /// The absence of a value.
    ///
    /// In code, the absence of a value is typically written using the `nil`
    /// literal rather than the explicit `.none` enumeration case.
    case none

    /// The presence of a value, stored as `Wrapped`.
    case some(Wrapped)
}
```

Реализация метода map

```
@inlinable
public func map<U>(_ transform: (Wrapped) throws -> U) rethrows -> U? {
    switch self {
    case .some(let y):
      return .some(try transform(y))
```

```
    case .none:
      return .none
    }
  }
```

Можно анврапнать через force ! или ?. Можно создавать цепочки опциональных вызовов (optional chaining) через ?. Если хотя бы один вызов наткнется вернет nil, то и вся оставшаяся цепочка вызвана не будет

Структуры, классы перечисления

Размещаются, как правило, на стеке

- Структуры, не поддерживают наследование, передаются в методы копированием. Могут иметь свои поля и методы. Из-за своего положения в памяти, на стеке, доступ к структурам осуществляется быстрее.
- Енумы, не поддерживают наследование, передаются копированием. Не имеют своих полей, но могут иметь методы и associated values

Размещаются в куче, но с оговорками в Stack promotions

- Классы, поддерживают наследование и полиморфизм, передаются по ссылке.
- Функции и замыкания – тоже ссылочные типы.

Структуры

- Лежат в стеке.
- Потокобезопасны, т.к. имеют свое стековое пространство для каждого потока, каждый поток будет работать со своей собственной копией структуры. Собственно это очевидно: при захвате параметра через [] или передаче как аргумента *без* inout модификатора.
  При передаче по ссылке и возможной мутабельности структуры потокобезопасность несколько смазывается.
- Память на стеке выделяется быстрее, поскольку фактически это операция инкремента(push) и декремента(pop) интового указателя на вершину стека.
- Функции в структуре требуют mutating чтобы изменять внутреннее состояние структуры.
- Если структура let, ее состояние нельзя никак изменить, даже через mutating
- Если структура хранит в себе ссылочные типы – ссылки на них так же копируются при копировании структуры. Это случай если структура внутри себя агрегирует объект класса. При копировании структуры скопируется и указатель на этот вложенный объект, значит через физически разные структуры мы будем обращаться внутри к 1 и тому же вложенному объекту класса по одной и той же ссылке. Таким образом, при копировании такой структуры удваивается и количество ссылок на вложенные объекты, что не очень хорошо.
- Если структура внутри себя содержит объект класса, то структура, скорее всего, будет размещаться на хипе, но это зависит от конкретных оптимизаций компилятора, надо смотреть sil

Касательно памяти есть нюансы в том как value and reference types представляются в системе. Если принтануть размер массива в 100000 интов элементов, то мы увидим 8 байт. Если принтануть очень длинную строку, мы увидим 16 байт.
Эмпирически получается, что система хранит на стеке структуру с указателем на объект массива. То же самое получается и со строкой. На стеке хранится структура String, но внутренний массив ее характеров физически сторится на хипе.
На самом деле то, где рантайм разместит строку и массив зависит от ситуации.

COW(copy on write)

Все value types в swift, в обычных ситуациях, передаются копированием. Это нормально до определенного момента, когда объем структуры является довольно большим, например массив с большим количеством элементов. Для этого используется COW оптимизация, это значит что при передаче такого массива в функцию, например, он не будет копироваться по умолчанию, а будет передаваться указатель на него. Копирование массива произойдет только в том случае, если в функции будет попытка что ни-будь записать в массив, то есть будет попытка изменения исходной структуры.
В общем *копирование буквально произойдет при попытке записи* в структуру данных.
Важно: не все value types имеют такое поведение, по умолчанию это будут только структуры из пространства Foundation, array, dictionary, set. Для кастомных структур требуется реализовывать такое поведение вручную.

```
var array1 = [1, 2, 3, 4]
print(address(of: array1))     // 0x102000020

var array2 = array1

print(address(of: array2))     // 0x102000020

array1.append(2)

print(address(of: array1))     // 0x102142d80
print(address(of: array2))     // 0x102000020
```

Для ручной имплементации COW потребуется сделать боксинг структуры, те завернуть ее в объект класса. Затем запакованную в класс структуру мы еще раз пакуем, но уже в объект стуктуры.

```
struct User {
    var identifier = 1
}

final class Ref<T> {
    var value: T
    init(value: T) {
        self.value = value
    }
}

struct Box<T> {
    private var ref: Ref<T>
    init(value: T) {
        ref = Ref(value: value)
    }

    var value: T {
        get { return ref.value }
        set {
            guard isKnownUniquelyReferenced(&ref) else {
                ref = Ref(value: newValue)
                return
            }
            ref.value = newValue
        }
    }
}

let user = User()

let box = Box(value: user)
var box2 = box                // box2 shares instance of box.ref

box2.value.identifier = 2      // Creates new object for box2.ref
```

Generics

Используются так же как и в других языках. Дженерики могут поддерживать некоторые ограничения для классов, например соответствие протоколу Equatable

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

2.    for (index, value) in array.enumerated() {

3.      if value == valueToFind {

4.        return index

5.      }

6.   }

7.  return nil

8.}
```

Если требуется сделать generic протокол, то используется ключевое слово associatedtype, используется чисто для протоколов

```
1.protocol Container {

2.  associatedtype Item: Equatable

3.  mutating func append(_ item: Item)

4.  var count: Int { get }

5.  subscript(i: Int) -> Item { get }

6.}
```

Some keyword (5.1)

Some является индикатором того, что тип является «непрозрачным»(opaque). Opaque types являются родственниками обычных generic types, но работают иначе, их часто описывают как reverse generics

- При использовании generic type placeholder (T for example), вызывающая функция уже знает конкретный тип данных, который скрывается за T
- В случая с opaque types, возвращаемый тип выглядит как протокол, но на самом деле реализация определяется конкретным типом-имплементацией.

```
func addition<T: Numeric>(a: T, b: T) -> T { return a + b }

// Adding two integers
let resultA = addition(a: 42, b: 99)

// Adding two doubles
let resultB = addition(a: 3.1415, b: 1.618)
```

В коде выше в качестве плейсходера T используются Int and Double, и в момент компиляции, компилятор уже знает какие именно типы подставить вместо T во время вызова `addition<T: Numeric>(a: T, b: T)`
То есть компилятор буквально *заменяет* T конкретным типом данных Int или Double.

В отличие от протоколов, opaque types сохраняют типоспецифичность. За протоколом может скрываться множество дочерних типов, за `some,` будет стоять конкретный тип, но мы точно не знаем какой именно, от нас эта информация как бы скрывается, но компилятор в курсе о типе-мплементации.
Можно сказать, что слово `some` является своего рода трюком, чтобы обмануть программиста, который за этим словом не видит технически конкретный тип, а видит как бы объект протокола.

Some полезен при использовании в протоколах с associated types

```
protocol Shape {
    associatedtype Color
    var color:Color { get }
    func describe() -> String
}

struct Square: Shape {
    typealias Color = String
    var color: Color
    func describe() -> String {
        return "I'm a square. My four sides have the same lengths."
    }
}

struct Circle: Shape {
    typealias Color = Int
    var color: Color
    func describe() -> String {
        return "I'm a circle. I look like a perfectly round apple pie."
    }
}

func makeShape() -> Shape {
    return Square(color: "")
// Protocol 'Shape' can only be used as a generic constraint because it has Self or associated
type requirements
}
```

Данная ошибка компилятора вызвана тем, что makeShape() не знает с каким именно associated type'ом требуется вернуть Shape. Можно избежать ошибки используя **some**

```
func makeShape() -> some Shape {
    return Square(color: "")
}
```

Для программиста это выглядит как будто вернулся объект протокола, но компилятор уже в курсе какой именно Shape с каким именно associated type возвращается.
Some может использоваться для

- объявления полей класса и сабскриптов
- объявления возвращаемого значения

Для объявления типов параметров использовать нельзя.

Type erase technique.

Нетривиальная техника в попытке "стереть" assotiated type, и привести объекты к более общему виду. Использует в своей реализации generic и паттерн декоратор. Проще привести листинг кода, чем описывать детали.

```
//Протокол жидкости
public protocol LiquidProtocol {
    var temperature: Float { get set }
    var viscosity: Float { get }
    var color: String { get }
}
//Реализация жидкостей
struct Coffee: LiquidProtocol {
    let viscosity: Float = 3.4
    let color = "black"
    var temperature: Float
}

//Протокол чашки
public protocol CupProtocol {
    associatedtype LiquidType: LiquidProtocol
    var liquid: LiquidType? { get }
    func fill (with liquid: LiquidType)
}
//Создаем generic имплементацию класса чашки
class CeramicCup<L: LiquidProtocol>: CupProtocol {
    var liquid: L?
    func fill(with liquid: L) {
        self.liquid = liquid
        self.liquid!.temperature -= 1
    }
}
class PlasticCup<L: LiquidProtocol>: CupProtocol {
    var liquid: L?
    func fill(with liquid: L) {
        self.liquid = liquid
        self.liquid!.temperature -= 10
    }
}

//Создаем абстрактную generic имплементацию класса чашки
//Будет использоваться для создания обертки над чашкой и реализации
//паттерна декоратора
class AbstractCup<L: LiquidProtocol>: CupProtocol {
    var liquid: L? {
        fatalError("Must implement")
    }
    func fill(with liquid: L) {
        fatalError("Must Implement")
    }
}

//Наследуемся от абстрактной чашки и
//используем паттерн декоратор для создания "обертки" над уже конкретной чашкой
//реализацию которой мы подкинем попозже.
class CupWrapper<C: CupProtocol>: AbstractCup<C.LiquidType> {
    var cup: C
    public init(with cup: C) {
        self.cup = cup
    }
```

```
    override var liquid: C.LiquidType? {
        return self.cup.liquid
    }
    override func fill(with liquid: C.LiquidType) {
        self.cup.fill(with: liquid)
    }
}

//мы получили массив абстрактных чашек с кофе, в которых могут использоваться любые наследники от
протокола CupProtocol
//и в которых как бы "стерт" assotiated type для Liquid.
//на самом деле он конечно есть и это Coffee
var cupsOfCoffee = [AbstractCup<Coffee>]()
cupsOfCoffee.append(CupWrapper(with: CeramicCup<Coffee>()))
cupsOfCoffee.append(CupWrapper(with: PlasticCup<Coffee>()))
//Фактически мы скрываем за враппером конкретные имплементации CupProtocol, но оставляем
//видимым то какой assotiated type используется в этих имплементациях

//Создаем еще одну обертку над абстрактной чашкой, с применением того самого декоратора
//да, вам не показалось, мы используем обертку внутри обертки
//потому что умеем в паттерны ¯\_(ツ)_/¯
final public class AnyCup<L: LiquidProtocol>: CupProtocol {
    private let abstractCup: AbstractCup<L>
    public init<C: CupProtocol>(with cup: C) where C.LiquidType == L {
        abstractCup = CupWrapper(with: cup)
    }
    public func fill(with liquid: L) {
        self.abstractCup.fill(with: liquid)
    }
    public var liquid: L? {
        return self.abstractCup.liquid
    }
}

var coffeeCups = [AnyCup<Coffee>]()
coffeeCups.append(AnyCup<Coffee>(with: CeramicCup<Coffee>()))
coffeeCups.append(AnyCup<Coffee>(with: PlasticCup<Coffee>()))
coffeeCups.forEach { (anyCup) in
    anyCup.fill(with: Coffee(temperature: 60.4))
}

//Проблемная строка, которая не компилируется
var cupsOfCoffee = [CupProtocol<Coffee>]()
//Превращается в строку
var coffeeCups = [AnyCup<Coffee>]()
```

Result type

Новый тип в swift 5.0, фактически это дженерик енам со связанным значением вида *success, failure.*

```
enum MagicError: Error {
  case spellFailure
}

func cast(_ spell: String) -> Result<String, MagicError> {
  switch spell {
  case "flowers":
    return .success("💐")
  case "stars":
    return .success("✨")
  default:
    return .failure(.spellFailure)
  }
}

let apiClient = APIClient()
apiClient.fetch(url: "/todos/1") { response: Result<Todo, Error> in
    switch response {
    case .failure(let error):
        print(error)
    case .success(let res):
        print("\(res.title) -> \(res.completed)")
```

```
    }
}
```

Слабые ссылки.

Сделаны для борьбы с возможными утечками памяти. Слабая ссылка всегда соберется первой, если на объект больше нет сильных ссылок. Если есть два объекта с зацикленными сильными ссылками – это может привести к утечке памяти, в таком случае один из объектов нужно сделать weak.
Weak применяется только к объектам классов, что логично, т.к. структуры передаваемые копированием неспособны порождать утечки зацикленностью.
Weak применяется к неконстантным опционалам, *unowned* в свою очередь может применяться к неопциональным константам, в остальном их поведение похоже.

Слабые ссылки и замыкания

Объекты классов передаются в замыкания по умолчанию по сильной ссылке, это может привести к утечке памяти.

```
1  class Human{
2      var name = "Человек"
3      deinit{
4          print("Объект удален")
5      }
6  }
7  var closure : (() -> ())?
8  if true{
9      var human = Human()
10     closure = {
11         print(human.name)
12     }
13     closure!()
14 }
15 print("Программа завершена")
```

Переменная для замыкания объявлена перед if, в блоке if присваивается замыкание, там же создается объект Human, которые передается в само замыкание. Казалось бы, что после выхода за пределы блока if, и вызова замыкания вызовется deinit для Human. Но нет. Поскольку замыкание присваивается в блоке if, и human объявлен в блоке if, а ссылка на замыкание идет во внешнем к if коде, то получается, что ARC не может сам понять когда можно удалить объект human. Т.к. он передан по сильной ссылке.
Чтобы этого избежать нужно объявить [unowned human] в самом замыкании

```
closure = {[unowned human] in
    print(human.name)
}
```

тогда компилятор поймет, что human передается по слабой ссылке, а это значит что замыкание не зависит от параметра и можно параметр удалить.
Так же будет интересный сайд эффект. Если попробовать позвать замыкание еще раз через closure!() - вывалится ошибка доступа к объекту human. Тк на этом этапе он был уже деаллоцирован. В общем, надо держать в голове какие внешние объекты я использую в замыкании. Более безопасно будет передавать human вообще как параметр, а не захватывать его из внешнего контекста.
Нужно быть осторожным со всеми замыканиями где как-то используется self, с RxSwift например, всегда надо передавать [weak self] иначе при биндинге у VC будут утечки.

```
class Person {
    var name: String
    init(_ name: String) {
        self.name = name
        debugPrint("init person")
    }

    deinit {
        debugPrint("deinit person")
    }

    //Все lazy значения будут проинициализированы только 1 раз, в момент первого вызова. Без lazy не получится использовать self внутри замыкания
    //в greetings1 присваивается замыкание. Будет утечка, в замыкании используется сильная ссылка на self, а self в свою очередь содержит ссылку на greetings1
    lazy var greetings1: () -> String = {
        return "Hello \(self.name)"
    }
    //в greetings2 присваивается замыкание. Не будет утечки, в замыкании используется слабая ссылка на self
```

```
    lazy var greetings2: () -> String = { [weak self] in
        return "Hello \(self?.name)"
    }
    //В greetings3 присваивается замыкание. Не будет утечки, поскольку замыкание получается escaped и в него передается внешний параметр person как аргумент, нет захвата текущего self
    var greetings3: (Person) -> String = { person in
        return "Hello \(person.name)"
    }
    //В greetings4 присваивание РЕЗУЛЬТАТА возврата от замыкания, а не само замыкание
    //Не будет утечки, поскольку в данном случае захват self при первом обращении произойдет по умолчанию по слабой ссылке. Но надо помнить, что это именно возврат от замыкания
    //т.е. изменив потом name, и вызвав greetings4 снова, мы не получим нового результата, поскольку значение в greetings4 уже было присвоено ранее. Замыкание не сработает снова и не реинициализирует переменную
    lazy var greetings4: String = {
        return "Hello \(self.name)"
    }()


    var retainCount: Int {
        return CFGetRetainCount(self) - 1
    }
}
if true {
    let person = Person("John")
    print(person.retainCount)
    var greeting = person.greetings4
    print(greeting)
    person.name = "Garfild"
    print(greeting)

    print(person.retainCount)
} else {

}
```

Extensions

Созданы для добавления нового функционала, но не для переопределения существующего.
Через экстеншены можно делать дефолтную реализацию протоколов.
Можно писать экстеншены для:

- *классов,*
- *структур*
- *перечислений*

Нельзя переопределять методы классов в экстеншенах

```
protocol MyProtocol {
    func anyFunc() -> Bool
}
extension MyProtocol {
    func anyFunc() -> Bool {
        return false
    }
}

class A {

}
extension A: MyProtocol {
//если не написать явную реализацию протокола
//вызов anyFunc()вернет false по дефолту
}

extension A: MyProtocol {
    func anyFunc() -> Bool {
        return true
    }
}
let a: MyProtocol = A()
a.anyFunc() //true

class B: A {
    override func anyFunc() -> Bool {
        return true //Overriding non-@objc declarations from extensions is not supported
надо наследоваться от класса A и anyFunc определять внутри класса A, а не внутри расширения
    }
}
```

Т.е. нельзя оверрайдить то, что имплементируется в экстеншене, оверрайдить можно только если наследование протокола идет в классе.

На самом деле есть один, не очень хороший способ, когда можно оверрайдить в экстеншене через ***@objc dynamic*** атрибут для метода**,** включив тем самым message dispatch(о нем ниже), но это плохая практика и нарушает смысл самих расширений.

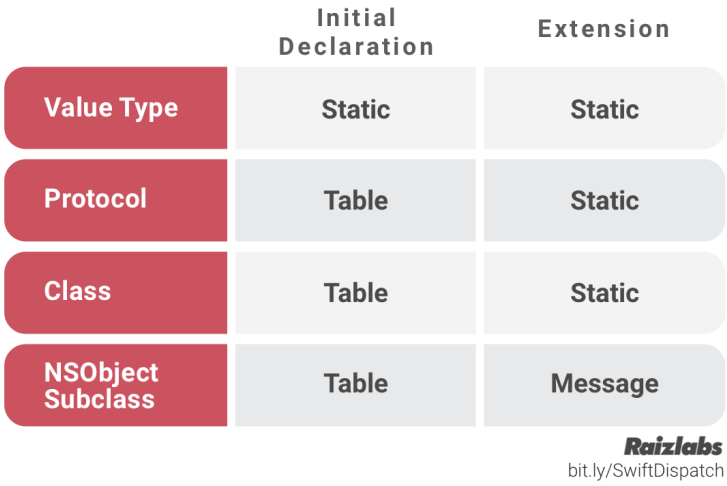
Диспетчеризация

Статья

Бывает *динамическая(virtual), статическая(direct) и динамическая в рантайме(message dispatch).*

- *Статическая(direct/static dispatch)* вызывается на объекте класса, который не имеет родителей. Как бы напрямую. Компилятор уже заранее знает какую инструкцию по какому адресу позвать.
- *Динамическая(virtual/dynamic/table dispatch)* необходима для поддержки наследования. При компиляции создается виртуальная таблица методов *VTable* для обычных классов и *Witness Table* для дженериков, в которой лежат указатели на методы, и их реализации в классах наследниках.
  При наследовании от класса или протокола, сабкласс копирует *v/witness table* базового класса/ протокола. Это значит что изначально в *v/witness* table сабкласса будут лежать адреса на функции-реализации из базового класса. При добавлении своих новых методов, в конец массива *v/witness* table этого класса будут добавляться новые ссылки на адреса с реализацией этих новых функций. При оверрайде родительских методов, наследник заменяет в своей копии *v/witness* table указатель метода родительского класса на предопределенный.
  По дефолту при наследовании объекта от NSObject в swift все его методы получают virtual dispatch, пока не будут применены @objc dynamic модификаторы
- *Динамическая в рантайме(message dispatch)* – наследие Obj-C, используется в KVO, CoreData. При компиляции создается таблица соответствия селекторов и адресов функций, называемая *dispatch table*. Для использования такой диспетчеризации внутри swift требуется использовать *@objc dynamic*.
  По умолчанию обычные классы, наследуемые от NSObject в swift получают именно virtual(not message) dispatch.
  По умолчанию экстеншены, наследуемые от NSObject получают message dispatch.

  Каждый класс, который наследуется от NSObject или NSProxy, имеет поле isa, которое является указателем на объект класса(не на объект экземпляра). Когда вы вызываете метод у какого либо объекта экземпляра, компилятор перепишет ваш вызов в вызов функции objc_msgSend(), передав функции в качестве аргументов объект, на котором вы делаете вызов и строковый селектор с названием вызываемого метода. Далее рантайм переходит по указателю isa на объект класса, и ищет в dispatch table адрес функции, которой соответствует переданный ранее селектор. Если он не находит такой функции, то он переходит на объект родительского класса и ищет эту функцию там. Так происходит до тех пор, пока нужная функция(метод) не будет найдена. Если функция не была найдена нигде, в том числе и в объекте класса NSObject, то выдается всем нам известное исключение: unrecognized selector sent to instance ...
  В качестве оптимизации, для ускорения работы таких вызовов, рантайм создает таблицу с кэшем ранее вызванных селекторов, в котором есть адреса на вызванные функции. Поэтому при повторном вызове данного метода – переход к области памяти с функцией будет осуществлен быстрее, так как вызов уже закэширован и рантайм не будет искать его во всей иерархии классов а вызовет из кэша.



Диспетчеризация в Extensions, @objc dynamic

- Методы описанные через экстеншен суперкласса ***НЕ*** добавляются в *v/witness* table класса, т.к. это статичная диспетчеризация. Поэтому их нельзя оверрайдить в дочерних классах.
- Внутри экстеншена можно оверрайдить методы суперкласса, *только* если метод помечен @objc dynamic, это включит message dispatch. *Но это плохая практика, не делай так.*
- По дефолту *экстеншены* для NSObject объектов получают message dispatch

```
class Person: NSObject {
    @objc dynamic func sayHi() {
        print("Hello")
    }
}
class MisunderstoodPerson: Person {}
extension MisunderstoodPerson {
    override func sayHi() {
        print("No one gets me.")
    }
}
```

Подводные камни Extensions и диспетчеризации

SR-103 bug
Этот баг вызывает дефолтную имплементацию метода, определенную в протоколе и сабклассах.

```
protocol Greetable {
    func sayHi()
}
extension Greetable {
    func sayHi() {
        print("Hello")
    }
}

class Person: Greetable {//gets PWT from Greetable
}
class LoudPerson: Person {//gets VTable from Person
    func sayHi() {
        print("HELLO")
    }
}

let person = Person()
person.sayHi()//Hello

var loud = LoudPerson()
loud.sayHi()// HELLO

var loud2: Greetable = LoudPerson()
loud2.sayHi()// Hello
```

Каждый класс/структура, который конформит протокол получает свою копию PWT(protocol witness table) от протокола, если протоколов несколько, то добавляется новая PWT таблица для других протоколов.
Сабкласс - получает копию *v/witness* table своего родителя, не PWT, PWT от протокола получает только первый класс в иерархии.
Делая *override* метода мы фактически заменяем в v/witness table текущего класса указатель на новую функцию-реализацию метода. Фактически вместо метода базового класса теперь мы в v/witness table потомка положили указатель на реализацию метода из текущего класса. Теперь будем отталкиваться от этого.

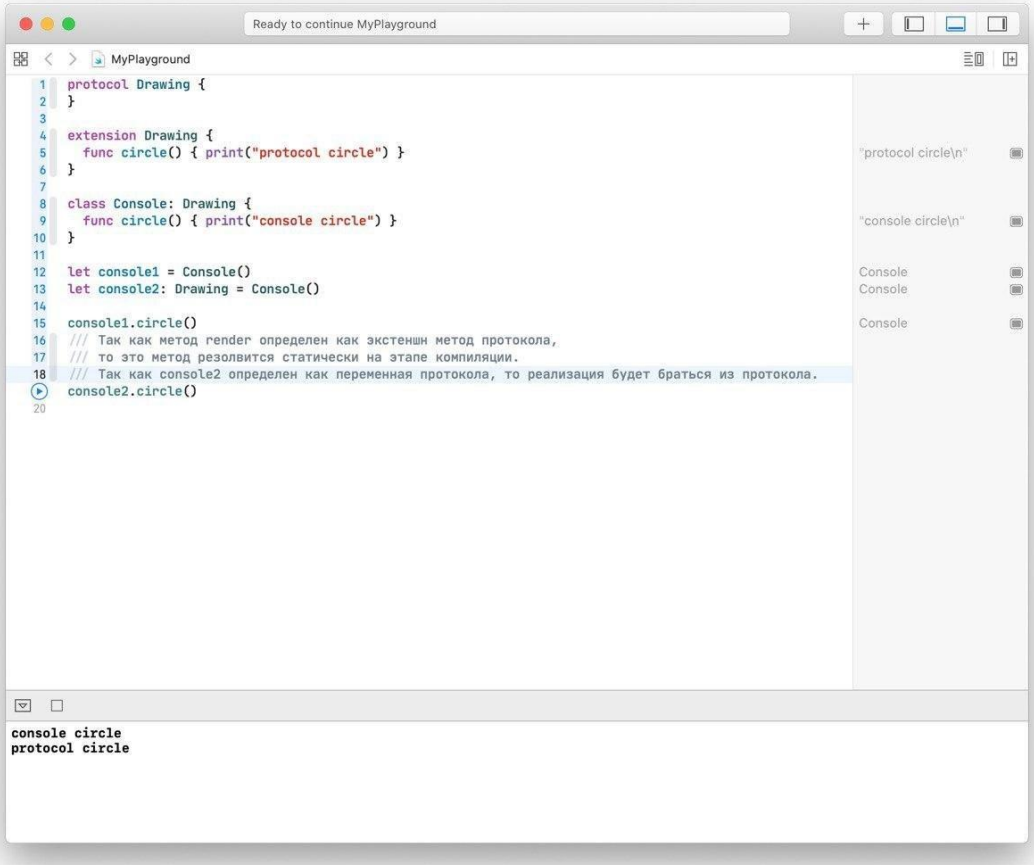
В примере выше Person, получает PWT таблицу соответствия протоколу Greetable. PWT таблицу наследует только первый класс в иерархии, остальные классы наследуют v-table(тк это не дженерик, дженерик наследует witness table суперкласса дженерика) самого первого класса, не PWT. Если бы Person определил внутри себя sayHi(), то он бы переписал в своей PWT таблице указатель с реализации функции экстеншена, на свою реализацию. Но Person этого не делает.
LoudPerson получил копию v-table от Person, а по содержанию указателей это копия PWT от Greetable. Поскольку LoudPerson не сделал override func sayHi() то это значит, что класс не зарегистрировал в своей witness table переопределение функции sayHi(). То есть фактически LoudPerson.sayHi() это НОВАЯ самостоятельная функция, она не используется ВМЕСТО Greetable.sayHi. *Теперь у нас ДВЕ sayHi() функции, одна в Greetable, другая в LoudPerson. И это, конечно, треш, но так работает рантайм.*
Зная эти факты становится понятным неожиданное поведение класса.
Вызывая LoudPerson.sayHi() фактически мы используем direct dispatch к указателю на конкретную область памяти в которой лежат машинные инструкции.
Вызывая LoudPerson.sayHi() через Greetable протокол мы используем dynamic(virtual) dispatch, то есть уже пытаемся позвать метод не напрямую, а через witness table класса LoudPerson. А в таблице лежит указатель на реализацию из родительского Person, который в свою очередь указывает на sayHi() в PWT таблице Greetable протокола и вуаля – мы получаем вызов базового метода, вместо метода в LoudPerson.

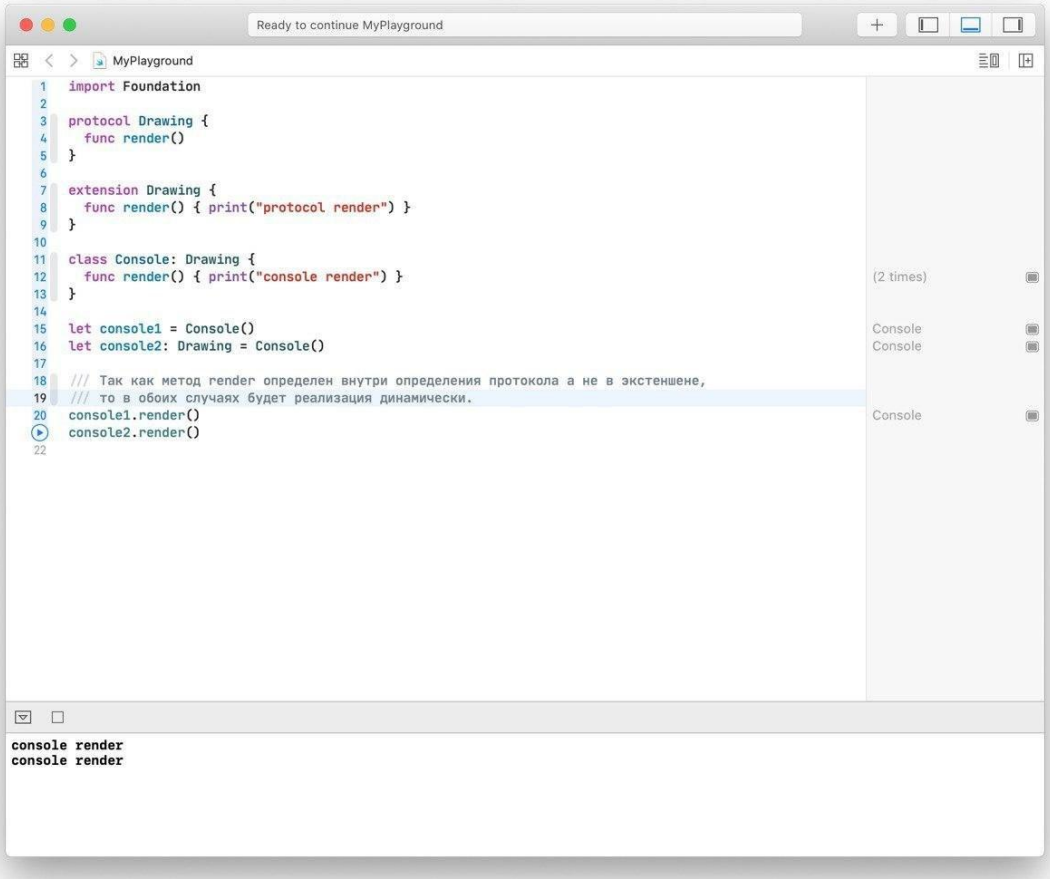
Давайте теперь усложним пример. Сделаем еще один класс

```
class VeryLoudPerson: LoudPerson {
//gets VTable from LoudPerson
    override func sayHi() {
        print("HEEEEELLLOOOOO!!!!")
    }
}

var veryLoud: Greetable = VeryLoudPerson()
veryLoud.sayHi()//Hello

var veryLoud2: LoudPerson = VeryLoudPerson()
veryLoud2.sayHi()//HEEEEELLLOOOOO!!!!
```

Класс VeryLoudPerson получает копию своей witness table от класса LoudPerson. В этой копии есть метод LoudPerson.sayHi(). И это НЕ ПЕРЕОПРЕДЕЛЕНИЕ Greetable.sayHi(). Это совершенно другой метод. В то же время VeryLoudPerson.sayHi() объявлен через override, это значит что внутри VeryLoudPerson.witness_table теперь есть переопределение для метода из LoudPerson.sayHi(). И это НЕ ПЕРЕОПРЕДЕЛЕНИЕ Greetable.sayHi(), я специально это прописал несколько раз.

В общем теперь вызывая VeryLoudPerson.sayHi() через Greetable протокол, все будет как и было раньше, мы стучимся в VeryLoudPerson.witness_table, не находим переопределения Greetable.sayHi() и идем выше по иерархии классов до первой доступной реализации.
Теперь зовем VeryLoudPerson.sayHi() через LoudPerson суперкласс, тоже стучимся в VeryLoudPerson.witness_table, и ищем переопределение для LoudPerson.sayHi() – и находим!, после этого вызываем переопределенную имплементацию для VeryLoudPerson


Ready to continue MyPlayground
MyPlayground
protocol Drawing {
}
extension Drawing {
func circle() { print("protocol circle") }
}
class Console: Drawing {
func circle() { print("console circle") }
}
let console1 = Console()
let console2: Drawing = Console()
console1.circle()
/// Так как метод render определен как экстеншн метод протокола,
/// то это метод резолвится статически на этапе компиляции.
/// Так как console2 определен как переменная протокола, то реализация будет браться из протокола.
console2.circle()
"protocol circle\n"
"console circle\n"
Console
Console
Console
console circle
protocol circle



Экзистенциальные контейнеры и хранение протокольных типов

Статья

Любой класс или структура, которые имплементят протокол – получают PWT таблицу этого протокола. Поскольку, например, структуры которые имплементят протокол имеют разный размер возникает вопрос как их размещать в одном массиве, в котором каждый элемент должен занимать одинаковое количество байт памяти. Для решения этого вопроса был придуман **existential container**

Все объекты/структуры, которые мы приводим к протоколу через *as* – помещаются в такой экзистенциальный контейнер. То есть все поля класса/структуры, вся передача параметров, везде где есть приведение к протоколу – физически упаковывается в такой контейнер и занимает по 40 бит независимо от того структура запакована или класс.

Контейнер занимает 5 машинных слов (в x64 битной системе 5 * 8 = 40 бит) и состоит из нескольких слоев

- *Value buffer* – пространство для хранения экземпляра-имплементации протокола. Если экземпляр может уместиться в буфере содержимого, то он в нем и хранится. Если экземпляр больше *3* машинных слов, то он не поместится в буфере и программа вынуждена выделить память на куче, независимо от того структура там или класс, сложить туда экземпляр, а в буфер содержимого положить указатель на эту память
- *Value witness table* так же, как и PWT, эта таблица есть у каждого типа, который соответствует протоколу. Она содержит в себе реализацию четырех методов, которыми управляется весь жизненный цикл объекта
    - Allocate – принимает решение о том где и как разместить объект, на стеке или хипе
    - Copy – помещает содержимое объекта в выбранную ранее соответствующую область памяти
    - Destruct – убавит до 0 счетчик ссылок, если требуется
    - Deallocate – деаллоцирует выделенную ранее память
- *PWT* таблица содержит в себе указатели на реализации требуемых протоколом методов для типа, к которому эта таблица привязана
- *PWT* таблица для соответствия другим протоколам, для случая если объект использует композицию протоколов и соответствует сразу нескольким.

Схематично контейнер можно представить вот так

```
struct ExistContDrawable {

    var valueBuffer: (Int, Int, Int)

    var vwt: ValueWitnessTable

    var pwt: ProtocolWitnessTable

}
```

Swizzling

Method Swizzling позволяет подменить метод вашим прямо в runtime, притом оставляя оригинальную имплементацию доступной.

Во всех материалах по Objective-C пишут, что подменять методы нужно в load(), а не в initialize(), потому что первый вызывается только один раз для класса, а второй — еще и для всех сабклассов. Но так как runtime при использовании Swift не вызывает load() вообще, приходится убеждаться, что этот initialize() вызван классом, а не кем-то из наследников, и что замена будет выполнена лишь однажды с помощью dispatch_once.

```
//Задача – заменить метод drawDots: фреймворка на метод drawStars:
//Мы желаем рисовать звездочки вместо точек
extension ChartRenderer {
    public override class func initialize() {
        struct Static {
            static var token: dispatch_once_t = 0
        }

        // убедимся, что это не сабкласс
        if self !== UIViewController.self {
            return
        }

        dispatch_once(&Static.token) {
            let originalSelector = Selector("drawDots:")
            let swizzledSelector = Selector("drawStars:")

            let originalMethod = class_getInstanceMethod(self, originalSelector)
            let swizzledMethod = class_getInstanceMethod(self, swizzledSelector)

            let didAddMethod = class_addMethod(self, originalSelector,
method_getImplementation(swizzledMethod), method_getTypeEncoding(swizzledMethod))

            if didAddMethod {
                class_replaceMethod(self, swizzledSelector,
method_getImplementation(originalMethod), method_getTypeEncoding(originalMethod))
            } else {
                method_exchangeImplementations(originalMethod, swizzledMethod)
            }
        }
    }

    //наш метод
    func drawStars(color: CGColor) {
       //рисуем звезды

       //также интересно, что если отсюда вызвать drawStars(...),
       //то отработает старый метод drawDots(...), отозвавшись на селектор "drawStars:"
    }
}
```

Стоит отметить, что существует и альтернативный вариант, более простой, но менее наглядный: вместо extension выполнять все эти операции в AppDelegate в application(_:didFinishLaunchingWithOptions:).

ARC

Automatic reference counting – автоматический подсчет ссылок. Система управления памятью. Всё что она делает — это при сборке приложения анализирует и расставляет retain/release в компилируемый код. Retain/Release в сворю очередь инкрементят/декрементят количество ссылок на объект. Как только мы выходим из скоупа, объекты объявленные в скоупе собираются ARC. Это происходит почти мгновенно. В отличие от классического GC, ARC не умеет ходить по зацикленным ссылкам, отсюда необходимость введения слабых ссылок. Если он натыкается на две сильные ссылки, которые зациклены друг на друге – это приведет к утечке, т.к. он не сможет решить какой объект освободить первым. GC в таких случаях просто соберет оба объекта.
Квалификаторы переменных.

- __unsafe_unretained (по умолчанию) - указывается ссылка, которая не сохраняет указанный объект живым и не устанавливается в nil, когда нет сильных ссылок на объект. Если объект, на который она ссылается, будет освобожден, то указатель остается болтаться со значением, которого уже нет. Значение unsafe_unretained используется по умолчанию, это самый простой вариант - свойству просто присваивается переданное значение.
- __strong - по умолчанию. Объект остается "живым", если на него есть сильный указатель.
- __weak - указывается (слабая) ссылка, которая не сохраняет указанный объект живым. Слабая ссылка устанавливается в nil, когда на объект нет сильных ссылок.
- __autoreleasing - используется для обозначения аргументов, которые передаются по ссылке (id *) и autoreleased по возвращении.

Будьте осторожны с применением __weak слабых ссылок, так, если Вы создадите объект только со слабой ссылкой, то нет никакой гарантии, что он доживет до своего применения и не освободится.

ARC и память под капотом, HeapObject

Статья

Есть три виртуальные области хранения данных:

1. Стек – там хранятся локальные переменные, value types
2. Global data – статичные переменные, константы, метаданные от типах
3. Heap – тут хранятся reference types

Swift runtime представляет каждый reference type объект в виде *HeapObject структуры*. Данная структура содержит всю необходимую рантайму информацию об объекте:

- метаданные типа
- количество ссылок на объект для всех трех типов ссылок:
  - unowned(non-zeroing)
  - weak(zeroing)
  - strong

В момент компиляции SIL(swift intermediate language), swiftc компилятор расставляет вызовы *swift_retain() и swift_release(),* в тех местах, где это требуется. В конечном счете данный процесс завершается перехватом инициализации и деаллокации объектов *HeapObject.*

Слабые ссылки и Side tables

Side table – механизм хранения слабых ссылок (НЕ unowned) в swift runtime. Зачастую, объекты не имеют слабых ссылок, поэтому расточительно резервировать память для счетчика слабых ссылок в каждом объекте. Информация о количестве слабых ссылок хранится в отдельной сторонней таблице: side table, выделение памяти под счетчик слабых ссылок происходит в таблице и *только* тогда, когда это действительно необходимо. В результате слабая ссылка объекта технически указывает на side table этого объекта, в которой хранится счетчик слабых ссылок и указатель на сам объект. Это решает две проблемы:

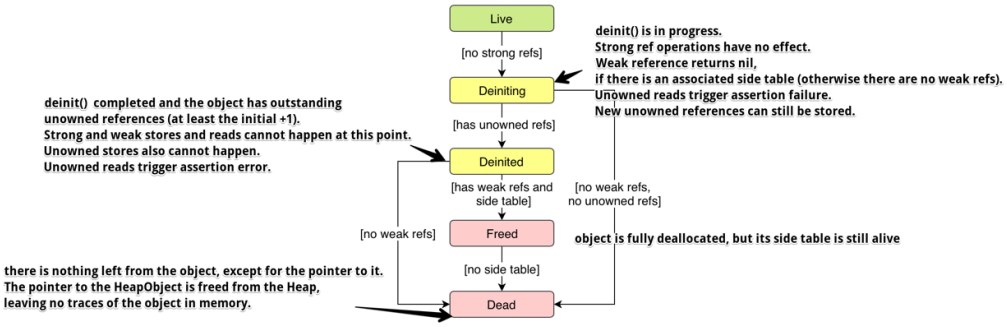
- Сохраняет память, не резервируя ее под счетчик слабых ссылок, пока это не становится необходимым.
- Позволяет безопасно занулять слабую ссылку, поскольку такая слабая ссылка не является техническим указателем на сам объект и это помогает избежать race condition.

Для каждого объекта, создается своя side table в тот самый момент, когда создается первая weak ссылка.
Важно: strong and unowned references ссылки *ссылаются на сам объект*, не на side table

В коде ниже видно, что side table хранит счетчик слабых ссылок на объект и указатель на него

```
HeapObjectSideTableEntry {

    std::atomic<HeapObject*> object;

    SideTableRefCounts refCounts;

    // Operations to increment and decrement reference counts

}
```

Swift object life cycle



Retain/Release

RunLoop и ARC, немного истории

Приведенная ниже информация актуальна для obj-c
В эру до появления ARC был просто RC. Разработчику требовалось вручную вызывать retain() release(), чтобы инкрементить и декрементить счетчик ссылок. Когда ссылок становилось 0, вызывался dealloc().
Чуть позже придумали autorelease pools – это специальная сущность в которой хранились объекты, которые могли быть деаллоцированы в будущем при «осушении» пула, но не прямо сейчас.
В obj-c *autorelease()* вызов добавлял объект в autorelease pool. По умолчанию авторелизпул осушался в конце runloop цикла текущего потока исполнения. Внутри авторелиз пула все объекты получали release() сообщение в момент когда пул осушался. Один и тот же объект можно было положить несколько раз в один пул, и он получал ровно столько же сообщений release(), сколько раз позвали autorelease(). Если вы хотите использовать добавленный в авторелиз пул объект вне этого пула, то необходимо было позвать явно retain().
Фактически теперь вместо вызовов retain() вызывали autorelease().
Application kit создает глобальный autorelease pool в main потоке приложения в начале каждой итерации run loop(надстройка над потоком, цикл обработки событий) и делает drain пула в конце итерации, посылая release всем объектам из пула, тем самым освобождая все autorelease объекты, созданные во время итерации цикла.

На самом деле autorelease blocks создают стек, новые объекты обычно ложатся в верхний пул из стека. В обычной жизни swift разработчика нам нет нужды создавать свои autorelease blocks, однако мы можем их создавать для **NSObject объектов(не чистых swift).** Делать локальный autorelease block для еще большей эффективности потребления памяти. Например, мы можем создавать свой локальный autorelease block для каких-нибудь циклов обработки данных, чтобы объекты мгновенно очищались после каждой итерации нашего цикла, не дожидаясь завершения общей итерации run loop. Однако, данный подход в общем случае не рекомендуется, т.к. частое создание и очистка пулов снижают производительность, поэтому лучше все оставить автоматике.

Важно:

- В современных swift проектах UIApplicationMain держит autorelease pool приложения, и мы недолжны использовать autorelease pool самостоятельно. По большей части этот пул висит для возможных NSObject объектов.
- Для чистых swift объектов авторелизпул не используется, поскольку бесполезен, т.к. компилятор уже расставил *swift_retain() и swift_release()* вызовы в нужных местах, это значит что *pure swift* объекты в пул не добавляются, и даже если их добавить вручную, ничего не изменится, тк release будет вызван при выходе объекта из скоупа видимости, то есть - не зависимо от того драинит система пул в конце итерации ранлупа или нет.

RunLoop и Таймеры

RunLoop – это своеобразная надстройка над потоком, цикл обработки событий, который планирует работу и управляет получением входящих событий. Runloop цикл держит поток занятым, когда есть работа в потоке, т.е события для обработки, и переводит поток в спящий режим, когда работы нет. Обычно нам не нужно создавать объекты runloop вручную, т.к. каждый созданный поток уже имеет свой связанный с ним объект runloop. Только при использовании вторичных потоков мы должны явно запустить их runloop.
У main thread тоже есть свой runloop, который создается автоматически, как часть стартап процесса приложения.

Runloop получает события из двух типов источников:

- Источник ввода(input source) – доставляет асинхронные события, обычно сообщения от других потоков или из других приложений
- Источник таймеров - доставляет события от таймеров, зарегистрированных на текущем runloop.

При создании таймера, он регистрируется в ранлупе потока, например потока main. В момент если в потоке мейна есть какие-то события на обработку, например скроллинг, то обработчик таймера вызываться не будет, т.к. поток мейна уже занят другой работой. Обработчик таймера вызовется потом, и уже со значением таймера, которое значительно меньше предполагаемого. Это тот случай, если у нас например таблица, а сверху лейбл с таймером, если скролить таблицу, то лейбл перестанет «тикать». Связано с загрузкой мейн потока обработкой интеракшенов с таблицы.
Лечится это установкой режима работы runloop mode в .common.

Режимы работы runloop

- Default – обрабатывает события от источников, которые НЕ NSConnectionObjects
- Common – режим в котором так же обрабатываются события от таймеров и наблюдателей
- Tracking – обрабатывает отзывчивый интерфейс приложени(UI). Например, отслеживание перемещений мышки

Взаимодействие между процессами iOS

Статья

В iOS возможно взаимодействие между процессами, оно построено на XPC (Cross process communication), как одной из реализаций IPC (inter process communication).
XPC - доступ к низкоуровневому (libSystem) механизму межпроцессного взаимодействия, основанному на сериализованных plist. *Приватен в iOS,* мы не имеем доступу к этому API.

MachPorts – один из ключевых механизмов IPC в ядре Mach, это нативный коммуникационный канал для общения между процессами.
Представлены в CoreFoundation

- CFMachPort - базовая обертка над Mach port (mach_port_t) ядра
- CFMessagePort – предоставляет коммуникационный канал для общения между процессами или тредами, более высокоуровневая обертка чем .CFMachPort.
- NSPort/NSMachPort/NSMessagePort – другие обертки над MachPort с различным функционалом

Статья
В 8 iOS появились AppExtensions – небольшие приложения, которые могли поставляться вместе с главным устанавливаемым приложением. Например приложение для watch. Для возможности шарить данные и ресурсы между экстеншеном и главным приложением ввели *Application Groups Identifier*, который прописывался в плисте каждого приложения. Теперь все экстеншеновые приложения могут работать со своим собственным сэндбоксом, но в то же время имеют доступ к сэндбоксу, директории и ресурсам главного приложения, например могут писать данные в общую sqllite базу.

BerkelySockets – способ взаимодействия между процессами посредством сокетов. Поскольку Darwin основан на ядре BSD, он унаследовал этот механизм. Мы можем создавать локальные сокеты для сабпроцессов внутри сэндбокса главного процесса(приложения) и настраивать их для обмена данными между сабпроцессами.

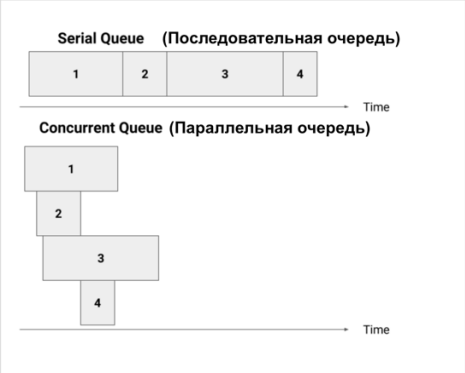
Также мы можем применять сокеты для общения между совершенно независимыми приложениями в ios. Проблема будет скорее в том, что в при сворачивании в бэкграунд один из процессов будет уходить в suspended состояние и сетевое взаимодействие станет в этом случае невозможно, придется изобретать что-то дополнительное. Но чисто технически такой способ взаимодействие возможен.

GCD

Grand central dispatch – высокоуровневый фреймворк, который предоставляет доступ к FIFO очередям исполнения, куда можно отправлять задачи в виде замыканий. Замыкание переданное в диспетчер будет обработано каким-то потоком из пула потоков, нет никаких гарантий каким именно потоком оно будет обработано. При работе с диспетчером мы имеем доступ только к очередям, а не к потокам.
Очереди - это не техническое, а абстрактное понятие, на самом деле это очередь представляет собой FIFO очередь из задач, которые обрабатываются ОС. Мы можем положить в определенную очередь наш собственный кусок кода (в виде замыкания), и он будет лежать в этой очереди пока не обработается. Надо помнить, что кроме наших замыканий в очереди могут быть и другие задачи.

Виды очередей

1. Serial – последовательные. Задачи в очереди выполняются одна, за другой, в порядке их входа в очередь.
2. Concurrent – параллельные. Задачи выполняются параллельно.



Как только очередь создана, в нее можно положить задание через две функции

1. Sync - синхронная функция sync возвращает управление на **текущую(!)** очередь только после полного завершения задания, тем самым блокируя текущую очередь.

2. async функция, в противоположность функции sync, возвращает управление на **текущую**(!) очередь немедленно после запуска задания на выполнение в другой очереди, не ожидая его завершения. Таким образом, асинхронная функция async не блокирует выполнение заданий на текущей очереди.

Имеется 5 основных типов качества обслуживания для очередей (quality of service). Качество обслуживаниях по сути расставляет приоретизацию очередей. Чем выше qos очереди, тем быстрее задачи из нее будут отправлены на обработку в cpu.

1. Main queue - последовательная глобальная очередь, в которой происходят все операции с пользовательским интерфейсом. Все манипуляции с интерфейсом, любыми объектами которые наследуют UIResponder, ОБЯЗАТЕЛЬНО делать только в этой очереди. Данная очередь имеет наивысший приоритет.

Очередь можно создать через DispatchQueue.global(qos: .userInteractive)

Ниже представлены различные qos и объясняется, для чего они предназначены:

2. .userInteractive — для заданий, которые взаимодействуют с пользователем в данный момент и занимают очень мало времени: анимация, выполняются мгновенно; пользователь не хочет этого делать на Main queue, однако это должно быть сделано по возможности быстро, так как пользователь взаимодействует со мной прямо сейчас. Можно представить ситуацию, когда пользователь водит пальцем по экрану, а вам необходимо просчитать что-то, связанное с интенсивной обработкой изображения, и вы размещаете расчет в этой очереди. Пользователь продолжает водить пальцем по экрану, он не сразу видит результат, результат немного отстает от положения пальца на экране, так как расчеты требуют некоторого времени, но по крайней мере Main queue все еще “слушает” наши пальцы и реагирует на них. Эта очередь имеет очень высокий приоритет, но ниже, чем у Main queue.
3. .userInitiated — для заданий, которые инициируются пользователем и требуют обратной связи, но это не внутри интерактивного события, пользователь ждет обратной связи, чтобы продолжить взаимодействие; может занять несколько секунд; имеет высокий приоритет, но ниже, чем у предыдущей очереди,
4. .utulity — для заданий, которые требуют некоторого времени для выполнения и не требуют немедленной обратной связи, например, загрузка данных или очистка некоторой базы данных. Делается что-то, о чем пользователь не просит, но это необходимо для данного приложения. Задание может занять от несколько секунд до нескольких минут; приоритет ниже, чем у предыдущей очереди,
5. .background — для заданий, не связанных с визуализацией и не критичных ко времени исполнения; например, backups или синхронизация с web — сервисом. Это то, что обычно запускается в фоновом режиме, происходит только тогда, когда никто не хочет никакого обслуживания. Просто фоновая задача, которая занимает значительное время от минут до часов; имеет наиболее низкий приоритет среди всех глобальных очередей.
6. Есть еще глобальная параллельная (concurrency) очередь по умолчанию .default, которая сообщает об отсутствие информации о «качестве обслуживания» qos. Она создается с помощью оператора: DispatchQueue.global()
   Если удается определить qos информацию из других источников, то используется она, если нет, то используется qos между .userInitiated и .utility.

Так же можно создавать пользовательские очереди. Label лучше указывать уникальной строкой инверсной DNS нотации. Данная очередь по умолчанию последовательная

```
//  Последовательная очередь  mySerialQueue
let mySerialQueue = DispatchQueue(label: "com.bestkora.mySerial")
```

А вот данная очередь уже параллельна

```
let workerQueue = DispatchQueue(label: "com.bestkora.worker_concurrent",
    qos: .userInitiated, attributes: .concurrent)
```

Пример. Создаем глобальную конкурентную очередь с высоким приоритетом

```
1 DispatchQueue.global(qos: .userInteractive).async {
        2 sleep(2)
        3 DispatchQueue.main.async {
           sleep(2)
           self.textLabel.text = “2”
         }
        4 let someInt = 18
}
```

1. Допустим мы вызвали код выше по нажатия на кнопку, значит в данный момент мы в очереди UIThread так сказать по дефолту. Мы запускаем DispatchQueue.global(qos: .userInteractive).async, это значит, что все замыкание которые мы передаём *не будет* блокировать uithread так как вызывается через async, сразу после добавления в очередь, возвращая управления uithread.
2. На другой очереди поток поспит 2 секунды, не блокируя юи.
3. Затем произойдет переключение на очередь uihtread через DispatchQueue.main. И мы заставим уже поток ui уснуть на 2 секунды.
4. Не ожидания когда «разморозится» ui thread идет присваивание let someInt = 18 на очереди из global. Присваивание someInt произойдет до того, как «разморозится» uithread, т.к мы использовали DispatchQueue.main.async, это значит, что задача для main будет выполнена асинхронно к текущей (global) очереди.

ВАЖНО. Самый простой способ получить deadlock – это вызвать DispatchQueue.main.sync {}, поскольку main – это serial очередь, то вызов блокирующей операции изнутри самой этой очередь приведет к взаимной блокировке

Создание барьеров средствами GCD

Если у нас есть задача выполнить какой либо блок кода, посреди параллельной очереди, то GCD предоставляет флаг, который работает как барьер.

```
let queue = DispatchQueue(
  label: "com.kentakodashima.app.sampleBarrier",
  attributes: .concurrent
)
//Можно добавить DispatchWorkItems которые должны выполнится до конца до барьера.
queue.async(flags: .barrier) {
  print("срабатывает барьер")
}
//Можно добавить задачи, которые выполняются ПОСЛЕ барьера
```

Таким образом барьер действует как собственно барьер, разделяя две группы параллельных задач в очереди.

DispatchGroup

Группы позволяют объединять набор задач и синхронизировать их поведение в группе. Вы добавляете несколько задач к группе и планируете их асинхронное выполнение в одной и той же очереди или в разных очередях. Когда все таски завершают выполнение, группа выполняет свой комплишн колбек *notify(: _),* тем самым уведомляя что все задачи в ней завершились. Вы также можете синхронно ожидать завершения всех задач в группе.
Добавлять задачи можно либо через
group.enter()
//твой код

либо через GCD
DispatchQueue.global(:).async(group: gr) {
//твой код
}
Группу можно ожидать через *wait()* на текущем потоке, данная функция морозит текущий поток до завершения задач в группе. Но это чревато блокировкой, лучше использовать *notify()*

Группу *нельзя прервать* в процессе работы. *Нельзя остановить*. Можно использовать *DispatchSemaphore*(обертка над обычным *Semaphore*) для синхронизации тасок в группе

OperationQueue

Более высокоуровневый способ группировки заданий для асинхронного выполнения, чем DispatchGroup
Используется для создания очереди из операций Operation. В отличие от DispatchGroup операции можно

- Канцеллить все операции (cancelAllOperations)
- Приостанавливать саму очередь (isSuspended)
- Ждать выполнения всех операций (waitUntilAllOperationsAreFinished), блокируя **текущий** поток исполнения
- Добавлять «барьеры»

Operation по сути может быть обычным замыканием вида () -> Void, а может быть более сложным объектом, наследником от Operation. Есть дефолтный `BlockOperation` например.
Операции добавляются через group.add()

Барьеры - это блоки кода, которые выполняются между постановкой на выполнение других задач. Как бы разделители между задачами. Поставили, например 5 задач в очередь, потом написали барьер и добавили

еще 5 задач. Вторые 5, не выполнятся пока не сработает код в барьере.

В классе ***Operation*** есть следующие основные методы:

- main()// делает основной код операции
- start()
- cancel()

ВАЖНО: при приостановке очереди, незавершенные операции могут привести к утечке памяти

Отличие обычных тредов(Thread/NSThread) и тредов из GCD

Исходя по обрывкам неподверженных документацией данных.
Отличий почти нет, кроме того, что треды из GCD создаются самим фреймворком, который создает под капотом некий тредпул. GCD оптимизирует создание потоков исходя из текущей загрузки процессора, и управляет ими исходя из QoS(quality of service), таким образом, *лучше* использовать GCD вместо ручного создания тредов.

Проблемы асинхронности

Есть несколько основных проблем асинхронного кода в ООП программировании:

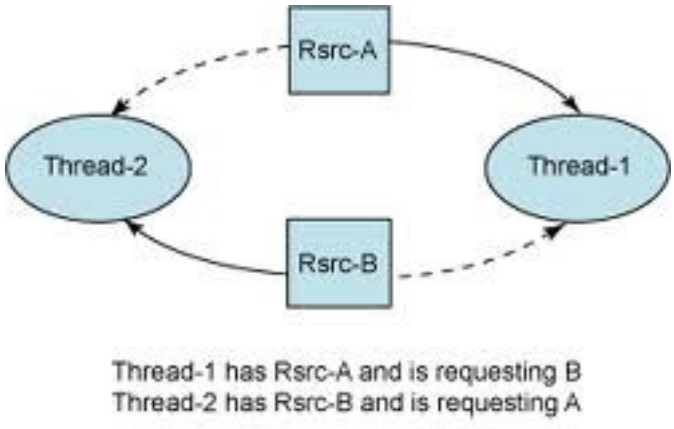
- **Гонка (race condition)** – состояние асинхронной системы, при котором доступ к одному общему ресурсу, пытаются получить несколько потоков одновременно. В результате ресурс захватит тот поток, который быстрее других «постучится» к этому ресурсу.
  Т.е мы заранее не сможем предсказать какая из асинхронных задач получит доступ к ресурсу (бд на запись например) первой. Это в свою очередь может порождать непредсказуемое состояние всей системы.

- **Инверсия приоритетов (priority inversion)** – состояние асинхронной системы, когда высокоприоритетная задача зависит, либо является результатом исполнения, низкоприоритетной задачи.
  Допустим есть два потока, которые обращаются к одному и тому же неделимому ресурсу, пусть будет H (высокоприоритетный) и L (низкоприоритетный). Если общий ресурс в данный момент времени занят потоком L, то поток H вынужден ждать, пока L завершит работу и освободит ресурс. То есть не смотря на пиритизацию, L завершит свою задачу быстрее, хотя это неверное поведение системы.

  GCD умеет иногда разруливать такие ситуации повышая приоритет низкоприоритетной задачи до приоритета ожидающей в очереди(высокоприоритетной) задачи.

- **Взаимоблокировка (Dead lock) -** состояние асинхронной системы, когда два и более потока держат свои различные ресурсы(тоже два например), и не могут их отпустить, т.к каждый из них ожидает возврата ресурса от другого.

  Решил, что проще показать картинкой:



  Есть два разных потока 1 и 2. Есть два разных ресурса A и B.
  Потоку 1 для работы требуется ресурс A, и B, захватывает первым A.
  Потоку 2 для работы требуется ресурс A и B, захватывает первым B.

Поток 1 ждет когда поток 2 отпустит ресурс B.
Поток 2 ждет когда поток 1 отпустит ресурс A.
И вот так они ждут друг дружку вечность, пока мы не вмешаемся.

Семафоры и мьютексы, синхронизация потоков

Для решения описанных выше проблем асинхронности используют средства синхронизации потоков.

```
let semaphore = DispatchSemaphore(value: 1) //количество потоков, которые имеют доступ к ресурсу
semaphore.wait() //лочим доступ к ресурсу
//работаем с ресурсом
semaphore.signal() // говорим семафору что ресурс освобожден
```

```
let higherPriority =
DispatchQueue.global(qos: .userInitiated)
let lowerPriority =
DispatchQueue.global(qos: .utility)



let semaphore = DispatchSemaphore(value: 1)



func asyncPrint(queue: DispatchQueue, symbol: String)
{
  queue.async {
    print("\(symbol) waiting")
    semaphore.wait()  // requesting the resource

    for i in 0...10 {
      print(symbol, i)
    }

    print("\(symbol) signal")
    semaphore.signal() // releasing the resource
  }
}



asyncPrint(queue: higherPriority, symbol: "🔴")
asyncPrint(queue: lowerPriority, symbol: "🔵")
```

Не смог найти в swift *мьютексов* как отдельного класса, поэтому в качестве его аналога используют просто последовательные очереди.
Вместо семафора можно юзать NSLock, но в отличие от семафора в NSLock нельзя указать сколько потоков он может лочить, то есть данный класс рассчитан только на 1 поток.

```
let serialQueue = DispatchQueue(label: "com.test.mySerialQueue")
```

```
serialQueue.sync {
    // code
}
```

iOS, UIKit

UIApplication life cycle

расположены в порядке вызовов

1. *Foreground*, когда на экране
2. *Inactive* – промежуточное состояние при переходе из foreground в background и наоборот. Еще данное состояние включается в момент если приложение было foreground и поступил входящий вызов, или приход пуша.
3. *Background*, когда приложение свернуто, но исполняется какой то код, например backgroundFetch
4. *Suspended*, когда приложение свернуто и код не исполняется
5. *Terminated*, not running, когда убито системой либо юзером, либо не запущено.

Методы AppDelegate в порядке их вызова:

1. didFinishLaunchingWithOptions() сразу после запуска
2. applicationDidBecomeActive ()– срабатывает уже после того, как приложение перешло в активное состояние

Нажимаем Home

1. applicationWillResignActive() – при переходе из активного в неактивное состояние
2. applicationDidEnterBackground() после сворачивания приложения, при уходе в бэкграунд. Тут можно очищать ресурсы

Разворачиваем, свернутое приложение

1. applicationWillEnterForeground() – срабатывает незадолго до перехода из бэкграунда в активное состояние
2. applicationDidBecomeActive ()

При попытке убить приложение из списка запущенных приложений

1. applicationWillTerminate() срабатывает перед убийством процесса, рекомендуется сохранять данные

VC life cycle

- LoadView - called when you try to call view first time. This method creates view for current VC. You can override this method and setup your custom view
- ViewDidLoad - called after the controller's view is loaded into memory. Great for initial setup and one-time-only work. Constraints here may are not initialized for current device screen.
  IMPORTANT(!)
  This method can be called twice in case if the system was unload view from memory(in case of didRecievedMemoryWarning). The system unload view automatically. In this case if user returns to this screen loadView() and viewDidLoad() will be called again.
  IMPORTANT(!!)
  You can face with incorrect frames and sizes in this method, because autloayout not do their work yet. It's not best practice, but you can use layoutIfNeeded() or setNeededLayout () to avoid this situation.
- ViewWillAppear - called right before your view appears, good for hiding/showing fields or any operations that you want to happen every time before the view is visible. Because you might be going back and forth between views, this will be called every time your view is about to appear on the screen.
- ViewDidAppear - called after the view appears - great place to start an animations or the loading of external data from an API.
- ViewWillDisappear/DidDisappear - same idea as ViewWillAppear/ViewDidAppear.
- ViewWillLayoutSubviews - when a view's bounds change, the view adjusts the position of its subviews. Your view controller can override this method to make changes before the view lays out its subviews. This method can be called multiple times.

UIWindow

UIWindow, наследует UIVIew не отображает контент само по себе, но является контейнером для вложенных Views

Часто в очень простых приложениях бывает только одно UIWindow. Однако, в приложении может быть и несколько UIWindow с вложенными навигейшен-контроллерами. Нужно просто создать UIWindow и

поставить makeKeyAndVisible().
UIWindow имеет уровни, которые регулируют позицию окна по оси Z.

Можно использовать дополнительные несколько UIWindow для решения такой задачи, как например, с вводом пароля юзера, если требуется авторизация в приложении. Т.е можно создать блокировочное окно для ввода пароля, сделать ему makeKeyAndVisible() и отобразить поверх всего контента приложения. Это удобно тем, что, что не надо думать о том, поверх какого контента ты отображаешь окно. Оно идет просто поверх текущего окна с любым контентом.

UIScreen

Обычно один на приложение, но может быть несколько, если апп шарит видео на телевизор например. Описывает характеристики физического экрана и его размеры.

Frame Bounds

Frame описывает размеры view относительно родительского контейнера. Bounds относительно внутренней системы координат самой view.
Принципиальное отличие между Frame and Bounds, с точки зрения практики, еще в том, что после срабатывания анимации с трансформированием view, координаты фрейма могут быть невалидными, в то время как Bounds остается неизменным.

Как увеличить зону тапа по кнопке.

Переопределить pointInside()
Если кнопка с картинкой, можно задать прост инсеты для картинки и увеличить фрейм кнопки.

Отличие UIView от CALayer:

UIView является высокоуровневой оберткой над CALayer. UIView содержит CAlayer, но в то же время UIView может работать с responding chain и поддерживает распознавание тачей и свайпов.

Segues

Объекты для перехода между VC у любого UIViewController есть метод prepareForSegue, который принимает сегу и позволяет из нее получить destionationVC и что-то с ним сделать

Layout methods

- *setNeedsDisplay(: rect)* - – регистрирует в main run loop «заявку» на то, чтобы rectangle and bounds данного вью были перерисованы во время следующего цикла перерисовки элементов. То есть ***не мгновенно***. Методу можно передать нужного размера прямоугольник, либо будет рассчитан дефолтный. Метод работает *асинхронно*.

- *setNeedsLayout()* – регистрирует в main run loop «заявку» на то, чтобы данное view и все subviews было перерисовано во время следующего цикла перерисовки элементов. Это ***асинхронный*** метод, который ***не вызывается мгновенно***. В дальнейшем будет вызван *layoutSubviews()*

- *layoutIfNeeded()* – вызывает ***мгновенную*** перерисовку view, без ожидания в очереди. Метод проверяет были ли изменения в позиционировании элемента или autoLauyout, и если они были вызовет *layoutSubviews()*.

- *layoutSubviews()* – реализация метода использует все доступные констреинты для определения размеров и позиционирования вложенных subviews. Обычно нет нужды вызывать этот метод напрямую, лучше воспользоваться *setNeedsLayout() или setNeedsLayout()*. Можно переопределить этот метод, если стандартное поведение autoLayout не подходит.

UILayoutGuides

UILayoutGuides: NSObject – определяет прямоугольную область внутри координатной системы родительской view, которая способна взаимодействовать с AutoLayout. В отличие от обычного UIView, UILaoutGuide не отрисовывается на экране, а значит не тратит ресурсное время процессора. Можно задать констреинты для UILayoutGuide. Нельзя добавлять сабвью, тк UILayoutGuide не является объектом view. Вообще, данная концепция была введена с 9 оси, до того как были добавлены StackViews, и используется, например в следующих случаях:

- У нас есть три кнопки, и надо сделать так, чтобы расстояния между ними были одинаковыми на разных устройствах. Кроме программного расчёта констреинтов исходя из ширины экрана, раньше использовали, разделительные прозрачные UIView, констреинт ширины которых, определенным

образом соотносился с шириной экрана, таким образом позволяя разделять кнопки.
Теперь для этой задачи можно использовать UILayoutGuide, делая отступы кнопок от него.

Констреинты и их резолв

Эпл понимает концепцию констреинтов, в общем виде, как систему линейных уравнений.
Выглядит примерно так:

```
// Setting a constant height
View.height = 0.0 * NotAnAttribute + 40.0
// Setting a fixed distance between two buttons
Button_2.leading = 1.0 * Button_1.trailing + 8.0
```

Важно отметить, что механизм AutoLayout не просто высчитывает решение исходя из того, чтобы левая сторона уравнения численно равнялась правой, а сам рассчитывает конкретные численные значения для `Button_2.leading и Button_1.trailing` таким образом, чтобы уравнение имело решение.

Это значит, что следующая конструкция тоже будет валидной

```
Button_1.trailing = 1.0 * Button_2.leading - 8.0
```

Иногда система представляет собой наоборот неравенство, для случаев когда констреинт может иметь решение в определенном диапазоне значений, например:

```
// Setting the minimum width
View.width >= 0.0 * NotAnAttribute + 40.0
Setting the maximum width
View.width <= 0.0 * NotAnAttribute + 280.0
```

Если констреинты противоречат друг другу, так называемый conflicting, UIKit в рантайме пытается разрешить эти коллизии и удалить тот констреинт, который имеет более низкий приоритет. Если приоритеты одинаковые, он удалит рандомный.
Если констреинтов не хватает ambiguous, UIKit добавит недостающие констреинты сам.

В основном для обычного UIVIew достаточно 4 констреинтов(x, y, height, width) для установки позиции. Но есть некоторые UI элементы, которые могут потребовать больше или меньше констреинтов.
Например UILabel будет требовать только x,y потому что у него есть IntristicContentSize и он сам определяет свой внутренний размер.
А UIScrollView потребует дополнительные констреинты позиционирования, если внутри него выставлены свои вьюхи со своими внутренними констреинтами.

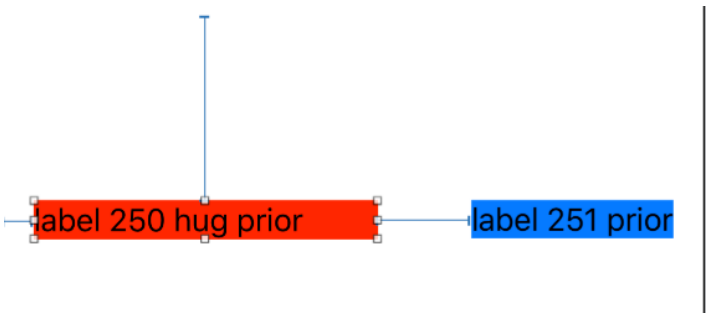
Приоритеты констреинтов

У констреинтов можно выставить приоритеты от 0 до 1000. 1000 – значит что констреинт должен применяться без всяких оговорок. Посредством приоритетов, UIKit может понимать какие констреинты обязательны к применению, а какие опциональны, или можно применить в зависимости от каких то специфических условий.

Intrinsic content size, hugging, resistance priority.

IntristicContentSize - ожидаемый размер контрола в зависимости от контента. Характерен для контролов с динамическим контентом. Лейблы, текстовые поля и т.д.

Content hugging priority – устанавливает взаимоотношения между текущим контролом, лейблом например и другим лейблом. Более высокое значение, говорит о том, что мы не хотим чтобы контрол растягивался больше чем его intrinsic size. Контрол с большим приоритетом будет иметь размер, как его intrinsic size, в свою очередь растягивая соседа.



Content compressions resistance priority – более высокое значение говорит о том, что мы не хотим чтобы контрол уменьшался меньше, чем его intrinsic размер

На пальцах, это значит, что чем выше приоритет compressions resistance, тем размер контента данного лейбла приоритетнее, перед размером контента соседнего. Мы можем увеличивать размер контента синего лейбла, до тех пор, пока красный не займет например 1 символ. В то же время красный лейбел может занять только то свободное место, которое еще не занято синим.

UIKit объекты и потоки.

Работая с объектами, наследниками **UIResponder** – это объекты UIKit(UIWindow, UIView и тд) мы должны осуществлять к ним доступ *только* через main thread. Это сделано для того, чтобы не создавать race condition на объектах интерфейса, и не блокировать его Более того, мы не можем даже создать локальный объект UIKit в бэкграунд потоке, вызов init() вызовет ошибку MainThreadChecker'a. Мы не можем ни создать объект UIKit, никаким либо другим образом провзаимодействовать с ним из не мейн потока.
MainThreadChecker - это специальная системна тула iOS, которая проверяет, невалидное использование объектов UIKit, AppKit и некоторых других объектов из бэкграундного потока.
То есть мы можем, например, создать объект UIImage: NSObject на бэкграунд потоке и отрендерить его, а потом положить его в UIImageView: UIResponder на мейне, главное осуществить доступ к UIImageView через main thread. Связано это с наследованием, UIImage: NSObject – можно работать в бэкграунде, UIImageView: UIView: UIResponder – нельзя

Работа с таблицами.

```
tableView.setEditing(true, animated: true) // переход в edit mode

func tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath) // говорит можно ли изменять какую либо ячейку

func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCell.EditingStyle, forRowAt indexPath: IndexPath) // надо оверрайдить для того чтобы стал доступен slide to delete. Вызывать tableView.setEditing при этом не обязательно
 срабатывает после выбора действия (удаления например) тут можно отследить стиль и индекс ячейки и что-то сделать с ней

tableView(_ tableView: UITableView, editActionsForRowAt indexPath: IndexPath) -> [UITableViewRowAction]? // возврат действий для изменения. Действие удаления присваивается автоматически. Если нужно только удаление, можно вернуть nil по умолчанию

override func tableView(_ tableView: UITableView, moveRowAt sourceIndexPath: IndexPath, to destinationIndexPath: IndexPath) // для включения режима перетаскивания ячеек

tableView.allowsMultipleSelectionDuringEditing = true // для выбора многих ячеек в edit mode
```

Следует помнить, что если включен allowsMultipleSelectionDuringEditing, то знак – (удаление слева ячейки) по дефолту будет отключен, вместо него будет чекбокс с галочкой. Если setEditing и allowsMultipleSelectionDuringEditing идут вместе сразу после нажатия на кнопку edit, то надо перезагрузить таблицу.

tableView.allowsSelectionDuringEditing// для выделения одной ячейки во время изменения, по умолчанию false. Чревато тем, что во время изменения можно случайно вызвать экшен выделения ячейки и куда то перейти

Обтекание текста

UIBezierPath * imgRect = [UIBezierPath bezierPathWithRect:CGRectMake(0, 0, 100, 100)];
self.textView.textContainer.exclusionPaths = @[imgRect];

Responder chain

Реализует паттерн цепочка обязанностей.
hitTest() -> point(inside:) -> touchesBegan() -> next() -> touchesEnded()

Как заменить дефолтный AppDelegate своим и наследовать свой UIApplication

Создаётся кастомный application класс – например `RCApplication: UIApplication`
Создается кастомный appdelegate класс `RCAppDelegate: UIResponder, UIApplicationDelegate`
Создается файл main.swift в проекте туда вставляем следующее

```
UIApplicationMain(
    CommandLine.argc,
    CommandLine.unsafeArgv,
    NSStringFromClass(RCApplication.self),
    NSStringFromClass(RCAppDelegate.self)
)
```

Static and dynamic frameworks and libraries

Подробнее тут
Library – библиотека – это кода, который не является непосредственным кодом текущего приложения. То есть, это некий независимый внешний код, который мы можем использовать.
Linking – процесс мержа кода библиотек и кода проекта.

- *Static library* *.a, фактически это архив с объектными файлами в виде машинного кода. Статичная библиотека копируется в исполняемый файл приложения, образуя единый монолитный файл.
- *Dynamic Library* *.dylib В отличие от статичной не копируется в файл приложения, а подключается рантаймом ОС во время исполнения программы. Динамические библиотеки могут шариться между приложениями. Все системные библиотеки iOS являются динамичными. UIKit, Foundation, CoreLocations etc. Другими словами эти библиотеки хранятся на устройстве, и мы не копируем их содержимое в каждое приложение. Эппл может обновлять эти библиотеки вместе с обновлением ОС.
  Важно: Девелоперы не могут создавать динамические библиотеки, только сам Apple.

Framework – содержит те же ресурсы, что и библиотека: хэдеры, сториборды, картинки и т.д., но в виде Bundle - иерархии директорий. Фреймворки могут содержать как библиотеки, так и другие фреймворки. Bundle – в том числе, является и классом в коде, через который можно получить доступ к ресурсам внутри фреймворка. Фреймворки поддерживают версионность.

- *Static framework* - содержит static library
- *Dynamic framework* - содержит dynamic library. Динамические фреймворки, в отличие от динамических библиотек, можно создавать начиная с iOS 8. В отличие от системных dynamic libraries, dynamic framework доступен как бандл только внутри конкретного приложения. Такой фреймворк архивируется в *.ipa, что в свою очередь увеличивает общий загружаемый объем приложения.

RxSwift basics

RxSwift – аналог RxExtensions из .net, библиотека для создания асинхронных и событийно-ориентированных программ с использованием паттерна Наблюдатель. Широко использует паттерн Наблюдатель и потоки последовательных направленных событий, на которые реагируют подписчики.

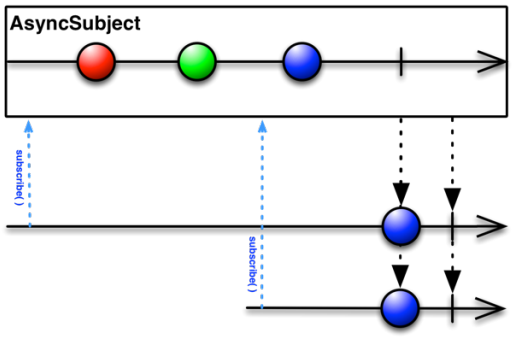
RxCocoa – экстеншены над элементами UIKit, которые позволяют удобно использовать Rx на элементах интерфейса.

Observable<T> - представляет собой источник последовательности событий или значений. Может посылать события асинхронно. На него могут быть подписаны подписчики.

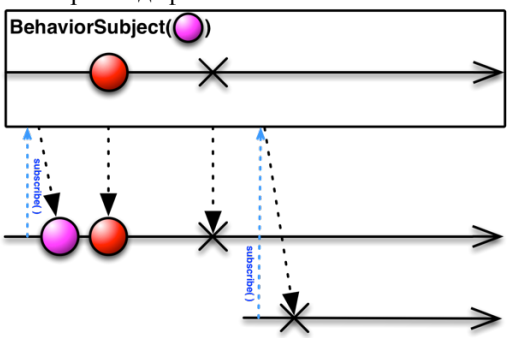
Subjects

Subjects – это своего рода прокси-объекты, которые могут выступать в роли как издателя – Observable, так и подписчика – Observer.

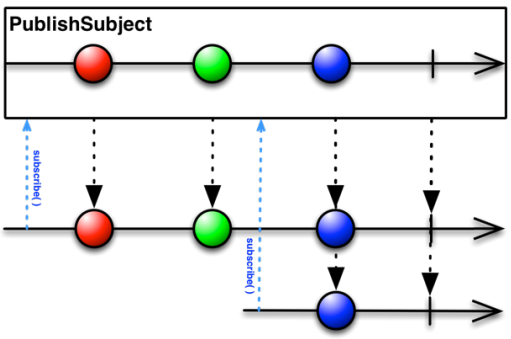
- *AsyncSubject<T>* - посылает последнее и ***только*** последнее значение для своих подписчиков. В случае если последовательность событий завершилась ошибкой – посылает ошибку.
  Не имеет значения по умолчанию.



- *BehaviourSubject<T>* - посылает подписчикам последнее текущее значение в последовательности, и все последующие значения от момента подписки. Если последовательность заканчивается ошибкой, то сабжект посылает ошибку. С практической точки зрения, нас интересует что данный сабжект может быть инициализирован с первым дефолтным значением.



- *PublishSubject<T>* - посылает подписчикам новое текущее значение в последовательности, которое было сгенерировано ***после*** момента подписки и все последующие от момента подписки.
  * Разница по сравнению с BehaviourSubject здесь в том, что BS сразу же после подписки пошлет последнее текущее значение в последовательности, в то время как PS нет. PS не пошлет никаких событий пока не будет сгенерировано *новое* значение от момент подписки.
  Если последовательность заканчивается ошибкой, то сабжект посылает ошибку.
  Не имеет значения по умолчанию.



- *ReplaySubject<T>* - посылает подписчикам всю последовательность, которая была сгенерирована до момента подписки. После подписки – посылает все значения которые генерируются после.

- *PublishRelay<T>, BehaviourRelay<T>* - обертки над соответствующими сабжектами, в отличие от сабжектов:
    - Не посылают ошибок.
    - Последовательность не имеет окончания.

Traits

Это просто обертки, которые могут “конвертировать” Observable в другой подтип, с целью предоставлять доступ к последовательности событий через удобные и подходящие к текущей ситуации интерфейсы.

- *Single* - это Observable, который вместо последовательности событий/значений возвращает только 1 или ошибку.

- *Completable* - это Observable, который вместо самой последовательности событий/значений возвращает только уведомление, что последовательность завершена, либо возвращает ошибку.

- *Maybe* - это Observable между Single and Completable. Может возвращать либо 1 значение, либо уведомление о завершении последовательности, либо ошибку.

- *Driver* – это Observable, который возвращает последовательность на Main Scheduler, проще говоря на UI потоке приложения. Эта обертка предназначена для использования подписок для элементов UI. Наиболее часто используемый тип. Не может посылать ошибки.

- *Signal* – имеет почти такое же поведение как и Driver. В отличие от Driver *HE* посылает последнее значение в последовательности.

Schedulers

Rx позволяет осуществлять подписки и посылать события на разных потоках. Например мы можем подписаться на обработку событий на UI потоке с использованием планировщика MainScheduler.

```
let observable = PublishSubject<Int>()
observable.subscribeOn(MainScheduler.instance).subscribe(onNext: { element in

})
```

Планировщики – это такие абстрактные сущности, которые предоставляют механизмы для выполнения асинхронных вычислений с применением current thread, dispatch queues, operation queues, new threads, thread pools, and run loops. То есть планировщик может быть и кастомным. Но я рассмотрю только те планировщики, которые уже идут в поставке Rx.

- *CurrentThreadScheduler(Serial scheduler)* – выполняет работу на текущем потоке. Это дефолтный планировщик по умолчанию.

- *MainScheduler(Serial scheduler)* - выполняет работу на UI потоке. Driver использует именно его.

- *SerialDispatchQueueScheduler (Serial scheduler)* – выполняет работу на заранее указанной последовательной очереди.

- *ConcurrentDispatchQueueScheduler (Concurrentscheduler)* – выполняет работу на заранее указанной параллельной очереди.

- *OperationQueueScheduler (Concurrent scheduler)* – выполняет работу на заранее указанной OperationQueue.

Сравнение RxSwift и Combine

https://github.com/CombineCommunity/rxswift-to-combine-cheatsheet

Все остальное

Битовые операции

Битовое И &
1 & 1 = 1
1 & 0 = 0
0 & 0 = 0
Битовое или |
1 | 0 = 1
1 | 1 = 1
0 | 0 = 1
Исключающее или XOR ^
Бит результата равен 1 в том случае, если ровно один из двух соответствующих битов входных байтов равен 1.
1 ^ 0 = 1
1 ^ 1 = 0
0 ^ 0 = 0

Лайфхаки

To Skip running pod repo update before install.Use : `pod install --no-repo-update`

To remove your current version you could just run:
`>sudo gem uninstall cocoapods`

You can install a specific version of cocoa pods via the following command:
`>sudo gem install cocoapods -v 0.25.0`

You can use older installed versions with following command:
`>pod _0.25.0_ setup`