Оглавление

[**1. Архитектура macOS** 3](#_Toc41118201)

[**2. Виртуализация, инструменты и запуск macOS в виртуализированном окружении** 6](#_Toc41118202)

[**3. Структура файловой системы ОС macOS, стандартные каталоги** 8](#_Toc41118203)

[**4. Гибкие методологии управления проектами и основные понятия систем управления в стиле Kanban** 9](#_Toc41118204)

[**6. Библиотека SQLite, управление базой данных** 20](#_Toc41118205)

[**7. Библиотека SQLite, синтаксис запросов** 25](#_Toc41118206)

[**8. Системы контроля версий, их виды и базовые команды git** 34](#_Toc41118207)

[**9. Ветвление в git, внешние репозитории и метки** 37](#_Toc41118208)

[**10. История развития iOS, архитектура и слои абстракции** 39](#_Toc41118209)

[**11. Структура проекта Single View Application** 40](#_Toc41118210)

[**12. Структура проекта согласно шаблону MVC и расширенная структура проекта** 42](#_Toc41118211)

[**13. Разработка интерфейса приложения в Interface Builder: макеты и автоматическая компоновка AutoLayout) и связь с кодом** 43](#_Toc41118212)

[**14. Язык программирования Objective-C: структура проекта согласно модели КИС** 44](#_Toc41118213)

[**15. Язык программирования Objective-C, базовый синтаксис языка, строковые объекты и коллекции** 49](#_Toc41118214)

[**16. Язык программирования Objective-C, классы и протоколы** 50](#_Toc41118215)

[**17. Язык программирования Swift, базовые операторы, переменные и константы, правила объявления переменных и констант, вывод текстовой информации (NSLog, println, print) и комментарии.** 52](#_Toc41118216)

[**18. Язык программирования Swift, числовые и текстовые типы данных, приведение числовых и текстовых типов, объединение строк и свойства строкового типа данных** 53](#_Toc41118217)

[**19. Язык программирования Swift, логические значения, логический тип данных и логические операторы, псевдонимы типов, операторы сравнения и диапазона** 56](#_Toc41118218)

[**20. Язык программирования Swift, кортежи (объявление кортежа, тип данных и взаимодействие с элементами кортежа)** 58](#_Toc41118219)

[**21. Язык программирования Swift, массивы (объявление, типы данных, доступ к элементам массива, сравнение и слияние массивов)** 59](#_Toc41118220)

[**22. Язык программирования Swift, массивы (многомерные массивы, базовые свойства и методы массивов)** 62](#_Toc41118221)

[**23. Язык программирования Swift, наборы (объявление, доступ и модификация набора, эквивалентность наборов)** 64](#_Toc41118222)

[**24. Язык программирования Swift, словари (объявление, взаимодействие с элементами словаря, явное указание типа данных, базовые свойства и методы словарей)** 66](#_Toc41118223)

[**25. Язык программирования Swift, опциональные типы данных и извлечение опционального значения** 70](#_Toc41118224)

[**26. Язык программирования Swift, функции и их возможности (объявление функций, аргументы функции и возвращаемое значение)** 71](#_Toc41118225)

[**27. Библиотека UIKit Framework, её основные структуры, классы и протоколы** 74](#_Toc41118226)

[**28. Контроллер представления и основные классы и методы** 76](#_Toc41118227)

[**29. Представления View, их типы и элементы управления, примеры и основные** 79](#_Toc41118228)

[**30. Интернационализация приложений для платформы iOS** 83](#_Toc41118229)

# **1. Архитектура macOS**

Прежде, чем говорить про архитектуру macOS, я бы хотел рассказать вообще о том, какие бывают типы у ядер различных операционных систем (Linux, Windows, macOS и других), потому что от них напрямую зависят особенности архитектуры. Ядро (иногда говорят kernel) является центральной частью операционной системы, о которой идёт речь, которая обеспечивает приложениям в этой системе так называемый координируемый доступ к тем или иным ресурсам компьютера, с которым мы работаем. К ним можно отнести процессорное время, память, внешнее аппаратное обеспечение, ну и соответственно внешнее устройство ввода или вывода информации. Говоря про типы ядер, можно выделить: монолитное, модульное ядра, а также менее распространённые микроядра (u-ядра), экзоядра, наноядра и гибридные ядра. В целом, можно сказать, что ядро обменивается информацией с серверами, а сервера в свою очередь с программным обеспечением, и таким образом все это слаженно работает в одной системе. В macOS используется ядро XNU, основанное на микроядре Mach и содержащее программный код, разработанный компанией Apple, а ещё код из ОС NeXTSTEP и FreeBSD. Стоит отметить, что версии примерно 10.3 данная операционная система работала только на компьютерах с процессорами PowerPC. Вроде как дальнейшие разработки поддерживали как PowerPC-, так и Intel-процессоры. Однако начиная с версии 10.6, macOS работает только с процессорами Intel. В качестве графического интерфейса используется разработка Aqua. Среди других компонентов операционной системы macOS можно выделить библиотеку Core Foundation, которая включает в себя такие компоненты как Carbon API, Cocoa API и Java API. Графическая среда представлена использованием таких технологий как QuickTime, Quartz Extreme и OpenGL. Ну и наконец стоит упомянуть поддерживаемые языки программирования. Это уже известные нам языки программирования: Си, C++, Objective-C, Swift, Ruby, Java и другие. Также стоит упомянуть тот факт, что данная операционная система поддерживается только на «мак-буках» компании Apple, что является, пожалуй, одним из серьёзнейших её недостатков.

Рассмотрим назначение каждого из уровней архитектуры.

**Cocoa Layer**

Уровень Cocoa, в основном, отвечает за внешний вид приложений и их способности реагировать на действия пользователя. Кроме того, этот же слой отвечает за многие функции, которые определяют пользователи OS X, такие как полноэкранный режим, AutoSave и т.д.

**Cocoa Layer** содержит фреймворк **AppKit**. AppKit является ключевой основой для разработки приложений. Классы и интерфейсы AppKit реализуют пользовательский интерфейс (UI) приложений, в том числе окон, диалоговых окон, элементы управления, меню и обработку событий. Они также управляют окнами, документами, диалогами типа Открыть и Сохранить, Копировать в буфер обмена и т.д. В дополнение к классам окон, меню и обработки событий в AppKit*,* также содержатся классы для работы со шрифтами, цветами, изображениями и т.д.

Также в Cocoa Layer «живут» такие фреймворки как:

GameKit

Preferences Panes

Screen Saver Framework

SecurityInterface

**Media Layer**

С Media Layer разработчики получают возможность разрабатывать красивые приложения с различными эффектами, воспроизведением видео/аудио и т.д. В общем, если AppKit в Cocoa дает нам возможность разрабатывать приложение вообще, то в Media Layer содержатся фреймворки, используя которые можно сделать красивое и функциональное приложение. Например, в Media Layer мы можем найти такие фреймворки:

AudioToolbox

AudioUnit

AVFoundation

CoreAudio

CoreAudioKit

CoreMIDI

CoreMIDIServer

DiscRecording

DiscRecordingUI

DVDPlayback

GLKit

IMServicePlugIn

InstantMessage

OpenAL

OpenGL

QTKit

Quartz

QuartzCore

QuickTime

SceneKit

**Core Services Layer**

На уровне Core Services располагаются фреймворки, позволяющие взаимодействовать с ОС на более низком уровне — получать доступ к различным типам данных, к сокетам для работы с сетью и т.д. Т.е. на этом слое мы начинаем углубляться в детали OS X. Так, например, используя фреймворк Foundation, мы можем работать с XML или получать доступ к рабочему пространству пользователя, например, чтобы запустить какое-либо приложение, получать информацию о запущенных приложениях и взаимодействовать с ними и т.д. Итак, какие фреймворки содержатся в Core Services:

Accounts

AddressBook

ApplicationServices

Automator

Carbon

Collaboration

CoreData

CoreFoundation

CoreLocation

CoreMedia

CoreServices

EventKit

Foundation

InputMethodKit

JavaScriptCore

LatentSemanticMapping

PublicationSubscription

QuickLook

ScriptingBridge

Security

SecurityFoundation

ServiceManagement

Social

StoreKit

WebKit

**Core OS Layer**

На уровне Core OS мы получаем в свое распоряжение классы интерфейсы для низкоуровневой работы с OS X, которых нам могло не хватить при работе с Core Services. Так, например, Core OS Layer содержит фреймворк SystemConfiguration с помощью которого мы можем получить доступ к настройкам самой операционной системы, управлять сетевыми соединениями и т.д.

На этом уровне работают следующие фреймворки:

CoreBluetooth

IOSurface

Accelerate

AGL

DirectoryService

DiskArbitration

ExceptionHandling

OpenCL

SystemConfiguration

XgridFoundation

**Kernel & Driver Layer**

На этом уровне реализуются классы и интерфейсы для доступа к ядру OS X и «железу» компьютера. На уровне Kernel & Driver используются следующие фреймворки:

CoreWLAN

ForceFeedback

FWAUserLib

ImageCaptureCore

IOBluetooth

IOBluetoothUI

IOKit

Kernel

System

# **2. Виртуализация, инструменты и запуск macOS в виртуализированном окружении**

**Виртуализация** — предоставление набора вычислительных ресурсов или их логического объединения, абстрагированное от аппаратной реализации, и обеспечивающее при этом логическую изоляцию вычислительных процессов, выполняемых на одном физическом ресурсе.

Примером использования виртуализации является возможность запуска нескольких операционных систем на одном компьютере, при том, каждый из экземпляров таких гостевых операционных систем работает со своим набором логических ресурсов (процессорных, оперативной памяти, устройств хранения информации), предоставлением которых из общего ресурса, доступного на уровне оборудования, управляет материнская операционная система или гипервизор. Также могут быть подвергнуты виртуализации сети передачи данных, сети хранения данных, платформенное и прикладное программное обеспечение.

Для виртуализации операционных систем применяется серия подходов, которые по типу реализации подразделяются на программные и аппаратные.

Начнем с программных методов.

**Динамическая трансляция** подразумевает под собой перехват команд гостевой операционной системы, вследствие чего гипервизор модифицирует их и возвращает гостевой ОС. Таким образом, гостевая операционная система фактически становится одним из приложений основной операционной системы, из-под которой она запущена. Гостевая система фактически считает, что работает на реальной физической платформе.

**Паравиртуализация** — технология виртуализации, при которой гостевые операционные системы подготавливаются для исполнения в виртуализированной среде, для чего их ядро незначительно модифицируется. Операционная система взаимодействует с программой гипервизора, который предоставляет ей гостевой API. Делается это для того, чтобы различные виртуальные машины могли работать с аппаратурой не конфликтуя с другими виртуальными машинами. Метод паравиртуализации позволяет добиться более высокой производительности, чем метод динамической трансляции.

В случае **полной виртуализации** используются не модифицированные экземпляры гостевых операционных систем. С целью поддержки работы этих гостевых систем используют общий слой эмуляции поверх основной операционной системы. Подобная технология применяется, к примеру, в таких приложениях как VMware Workstation, Parallels Desktop, MS Virtual PC, Virtual Iron. Среди достоинств данного метода виртуализации можно выделить относительную простоту реализации. Данное решение является достаточно надежным и универсальным.

Наиболее распространенным, в данный момент, формой виртуализации является **виртуализация операционной системы**. Виртуальная операционная система представляет собой совмещение нескольких операционных систем, функционирующих на одной аппаратной основе. Основным достоинством данного метода является высокая эффективность использования аппаратных ресурсов.

Классический процесс программной виртуализации подразумевает под собой наличие основной операционной системы, поверх которой запускается платформа виртуализации. Именно эта платформа берет на себя работу по эмуляции аппаратных компонентов и управляет ресурсами в отношении гостевой системы.

В отличии от программных методов, с помощью аппаратных средств виртуализации возможно получение изолированных гостевых систем, управляемых гипервизором **напрямую**.

Аппаратный процесс виртуализации практически не имеет никаких кардинальных отличий от программного. По факту, это процесс виртуализации, подкрепленный аппаратной поддержкой.

Оборудование

Эмуляция

Операционные системы

Программная виртуализация

Аппаратная виртуализация:

Intel VT — Intel Virtualization Technology for x86 (VT-x)

AMD-V — AMD Pacific

Виртуализация на уровне операционной системы: котейнеры Docker, LXC/LXD

**Виртуальная машина** — это окружение, которое предоставляется для

«гостевой» операционной системы, как аппаратное.

Процесс виртуализации действительно достаточно распространён и имеет особую важность. Как известно, операционная система macOS поддерживается только на «мак-буках» компании Apple, поэтому с данной операционной системой частенько работают на виртуальных машинах. В этом вопросе мы обсудим основные аспекты процесса виртуализации macOS и его запуск в виртуализированном окружении. Начнем с того, что дадим определение этому понятию. Виртуализация — это предоставление набора некоторых вычислительных ресурсов или их логического объединения, абстрагированное от аппаратной реализации, и обеспечивающее при этом логическую изоляцию друг от друга вычислительных процессов, выполняемых на одном физическом ресурсе. Другими словами, чтобы запустить macOS, например, на Windows, нам потребуется разделить ресурсы компьютеры на части, чтобы каждой операционной системе досталось какое-то определённое количество для работы. Обычно, на практике ресурсы делят поровну. Например, в моём компьютере оперативная память около 16 ГБ, и я разделил её таким образом, что оставил родной операционной системе Windows 8 ГБ, а на macOS соответственно выделил остальные 8 ГБ. Так, среда виртуализации связывает хостовую и гостевую операционные системы. Здесь я пользовался программным типом виртуализации (с помощью VMware Workstation, если быть точным), однако существуют и другие типы виртуализации, например, аппаратная (вспомним Intel VT – Intel Virtualization Technology for x86 или AMD-V – AMD Pacific) или даже на уровне операционной системы (через контейнеры Docker, LXC/LXD). Помимо виртуализации можно также с помощью оборудования проводить эмуляцию. Виртуализировать можно как ресурсы, так и отдельные приложения. Что касается виртуальных машин (одну я уже упомянул выше), то они являются по сути окружением, которое предоставляется для «гостевой» операционной системы как аппаратное. Среди известных виртуальных машин можно выделить VMware Workstation, VMware Workstation Player, VirtualBox, Microsoft Virtual PC, Microsoft Hyper-V и многие другие. Каждая из них имеет свою достоинства и недостатки. Таким образом, если у вас нету «мак-бука» для работы в macOS, можно установить одну из вышеперечисленных виртуальных машин и познавать операционную систему macOS.

# **3. Структура файловой системы ОС macOS, стандартные каталоги**

Файловая система имеет **древовидную иерархическую структуру**. Для её организации используются **директории** (они же папки или каталоги) — *объекты фaйловой системы, содержащие внутри себя файлы и другие директории*. Начинается файловая система с корня или корневой директории, которая находится в «начале» жёсткого диска и в ней содержатся уже все другие директории и файлы на жёстком диске.

Путь к файлу

Таким образом, у каждой папки или файла имеется адрес, по которому их можно найти, этот адрес называется путь.Начинается он с корневой директории, обозначенной символом слеш «/», далее идут названия директорий, разделённые слешем, до места назначения — директории или файла.

Относительный и абсолютный путь

Путь может быть относительным. Это значит, что он начинается от текущей директории, в которой вы находитесь, то есть, он относителен по отношению к ней.

**Абсолютный путь** — это путь, который начинается от корневой директории, он обязательно начинается со слеша.

**Существует четыре области файловой системы:**

**User**. Пользовательская область.Здесь хранятся ресурсы каждого отдельного пользователя. Физически, эта область определена домашней директорией пользователя в папке /Users на системном или же на сетевом разделе.

**Local**. Локальная область.Содержит общие ресурсы **для многих пользователе**й (например, общие документы и программы). Представлена множеством папок на системном разделе. Изменять информацию в этой области может лишь пользователь с правами администратора.

**Network**. Сетевая область.Сюда входят ресурсы, общие для пользователей локальной сети.

**System**. Системная область.Содержит ресурсы, необходимые для работы Mac OS. Физически они размещены в папке /System на системном диске. Изменять данные в этой папке может только root-пользователь.

Имя Описание

Applications Это папка, в которой лежат все установленные приложения. Собственно, большинство приложений устанавливаются простым перетаскиванием файла приложения в директорию Applications. Проще всего получить к ней доступ через сайдбар (левая панель) Finder’а.

Library В ней хранятся всевозможные настройки, [кэши](http://ru.wikipedia.org/wiki/%D0%9A%D1%8D%D1%88), а также туда часто приложения складывают какие-либо файлы, которые нужны им для работы. Чего там только нет, начиная от шрифтов и закачивая звуковыми семплами.

System Название говорит само за себя. Содержит только одну папку Library, которая похожа на предыдущую Library, но только тут всё относится уже не к установленным приложениям, а к самой Mac OS X. Кстати, приложение Finder находится именно там:

/System/Library/CoreServices/Finder.app

Users В этой директории хранятся домашние директории всех пользователей, зарегистрированных на данном компьютере. А в домашних директориях уже хранятся ваши документы, музыка, фото с котиками и прочее.

# **4. Гибкие методологии управления проектами и основные понятия систем управления в стиле Kanban**

**Управление проектами** — это приложение знаний, навыков, инструментов и методов к операциям проекта для удовлетворения предъявляемых к проекту требований.

Совокупность процессов управления проектами:

инициация;

планирование;

исполнение;

мониторинг и управление;

завершение.

**Жизненный цикл проекта** — совокупность фаз проекта, которые связывают его начало и окончание.



Последовательность фаз в жизненном цикле проекта

Фаза проекта характеризуется завершением и одобрением одного или нескольких её результатов.

Результат фазы — это измеримый, проверяемый продукт работы.

Основные методологии разработки

Waterfall — традиционный подход.

RUP (Rational Unified Process) — рациональный.

Crystal Clear — подход с уравниванием разработчиков в коллективе.

Spiral — спиральный метод.

DSDM (Dynamic Systems Development Model) — динамическая модель.

FDD (Feature Driven Development) — методология, рассматривающая будущие изменения.

JAD (Joint Application Development) — ориентированный на пользователя подход.

RAD (Rapid Application Development) — модель быстрой разработки.

Agile — общая методология гибкой разработки:

a.Scrum — концепция работы в условиях сорванных сроков и идеологического кризиса.

Kanban — концепция улучшения процессов для команды и способы разработки ПО.

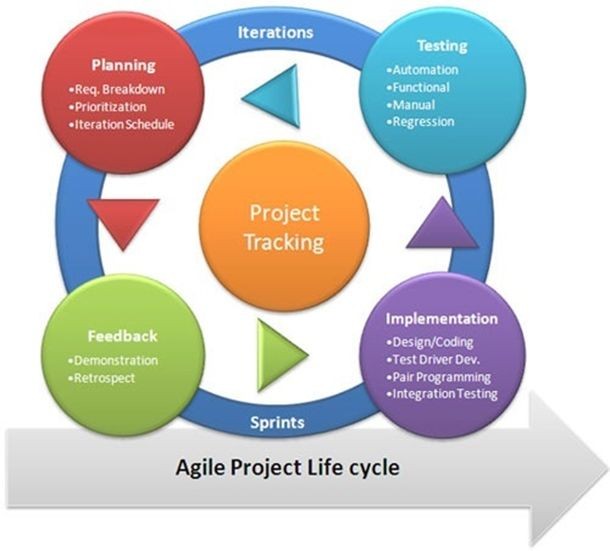
XP (Extreme Programming) — экстремальная разработка в динамической среде.

LD (Lean Development) — метод, предполагающий бережное отношение ко всем участникам процесса.

**Гибкая методология разработки** (англ. Agile software development, agile-методы) — серия подходов к разработке программного обеспечения, ориентированных на использование итеративной разработки, динамическое формирование требований и обеспечение их реализации в результате постоянного взаимодействия внутри самоорганизующихся рабочих групп, состоящих из специалистов различного профиля.

1970 г. — Уинстон Райс «Управление разработкой больших программных систем»

2001 г. — Манифест Agile (Agile Manifesto)



**Преимущества**

1.Максимальная удовлетворенность клиента 2.Благоприятная атмосфера в коллективе 3.Высокие цели

4.Безопасность

**Недостатки**

Слабое внимание к документации

Проблемы с реализацией комплексных продуктов 3.Вероятность отказа клиентов в процессе разработки 4.Влияние иерархии

**Scrum** — гибкий метод управления проектами, целью которого является повышение производительности труда в командах.

**Преимущества**

1.Быстрая публикация поставок 2.Коммуникация в команде 3.Поставки небольшие по размеру 4.Поставки легко отслеживать **Недостатки**

Требователен к команде проекта

Участник команды обладает более чем одной компетентностью

Командные игроки 4.Подходит не для всех игроков

**Экстремальное программирование** — возможность вести разработку в условиях постоянно меняющихся требований.

Игра в планирование.

Высокая частота релизов.

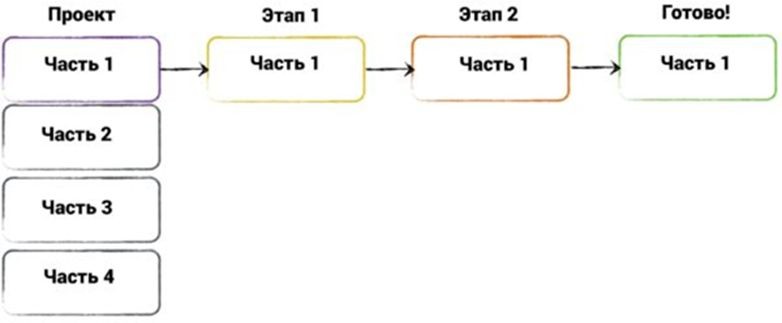
Контакт с клиентом.

Рефакторинг.

Стандарт выполнения кода.

Коллективная ответственность.

**Бережливая разработка ПО** — ещё одно ответвление гибкой методологии, предполагающее сохранение высокого морально-функционального состояния разработчиков.



**Kanban** — система организации производства и снабжения, позволяющая реализовать принцип «точно в срок».



Основа Kanban

Карточки

Ограничение на количество задач на этапе

Непрерывный поток

Постоянной улучшение (kaizen)

**Преимущества**

Подходит для сплоченных команд с хорошей коммуникацией и опытных команд

Расчет нагрузки на команду 3.Правильная расстановка ограничений 4.Концентрация на постоянном улучшении

**Недостатки**

Подходит для команд, участники которых обладают пересекающимися навыками

Отсутствие жестких дедлайнов

**Канбан** — метод управления разработкой, реализующий принцип «[точно в срок](https://ru.wikipedia.org/wiki/%D0%A2%D0%BE%D1%87%D0%BD%D0%BE_%D0%B2_%D1%81%D1%80%D0%BE%D0%BA)» и способствующий равномерному распределению нагрузки между работниками. При данном подходе весь процесс разработки прозрачен для всех членов команды. Задачи по мере поступления заносятся в отдельный список, откуда каждый разработчик может извлечь требуемую задачу. Основа подхода – снижение количества выполняемых в данный момент задач.

**Основные принципы Kanban**

Определение этапов работы. Их изображают в виде столбцов на доске (электронной или реальной). Задачи представляются карточками, которые перемещаются по этапам. После прохождения каждого из них мы получаем готовый к поставке заказчику элемент продукта или целый продукт.

Отсутствие деления процесса выполнения задач по времени (нет спринтов). То есть каждый участник команды просто выполняет задачу из общего пула, работает над ней с самого начала и до завершения. Процесс считается выполненным, задача готова.

Количество задач уменьшается за счет увеличения числа пунктов в каждой из них.

Ограничение на числа задач на конкретном этапе. Вы сами можете изменять количество задач. Это позволит быстро выявить и решить любой «затор» или недостаток работы.

Непрерывный поток. Задачи попадают в очередь в порядке приоритета. Поэтому работа никогда не прекращается.

**Сильные стороны Kanban:** подходит для сплоченных и мотивированных команд, нет фиксированных дедлайнов, идеальный расчет нагрузки между специалистами, четкая расстановка ограничений и акцент на постоянном улучшении, экономия трудовых ресурсов, максимальная гибкость.

**Слабые стороны Kanban:** для максимального эффекта от работы навыки специалистов из команды должны пересекаться (для совместного решения сложных задач), отсутствие четких сроков расхолаживает.

**Главная задача Kanban** – это уменьшение времени прохождения задачи от начала до стадии готовности.

Итого:

**Задачи**: Могут изменяться на любом этапе

**Встречи:** Проводятся только по завершении задачи или не проводятся совсем

**Самый важный параметр:** Время на выполнение задачи

**Команды:** Максимально сплоченные

**Добавление задач:** На любом этапе

**Роли:** Определяются условно или не требуются

**Ограничения:** По количеству работ в один временной промежуток

**Этапы работы:** Нет фиксированных этапов

**5. Многофайловое программирование на С и сборка приложений**

Если исходный код сколько-нибудь серьезной программы уместить в одном файле, то такой код станет просто нечитаемым. К тому же если программа компилируется достаточно долго, то после исправления одной ошибки, нужно перекомпилировать весь код.

Куда лучше разбросать исходный код по нескольким файлам (осмысленно, по какому-нибудь критерию), и компилировать каждый такой файл отдельно.

Сначала разберёмся, как из исходного файла получается бинарник. После компиляции создаётся ***объектный код***. Это исполняемый код с некоторыми "вкраплениями".

Эти самые "вкрапления" (самое главное среди них — таблица символов) позволяют объектному коду "пристыковываться" к другому объектному коду. Такую пристыковку или объединение выполняет *компоновщик* (*линковщик*) — программа, которая объединяет объектный код, полученный из "разных мест", удаляет все лишнее и создает полноценный бинарник. Этот процесс называется *компоновкой* или *линковкой*.

Чтобы откомпилировать мультифайловую программу, надо сначала добыть объектный код из каждого исходного файла в отдельности. Каждый такой код будет представлять собой *объектный модуль*. Каждый объектный модуль записывается в отдельный *объектный файл*. Затем объектные модули надо скомпоновать в один бинарный файл.

Автоматическая сборка

Утилита make

Сценарий записывается в файле с именем Makefile

Помещается в репозиторий (рабочий каталог) проекта.

Набор утилит GNU Autotools для генерации Makefile

Самое главное преимущество make - это "интеллектуальный" способ рекомпиляции: в процессе отладки make компилирует только измененные файлы.

**Сборка проекта** — выполнение утилиты make

**Сборщик** — утилита make

Makefile

комментарии:

однострочные комментарии, начинающиеся с #

макроопределения:

макроопределения позволяют назначить имя любой строке

целевые связки:

цель, зависимости (реквизиты) и правила (команды).

# Индентация осуществляется исключительно при помощи символов табуляции, # каждой команде должен предшествовать отступ

<цели>: <реквизиты>

<команда #1>

...

<команда #n>

# Makefile for Hello World project

hello: main.o hello.o

gcc -o hello main.o hello.o

main.o: main.c

gcc -c main.c

hello.o: hello.c

gcc -c hello.c

clean:

rm -f \*.o hello

**Модель КИС**

**Репозиторий** — рабочий каталог, в котором находятся исходники, сценарии сборки (Makefile) и прочие файлы, относящиеся к проекту.

**Модель КИС** (Клиент-Интерфейс-Сервер) — это элегантная концепция распределения исходного кода в репозитории, в рамках которой все исходники можно поделить на *клиенты*, *интерфейсы* и *серверы*. Сборка (building) программного проекта – набор правил и процедур, направленный на получение исполняемой программы

Выпуск (release) программного продукта – процесс отчуждения программы от разработчика и заключающийся в

Сборке программного проекта

Формировании инсталляционного пакета

Формировании документации

Формировании аннотации релиза

Задачи сборки и выпуска проектов

Трансляция всего проекта

Сборка дистрибутива

Подготовка исходных текстов

Подготовка документации Причины сборки проектов

Проверка работоспособности

Очередная периодическая сборка

Подготовка версии к автоматическому тестированию

Подготовка дистрибутива

Инсталляция дистрибутива (из исходных текстов)

Проблемы сборки проектов

Проблемы с исходными текстами

Проблемы с подключаемыми файлами

Проблемы с используемыми библиотеками

Проблемы с процедурами сборки

Проблемы со средствами (утилитами) сборки

Проблемы с системной средой и аппаратной платформой сборки

Окружение для сборки

Аппаратная платформа

Системное окружение

Операционная система

Системные файлы

Библиотечное окружение

Подключаемые файлы

Библиотечные файлы

Исходные файлы в требуемых каталогах

Средства сборки

Построение окружения

Установка всех требуемых библиотек

Из дистрибутивов (версия разработчика)

Установка всех средств сборки

Из дистрибутива

Размещение всех исходных файлов

Из СКВ

Из архива

Общие требования для сборки

Сборка должна проводиться на любом компьютере с подготовленным окружением

Сборка должна проводиться отдельно от рабочего места разработчика

Версии подключаемых файлов

Версии библиотек

Расположение исходных файлов

Версии средств сборки

Влияние временных файлов

Объектные файлы

Исполняемые файлы

Предкомпилированные заголовки

И т.п.

Процедура сборки должна быть:

Документирована

Прозрачна

Повторяема

Способы сборки программных проектов

С помощью среды разработки

С помощью запуска компилятора командной строки

С помощью сценариев

С помощью различных make tools

make

nmake

Ant

Maven

И т.п.

Сборка компонента проекта

Для каждого компонента проекта должна быть сформирована локальная процедура сборки

Например, makefile

В каждом компоненте проекта сборка должна проходить одной командой

Например, make Требования к процедурам сборки

Процедуры сборки должны

находиться под управлением СКВ

помечаться и в

етвиться аналогично файлам проекта

изменяться в соответствии с изменениями проекта Утилиты сборки

Make tools:

GNU Make

Nmake

Ant

Maven

...

Генераторы make-файлов

GNU Automake

Makedep

Makedepend Средства сборки

Средства управления сборкой:

IBM Rational Build Forge

FinalBuilder

TeamCity

Visual Build

Codefast PerfectBuild

OpenMake

Anthill Pro

...

# **6. Библиотека SQLite, управление базой данных**

(сначала немного о SQL)

**База данных** (БД, database — DB) — поименованная совокупность структурированных данных, относящихся к определённой предметной области.

**Предметная область** — некоторая часть реально существующей системы, функционирующая как самостоятельная единица.

**Система управления базами данных** (СУБД, database management system — DBMS)

комплекс программных и языковых средств, необходимых для создания и модификации базы данных.

Классификация баз данных

По типу хранимой информации:

документальные,

фактографические,

лексикографические.

По характеру хранения и обращения:

локальные (персональные),

общие (интегрированные, централизованные),

распределенные базы данных.

По характеру организации данных:

неструктурированные,

частично структурированные,

структурированные.

По типу используемой модели:

иерархические,

сетевые,

реляционные,

смешанные,

мультимодельные.

**Реляционная база данных** — тип баз данных, состоящих из таблиц, связанных по ключевым значениям.

**Таблица базы данных (table)** - регулярная структура, которая состоит из однотипных строк (записей, records), разбитых на столбцы (поля, fields).

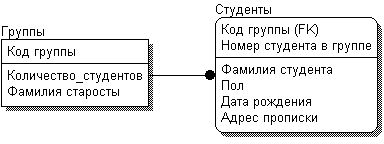
**Ключевой элемент таблицы** (ключ, regular key) — такое её поле (простой ключ) или строковое выражение, образованное из значений нескольких полей (составной ключ), по которому можно определить значения других полей для одной или нескольких записей таблицы.

**Первичный ключ** — поле или набор полей, однозначно идентифицирующих запись.

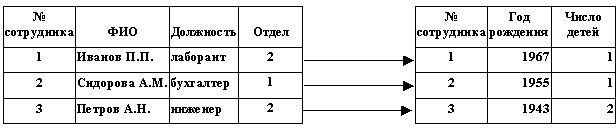
**Виды ключей** — первичный, альтернативный, уникальный.

Реляционные связи между таблицами

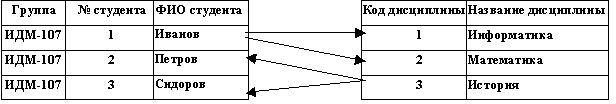
Один-ко-многим



Один-к-одному



Многие-ко-многим



**Внешний ключ** (foreign key) — ключевой элемент подчиненной (внешней, дочерней) таблицы, значение которого совпадает со значением первичного ключа главной (родительской) таблицы.

**Индекс** — механизм быстрого доступа к данным в таблицах БД.

Методы доступа:

последовательный;

индексно-последовательный.

*Определение первичных и внешних ключей таблиц БД приводят к созданию индексов по полям, объявленным в составе первичных или внешних ключей.*

SQLite — компактная встраиваемая реляционная база данных. SQLite — C-библиотека, реализующая движок базы данных SQL. SQLite — все данные хранятся в одном файле.

Преимущества SQLite

С точки зрения пользователя — SQLite, это одиночный файл на диске.

Движок SQLite и интерфейс к ней реализованы в одной библиотеке.

Нет проблем с развертыванием и переносом базы данных с одного компьютера на другой.

Реализует основные возможности стандарта SQL 92 (включая триггеры и транзакции).

SQLite обеспечивает поддержку объема данных до 2-х терабайт.

Поддерживается ограниченное количество типов данных.

Типы данных SQLite

|  |  |
| --- | --- |
| **Объявление** | **Описание** |
| NULL | NULL-значение или отсутствующее значение |
| INTEGER | Целые положительные и отрицательные числа переменного размера от 1 до 8 байтов. |
| REAL | Числовой тип данных размером 8 байтов. Позволяет хранить любые числа. |
| TEXT | Текстовый тип данных который может хранить текстовые строки произвольной длины в кодировке UTF-8 или UTF-16. |
| BLOB | Тип данных для хранения двоичных объектов. |

Отличие SQLite от других баз данных

нет привязки к типам данных;

все данные сохраняются как строки оканчивающиеся символом NULL;

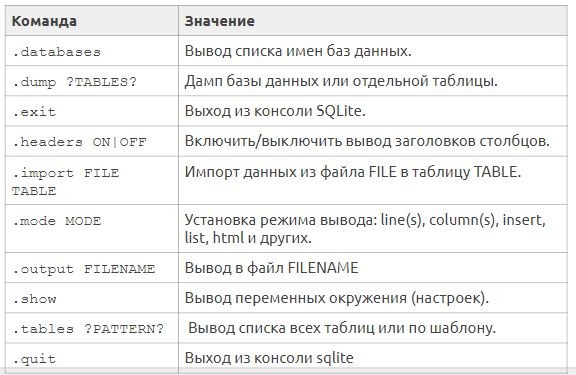
SQLite поддерживает тип спецификации в конструкциях CREATE TABLE, например, такой как INT, CHAR, FLOAT, TEXT;

внутри базы данных SQLite только делает различие между строковыми и целочисленными данными во время сортировки.

Установка в macOS

$ sudo port install sqlite3 ИЛИ

$ brew install sqlite3



Создание базы данных

$ sqlite3 db.sqlite

Импорт базы данных

$ sqlite3 testbdsqlite.db < dbcreate164.sql

Создание базы данных и таблиц

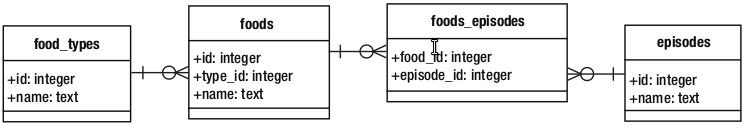
$ sqlite3 foods.db

create table episodes (id integer primary key,season int,name text );

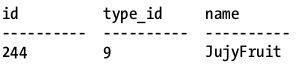
create table foods\_episodes(food\_id integer,episode\_id integer);

create table foods(id integer primary key,type\_id integer,name text );

create table food\_types(id integer primary key,name text );



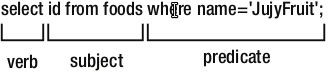
Настройка вывода

.echo on

.mode column

.headers on

.nullvalue NULL

Выборка

sqlite> select \*

...> from foods

...> where name='JujyFruit'

...> and type\_id=9;

Изменение таблицы

Синтаксис

alter table table { rename to name | add column column\_def }

Пример добавления полей в таблицу

sqlite> alter table contacts

add column email text not null default '' collate nocase;

INSERT

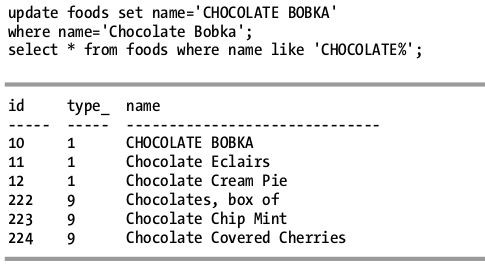
Форма записи

insert into table (column\_list) values (value\_list);

Вставка одной строки

sqlite> insert into foods (name, type\_id) values ('Cinnamon Bobka', 1);

Обновление записей



Удаление записей и таблиц

delete from table where predicate; drop table table;

Менеджеры для баз данных SQLite

SQLite Database Browser (DB Browser for SQLite)

Valentina Studio

Navicat for SQLite

DbVisualizer

# **7. Библиотека SQLite, синтаксис запросов**

Про библиотеку SQLite и концепцию баз данных можно прочитать в вопросе выше (номер 9).

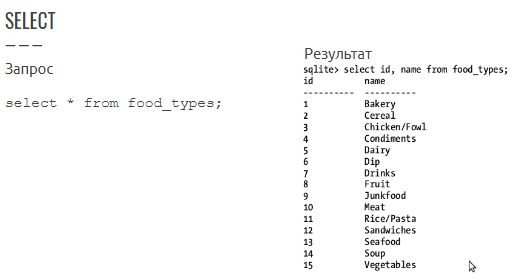
**Запрос SELECT**

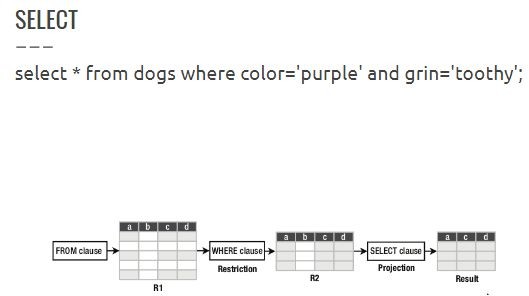
Общий синтаксис запроса select можно описать следующим образом:

select [distinct] heading from tables

where predicate group by columns having predicate order by columns limit count, offset;

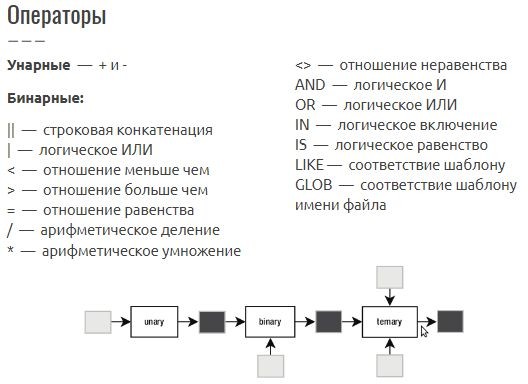
В общем случае все уточняющие параметры после записи from tables являются опциональными и влияют лишь на результат вывода, а не на работоспособность как таковую.

**Примеры:**



Рассмотрим более детально возможные дополнения команды select.

Нетрудно видеть, что во втором примере мы объединили два значения поля с помощью оператора and.

**Рассмотрим все операторы:**

**Операции LIKE и GLOB**

Эти два операторы похожи и служат для сопоставления только текстовых значений с шаблоном с помощью подстановочных знаков.

В отличие от оператора LIKE, GLOB учитывает регистр символов и следует синтаксису UNIX для указания следующих подстановочных знаков:

\*

?

Знак звездочки ( \* ) у GLOB обозначает ноль или несколько цифр или символов. Вопросительный знак (?) представляет одно число или символ.

**Примеры:**

**WHERE SALARY GLOB '200\*'**

Ищет любое значение, начинающееся с 200.

**WHERE SALARY GLOB '\*200\*'**

Ищет любое значение, где 200 содержится в любой позиции.

**WHERE SALARY GLOB '?00\*'**

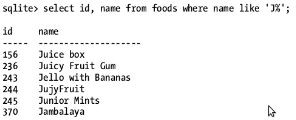
Ищет любое значение, где нулю стоят на второй и третьей позиции.

**WHERE SALARY GLOB '2??'**

Ищет любое значение, начинающееся с 2 и имеющее в своём составе хотя бы 3 символа.

**WHERE SALARY GLOB '?2\*3'**

Ищет любое значение, где на втором месте стоит 2, а в конце - 3.

У LIKE же аналогами служат знак % для \* и \_ для ? соответственно. Пример использования:

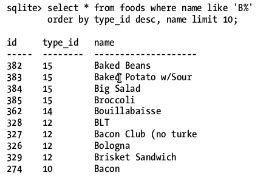
Выбирает значения, где имя начинается с J.

**Операции LIMIT и ORDER**

LIMIT служит для того, чтобы выбрать ограниченный набор результата поиска. Например, результатов 15, а выбрать

надо лишь первые 10, LIMIT тут и пригодится. ORDER позволяет сортировать результат запроса.

Пример:



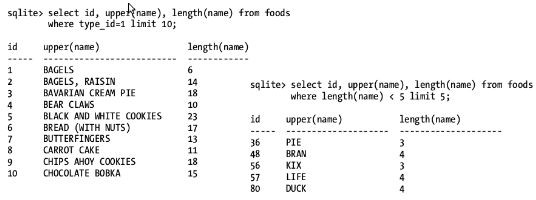
Тут мы сортируем по номеру type\_id, после чего дополнительно сортируем группки с одинаковыми значениями type\_id ещё и лексикографически. Обращаю внимание, что здесь сортировка по type\_id идёт в обратном порядке благодаря параметру desc, который задает обратный порядок сортировки.

Для дальнейшего продвижения надо ознакомиться с функциями:

abs(), upper(), lower(), length(). abs - модуль;

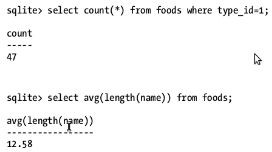
upper - переводит символы в верхний регистр; lower - в нижний;

length - длина. Пример:



Также есть и функции агрегации, которые позволяют получить совокупный результат в зависимости от содержимого полей результата запроса.

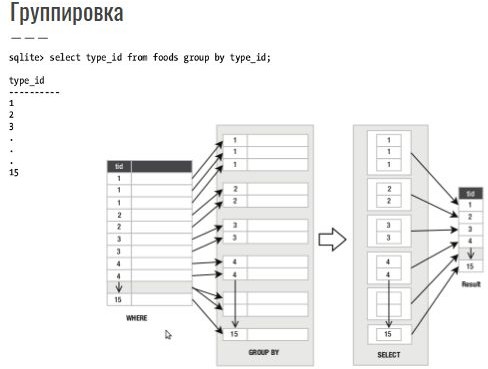
Функции: sum(), avg(), count(), min(), max(). Соответственно сумма, среднее, количество, минимальное и максимальное.

Например:

**GROUP**

С помощью команды GROUP BY можно группировать элементы по значению какого-либо поля.

Пример:



На первый взгляд может показаться, что функция группировки в данном случае сработала прямо как ORDER BY. И в принципе возникшие сомнения в целесообразности её существования обоснованы.

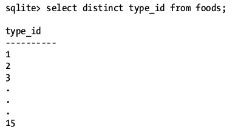
Но всё же есть ситуации, где функция группировки оказывается полезной.

Например, у нас в магазине есть множество экземпляров товаров каждого вида, а таких видов несколько. Нам нужно посчитать количество единиц каждого вида товара. Первое, что приходит на ум, - это использовать функцию count, однако в чистом виде она подсчитает нам сумму ВСЕХ товаров в магазине независимо от вида. И тут нам на помощь приходит группировка:

Нетрудно видеть, что в данном примере товары сначала группируются по типу, а потом уже в каждой группе считается количество элементов. То, что нам было нужно.

**DISTINCT**

Эта команда применяется для фильтрации дублей:



**JOIN**

SQL определяет три основных типа соединений:

Cross join

Inner join

Outer join

Cross join сопоставляет каждую строку первой таблицы с каждой строкой второй таблицы. Если входные таблицы имеют строки x и y, соответственно, результирующая Таблица будет иметь x \* y строк.

Поскольку cross join’ы потенциально могут создавать чрезвычайно большие таблицы, следует использовать их только в соответствующих случаях.

Синтаксис cross join:

Inner join создает новую таблицу результатов путем объединения значений столбцов двух таблиц (table1 и table2) на основе предиката соединения. Запрос сравнивает каждую строку таблицы 1 с каждой строкой таблицы 2, чтобы найти все пары строк, удовлетворяющих предикату соединения. Когда предикат соединения удовлетворен, значения столбцов для каждой пары строк A и B объединяются в строку результата.

Inner join является наиболее распространенным и стандартным типом соединения. Можно использовать ключевое слово INNER опционально.

Синтаксис inner join:

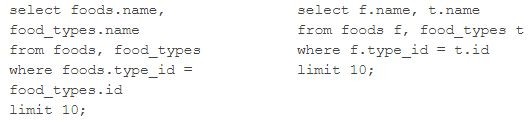
Outer join является расширением inner join. Хотя стандарт SQL определяет три типа внешних соединений (outer join): левое (left),

правое (right) и полное(full), SQLite поддерживает только LEFT OUTER JOIN.

В отличие от inner join в результате left outer join выводятся не только совпавшие записи из обеих таблиц, но и записи из левой таблицы, для которых совпадений в правой не нашлось.

Ненайденные поля правой таблицы заполняются пробелом либо NULL.

Синтаксис:

**Имена и псеводнимы**

В примере выше слева используются явные таблиц имена, к полям которых мы доступ получить хотим, справа же псевдонимы их используем мы.

 смешно:) очень

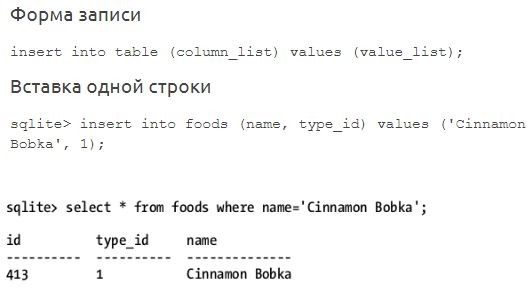
**CASE**

Case команда используется аналогично команде switch в обычных языках программирования:

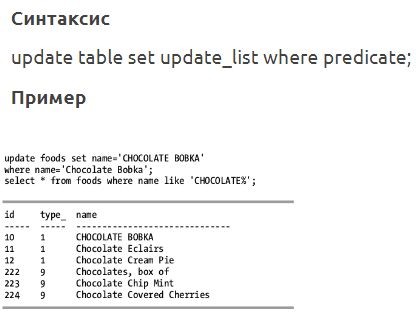


Кроме SELECT есть и другие типы запросов.

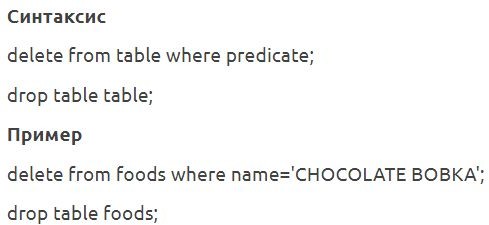
**INSERT**



**UPDATE**



**DELETE**



# **8. Системы контроля версий, их виды и базовые команды git**

Командная работа над проектом включает в себя:

Общий доступ к файлам(совместное редактирование)

Сохранение истории модификации файла (Версия, автор, время, суть и причина изменения)

Пометка отдельных версий файла

Поддержание и развитие нескольких параллельных историй файла:

Версия, поставленная заказчику

Версия в разработке

Конфигурации для разной программной и аппаратной платформ

Эти задачи в проекте помогают выполнять системы контроля версий(СКВ) СКВ предназначены для автоматизации групповой работы и управления версионированием проектов и обеспечивают:

Репозиторий (или репозитории) хранения проектов

Стандартные операции обеспечения групповой работы

Клиенты для выполнения операций

Типы систем контроля версий:

локальная система контроля версий(простая БД, где хранятся все изменения нужных файлов)

централизованная система контроля версий(Единое централизованное хранилище с Клиент-серверным доступом)

распределенная система контроля версий

Про распределенные СКВ(к коим и относится гит):

Репозиторий хранится на каждом компьютере

Сетевая синхронизация репозиториев посредством заплаток (слияний, патчей, change sets и т.п.)

Используется в интернет-проектах, когда разработчики существенно удалены друг от друга

Git - программное обеспечение для управления версиями, разработанное Линусом Торвальдсом для использования в управлении разработкой ядра Linux.

Для большинства операций git необходимы только локальные файлы и ресурсы.

История проекта хранится на диске локально.

Изменения между текущей версией файла и другой вычисляются локально.

Изменения вносятся локально и публикуются при наличии сети.

Для сохранения целостности данных перед каждым сохранением файла вычисляется контрольная сумма(результат хеш-функции SHA-1)

Почти все действия с гитом добавляют данные в БД. ПОсле фиксации их достаточно трудно потерять. Хорошая практика - часто добавлять(коммитить) файлы.

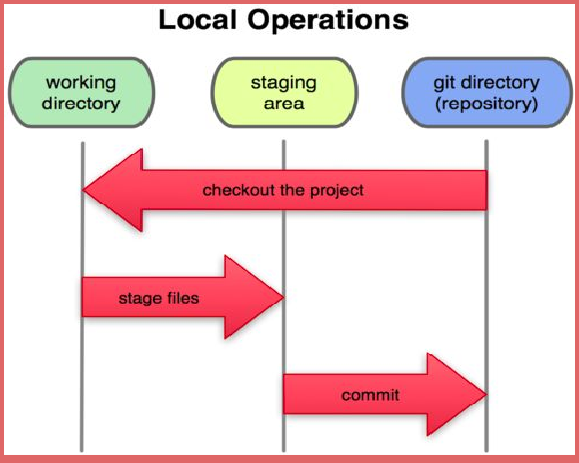
В гите есть 3 состояния файлов: 1)Зафиксированное 2)Измененное 3)Подготовленное

И 3 области:

каталог git (git directory)

рабочий каталог (working directory)

область подготовленных файлов (index or staging area)



**Основные команды**

После установки гита, нужно будет залогиниться git config --global user.name "John Doe"

$ git config --global user.email [johndoe@example.com](mailto:johndoe@example.com)

**Помощь**

$ git help <команда>

$ git <команда> --help

$ man git-<команда>

**Создание (инициализация) репозитория:**

$ git init

Результат — каталог .git

**Добавление существующих файлов в репозиторий**

$ git add \*.c

$ git add README

$ git commit -m 'initial project version'

**Клонирование репозитория**

git clone [url]

$ git clone git://github.com/libgit2/rugged.git

$ git clone git://github.com/libgit2/rugged.git myrugget

**Состояние файлов и история изменений**

$ git status

$git log

**Индексация(добавление в локальный репоз) измененных файлов**

$ git add benchmarks.rb

**Фиксация изменений**

$ git commit (и -m “” для добавления своего комментария)

**Удаление из отслеживаемых файлов**

$ git rm

**Перемещение файлов**

$ git mv file\_from file\_to

**Добавление внешнего репозитория**

$ git remote add [сокращение] [url]

**Получение и отправка данных во внешний репоз**

$ git pull — извлекает (fetch) данные с сервера и автоматически пытается слить (merge) их с локальным кодом

$ git push [удал. сервер] [ветка] — отправка наработок во внешний репозиторий

$ git push origin master — отправка ветки master на сервер origin

**Для работы с ветвями**

$git branch branchName - создать ветку

$git checkout branchName - переключиться на ветку

Это основные команды, у каждой есть свой набор опций, который можно узнать с помощью команды **git help команда**

# **9. Ветвление в git, внешние репозитории и метки**

Ветка в git - легковесный подвижный указатель(HEAD) на один из коммитов. Ветка по умолчанию в Git’е - master.

GIt branch [название ветки] - создание ветки, git branch (-v)(--merged) (--no-merged) - вывод всех веток (просмотр последнего коммита) (вывод всех веток, которые уже слили)(вывод всех веток, которые еще не слили в главную), git checkout - переключение между ветками, git rebase - перемещение измененией на другую ветку, git merge - слияние веток. Rebase применяется для избежания или разрешения конфликтов.

git tag (-a v1.4 -m 'my version 1.4')( -s v1.5 -m 'my signed 1.5 tag' ) (-v

[имя метки] ) — просмотр меток (добавление аннотированной метки)( добавление подписанной метки ключом GPG)(верификация подписанной метки), git show - просмотр меток вместе с коммитом

git remote ( -v) ( add [сокращение] [url]) — перечисляет список имён-сокращений для всех уже указанных удалённых дескрипторов (соответствие URL сокращенному имени)( добавление внешнего репозитория, например)

git fetch [имя удал. сервера]- получение данных

git fetch origin — извлекает все наработки, отправленные (push) на сервер после его клонирования

git pull — извлекает (fetch) данные с сервера и автоматически пытается слить (merge) их с локальным кодом

git push [удал. сервер] [ветка] — отправка наработок во внешний репозиторий

git push origin master — отправка ветки master на сервер origin (Дальше из книги Pro Git)

Как и большинство СКВ, Git имеет возможность помечать (tag) определённые моменты в истории как важные. Как правило, этот функционал используется для отметки моментов выпуска версий (v1.0, и т.п.).

Просмотр имеющихся меток (tag) в Git'е делается просто. Достаточно набрать git tag

$ git tag -l 'v1.4.2.\*' v1.4.2.1

Для меток вы также можете осуществлять поиск по шаблону. Например, репозиторий Git'а содержит более 240 меток. Если вас интересует просмотр только выпусков 1.4.2, вы можете выполнить следующее:

v1.4.2.2

v1.4.2.3

v1.4.2.4

Git использует два основных типа меток: легковесные и аннотированные. Легковесная метка — это что-то весьма похожее на ветку, которая не меняется — это просто указатель на определённый коммит. А вот аннотированные метки хранятся в базе данных Git'а как полноценные объекты. Они имеют контрольную сумму, содержат имя поставившего метку, e-mail и дату, имеют комментарий и могут быть подписаны и проверены с помощью GNU Privacy Guard (GPG). Обычно рекомендуется создавать аннотированные метки, чтобы иметь всю перечисленную информацию; но если вы хотите сделать временную метку или по какой-то причине не хотите сохранять остальную информацию, то для этого годятся и легковесные метки.

Ветка в Git — это простой файл, содержащий 40 символов контрольной суммы SHA-1 коммита, на который она указывает; поэтому операции с ветками являются дешёвыми с точки зрения потребления ресурсов или времени. Создание новой ветки в Git происходит так же быстро и просто как запись 41 байта в файл (40 знаков и перевод строки).

Это принципиально отличает процесс ветвления в Git от более старых систем контроля версий, где все файлы проекта копируются в другой подкаталог. В зависимости от размера проекта, операции ветвления в таких системах могут занимать секунды или даже минуты, когда в Git эти операции мгновенны. Поскольку при коммите мы сохраняем указатель на родительский коммит, то поиск подходящей базы для слияния веток делается автоматически и, в большинстве случаев, очень прост. Эти возможности побуждают разработчиков чаще создавать и использовать ветки.

# **10. История развития iOS, архитектура и слои абстракции**

**iOS** (до 24 июня 2010 года — iPhone OS) — мобильная операционная система для смартфонов, электронных планшетов, носимых проигрывателей и некоторых других устройств, разрабатываемая и выпускаемая американской компанией Apple.

iOS 1.0 (iPhoneOS 1) — 9 января 2007

iOS 2.0 (iPhoneOS 2) — 11 июля 2008

iOS 3.0 (iPhoneOS 3) — 17 июня 2009 4. iOS 4.0 — 21 июня 2010

iOS 5.0 — 12 октября 2011

iOS 6.0 — 19 сентября 2012

iOS 7.0 — 18 сентября 2013

iOS 8.0 — 17 сентября 2014

iOS 9.0 — 9 сентября 2015

iOS 10.0 — 13 сентября 2016

iOS 11.0 — 19 сентября 2017

Пользовательский интерфейс iOS основан на концепции прямого взаимодействия с использованием жестов «мультитач».

Использует набор основных компонентов Darwin, совместимый со стандартом POSIX.

Слои абстракции iOS:

Core OS;

Core Services;

Media Layer;

Cocoa Touch.

iOS представляет собой облегченную версию настольной Mac OS X. В основе Mac OS X лежит POSIX-совместимая ОС Darwin. основа Darwin — это ядро XNU.

Условно начинку OS X / iOS можно разделить на три логических уровня: ядро XNU, слой совместимости со стандартом POSIX (плюс различные системные демоны/сервисы) и слой NeXTSTEP, реализующий графический стек, фреймворк и API приложений.

Уровнем выше ядра в Darwin располагается слой UNIX/BSD, включающий в себя набор стандартных библиотек языка си.

На этом открытая часть ОС под названием Darwin заканчивается, и начинается слой фреймворков, которые как раз и образуют то, что мы привыкли считать OS X / iOS

Darwin реализует лишь базовую часть Mac OS / iOS, которая отвечает только за низкоуровневые функции

Как и во многих других ОС, API Mac OS и iOS разделен на публичный и приватный. Сторонним приложениям доступен исключительно публичный и сильно урезанный API, однако jailbreak-приложения могут использовать и приватный.

Набор базовых фреймворков для разработки графических приложений объединен в так называемый Cocoa API, своего рода метафреймворк, позволяющий получить доступ к основным возможностям ОС. В iOS он носит имя Cocoa Touch и отличается от настольной версии ориентацией на сенсорные дисплеи.

Далеко не все фреймворки доступны в обеих ОС. Многие из них специфичны только для iOS. В качестве примеров можно привести AssetsLibrary, который отвечает за работу с фотографиями и видео.

Все стандартные системные фреймворки можно найти в системном каталоге

/System/Library/Frameworks/. Каждый из них находится в своем собственном каталоге, называемом бандлом (boundle)

Уровнем выше находятся приложения, системные и устанавливаемые из магазина приложений. Центральное место среди них занимает, конечно же, SpringBoard (только в iOS), реализующее домашний экран.

SpringBoard — это связующее звено между операционной системой и ее пользователем, графический интерфейс, позволяющий запускать приложения и переключаться между ними, просматривать уведомления и управлять некоторыми настройками системы (начиная с iOS 7).

Располагается в каталоге /System/Library/CoreServices/ Использует плагины из /System/Library/SpringBoardPlugins/

Начиная с iOS 6 SpringBoard разделен на две части: рабочий стол и сервис BackBoard

# **11. Структура проекта Single View Application**

AppDelegate.swift

Вообще говоря это некая точка входа в ваше приложение. Этот класс управляет состояниями вашего приложения. Он так же содержит главное окно вашего приложения, то бишь UIWindow. В этом классе настраиваются все основные потребности вашего приложения, будь то Push Notifications или Voice Calls и так далее. Я бы сказал это Core вашего приложения.

Начнем с «главного» метода application:didFinishLaunchingWithOptions:. Главным я его назвал лишь потому, что этот метод срабатывает только раз: при открытии приложения (загрузке его в память). Из комментариев становится ясно, что он предназначен для конфигурации аппа. К примеру, здесь можно кастомизировать глобальный интерфейс приложения, загружать необходимые данные из базы данных, либо же определять какой экран нужно загрузить первым (в обход storyboard).

Переходим дальше и на очереди у нас applicationWillResignActive:. Делегат вызывает данный метод когда юзер только начал сворачивать приложение. Также он вызывается при входящих звонках, начале работы будильника и т.д. В данном методе Apple советует останавливать любые активные задачи, таймеры и тому подобное.

Метод делегата applicationDidEnterBackground: срабатывает, когда приложение уже закончило переход в background (фоновый режим работы). Здесь лучше всего сохранять какие-нибудь промежуточные данные пользователя, останавливать работу различных СДК, либо же сохранять идентификатор последнего активного окна, если Вам нужно открыть его при возобновлении работы приложения.

Метод applicationWillEnterForeground: делегат приложения вызывает, когда апп возвращается в foreground из бэкграунда. В этот момент пользователь еще не видит интерфейс приложения, поэтому здесь Вы можете восстановить все необходимые данные и вернуть изменения, сделанные в предыдущем методе.

applicationDidBecomeActive: срабатывает когда приложение уже стало активным и пользователь видит интерфейс аппа. Обновляйте здесь интерфейс различными данными, восстанавливайте работу таймеров, сторонних СДК и т.д.

Последний метод из списка — applicationWillTerminate: вызывается когда пользователь «убивает» приложение (закрывает и оно удаляется из памяти системы). Если у Вас нет необходимости сохранять важные данные в applicationDidEnterBackground:, то обязательно делайте это здесь. Кстати, если апп будет поддерживать активную работу в бэкграунде, то applicationWillTerminate: не сработает и вместо него вызовется всё тот же applicationDidEnterBackground:.

ViewController.swift

Main.storyboard

Assets.xcassets

LaunchScreen.storyboard

info.plist

# **12. Структура проекта согласно шаблону MVC и расширенная структура проекта**

**MVC** расшифровывается как **Model-View-Controller**

(Модель-Отображение-Контроллер)

и является основой построения программных продуктов

в среде **Xcode**

**Модель** — классы, которые обеспечивают хранение данных ваших приложений.

**Отображение** (Представление) — позволяют создать различные графические элементы, которые видит пользователь при работе с приложением.

**Контроллер** — обеспечивает совместную работу «отображения» и «модели». Данный блок содержит логику работы приложения

**Структура проекта:**

AppDelegate.swift

ViewController.swift

Main.storyboard

Assets.xcassets

LaunchScreen.storyboard

info.plist

**Расширенная структура** проекта в соответствии с MVC:

CoreData

Extension

Helper

Model

Services

View

Controller

Каждый объект, который вы создаете в своей программе, может быть легко отнесен к одной из категорий, но при этом не должен реали- зовать какие-либо функции, присущие двум другим . Например эк- земпляр класса UIButton, обеспечивающий отображение кнопки, не должен содержать код, выполняемый при нажатии на нее, а код, про- изводящий работу с базой аккаунтов, не должен рисовать таблицу на экране смартфона .

MVC позволяет достичь максимального разделения трех основных категорий, что впоследствии позволяет удобно обновлять и пере- рабатывать программу, а также повторно использовать отдельные компоненты . Так, например класс, который обеспечивает отображение кнопки, без труда может быть дополнен, расширен и многократно ис- пользован в любых приложениях .

Все возможности по разработке iOS-приложений предоставляет iOS SDK (software development kit, комплект средств разработки), который входит в состав Xcode . Данный SDK предоставляет огромное число ресурсов, благодаря которым вы можете строить UI, организовывать мультитач управление, хранение данных в БД, работу с мультимедиа, передачу данных по сети, использование функций устройств (аксе- лерометр и т . д .) и многое-многое другое . В состав iOS SDK входит фреймворк Cocoa Touch, который как раз построен на принципе MVC .

Во время разработки приложений в Xcode вы работали с категорией

«Отображения» с помощью InterfaceBuilder, но при этом не обеспечи- вали решения каких-либо бизнес-процессов с помощью графических элементов .

Категория «Контроллер» включает в себя специфические классы, обеспечивающие функциональность ваших приложений, например UIViewController, а точнее, его потомок ViewController, с которым вы работали ранее .

# **13. Разработка интерфейса приложения в Interface Builder: макеты и автоматическая компоновка AutoLayout) и связь с кодом**

Создание базового UI

Предварительные настройки:

1.**Editor > Canvas** и **Show Bounds Rectangles**

Открыть библиотеку объектов в разделе утилит или с помощью меню **View > Utilities > Show Object**

AutoLayout

**Auto Layout** занимается динамическим вычислением позиции и размера всех view в view иерархии, на основе *constraints* — правил заданных для того или иного view.

Самый большой и очевидный плюс для разработчика в использовании Auto Layout в

том, что исчезает необходимость в подгонке размеров приложения под определенные устройства — Auto Layout делает это за вас, динамически изменяя интерфейс в зависимости от внешних или внутренних изменений. Auto Layout самостоятельно вычисляет расположение элемента интерфейса в приложении и изменяет его относительно окружения.

Если вы по каким-либо причинам не хотите использовать правила(constraints) или ваш интерфейс содержит в себе множество элементов расположение которых можно изменять бесконечно, на помощь придет Stack View.

Он может расставлять элементы внутри себя с данными параметрами:

**axis** (только UIStackView) — определяет ориентацию, горизонтально или вертикально;

**orientation** (только NSStackView) — тоже что и axis у UIStackView;

**distribution** — определяет расположение элементов в данной ориентации; **alignment** — определяет расположение элементов перпендикулярно ориентации StackView;

**spacing** — определяет расстояние между соседними элементами;

**Constraints**

По сути constraint - это выражение, задающее расположение элемента интерфейса. Выглядит это приблизительно так:

*Кнопка.Верх = ВысшаяТочкаИнтерфейса.Низ + 100*

Вычисления Auto Layout всегда происходят относительно ближайших элементов, будь это граница экрана или соседняя кнопка.

В своих вычислениях *constraints* используют множители, ближайшие объекты и константы. Также при создании правил не обязательно, чтобы это были равенства, можно использовать >= или <=.

При создании *layout* желательно указывать два правила для каждого измерения на элемент.

При создании *constraints* можно устанавливать приоритетность самих *constraints*. При вычислении, Auto Layout старается удовлетворить все *constraint*'ы в порядке приоритетности. Приоритет = 1000 — обязателен. Все остальные, менее приоритетные правила можно устанавливать для придания четкости обработке расположения элементов вашего интерфейса. В случае, если один из *constraint*'ов будет неправильно вычислен, Auto Layout использует ближайший *constraint* и начнет отталкиваться от него.

Создавать *constraint*'ы можно 3-мя способами:

**CTRL + Перетаскивание, например, от *label* к верхней границе.**

**Используя Stack, Align, Pin и Resolve Tools.**

Stack — кнопка, с помощью которой вы можете поместить выделенные детали интерфейса в StackView.

Align — меню, которое позволит вам установить элементы четко по определенной линии, будь то сбоку, вертикально по центру или снизу.

Pin — меню, позволяющее вам задать жесткие рамки относительно своего размера или ближайшего предмета.

Resolve Tools — самый лучший помощник в отладке *constraint*'ов. Основные возможности этого меню: убрать все правила, добавить предположительные *constraints*(Interface Builder построит все правила за вас), добавить отсутствующие *constraints*, обновить *constraints* или *frames*(положение объектов).

**Предоставить *Interface Builder* построить constraints вместо вас.**

[**https://habr.com/post/312782/**](https://habr.com/post/312782/)

# **14. Язык программирования Objective-C: структура проекта согласно модели КИС**

**Модель КИС**

**Репозиторий** — рабочий каталог, в котором находятся исходники, сценарии сборки (Makefile) и прочие файлы, относящиеся к проекту.

**Модель КИС** (Клиент-Интерфейс-Сервер) — это элегантная концепция распределения исходного кода в репозитории, в рамках которой все исходники можно поделить на *клиенты*, *интерфейсы* и *серверы*.

* Сборка (building) программного проекта – набор правил и процедур, направленный на получение исполняемой программы
* Выпуск (release) программного продукта – процесс отчуждения программы от разработчика и заключающийся в
  + Сборке программного проекта
  + Формировании инсталляционного пакета
  + Формировании документации
  + Формировании аннотации релиза

Задачи сборки и выпуска проектов

* Трансляция всего проекта
* Сборка дистрибутива
* Подготовка исходных текстов
* Подготовка документации Причины сборки проектов
* Проверка работоспособности
* Очередная периодическая сборка
* Подготовка версии к автоматическому тестированию
* Подготовка дистрибутива
* Инсталляция дистрибутива (из исходных текстов)

Проблемы сборки проектов

* Проблемы с исходными текстами
* Проблемы с подключаемыми файлами
* Проблемы с используемыми библиотеками
* Проблемы с процедурами сборки
* Проблемы со средствами (утилитами) сборки
* Проблемы с системной средой и аппаратной платформой сборки

Окружение для сборки

* Аппаратная платформа
* Системное окружение
  + Операционная система
  + Системные файлы
* Библиотечное окружение
  + Подключаемые файлы
  + Библиотечные файлы
* Исходные файлы в требуемых каталогах
* Средства сборки
* Построение окружения
  + Установка всех требуемых библиотек
    - Из дистрибутивов (версия разработчика)
  + Установка всех средств сборки
    - Из дистрибутива
  + Размещение всех исходных файлов
    - Из СКВ
    - Из архива

Общие требования для сборки

* Сборка должна проводиться на любом компьютере с подготовленным окружением
* Сборка должна проводиться отдельно от рабочего места разработчика
  + Версии подключаемых файлов
  + Версии библиотек
  + Расположение исходных файлов
  + Версии средств сборки
  + Влияние временных файлов
    - Объектные файлы
    - Исполняемые файлы
    - Предкомпилированные заголовки
    - И т.п.
* Процедура сборки должна быть:
  + Документирована
  + Прозрачна
  + Повторяема

Способы сборки программных проектов

* С помощью среды разработки
* С помощью запуска компилятора командной строки
* С помощью сценариев
* С помощью различных make tools
  + make
  + nmake
  + Ant
  + Maven
  + И т.п.

Сборка компонента проекта

* Для каждого компонента проекта должна быть сформирована локальная процедура сборки
  + Например, makefile
* В каждом компоненте проекта сборка должна проходить одной командой
  + Например, make Требования к процедурам сборки
* Процедуры сборки должны
  + находиться под управлением СКВ
  + помечаться и в
  + етвиться аналогично файлам проекта
  + изменяться в соответствии с изменениями проекта Утилиты сборки
* Make tools:
  + GNU Make
  + Nmake
  + Ant
  + Maven
  + ...
* Генераторы make-файлов
  + GNU Automake
  + Makedep
  + Makedepend Средства сборки
* Средства управления сборкой:
  + IBM Rational Build Forge
  + FinalBuilder
  + TeamCity
  + Visual Build
  + Codefast PerfectBuild
  + OpenMake
  + Anthill Pro
  + ...

# **15. Язык программирования Objective-C, базовый синтаксис языка, строковые объекты и коллекции**

**Objective-C** — компилируемый объектно-ориентированный язык программирования корпорации Apple, построенный на основе языка Си и парадигм Smalltalk.

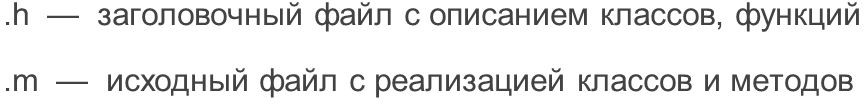
Совместим с С

Используется отправка сообщений в стиле Smalltalk

Компилятор Objective-C входит в GCC

Используется для Mac OS X (Cocoa) и GNUstep

Структура именования файлов в Objective-C:



Синтаксис Objective-C:

Директива #import.

Для обозначения объектов используется специальный тип id.

Для обозначения нулевого указателя на объект используется константа nil.

Для посылки сообщений используется следующий синтаксис:

[receiver message]; receiver - указатель на объект message - имя метода

Все директивы, используемые Objective-C начинаются с символа @

Строки в Objective-C:

Строковые объекты представляют собой символьные строки в Cocoa frameworks.

Строковый объект реализован в виде массива символов Юникода.

класс NSString — для создания и управления неизменяемой строкой:

метод length

метод characterAtIndex:

класс NSMutableString — для создания и управления изменяемой строкой

Коллекции в Objective-C:

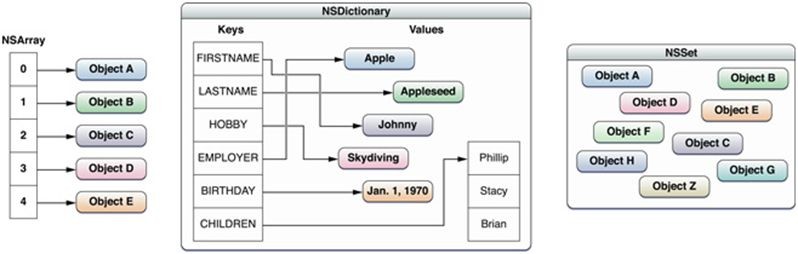
Большинство коллекций содержат только объекты и могут быть

**изменяемыми** (mutable) и **неизменными**.

Обладают рядом общих задач:

Перечисление объектов в коллекции

Определение, находится ли объект в коллекции

Доступ к отдельным элементам в коллекции

Изменяемые (mutable) коллекции позволяют рассмотреть также некоторые дополнительные задачи:

Добавление объектов в коллекцию

Удаление объектов из коллекции

Доступ по индексам и легкое перечисление элементов: Массивы (например, NSArray и NSMutableArray)

Ассоциирование данных с произвольным ключем: Словари (например NSDictionary и NSMutableDictionary)

Предоставление быстрых вставки, удаления и проверки членства: Наборы (например, NSSet, NSMutableSet и NSCountedSet)

Хранение подмножества массива: Индексные наборы (например, NSIndexSet и NSMutableIndexSet)

# **16. Язык программирования Objective-C, классы и протоколы**

Классы

Объявление классов в .h-файле: @interface имя\_класса : super\_Class

{

объявление переменных; (имеют зону видимости private)

}

объявление методов;

@end

Реализация класса создается в файле с расширением .m:

#import "имя\_класса.h" @implementation имя\_класса

реализация методов; @end

Протоколы

Протоколы описывают методы, которые могут быть реализованы каким-либо классом. Цели для которых используются протоколы:

Ожидание, что класс поддерживающий протокол выполнит описанные в протоколе функции

Поддержка протокола на уровне объекта, не раскрывая методы и реализацию самого класса

Ввиду отсутствия множественного наследования — объединить общие черты нескольких классов

Описание протокола:

@protocol ProtocolName объявление метода; @end

Протокол может быть унаследован от произвольного количества других протоколов: @protocol MyProtocol <Protocol1, Protocol2, Serializable, Drawable>

Добавление поддержки протокола классу: #import "protocolName.h"

@interface MyClass : SuperClass <protocolName> описание класса

@end

В файле реализации класса реализовать все методы протокола: @protocol asdfg

(void) readMy;

@optional

(void) saveReadMy;

(NSString\*) loadReadMy;

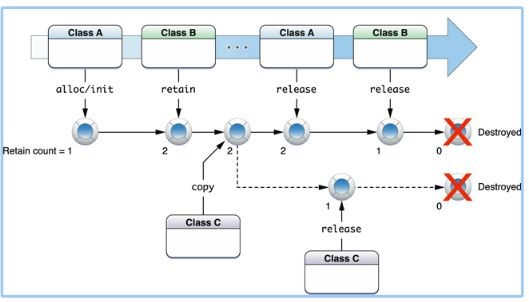
@required

(NSInteger) countReadMy; @end

Управление памятью

"manual retain-release" (ручное сохранение-освобождение) или MRR.

Automatic Reference Counting (автоматический подсчет ссылок) или ARC.

Garbage collection (сбор мусора)

# **17. Язык программирования Swift, базовые операторы, переменные и константы, правила объявления переменных и констант, вывод текстовой информации (NSLog, println, print) и комментарии.**

**Swift** — это быстрый, современный, безопасный и удобный язык программирования. Используется для разработки приложения для платформ:

iOS

macOS

watchOS

tvOS

**Выражение** — завершенная команда на языке Swift.

Выражения состоят из операторов, модификаторов и других выражений.

**Оператор** — это минимальная автономная единица, выполняющая определенную команду.

**Модификаторы** — это функциональные единицы, расширяющие возможности операторов.

**Объявление** — это создание нового объекта, с которым планируется взаимодействие.

**Инициализация** — это присвоение объявленному объекту определенного значения.

**Операторы** — это минимальные автономные функциональные единицы, выполняющие некоторую команду.

Операторы классифицируются на **простые** и **структурные**:

Простые (унарные и бинарные)

Структурные (тернарные)

**Оператор присваивания** — особый бинарный оператор. Он используется в типовом выражении a = b, инициализирует значение объекта a значением объекта b.

**Переменная** — это некая именованная область (хранилище), в которой может храниться значение определенного типа данных.

Объявление переменной:

*var имяПеременной = значениеПеременной*

Объявление константы:

*let имяКонстанты = значениеКонстанты*

Правила объявления переменных и констант:

Переменные и константы следует именовать в стиле **camelCase**: myBestText, theBestCountry, highScore

Имена переменных должны быть **уникальными**

Использование зарезервированных слов в качестве переменной:

var `var` = "Пример переменной в апострофах"

**Область видимости** определяет, где именно в коде можно использовать тот или иной объект.

Глобальные объекты — это объекты, доступные в любой точке программы.

Локальные объекты — это объекты, доступные в пределах родительского объекта.

**Вывод информации**

Функция **print** печатает сообщения в консоли Xcode при отладке приложений.

Функция **println** - это вариация, которая была удалена в Swift 2 и больше не используется

Функция **NSLog**:

NSLog работает медленнее,

NSLog добавляет метку времени и идентификатор к выходу, тогда как print не будет;

Операторы NSLog отображаются как на консоли устройства, так и на консоли отладчика, тогда как print отображается только в консоли отладчика.

NSLog использует строки стиля printf, например: NSLog("%0.4f", CGFloat.pi)

**Стандартные комментарии:**

// это — однострочный комментарий

/\* это - многострочный комментарий \*/

Комментарии в markdown стиле:

//: комментарий

# **18. Язык программирования Swift, числовые и текстовые типы данных, приведение числовых и текстовых типов, объединение строк и свойства строкового типа данных**

Неявное определение типа

var someText = "Свифт"

Явное определение типа

// объявляем пустую переменную var dragonsName: String

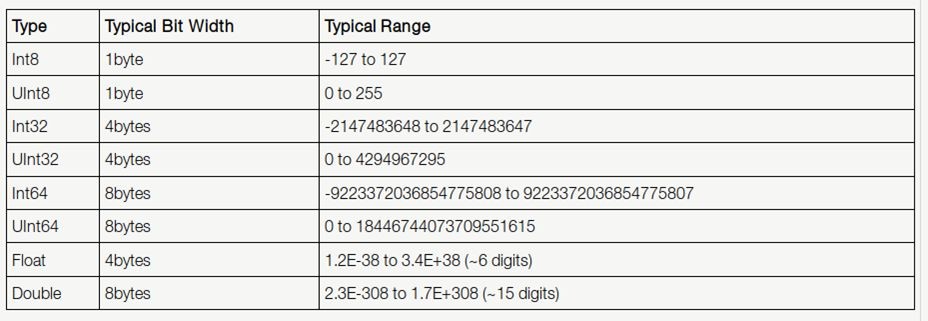
|  |  |  |  |
| --- | --- | --- | --- |
| // создаем непустую | переменную orksName | с | неявным |
| определением типа |  |  |  |
| var orksName = "Орк | " |  |  |
| // создаем непустую | переменную elfsName | с | явным |
| определением типа |  |  |  |

var elfsName: String = " Эльф "

Копирование типа данных

var groupOne = 23

var groupOneCopy = groupOne print(groupOneCopy) groupOneCopy = 25 print(groupOneCopy) print(groupOne)



// объявим переменную знакового целочисленного типа var signedNum: Int

// и присвоим ей значение signedNum = -32

// объявим переменную беззнакового целочисленного типа

var unsignedNum: UInt = 128

// минимальное значение параметра типа Int8 var minInt8 = Int8.min

// дробное число типа Float с явным указанием типа

var numFloat: Float = 104.3

// пустая константа типа Double let numDouble: Double

// дробное число типа Double с неявным указанием типа

var someNumber = 8.36

Swift — арифметические операторы

Бинарный оператор сложения — ( a + b ) Унарный оператор сложения — +a Бинарный оператор вычитания — ( a - b ) Унарный оператор “минус” — -a

Бинарный оператор умножения — ( a \* b ) Бинарный оператор деления — ( a / b )

Swift — приведение числовых типов

Int(\_:) — Преобразование в тип данных Int. Double(\_:) — Преобразование в тип данных Double. Float(\_:) — Преобразование в тип данных Float.

// переменная типа Int var numInt = 19

//переменная типа Double var numDouble = 3.13

// операция перемножения чисел

var resD = Double(numInt) \* numDouble var resI = numInt \* Int(numDouble) Swift — составной оператор присваивания

/ переменная типа Int var someNumInt = 19

// прибавим к ней произвольное число

someNumInt += 5

/\* эта операция аналогична выражению someNumInt = someNumInt+5 \*/

// умножим его на число 3 someNumInt \*= 3

/\* эта операция аналогична выражению

someNumInt = someNumInt\*3 \*/

Swift — текстовые типы данных

Character — предназначен для хранения отдельных символов

String — предназначен для хранения произвольной текстовой информации

var char: Character = "a"

// переменная типа String var stringOne = "Dragon" Пустые строковые литералы

// с помощью пустого строкового литерала var emptyString = ""

// с помощью инициализатора типа String

var anotherEmptyString = String()

Swift — объединение строк (интерполяция)

// переменная типа String var name = "Дракон"

// константа типа String c использованием интерполяции

let hello = "Привет, меня зовут \(name)!"

// интерполяция с использованием выражения

var meters: Double = 10

let text = "Моя длина \(meters \* 3.28) фута" "Моя длина 32.8 фута"

Swift — объединение строк (конкатенация)

// константа типа String let firstText = "Мой вес "

// переменная типа Double var weight = 12.4

// константа типа String let secondText = " тонны"

// конкатенация строк при инициализации значения новой переменной

var text = firstText + String(weight) + secondText

Swift — коллекция символов в строке

// переменная типа String var str = "Hello, Troll!"

// получаем коллекцию символов var collection = str.characters

// количество символов в строке collection.count

# **19. Язык программирования Swift, логические значения, логический тип данных и логические операторы, псевдонимы типов, операторы сравнения и диапазона**

В Swift существует специальный логический тип данных, называемый Bool и способный хранить одно из двух значений: «истина» или «ложь» . Значение

«истина» обозначается как true, а «ложь»— как false. Объявим переменную и константу логического типа данных (листинг 5 .32) .

Листинг 5.32

// логическая переменная с неявно заданным типом var isDragon = true

// логическая константа с явно заданным типом let isTroll: Bool = false

Как и для других типов данных в Swift, для Bool возможно явное и неявное определение типа, что видно из приведенного примера .

Логические операторы

Логические операторы проверяют истинность какого-либо утверждения и возвращают соответствующее логическое значение. Swift поддерживает три стандартных логических оператора:

логическое НЕ (!a); логическое И (a&&b); логическое ИЛИ (a||b) .

Swift предоставляет возможность создания псевдонима для любого типа данных.Псевдонимом типа называется дополнительное имя, по которому будет происходить обращение к данному типу. Для этого используется оператор typealias. Псевдоним необходимо применять тогда, когда существующее имя типа неудобно использовать в контексте программы (листинг 5 .39) .

Листинг 5.39

// определяем псевдоним для типа UInt8 typealias ageType = UInt8

/\* создаем переменную типа UInt8, используя псевдоним \*/ var myAge: ageType = 29

В результате будет создана переменная myAge, имеющая значения типа UInt8.

У типа может быть произвольное количество псевдонимов . И все псевдонимы вместе с оригинальным названием типа можно использовать в программе Созданный псевдоним обладает теми же самыми возможностями, что и сам тип данных . Однажды объявив его, вы сможете использовать данный псевдоним для доступа к свойствам и методам типа

(листинг 5 .41) .

Листинг 5.41

// объявляем псевдоним typealias ageType = UInt8

/\* используем свойство типа UInt8 через его псевдоним \*/ var maxAge = ageType.max

Для Swift обращение к псевдониму равносильно обращению к самому типу данных. Псевдоним — это ссылка на тип . В данном примере используется псевдоним maxAge для доступа к типу данных UInt8.

Операторы сравнения

Swift позволяет производить сравнение однотипных значений друг

с другом . Для этой цели используются операторы сравнения, результатом работы которых является значение типа Bool. Всего существует шесть стандартных операторов сравнения:

Бинарный оператор эквивалентности (a==b) возвращает true,когда значения обоих операндов эквивалентны .

Бинарный оператор неэквивалентности (a!=b) возвращает true,когда значения операндов различны .

Бинарный оператор «больше чем» (a>b) возвращает true , когда значение первого операнда больше значения второго операнда .

Бинарный оператор «меньше чем» (a<b) возвращает true, когда значение первого операнда меньше значения второго операнда .

Бинарный оператор «больше или равно» (a>=b) возвращает true,когда значение первого операнда больше значения второго операнда или равно ему. Бинарный оператор «меньше или равно» (a<=b) возвращает true,когда значение первого операнда меньше значения второго операнда или равно ему

.

Каждый из приведенных операторов возвращает значение, указывающее на справедливость некоторого утверждения .

В Swift существуют специальные операторы, с

помощью которых можно объединить множество последовательных числовых значений (например, 1, 2, 3, 4), то есть создать диапазон (например, от 1 до 4) .

Такие операторы называются операторами диапазона (range operators) .

Swift предлагает два оператора диапазона .

Закрытый оператор a...b

определяет диапазон, который содержит все числа от a до b , включая a и b. Например, для целочисленного типа диапазон 1...4 включает в себя числа 1, 2, 3, 4 .

Полуоткрытый оператор a..<b

определяет диапазон чисел от a до b, включая только a. Например, для целочисленного типа диапазон

2..<5

включает в себя числа 2, 3, 4 .

Односторонний оператор a.. a>. <b ..b

определяет диапазон чисел, ограниченный лишь с одной из сторон . Вторая сторона вычисляется автоматически . Одним из примеров может служить диапазон символов в строке с третьего до последнего . Его особенность заключается в том, что длина строки изначально неизвестна .

# **20. Язык программирования Swift, кортежи (объявление кортежа, тип данных и взаимодействие с элементами кортежа)**

**Swift** — это быстрый, современный, безопасный и удобный язык программирования. Используется для разработки приложения для платформ:

iOS

macOS

watchOS

tvOS

**Кортеж** — это особый объект, который группирует значения различных типов в пределах одного составного значения.

Синтаксис

(значение\_1, значение\_2, )

let(или var) имяКонстанты = (значение\_1, значение\_2, ...)

Пример объявления кортежа

let myProgramStatus = (200, "In Work", true)

Тип данных кортежа — это фиксированный упорядоченный набор типов данных входящих в него значений, который записывается в скобках и элементы которого отделяются запятыми друг от друга.

// объявляем кортеж с явно заданным типом let myProgramStatus: (Int, String, Bool) = (200, "In Work", true)

Вместо того чтобы сразу инициализировать можно инициализировать после объявления кортежа с явно заданным типом так:

myProgramStatus = (200, "In Work", true)

Взаимодействие с элементами кортежа

// записываем значения кортежа в переменные var (statusCode, statusText,

statusConnect) = myProgramStatus

И взаимодействуем с полученными varами

Взаимодействие с элементами кортежа при передаче набора значений

var (myName, myAge) = ("Иван", 40)

Игнорирование элементов кортежа

/\* получаем только необходимые значения кортежа \*/

var (statusCode, \_, \_) = myProgramStatus

Индексы для доступа к элементам кортежа

// выводим информацию с использованием индексов print("Код ответа — \(myProgramStatus.0)") print("Текст ответа — \(myProgramStatus.1)") print("Связь с сервером — \(myProgramStatus.2)")

Имена для доступа к элементам кортежа

let myProgramStatus = (statusCode: 200, statusText: "In Work", statusConnect: true)

// выводим информацию с использованием print(" Код ответа —

\(myProgramStatus.statusCode)") print(" Текст ответа —

\(myProgramStatus.statusText)") print(" Связь с сервером —

\(myProgramStatus.2)")

/\* объявляем кортеж с указанием имен элементов в описании типа \*/

let myProgramStatus: (statusCode: Int,

statusText: String, statusConnect: Bool) = (200, "In Work", true)

Изменение значений кортежа

// объявляем пустой кортеж

var myFirstTuple: (Int, String)

// создаем кортеж со значением var mySecondTuple = (100, "Код")

// передаем значение кортежа myFirstTuple = mySecondTuple

// объявляем кортеж

var someTuple: (200, true)

// изменяем значение отдельного элемента someTuple.0 = 404

someTuple.1 = false

Сравнение кортежей

(1, "alpha") < (2, "beta")

// истина, так как 1 меньше 2.

// вторая пара элементов не учитывается (4, "beta") < (4, "gamma")

// истина, так как "beta" меньше "gamma". (3.14, "pi") == (3.14, "pi")

//истина, так как все соответствующие элементы идентичны

# **21. Язык программирования Swift, массивы (объявление, типы данных, доступ к элементам массива, сравнение и слияние массивов)**

**Массив** — это упорядоченная коллекция однотипных элементов, для доступа к которым используются индексы (номера) . Упорядоченной называется коллекция, в которой элементы располагаются в порядке, заложенном разработчиком.

Значение массива задается с помощью литерала массива, в котором перечисляются входящие в состав элементов значения .

СИНТАКСИС

[значение\_1, значение\_2, ...,значение\_N]

Литерал массива указывается в квадратных скобках, а значения отдельных элементов в нем разделяются запятыми. Массив может содержать произвольное количество элементов одного типа. Массивы хранятся в переменных и константах, поэтому для их объявления используются операторы var и let.

Типом массива является тип-значение (value type), а не тип-ссылка (reference type) . Это означает, что при попытке скопировать или передать значение массива создается его копия, с которой и происходит вся дальнейшая работа.

Вы можете создать массив с помощью функции Array(arrayLiteral:), или, точнее, инициализатор типа данных Array.

СИНТАКСИС

Array(arrayLiteral: значение\_1, значение\_2, ...,значение\_N)

Литерал массива передается в виде входных параметров функции Array(arrayLiteral:).

Пример создания массива:

let alphabetArray = ["a", "b", "c"]

let alphabetArray = Array(arrayLiteral:"a", "b","c")

При необходимости создания массива, состоящего из нескольких последовательных целочисленных значений, можно использовать оператор диапазона

.

СИНТАКСИС

Array(значение\_1...значение\_N) Array(значение\_1..<значение\_N) example: let lineArray = Array(0...9)

Третий способ создания массива используется в том случае, когда необходимо объявить массив, состоящий из нескольких одинаковых значений .

СИНТАКСИС

Array(repeating: значение, count: количество)

**Тип данных массива**

Тип данных массива всегда основывается на типе данных входящих в него элементов . Существует два способа написания типа данных массива .

СИНТАКСИС

Полная форма записи: Array<ТипДанных> Краткая форма записи: [ТипДанных]

Обе формы имеют один и тот же смысл: они указывают, что массив имеет тип данных [типДанных] или Array<типДанных> и состоит из элементов, тип данных которых — типДанных.

При необходимости можно явно указать тип данных массива. СИНТАКСИС

Полная форма записи: var имяМассива: Array<ТипДанных> Краткая форма записи: var имяМассива: [ТипДанных]

**Доступ к элементам** массива Для доступа к отдельному элементу массива необходимо использовать индекс данного элемента, заключенный в квадратные скобки и указанный сразу после имени массива. Способ доступа к элементам с использованием ключевых наименований, в данном случае индексов, называется сабскриптом.

При использовании оператора диапазона можно получить доступ сразу ко множеству элементов в составе массива, то есть к его подмассиву. Данный оператор должен указывать на индексы крайних элементов выделяемого множества

**Сравнение массивов**

Массивы, так же как и значения фундаментальных типов данных, можно сравнивать друг с другом . Два массива являются эквивалентными:

* если количество элементов в сравниваемых массивах одинаково;
* каждая соответствующая пара элементов эквивалентна .

let a1 = 1 let a2 = 2 let a3 = 3

if [1, 2, 3] == [a1, a2, a3] {

print("Массивы эквивалентны")

} else {

print("Массивы не эквивалентны")

}

Консоль:

Массивы эквивалентны

**Слияние массивов**

Со значением массива, как и со значениями фундаментальных типов данных, можно проводить различные операции . Одной из них является операция слияния, при которой значения двух массивов сливаются в одно, образуя новый массив . Обратите внимание на несколько моментов:

* результирующий массив будет содержать значения из обоих массивов, но индексы этих значений могут не совпадать с родительскими;
* значения элементов подлежащих слиянию массивов должны иметь один и тот же тип данных .

Операция слияния производится с помощью уже известного оператора сложения (+)

let charsOne = ["a", "b", "c"]

let charsTwo = ["d", "e", "f"]

let charsThree = ["g", "h", "i"] var alphabet = charsOne + charsTwo

alphabet += charsThree //["a", "b", "c", "d", "e", "f", "g", "h", "i"]

# **22. Язык программирования Swift, массивы (многомерные массивы, базовые свойства и методы массивов)**

Элементами массива могут быть не только значения фундаментальных типов, но и другие массивы Массивы, содержащие в себе другие массивы, называются **многомерными.** Необходимо обеспечить единство типа всех вложенных массивов.

*var arrayOfArrays = [[1,2,3], [4,5,6], [7,8,9]]*

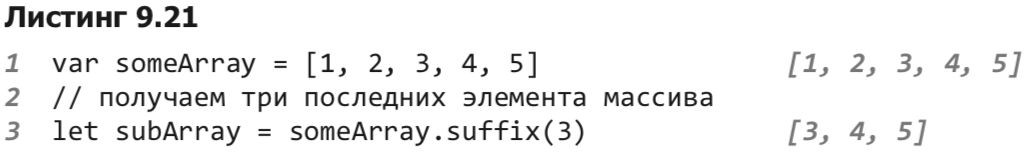
В данном примере создается массив, содержащий массивы типа [Int] в качестве своих элементов Типом основного массива arrayOfArrays является [[Int]] (с удвоенными квадратными скобками с каждой стороны)

Для доступа к элементу многомерного массива необходимо указывать несколько индексов, например arrayOfArrays[2][1].

Базовые свойства и методы массивов

* 1. Свойство count возвращает количество элементов в массиве
  2. Если значение свойства count равно нулю, то и свойство isEmpty воз- вращает true
  3. Вы можете использовать свойство count для того, чтобы получить требуемые элементы массива, например var newArray = someArray[someArray.count-3... someArray.count-1]
  4. Другим средством получить множество элементов массива явля- ется

метод suffix(\_:) — в качестве входного параметра ему передается количество элементов, которые необходимо получить Элементы отсчитываются начиная с последнего элемента массива, например



* 1. Свойства **first** и **last** возвращают первый и последний элементы массива
  2. С помощью метода **append(\_:)** можно добавить новый элемент в конец массива
  3. Если массив хранится в переменной (то есть является изменяемым), то метод **insert(\_:atIndex:)** вставляет в массив новый одиночный элемент с указанным индексом. При этом индексы массива пересчитываются, чтобы обеспечить их последовательность
  4. Так же как в случае изменения массива, методы **remove(at:)**, **removeFirst()** и **removeLast()** позволяют удалять требуемые элементы При этом они возвращают значение удаляемого элемента. После удаления индексы оставшихся элементов массива перестраиваются.
  5. Если массив является неизменяемым (хранится в константе), то мож-но использовать методы **dropFirst(\_:)** и **dropLast(\_:)**, возвращающие новый массив, в котором отсутствует несколько первых или послед- них элементов.
  6. Метод **contains(\_:)** определяет факт наличия некоторого элемента в массиве и возвращает Bool в зависимости от результата.
  7. Метод **index(of:)** сообщает индекс первого вхождения искомого эле- мента в рассматриваемом массиве Так как искомый элемент может от- сутствовать, метод возвращает опциональное значение. Если элемент отсутствует, то возвращается **nil**.
  8. Для поиска минимального или максимального элемента в массиве применяются методы **min ()** и **max ()** Данные методы работают толь- ко в том случае, если элементы массива можно сравнить между собой.

*ПРИМЕЧАНИЕ Сопоставлять можно значения тех типов данных, которые являются хешируемыми, то есть в них должен присутствовать функционал вычисления хеша для значения. Большинство фундаментальных типов данных поддерживают хеширование. Хеш обеспечивает возможность сравнения (сопоставления) различ- ных значений одного и того же типа. При его*

*вычислении собственное значение параметра по специальному алгоритму преобразуется в числовое значение и поме- щается в свойство hashValue параметра.*

* 1. Чтобы изменить порядок следования всех элементов массива на про- тивоположный, используйте метод **reverse()**

# **23. Язык программирования Swift, наборы (объявление, доступ и модификация набора, эквивалентность наборов)**

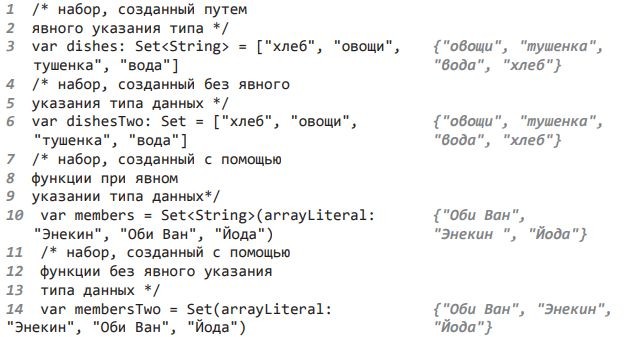
**Набор** — это неупорядоченная коллекция уникальных элементов.У элементов набора нет какого-либо определенного порядка следования, важен лишь факт наличия некоторого значения в наборе. Набор создается с помощью литерала набора. В плане синтаксиса он идентичен литералу массива, но при этом гарантирует отсутствие дубликатов значений.

СИНТАКСИС

**[значение\_1, значение\_2, ...,значение\_N]**

Литерал набора указывается в квадратных скобках, а значения отдельных элементов в нем разделяются запятыми. Литерал может содержать произвольное количество уникальных элементов одного типа.

*!!!!!!!Самое важное, чтобы при создании набора вы не забывали использовать ключевое слово Set, иначе в результате будет создан массив.!!!!!!*

Все возможные способы объявления набора:

Пустой набор, то есть набор, значение которого не имеет элементов, создается с помощью пустого литерала набора []. Вы также можете передать данный литерал для удаления всех существующих элементов набора.

Для добавления нового значения в наборе применяется метод **insert(\_:)**, которому передается создаваемое значение.

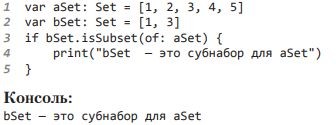
Для удаления элемента из набора используется метод **remove(\_:)**, который удаляет элемент с указанным значением и возвращает удаленное значение или *nil*, если удаляемого элемента не существует. Также вы можете задействовать метод **removeAll()** для удаления всех элементов набора.

Проверка факта наличия значения в наборе осуществляется методом **contains(\_:)**. Данный метод возвращает значение типа **Bool** в зависимости от результата проверки.

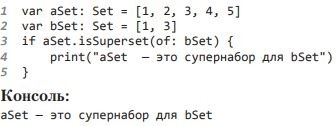
* получить все общие для обоих наборов элементы **(intersection )**;
* получить все непересекающиеся (не общие) для обоих наборов элементы (**symmetricDifference**);
* получить все элементы обоих наборов (**union**);
* получить разницу элементов, то есть элементы, которые входят в первый набор, но не входят во второй (**subtracting)**.

Эквивалентность наборов

Два набора считаются эквивалентными, если у них один и тот же набор элементов. Эквивалентность наборов проверяется с помощью оператора эквивалентности **(==).**

**М**етод **isSubset(of:)** определяет, является ли один набор субнабором другого, как bSet для aSet

Метод **isSuperset(of:)** вычисляет, является ли набор супернабором для другого набора



Метод **isDisjoint(with:)** определяет, существуют ли в двух наборах общие элементы, и в случае их отсутствия возвращает true

Методы **isStrictSubset(of:) и isStrictSuperset(of:)** определяют, является набор субнабором или супернабором, не равным указанному множеству

# **24. Язык программирования Swift, словари (объявление, взаимодействие с элементами словаря, явное указание типа данных, базовые свойства и методы словарей)**

##### Swift - это быстрый, современный и безопасный язык программирования.

Его используют для разработки приложений для платформ:

* iOS
* macOS
* watchOS
* tvOS

Словарь - это неупорядоченная коллекция элементов одного и того же типа, для доступа к значением которых используются ключи.

**Объявление словаря**

В Swift тип словаря в полной форме пишется как Dictionary<Key, Value>, где Key это тип значения который используется как ключ словаря, а Value это тип значения который словарь может хранить для этих ключей. Вы можете также написать словарь в сокращенной форме как [Key: Value].

Пример:



Во втором примере для объявления используется литерал словаря. Литерал словаря - это краткий способ написать одну или несколько пар ключ-значение в виде коллекций словаря.

##### Обновление значения

var countryDict = ["RUS":"Россия", "BLR": "Беларусь", "UKR":"Украина"]

// изменяем значение элемента countryDict.updateValue("Российская Федерация", forKey: "RUS")

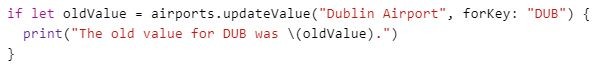
Метод updateValue(forKey:) возвращает опциональное значение соответствующее типу значения словаря. Например, для словаря, который хранит String значения, метод возвратит String? тип, или » опциональный String«. Это опциональное значение содержит старое значение для этого ключа, если оно существовало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

##### //Также можно обновить значение по ключу следующим способом

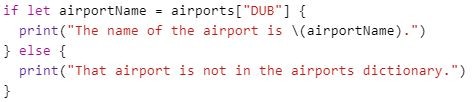
countryDict["RUS"] = "Московия"

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

Исходя из вышесказанного можно следующими способами проверить наличие пары ключа и значения в словаре:



После записи сверху по данному ключу уже будет находится значение, так был использован метод updateValue.



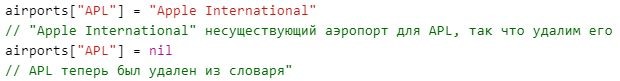
##### Добавление значения

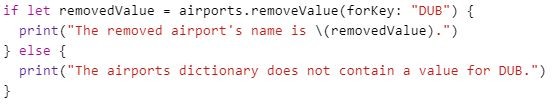
var countryDict = ["RUS":"Россия", "BLR": "Беларусь", "UKR":"Украина"]

// создание нового элемента словаря countryDict["POL"] = "Польша"

Удаление значения

Вы можете использовать синтаксис индексов для удаления пары

ключ-значение из словаря путем присвоения nil значению для этого ключа:

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

**Базовые свойства и методы**

##### Свойство count помогает определить количество элементов в словаре:

Свойство isEmpty помогает определить, пуст ли массив:



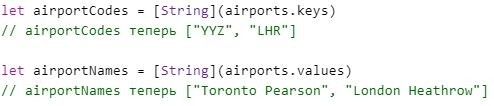
**Итерация по словарю**

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

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



Если вам нужно использовать ключи или значения словаря вместе с

каким-либо API, которое принимает объект Array, то можно инициализировать новый массив с помощью свойств keys и values:

# **25. Язык программирования Swift, опциональные типы данных и извлечение опционального значения**

**Опциональные типы данных** (опционалы) — это особый тип данных, который говорит о том, что некоторая переменная или константа либо имеет значение определенного типа, либо вообще не имеет никакого значения.

// переменная с числовым строковым литералом let possibleString = "1945"

/\* при попытке конвертации преобразуется в целочисленный тип \*/

let convertPossibleString = Int(possibleString)

// переменная со строковым литералом let unpossibleString = "Привет!"

/\* при попытке конвертации не преобразуется в целочисленный тип \*/

let convertUnpossibleString = Int(unpossibleString)

Обозначение опционального типа, например Int?

/\* переменная с опциональным типом Int и с установленным значением \*/

var dragonAge: Int? = 230

// уничтожаем значение переменной dragonAge = nil

Объявление параметра опционального типа с помощью функции

Optional

// опциональная переменная с установленным значением var optionalVar = Optional ("stringValue") optionalVar

// уничтожаем значение опциональной переменной optionalVar = nil

optionalVar

Извлечение опционального значения: принудительное и косвенное

/\* опциональная переменная

с установленным значением \*/ var trollAge: Int? = 95

trollAge = trollAge + 10 // ОШИБКА

Принудительное извлечение (forced unwrapping)

/\* опциональная переменная

с установленным значением \*/ var trollAge: Int? = 95

// проведение арифметической операции trollAge = trollAge! + 10

trollAge

Извлечение опционального значения: принудительное и косвенное

Косвенное извлечение (implicitly unwrapping)

var type: String

// принудительное извлечение опционального значения

let monsterOneType: String? = "Дракон" type = monsterOneType!

type

// косвенное извлечение опционального значения let monsterTwoType: String! = "Тролль"

type = monsterTwoType type

let pointCoordinates: (Int, Int)! = nil coordinates = pointCoordinates // ОШИБКА

# **26. Язык программирования Swift, функции и их возможности (объявление функций, аргументы функции и возвращаемое значение)**

**Функция** — это именованный фрагмент программного кода, к которому можно многократно обращаться .

Синтаксис

func имяФункции (входные\_параметры) -> ТипВозвращаемогоЗначения

{

тело\_функции

}

Список входных параметров заключается в круглые скобки и состоит из разделенных запятыми элементов . Каждый отдельный элемент описывает один входной параметр и состоит из имени и типа этого параметра, разделенных двоеточием .

Количество входных параметров может быть произвольным (также они могут вовсе отсутствовать) .

Аргументы будут существовать только в пределах тела функции . По окончании ее работы данные параметры будут уничтожены и станут недоступными .

После списка параметров следует стрелка (->), за которой указывается тип данных возвращаемого функцией значения . В качестве типа данных вы можете задать как фундаментальный тип, так и тип массива или кортежа .

Тело функции заключается в фигурные скобки и содержит в себе всю логику ее работы . Если функция возвращает какое-либо значение, то в ее теле должен присутствовать оператор return, за которым следует возвращаемое значение . После выполнения программой оператора return работа функции завершается и происходит выход из нее .

Процесс обращения к объявленной функции с целью получения ре- зультата ее работы называется вызовом функции .

Если функция не возвращает никакого значения (как в примере ра- нее), то в качестве типа данных возвращаемого значения необходимо указать тип Void .Существует альтернативная ключевому слову Void форма записи — пустые скобки, которые также сообщают о том, что функция не возвращает значение . Кроме того, если функция не возвращает никакого значения, на- писание какой бы то ни было конструкции можно вовсе опустить .

Вызов функции

myFunction(параметры)

Пример

func myFunc (param1: String, param2: Int) -> Int

{

print(param1) return param2+1

}

При вызове функции необходимо указывать имя для каждого параметра при передаче его значения . При этом значения для параметров передаются точно в том же порядке, в каком были описаны параметры при определении функции .

Существует способ сообщить Swift, что для определенных параметров нет необходимости указывать имена при функции . Для этого перед именем таких параметров необходимо поставить символ нижнего под- черкивания, отделив его от имени пробелом

Все входные параметры функции — константы . При попытке изме- нения их значения внутри тела функции происходит ошибка . При необходимости изменения переданного входного значения внутри функции потребуется создать новую переменную и присвоить пере- данное значение ей.

Для того чтобы входные аргументы сохранили свои значения даже после завершения вызова функции, необходимо использовать сквозные параметры .

Чтобы преобразовать входной параметр в сквозной, перед описанием его типа необходимо указать модификатор inout . Сквозной параметр передается в функцию, изменяется в ней и сохраняет свое значение при завершении работы функции, заменяя собой исходное значение . При вызове функции перед передаваемым значением аргумента не- обходимо ставить символ амперсанда (&), указывающий на то, что параметр передается по ссылке .

**Листинг 11.6**

func changeValues(\_ a: inout Int, \_ b: inout Int) -> () {

//какой-то код

}

var a = 150, b = 45 changeValues(&a, &b)

Вы можете использовать возвращаемое некоторой функцией значение в качестве значения входного аргумента другой функции .

print(generateString(code: 200, message: "Сервер доступен"))

Функция generateString(code:message:) возвращает значение типа String . В связи с этим в конце функции используется ключевое слово return с указанием возвращаемого параметра .

При знакомстве с кортежами мы говорили о том, что их сильной сторо- ной является возможность группировки нескольких значений в одном . Эту группу значений можно использовать в качестве возвращаемого значения функции .

Для любого входного параметра можно задать его внешнее имя, то есть имя, которое указывается при вызове функции . Оно пишется перед внутренним именем аргумента и отделяется от него пробелом

# **27. Библиотека UIKit Framework, её основные структуры, классы и протоколы**

##### **UI Framework** — библиотека для создания и управления графическим пользовательским интерфейсом для платформ iOS или tvOS.

<https://developer.apple.com/documentation/UIKit>

The UIKit framework provides the required infrastructure for your iOS or tvOS apps. It provides the window and view architecture for implementing your interface, the event handling infrastructure for delivering Multi-Touch and other types of input to your app, and the main run loop needed to manage interactions among the user, the system, and your app. Other features offered by the framework include animation support, document support, drawing and printing support, information about the current device, text management and display, search support, accessibility support, app extension support, and resource management.

гугл транслате:

Рамка UIKit обеспечивает необходимую инфраструктуру для iOS или приложения tvOS. Он предоставляет архитектуру окна и представления для

реализации вашего интерфейса, инфраструктуры обработки событий для доставку Multi-Touch и других типов ввода в ваше приложение, а также основной цикл запуска, необходимый для управления взаимодействием

между пользователем,

системы и вашего приложения. Другие функции, предлагаемые структурой

поддержка анимации, поддержка документов, рисование и печать поддержка, информация о текущем устройстве, управление текстом и отображение, поддержка поиска, поддержка доступности, поддержка расширения приложений,

и управление ресурсами

UIKit руководит взаимодействие вашего проекта с системой и обеспечивает классы, чтобы вы могли управлять данными и ресурсами вашего приложения

* *Структура приложения*
  + Core App - управляйте моделью данных вашего приложения и его взаимодействием с системой.
  + Resource Management - Процесс использования ресурсов компании в наиболее эффективном способе.
  + App Extensions - предоставляют пользователям доступ к функциям и контенту вашего приложения в iOS и macOS.
* *Пользовательский интерфейс*
  + Представления и взаимодействие (управление)
  + Контроллеры представлений
  + Макеты представлений
  + Анимация и Haptic Feedback
  + Окна и экраны
* *Взаимодействие с пользователем*
  + Прикоснования, нажатия и жесты (Touches, presses and gestures)
  + Drag and drop
  + Focus Interactions
  + Peek and pop
  + Меню и клавиатура
  + Accessibility
* *Графика, рисование и печать*
  + Images and pdf
  + Drawing
  + Printing
* *Текст*
  + Вывод текста и шрифты
  + Хранение текста
  + Клавиатура и методы ввода
* Структуры
  + struct UIDocumentBrowserError
* Классы
  + class UICollectionViewDropPlaceholder class UICollectionViewPlaceholder class UITableViewDropPlaceholder class UITableViewPlaceholder

**class UICollectionViewDropPlaceholder** - интерфейс для координации ваших пользовательских действий, связанных с кадром, с представлением коллекции.

Вы сами не создаете экземпляры этого класса. Когда в представлении коллекции появляется капля, UIKit создает экземпляр этого класса и передает его вашему методу. Используйте этот объект, чтобы представление коллекции узнало, как вы хотите анимировать упавшие элементы в нужное положение.

Капля? Да? Капля!

# **28. Контроллер представления и основные классы и методы**

К контроллерам представления относятся(ХОТЯ ХЗ ТО ЛИ ЭТО. В НЕТЕ НИЧЕГО, А ПРЕЗЕНТАШКАХ ТОЛЬКО ЭТА ТАБЛИЦА):

|  |  |
| --- | --- |
| **Категория** | **Элементы** |
| **Настраиваемые контроллеры представлений** | class UIViewController  class UITableViewController class UICollectionViewController protocol UIContentContainer |
| **Интерфейс Split View** | class UISplitViewController |
| **Интерфейс навигации** | class UINavigationController class UINavigationBar  class UINavigationItem |
| **Интерфейс Page View** | class UIPageViewController |

|  |  |
| --- | --- |
| **Интерфейс Tab View** | class UITabBarController class UITabBar  class UITabBarItem |
| **Интерфейс поиска** | class UISearchContainerViewController class UISearchController  class UISearchBar  protocol UISearchResultsUpdating |
| **Изображения и видео** | class UIImagePickerController class UIVideoEditorController |
| **Менеджер документов** | Adding a Document Browser to Your App Building a Document Browser-Based App class UIDocumentBrowserViewController class UIDocumentPickerViewController |

|  |  |
| --- | --- |
| **Просмотр документа** | class UIDocumentInteractionController |
| **Доступ к iCloud** | class UICloudSharingController |
| **Интерфейс Activities** | class UIActivityViewController class UIActivityItemProvider protocol UIActivityItemSource class UIActivity |
| **Интерфейс поиска слов** | class UIReferenceLibraryViewController |
| **Интерфейс ориентации** | func UIInterfaceOrientationIsPortrait(UIInterfaceOrientation) func UIInterfaceOrientationIsLandscape(UIInterfaceOrientation) |

**UIKit** — это набор основных возможностей, которые разработчики используют при построении интерфейса практически любого приложения.

В библиотеке **UIKit описан класс UIViewController**, предназначенный для описания инфраструктуры и управления отображениями приложения.

Основной функцией UIViewController (а также его потомков) является обновление содержимого отображений, в том числе в ответ на действия пользователей.

Два метода, описанных в классе ViewController, которые при создании приложения, используя шаблон Single View Application, автоматически будут переопределкны в коде класса ViewController: viewDidLoad() и didReceiveMemoryWarning()

Класс **ViewController** является **подклассом UIViewController**

Если в качестве шаблона приложения был выбран Single View Application, то класс ViewController создается автоматически(в файле ViewController.swift) **Для создания связи и организации взаимодействие кода и элементов интерфейса надо связать объект View Controller, отображенный в Document outline и входящий в состав сцены, и класс ViewController, описанный в файле ViewController. swift. Это позволит влиять на элементы интерфейса с помощью программного кода.**

**View Controller** в действительности не входит в состав сцены, он лишь обеспечивает ее отображение. **View Controller — это внешний объект, предназначенный для управления ассоциированной с ним сценой и набором отображений, входящих в эту сцену.**

или(еще одно определение)

View Controller **—** объект, от которого во многом зависит функционирование сцены. Он загружает сцену на экран вашего устройства, а также следит за происходящим на сцене. Дополнительно он обеспечивает эффективный и однозначный доступ к любому из объектов, созданных в пределах сцены (например, к кнопкам).

Средствами View Controller на экран вашего устройства загружается и отображается экземпляр класса UIView(который является визуальным макетом определяющим непосредственно саму сцену)

**Основные методы(если кто считает основными ещё какие методы можно сюда добавить, их слишком куча)**

Создание View Controller программно

init(nibName: String?, bundle: Bundle?)

**Возвращает новый инициализированный контроллер представления с nib файлом в указанном bundle(комплекте?).**

**init?(coder: NSCoder)**

**Создание view**

**(BOOL)isViewLoaded**

**loadView**

***viewDidLoad***

Обработка изменения состояния view

**viewDidLoad**

**viewWillAppear:(BOOL)animated**

**viewDidAppear:(BOOL)animated**

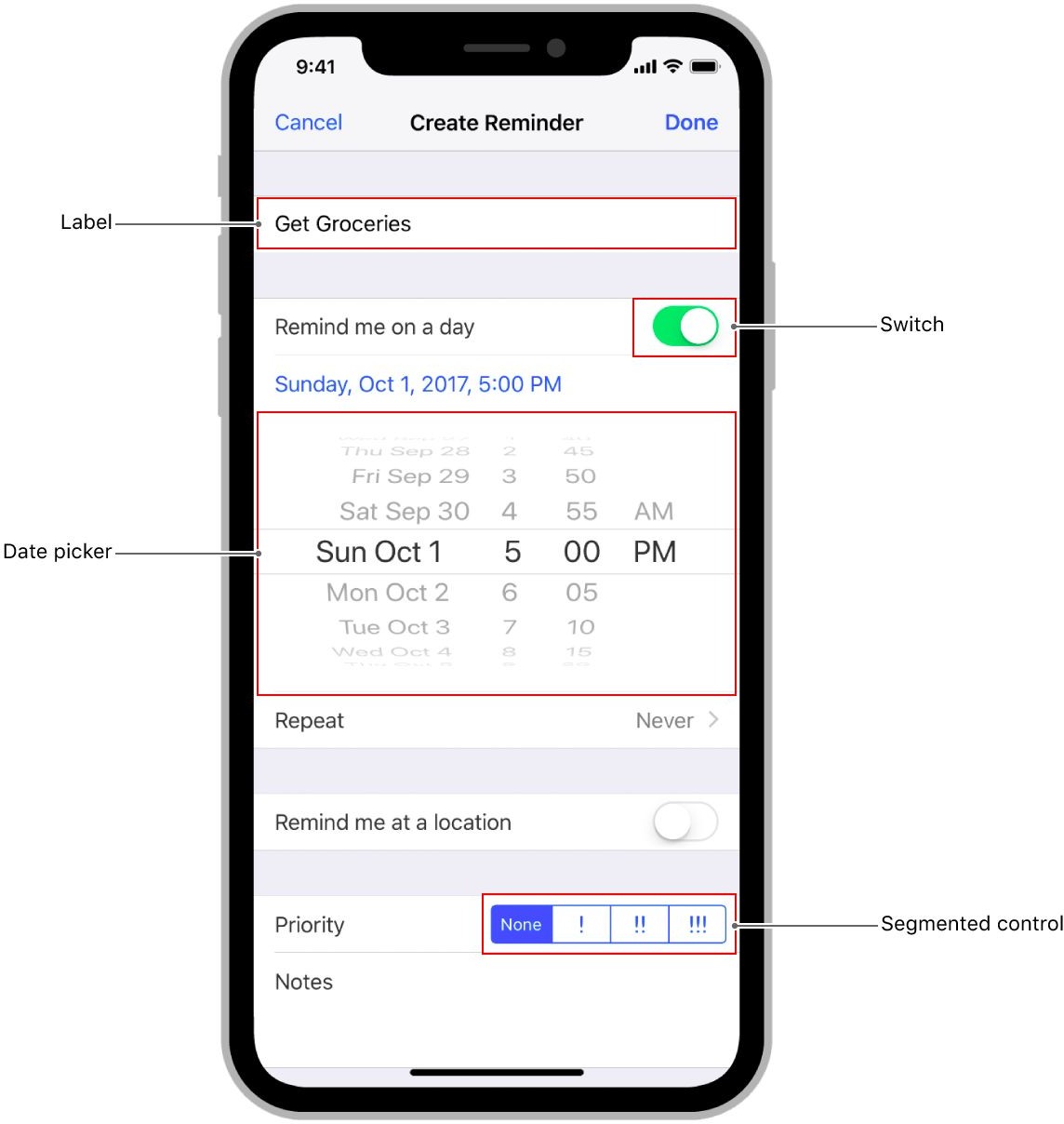
**viewWillDisappear:(BOOL)animated**

**viewDidDisappear:(BOOL)animated Обработка memory warning**

**didReceiveMemoryWarning**

**29. Представления View, их типы и элементы управления, примеры и основные классы**

Представления и управляющие элементы (Views and Controls)



[UIView](https://developer.apple.com/documentation/uikit/uiview) is the root class for all views and defines their common behavior.

[UIControl](https://developer.apple.com/documentation/uikit/uicontrol) defines additional behaviors that are specific to buttons, switches, and other views designed for user interactions.

**Категория Элементы**

**View Fundamentals**

class UIView

**Container Views** Collection Views

Table Views

class UIStackView class UIScrollView

**Content Views** class UIActivityIndicatorView

class UIImageView class UIPickerView class UIProgressView class UIWebView

**Управляющ ие элементы**

class UIControl class UIButton class UIDatePicker

class UIPageControl

class UISegmentedControl class UISlider

class UIStepper class UISwitch

**Text Views** class UILabel

class UITextField class UITextView

Drag and Drop Customization

**Визуальные эффекты**

class UIVisualEffect class UIVisualEffectView class UIVibrancyEffect class UIBlurEffect

**Панели** class UIBarItem

class UIBarButtonItem

class UIBarButtonItemGroup class UINavigationBar

class UISearchBar class UIToolbar class UITabBar class UITabBarItem

protocol UIBarPositioning

protocol UIBarPositioningDelegate

**Настройка внешнего вида**

protocol UIAppearance

protocol UIAppearanceContainer

[https://developer.apple.com/documentation/uikit/vie](https://developer.apple.com/documentation/uikit/views_and_controls) [ws\_and\_controls](https://developer.apple.com/documentation/uikit/views_and_controls)

# **30. Интернационализация приложений для платформы iOS**

**Интернационализация** — это процесс разработки и создания вашего приложения с обеспечением международной совместимости.

Интернационализация делает ваше приложение *локализуемым.*

**Локализация** — это процесс перевода пользовательского интерфейса и ресурсов приложения на разные языки.

Интернационализация выполняется разработчиком приложения и должна

обеспечивать 3 вещи:

правильную работу системы **Autolayout**

обертку литеральных строк, участвующих в формировании пользовательского интерфейса, в вызовы **NSLocalizedString** и **String.localizedStringWithFormat**

использование **NSNumberFormatter, NSDateFormatter, NSDateСomponentsFormatter** и других для представления чисел, дат, валюты, расстояния и т.д. на экране

Далее в **Xcode** работает система локализации, которая автоматически объединяет все, что подготовлено разработчиком для перевода, и экспортирует это в **.xliff** файлы, являющиеся индустриальным стандартом для «локализаторов».

Локализаторы, пользуясь своим инструментарием, возвращтают **.xliff** файлы с результатами перевода, которые импортируются назад в приложение. Используя режим **Preview** для storyboard, а также **Run** Схемы можно очень быстро просмотреть результаты локализации даже для большого количества языков.

<https://www.raywenderlich.com/180356/ios-internationalization>пример локализации, не знаю, что тут еще написать