GeekBrains

Лекция 6. Работа с файлами. Обработка ошибок. Objective-C Runtime. Жизненный цикл контроллера.

Работа с файлами при программировании на iOS. Обработка ошибок. Обзор Objective-C Runtime. Обзор жизненного цикла приложения и контроллера на iOS.

Оглавление

Работа с файлами

В большинстве программ необходимо сохранять пользовательские данные для корректной работы. Так или иначе, программе приходится работать с файлами. У языка C для этого существует несколько функций: создания, чтения и записи файлов. В среде Cocoa есть специальный интерфейс Core Data, где разработчику не приходится работать с файлами напрямую. В этом уроке разберем работу со списками свойств и кодирование объектов.

Списки свойств

В среде Cocoa существует специальный вид файлов, известный как список свойств (property list или plist). Он может хранить в себе все разновидности объектов Objective-C, которые представлены в Cocoa:

- NSString;
- NSNumber;
- NSDate;
- NSData;
- NSArray;
- NSDictionary.

Класс NSData

Это тип в Cocoa, который может хранить в себе данные. Это своеобразная «обертка» для блока байтов. Можно получить указатель на начало блока и его длину.

Рассмотрим пример преобразования строки в данные:

```objc
#import <Foundation/Foundation.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSString *string = @"Hello!";
        NSData *data = [string dataUsingEncoding:NSUTF8StringEncoding];
        NSLog(@"%@ - %@", string, data);

        const char *anotherString = "HELLO!";
        NSData *anotherData = [NSData dataWithBytes:anotherString length:strlen(anotherString) + 1];
        NSLog(@"%s - %@", anotherString, anotherData);
    }
    return 0;
}


```

Результат работы:

```
Hello! - <48656c6c 6f21>
HELLO! - <48454c4c 4f2100>
```

Представленные данные – это шестнадцатеричные цифры, которые являются адресами для букв в таблице ASCII.

Объекты класса **NSData** являются неизменяемыми. Соответственно, после создания можно его использовать, но изменить невозможно. Также существует его изменяемая версия – **NSMutableData**.

Запись и чтение списков свойств

Для записи в файл у объектов **NSArray** и **NSDictionary** есть специальный метод, который преобразует их в списки свойств. Он применяется и у типов **NSString** и **NSData**, но сохраняет не списки свойств, а строки и блоки байтов.
Рассмотрим пример сохранения массива как списка свойств. Для наглядности сохраним его в обычный текстовый файл:

```
NSString *path = [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES) firstObject];
path = [path stringByAppendingString:@"/array.txt"];

NSArray *elements = [NSArray arrayWithObjects:@1, @2, @3, @4, nil];
[elements writeToFile:path atomically:YES];

```

В содержимом сохраненного файла увидим следующее:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
    <integer>1</integer>
    <integer>2</integer>
    <integer>3</integer>
    <integer>4</integer>
</array>
</plist>

```

Это структура списка свойств. Чтобы сохранить файл в формате списков свойств, необходимо поменять тип файла (**.txt**) на **plist**. Xcode позволяет просматривать и изменять такие списки.
Для чтения списка свойств воспользуемся специальными методами **arrayWithContentsOfFile** или **dictionaryWithContentsOfFile**.

```
NSString *path = [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES) firstObject];
path = [path stringByAppendingString:@"/array.plist"];

NSArray *elements = [NSArray arrayWithContentsOfFile:path];
NSLog(@"%@", elements);

```

У этих функций есть существенный минус: они не могут вернуть информацию об ошибке при загрузке объекта из файла. Если невозможно получить объект, то будет возвращен указатель **nil** – без возможности определить причину неисправности.

Кодирование объектов

К сожалению, напрямую сохранить собственный объект в память нельзя. Для этого необходимо реализовать протокол **NSCoding**. Объекты сохраняются путем кодирования в **NSData**.

Для принятия протокола **NSCoding** реализуем методы **encodeWithCoder:** и **initWithCoder:**.

Он выглядит следующим образом:

```
@protocol NSCoding

- (void)encodeWithCoder:(NSCoder *)aCoder;
- (nullable instancetype)initWithCoder:(NSCoder *)aDecoder; // NS_DESIGNATED_INITIALIZER

@end

```

Для кодирования объекта в **NSData** применяется **NSKeyedArchiver**, который принимает объект, подписанный на **NSCoding**, и возвращает объект данных. Для декодирования используют **NSKeyedUnarchiver**, которой принимает объект типа **NSData** и возвращает декодированный объект.

Создадим класс и реализуем протокол **NSCoding**. Он будет называться **Student** и обладать двумя свойствами: для имени и фамилии (**name** и **surname**):

```
// Student.h

@interface Student : NSObject <NSCoding>

@property (nonatomic, strong) NSString *name;
@property (nonatomic, strong) NSString *surname;

@end


// Student.m

#import "Student.h"

@implementation Student

- (instancetype)initWithCoder:(NSCoder *)aDecoder {
    if (self = [super init]) {
        self.name = [aDecoder decodeObjectForKey:@"name"];
        self.surname = [aDecoder decodeObjectForKey:@"surname"];
    }
    return self;
}

- (void)encodeWithCoder:(NSCoder *)aCoder {
    [aCoder encodeObject:self.name forKey:@"name"];
    [aCoder encodeObject:self.surname forKey:@"surname"];
}

@end

```

При объявлении класс **Student** подписывается на протокол **NSCoding** (указывается в угловых скобках после родительского класса). Затем реализуются методы

протокола **initWithCoder** и **encodeWithCoder**. Значения кодируются по определенному ключу, по которому в будущем их можно будет декодировать.

Реализация файла **main.m**:

```objc
#import "Student.h"

NSString* directory() {
    return [[NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES) firstObject]
stringByAppendingString:@"/student.txt"];
}

void writeStudent(Student *student) {
    NSData *data = [NSKeyedArchiver archivedDataWithRootObject:student];
    [data writeToFile:directory() atomically:YES];
    NSLog(@"Сохранено!");
}
Student* readStudent() {
    NSLog(@"Прочитано!");
    return [NSKeyedUnarchiver unarchiveObjectWithFile:directory()];
}

void printStudent(Student *student) {
    NSLog(@"name - %@, surname - %@", student.name, student.surname);
}
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Student *student = [[Student alloc] init];
        student.name = @"Steve";
        student.surname = @"Jobs";
        printStudent(student);
        writeStudent(student);
        student = nil;
        printStudent(student);

        student = readStudent();
        printStudent(student);
    }
    return 0;
}

```

Была реализована функция для быстрого отображения директории, в которую необходимо сохранять файл. Также реализованы функции для сохранения и чтения объекта **Student** из файла.

При выполнении функции **main** создается объект **Student**, устанавливаются значения имени и фамилии. После этого объект выводится в консоль и сохраняется в файл. Далее объект **Student** уничтожается и выводится в консоль – для подтверждения того, что данные были удалены. После этого **Student** загружается из файла и снова выводится с сохраненными ранее данными.

Обработка ошибок

➢ Исключения

Для отслеживания нежелательных событий в программе существуют исключения. Они возникают в случаях, когда программа оказывается в затруднительной ситуации и не знает, как поступить. Тогда создается объект исключения и передается системе, которая определяет дальнейшие действия. Исключения в Cocoa представлены классом **NSException**.

Создание исключений и их обработка системой выполнения программ называется генерированием исключения. Класс **NSException** имеет несколько методов, которые начинаются с ключевого слова **raise**.

➢ Ключевые слова для работы с исключениями

- **@try** – определяет блок кода, который будет проверяться на необходимость генерирования исключения;
- **@catch()** – определяет блок кода для обработки сгенерированного исключения;
- **@finally** – определяет блок кода, который выполняется вне зависимости от того, было ли сгенерировано исключение или нет;
- **@throw** – генерирует исключение.

Как правило, все ключевые слова используются в единой структуре:

```
@try {
    // Код, который может вызвать генерирование исключения
} @catch (NSException *exception) {
    // Код для обработки исключения
}
@finally {
    // Код, который выполняется всегда (как правило, для освобождения памяти)
}

```

Структуру **@catch** можно применять многократно (как **elseif**). Будут перехватываться разные типы исключений.

➢ **Генерирование исключений**

При обнаружении исключения программа передает его в блок кода, который называется обработчиком исключения.

Пример обработки исключения:

```
id exception = nil;
NSDictionary *dictionary = [[NSDictionary alloc] initWithObjectsAndKeys:@"value", @"key", nil];
SomeObject* someObject;
@try {
    someObject = [SomeObject new];
    [self process: dictionary, and: someObject];
} @catch(NSException *e) {
    exception = e;
    @throw;
} @finally {
    someObject = nil;
    [exception autorelease];
}
```

Если не получится обработать метод **process**, то будет обработано исключение. Применение исключений – ресурсоемкий процесс, поэтому прибегать к нему нужно только при весомых основаниях.

🔥 **Выдержка из официальной документации Apple:** «Важно: вы не должны использовать исключения для общего управления потоком или просто для обозначения ошибок (например, если файл недоступен)».

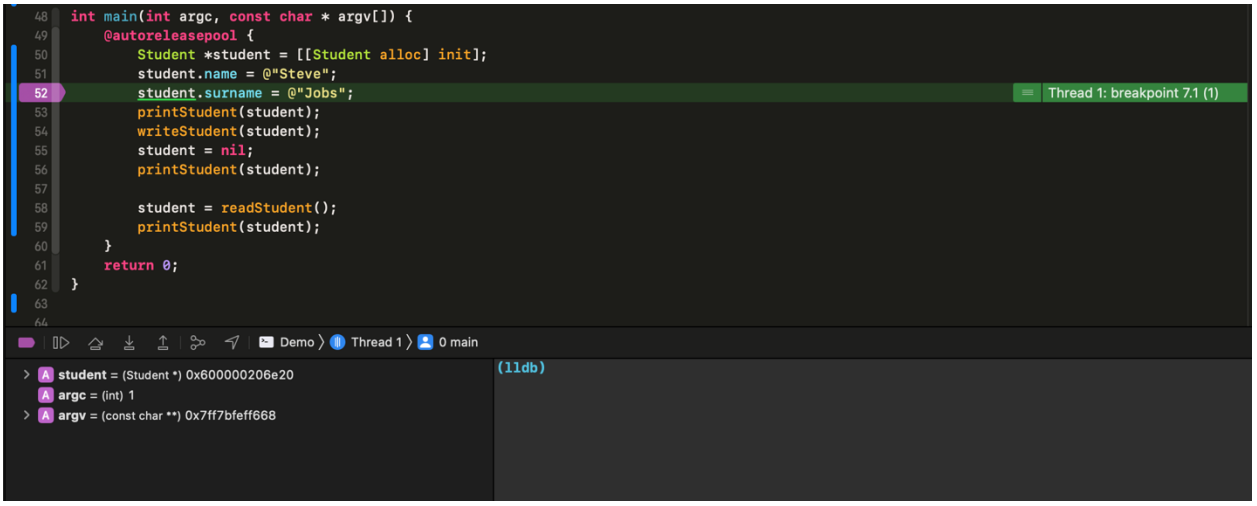
Кроме того, при использовании исключений возникают утечки памяти, из-за некорректной работы ARC.

Отладчик

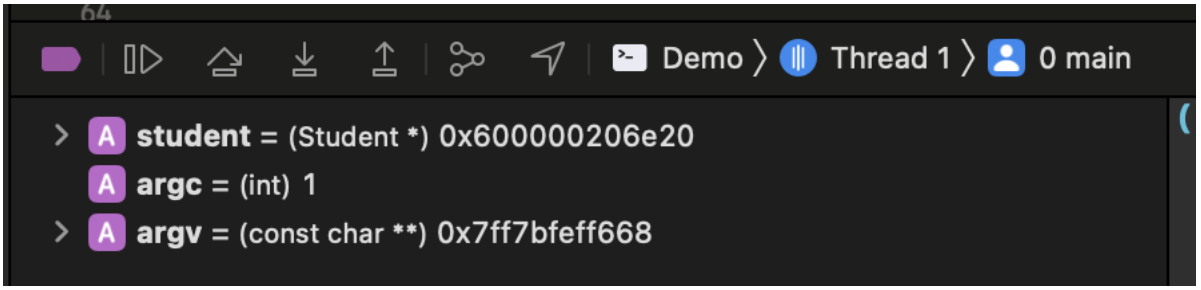
Отладчик необходим для поиска ошибок в приложении. Он позволяет выполнять код пошагово. Чтобы воспользоваться отладчиком, достаточно нажать на необходимую линию справа, и появится специальный индикатор:

```objc
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Student *student = [[Student alloc] init];
        student.name = @"Steve";
        student.surname = @"Jobs";
        printStudent(student);
        writeStudent(student);
        student = nil;
        printStudent(student);

        student = readStudent();
        printStudent(student);
    }
    return 0;
}
```

При выполнении программа остановится в указанном месте:



Чтобы управлять дальнейшим выполнением программы, можно воспользоваться специальными s кнопками в нижней панели:



Первая кнопка слева отвечает за включение и отключение точек остановки. Следующая – за продолжение выполнения до следующей точки, если она есть.

Третья – за переход на следующую строку. Четвертая кнопка позволяет перейти в выполнение метода или функции. Последняя – возвращает из метода или функции.

Подробнее можно рассмотреть все команды Debug, LLVM на сайте Apple: https://developer.apple.com/library/archive/documentation/IDEs/Conceptual/gdb_to_lldb_transition_guide/document/lldb-command-examples.html

Что такое Runtime?

Objective-C задумывался как надстройка над языком C, добавляющая к нему поддержку объектно-ориентированной парадигмы. Фактически, с точки зрения синтаксиса, Objective-C — это достаточно небольшой набор ключевых слов и управляющих конструкций над обычным C. Именно Runtime, библиотека времени выполнения, предоставляет тот набор функций, которые вдыхают в язык жизнь, реализуя его динамические возможности и обеспечивая функционирование ООП.

Базовые структуры данных

Функции и структуры Runtime-библиотеки определены в нескольких заголовочных файлах: objc.h, runtime.h и message.h. Сначала обратимся к файлу objc.h и посмотрим, что представляет из себя объект с точки зрения Runtime:

```
typedef struct objc_class *Class;
typedef struct objc_object {
    Class isa;
} *id;

```

Мы видим, что объект в процессе работы программы представлен обычной C-структурой. Каждый Objective-C объект имеет ссылку на свой класс — так называемый isa-указатель. Думаю, все видели его при просмотре структуры объектов во время отладки приложений. В свою очередь, класс также представляет из себя аналогичную структуру:

```
struct objc_class {
    Class isa;
};

```

Класс в Objective-C — это полноценный объект и у него тоже присутствует isa-указатель на «класс класса», так называемый мета класс в терминах Objective-C.

Аналогично, C-структуры определены и для других сущностей языка:

```
typedef struct objc_selector *SEL;
typedef struct objc_method *Method;
typedef struct objc_ivar *Ivar;
typedef struct objc_category *Category;
typedef struct objc_property *objc_property_t;

```

Функции Runtime-библиотеки

Помимо определения основных структур языка, библиотека включает в себя набор функций, работающих с этими структурами. Их можно условно разделить на несколько групп (назначение функций, как правило, очевидно из их названия):

- Манипулирование классами: class_addMethod, class_addIvar, class_replaceMethod
- Создание новых классов: class_allocateClassPair, class_registerClassPair
- Интроспекция: class_getName, class_getSuperclass, class_getInstanceVariable, class_getProperty, class_copyMethodList, class_copyIvarList, class_copyPropertyList
- Манипулирование объектами: objc_msgSend, objc_getClass, object_copy
- Работа с ассоциативными ссылками

Пример 1. Интроспекция объекта

Рассмотрим пример использования Runtime библиотеки. В одном из наших проектов модель данных представляет собой plain old Objective-C объекты с некоторым набором свойств:

```
@interface COConcreteObject : COBaseObject

@property(nonatomic, strong) NSString *name;
@property(nonatomic, strong) NSString *title;
@property(nonatomic, strong) NSNumber *quantity;

@end

```

Для удобства отладки хотелось бы, чтобы при выводе в лог печаталась информация о состоянии свойств объекта, а не нечто вроде `<COConcreteObject: 0x71d6860>`. Поскольку модель данных достаточно разветвленная, с большим количеством различных подклассов, нежелательно писать для каждого класса отдельный метод description, в котором вручную собирать значения его свойств. На помощь

приходит Objective-C Runtime:

```objc
@implementation COBaseObject

- (NSString *)description {
    NSMutableDictionary *propertyValues = [NSMutableDictionary dictionary];
    unsigned int propertyCount;
    objc_property_t *properties = class_copyPropertyList([self class], &propertyCount);
    for (unsigned int i = 0; i < propertyCount; i++) {
        char const *propertyName = property_getName(properties[i]);
        const char *attr = property_getAttributes(properties[i]);
        if (attr[1] == '@') {
            NSString *selector = [NSString stringWithCString:propertyName encoding:NSUTF8StringEncoding];
            SEL sel = sel_registerName([selector UTF8String]);
            NSObject * propertyValue = objc_msgSend(self, sel);
            propertyValues[selector] = propertyValue.description;
        }
    }
    free(properties);
    return [NSString stringWithFormat:@"%@: %@", self.class, propertyValues];
}

```

Метод, определенный в общем суперклассе объектов модели, получает список всех свойств объекта с помощью функции class_copyPropertyList. Затем значения свойств собираются в NSDictionary, который и используется при построении строкового представления объекта. Данный алгоритм работает только со свойствами, которые являются Objective-C объектами. Проверка типа осуществляется с использованием функции property_getAttributes. Результат работы метода выглядит примерно так:

```
2013-05-04 15:54:01.992 Test[40675:11303] COConcreteObject: {
name = Foo;
quantity = 10;
title = bar;
}
```

Сообщения

Система вызова методов в Objective-C реализована через посылку сообщений объекту. Каждый вызов метода транслируется в соответствующий вызов функции objc_msgSend:

```objc
// Вызов метода
[array insertObject:foo atIndex:1];
// Соответствующий ему вызов Runtime-функции
objc_msgSend(array, @selector(insertObject:atIndex:), foo, 1);

```

Вызов objc_msgSent инициирует процесс поиска реализации метода, соответствующего селектору, переданному в функцию. Реализация метода ищется в так называемой таблице диспетчеризации класса. Поскольку этот процесс может быть достаточно продолжительным, с каждым классом ассоциирован кеш методов. После первого вызова любого метода результат поиска его реализации будет

закеширован в классе. Если реализация метода не найдена в самом классе, дальше поиск продолжается вверх по иерархии наследования — в суперклассах данного класса. Если же и при поиске по иерархии результат не достигнут, в дело вступает механизм динамического поиска — вызывается один из специальных методов: `resolveInstanceMethod` или `resolveClassMethod`. Переопределение этих методов — одна из последних возможностей повлиять на Runtime:

```
+ (BOOL)resolveInstanceMethod:(SEL)aSelector {
    if (aSelector == @selector(myDynamicMethod)) {
        class_addMethod(self, aSelector, (IMP)myDynamicIMP, "v@:");
        return YES;
    }
    return [super resolveInstanceMethod:aSelector];
}

```

Здесь вы можете динамически указать свою реализацию вызываемого метода. Если же этот механизм по каким-то причнам вас не устраивает — вы можете использовать форвардинг сообщений.

Пример 2. Method Swizzling

Одна из особенностей категорий в Objective-C — метод, определенный в категории, полностью перекрывает метод базового класса. Иногда нам требуется не переопределить, а расширить функционал имеющегося метода. Пусть, например, по каким-то причинам нам хочется залогировать все добавления элементов в массив `NSMutableArray`. Стандартными средствами языка этого сделать не получится. Но мы можем использовать прием под названием method swizzling:

```
@implementation NSMutableArray (CO)

+ (void)load {
    Method addObject = class_getInstanceMethod(self, @selector(addObject:));
    Method logAddObject = class_getInstanceMethod(self, @selector(logAddObject:));
    method_exchangeImplementations(addObject, logAddObject);
}

- (void)logAddObject:(id)aObject {
    [self logAddObject:aObject];
    NSLog(@"Добавлен объект %@ в массив %@", aObject, self);
}

@end

```

Мы перегружаем метод `load` — это специальный callback, который, если он определен в классе, будет вызван во время инициализации этого класса — до вызова любого из других его методов. Здесь мы меняем местами реализацию базового метода `addObject:` и нашего метода `logAddObject:`. Обратите внимание на «рекурсивный» вызов в `logAddObject:` — это и есть обращение к перегруженной реализации основного метода.

Пример 3. Ассоциативные ссылки

Еще одним известным ограничением категорий является невозможность создания в них новых переменных экземпляра. Пусть, например, вам требуется добавить новое свойство к библиотечному классу `UITableView` — ссылку на «заглушку», которая будет показываться, когда таблица пуста:

```
@interface UITableView (Additions)

@property(nonatomic, strong) UIView *placeholderView;

@end

```

«Из коробки» этот код работать не будет, вы получите исключение во время выполнения программы. Эту проблему можно обойти, используя функционал ассоциативных ссылок:

```
static char key;

@implementation UITableView (Additions)

-(void)setPlaceholderView:(UIView *)placeholderView {
    objc_setAssociatedObject(self, &key, placeholderView, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}

-(UIView *) placeholderView {
    return objc_getAssociatedObject(self, &key);
}

@end

```

Любой объект вы можете использовать как ассоциативный массив, связывая с ним другие объекты с помощью функции `objc_setAssociatedObject`. Для ее работы требуется ключ, по которому вы потом сможете извлечь нужный вам объект назад, используя вызов `objc_getAssociatedObject`. При этом вы не можете использовать скопированное значение ключа — это должен быть именно тот объект (в примере — указатель), который был передан в вызове `objc_setAssociatedObject`.

Objective – C Runtime. Итог

Теперь вы располагаете базовым представлением о том, что такое Objective-C Runtime и чем он может быть полезен разработчику на практике. Для желающих

узнать возможности библиотеки глубже, могу посоветовать следующие дополнительные ресурсы:

1. Objective-C Runtime Programming Guide — документация от Apple
2. Блог Mike Ash — статьи гуру Objective-C разработки

Жизненный цикл приложения

➢ **Жизненный цикл**

Для написания грамотного кода мы должны знать, чего ожидать от системы. Рассмотрим основные статусы приложения:

1. Не запущено;
2. Неактивно;
3. Активно;
4. Находится в фоне;
5. Приостановлено.

При запуске и при каждом изменении статуса приложение посылает сообщения классу **AppDelegate**, в котором можно обработать эти изменения. Важно понимать, в какой момент и какой метод вызывается.

Изначально приложение находится в статусе «не запущено». При запуске пользователем оно переходит в **Foreground**, где изначально становится неактивным. На этом этапе выполняется код программы, но не обрабатываются события интерфейса. Затем приложение переходит в статус «активно»: выполняется код и обрабатываются все пользовательские события.

При запуске пользователем другого приложения текущее переходит в состояние неактивного, а после него – в **Background**. В этом состоянии код программы выполняется только ограниченное время (как правило, для синхронизации данных). События на данном этапе уже не обрабатываются. На этом этапе можно производить обновление контента, чтобы к моменту возвращения в активное состояние данные были актуальны.
По истечении предоставленного системой времени приложение переходит в статус «приостановлено». Далее система может уничтожить все его данные из памяти и

полностью закончить его работу.

➢ **Методы жизненного цикла приложения**

Рассмотрим подробнее методы **AppDelegate**, с помощью которых можно отслеживать изменения статуса приложения.

1. **Метод успешного запуска.**

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {

    return YES;
}

```

Он вызывается, когда система полностью подготовила все необходимые данные для работы приложения. Это значит, что по его завершении пользователь сможет приступить к работе с приложением.

2. **Метод перехода в неактивное состояние.**

```
- (void)applicationWillResignActive:(UIApplication *)application {

}

```

Как правило, в этом методе ставятся на паузу активные процессы и останавливается обновление пользовательского интерфейса.

3. **Метод перехода в активное состояние.**

```
- (void)applicationDidBecomeActive:(UIApplication *)application {

}

```

В этом методе необходимо обновить все процессы, которые были остановлены при переходе приложения в неактивное состояние.

4. **Метод перехода приложения в Background.**

```
- (void)applicationDidEnterBackground:(UIApplication *)application {

}

```

Этот метод сохраняет основные данные или состояние приложения, а также запускает процессы по обновлению его контента.

5. **Метод перехода приложения из Background в Foreground.**

```
- (void)applicationWillEnterForeground:(UIApplication *)application {

}

```

В этом методе можно остановить все процессы обновления контента.

6. **Метод, который отражает закрытие приложения пользователем.**

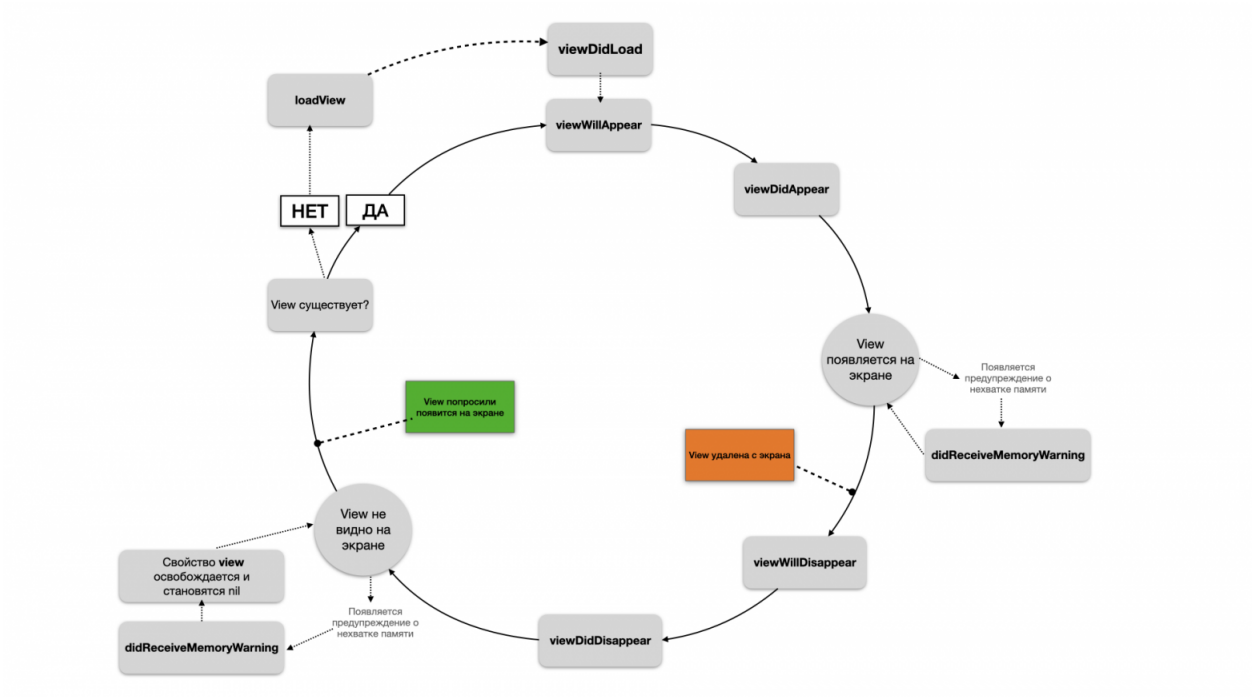
```
- (void)applicationWillTerminate:(UIApplication *)application {

}

```

Жизненный цикл контроллера

➢ **Жизненный цикл**

Рассмотрим жизненный цикл отдельного контроллера. Первоначально он инициализируется, и в него осуществляется переход. Затем контроллер устанавливает свои свойства, и только после этого появляется на экране. Когда контроллер закрывается, он уничтожает все свойства из памяти и освобождает ее.

➢ **Методы жизненного цикла контроллера**



Как и у **AppDelegate**, у каждого контроллера есть методы, которые отражают его текущее состояние:

1. **init(coder:)**

View Controller обычно создается из сторибордов. В этом случае вызывается инициализатор init(coder:), который необходимо переопределить. Он предоставляет экземпляр NSCoder в качестве параметра, который вам нужен, только если вы используете API-интерфейсы сериализации iOS. Это используется не часто, поэтому вы можете игнорировать этот параметр. Эта сериализация преобразует объект в поток байтов, который вы можете сохранить на диске или отправить по сети. На этапе инициализации ViewController'a вы обычно выделяете ресурсы, которые потребуются ViewController'y в течение его жизненного цикла. К ним относятся объекты модели или другие вспомогательные контроллеры, такие как сетевые контроллеры. Предыдущие контроллеры также могут передавать эти объекты текущему, поэтому вам не всегда нужно создавать их экземпляры в каждом контроллере.

Имейте в виду, что на данный момент вью ViewController’a еще не создано. Если вы попытаетесь получить к нему доступ через вью свойства ViewController’a, будет вызван метод loadView(). Это может привести к неожиданному поведению и ошибкам, поэтому безопаснее обращаться к вью на более позднем этапе жизненного цикла.

2. **init(nibName:bundle:)**

Иногда вы можете решить поместить интерфейс вашего ViewController’а в отдельный nib-файл вместо сториборда. Например, при работе в большой команде, где разные разработчики должны изменять интерфейсы ViewController’а, не влияя на работу друг-друга. У вас также может быть проект, который был создан, когда сторибордов еще не существовало, поэтому у каждого ViewController’а был свой собственный nib-файл. Обратите внимание, что, если ваш Main(Основной) Storyboard становится слишком большой, вы можете разделить его на несколько Storyboard’ов. Вам не нужно перемещать каждый ViewController в отдельный nib-файл. Если вы создаете ViewController из nib-файла, этот инициализатор вызывается вместо init(coder:).

3. **loadView()**

Это метод, который создает view для ViewController’а. Вы переопределяете этот метод только в том случае, если хотите построить весь интерфейс для ViewController’а из кода. Не используйте его, если нет уважительной причины. Если вы работаете со Stryboard'ми или nib-файлами, вам не нужно ничего делать с этим методом, и вы можете его игнорировать. Его реализация в UIVIewController загружает интерфейс из файла и подключает за вас все Outlets и Actions.

4. **Метод, который вызывается сразу после загрузки view.**

```
- (void)viewDidLoad {
    [super viewDidLoad];

}

```

При вызове этого метода все необходимые свойства контроллера уже созданы и готовы к использованию.

5. **Метод до отображения на экране.**

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];

}

```

Этот метод вызывается перед появлением представления на экране. Он подходит для обновления состояний или переменных, которые влияют на отображение пользовательского интерфейса.

6. **Метод после отображения на экране.**

```objc
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];

}

```

Этот метод вызывается сразу после отображения представления на экране. Подходит для конфигурации размеров или схожих величин, так как все компоненты уже появились на экране.

7. **Метод до закрытия контроллера.**

```objc
- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];

}

```

Этот метод вызывается перед закрытием контроллера. В нем можно обновить пользовательские настройки, которые были заданы в контроллере.

8. **Метод после закрытия контроллера.**

```objc
- (void)viewDidDisappear:(BOOL)animated {
    [super viewDidDisappear:animated];

}

```

Этот метод вызывается сразу после закрытия контроллера. В нем можно очистить данные, используемые в нем.

9. **Метод для обработки поворота экрана.**

```objc
- (void)viewWillTransitionToSize:(CGSize)size withTransitionCoordinator:
(id<UIViewControllerTransitionCoordinator>)coordinator {
    [super viewWillTransitionToSize:size withTransitionCoordinator:coordinator];

}

```

Этот метод вызывается при повороте экрана. Его можно применять для анимации.

10. **deinit()**

Как и любой другой объект, прежде чем ViewController будет удален из памяти, он деинициализируется. Обычно вы переопределяете deinit() для очистки ресурсов, выделенных ViewController, которые не освобождаются ARC (автоматический подсчет ссылок).

Вы также можете остановить задачи, которые вы не остановили в предыдущем методе, потому что вы хотели оставить их в фоновом режиме. ViewController исчезает с экрана, но это не означает, что впоследствии он будет освобожден. Многие контейнеры хранят viewController в памяти. Например, когда вы углубляетесь в navigation controller, все предыдущие viewController остаются в памяти. Navigation controller освобождает viewController'ы только при переходе вверх по иерархии. Поэтому вы должны иметь в виду, что viewController, который не находится на экране, все равно работает нормально и получает уведомления. Иногда это желательно, иногда нет, поэтому вам нужно помнить об этом при разработке приложения.

11. didReceiveMemoryWarning()

Устройства iOS имеют ограниченный объем памяти и мощности. Когда оперативная память начинает заполняться, iOS не использует дополнительно память на жестком диске для перемещения данных, как это делает компьютер. По этой причине вы несете ответственность за то, сколько ваше приложение занимает оперативной памяти. Если ваше приложение начнет использовать слишком много памяти, iOS уведомит об этом.

Поскольку viewController'ы выполняют управление ресурсами, эти уведомления доставляются им с помощью этого метода. Таким образом, вы можете предпринять действия, чтобы освободить часть памяти. Имейте в виду, что если вы проигнорируете предупреждения о нехватки памяти и память, используемая вашим приложением, превысит определенный порог, iOS завершит работу вашего приложения. Это будет выглядеть как краш для пользователя, и его следует избегать.