Урок 7

Работа с файлами. Обработка ошибок. Objective-C Runtime

Работа с файлами при программировании на iOS. Обработка ошибок. Обзор Objective-C Runtime. Начало создания игры «Пинг-понг» на Objective-C.

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:

```
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);
Результат:
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**:

```
#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:

При выполнении программа остановится в указанном месте:

Чтобы управлять дальнейшим выполнением программы, можно воспользоваться специальными 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.

Objective-C Runtime

Язык Objective-C – это надстройка над языком C, которая добавляет объектно-ориентированные парадигмы. По сути, Objective-C – это набор ключевых слов и различных конструкций над привычным C.

Базовые структуры

Все функции и структуры **Runtime** определены в нескольких файлах: **objc.h**, **runtime.h** и **message.h**. Рассмотрим, что представляет собой объект с точки зрения библиотеки Runtime:

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

Объект представлен в виде обычной структуры C. Каждый объект имеет ссылку на свой класс (isa-указатель). Класс представлен схожей структурой:

```
struct objc_class {
    Class isa;
};
```

В Objective-C класс является полноценным объектом, и у него есть isa-указатель на метакласс.

Функции Runtime-библиотеки

Для работы со всем этими структурами Runtime включает набор функций. Они разделяются на несколько групп:

- Управление классами;
- Создание новых классов;
- Интроспекция;
- Управление объектами;

- Работа с ассоциативными ссылками.

Рассмотрим пример интроспекции. Допустим, необходимо отображать свойства объекта и его значения при его передаче в **NSLog**. Если сейчас попробовать вывести объект, то в консоли отобразится только адрес этого объекта. Создадим необходимый объект и реализуем метод **description**:

```
@interface Person : NSObject

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

@end

@implementation Person

- (NSString *)description {
    NSMutableDictionary *values = [NSMutableDictionary new];
    unsigned int count;
    objc_property_t *properties = class_copyPropertyList([self class], &count);

    for (int i = 0; i < count; i++) {
        objc_property_t property = properties[i];
        const char *name = property_getName(property);
        const char *attributes = property_getAttributes(property);
        if (attributes[1] == '@') {
            NSString *selector = [NSString stringWithCString:name
encoding:NSUTF8StringEncoding];
            SEL sel = sel_registerName([selector UTF8String]);
            NSObject *value = objc_msgSend(self, sel);
            values[selector] = value.description;
        }
    }
    free(properties);
    return [NSString stringWithFormat:@"%@: %@", self.class, values];
}

@end
```

Теперь создадим объект, установим значение и выведем объект в консоль:

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {

        Person *person = [[Person alloc] init];
        person.name = @"Steve";
        person.surname = @"Jobs";
        NSLog(@"%@", person);

    }
    return 0;
}
```

Результат в консоли:

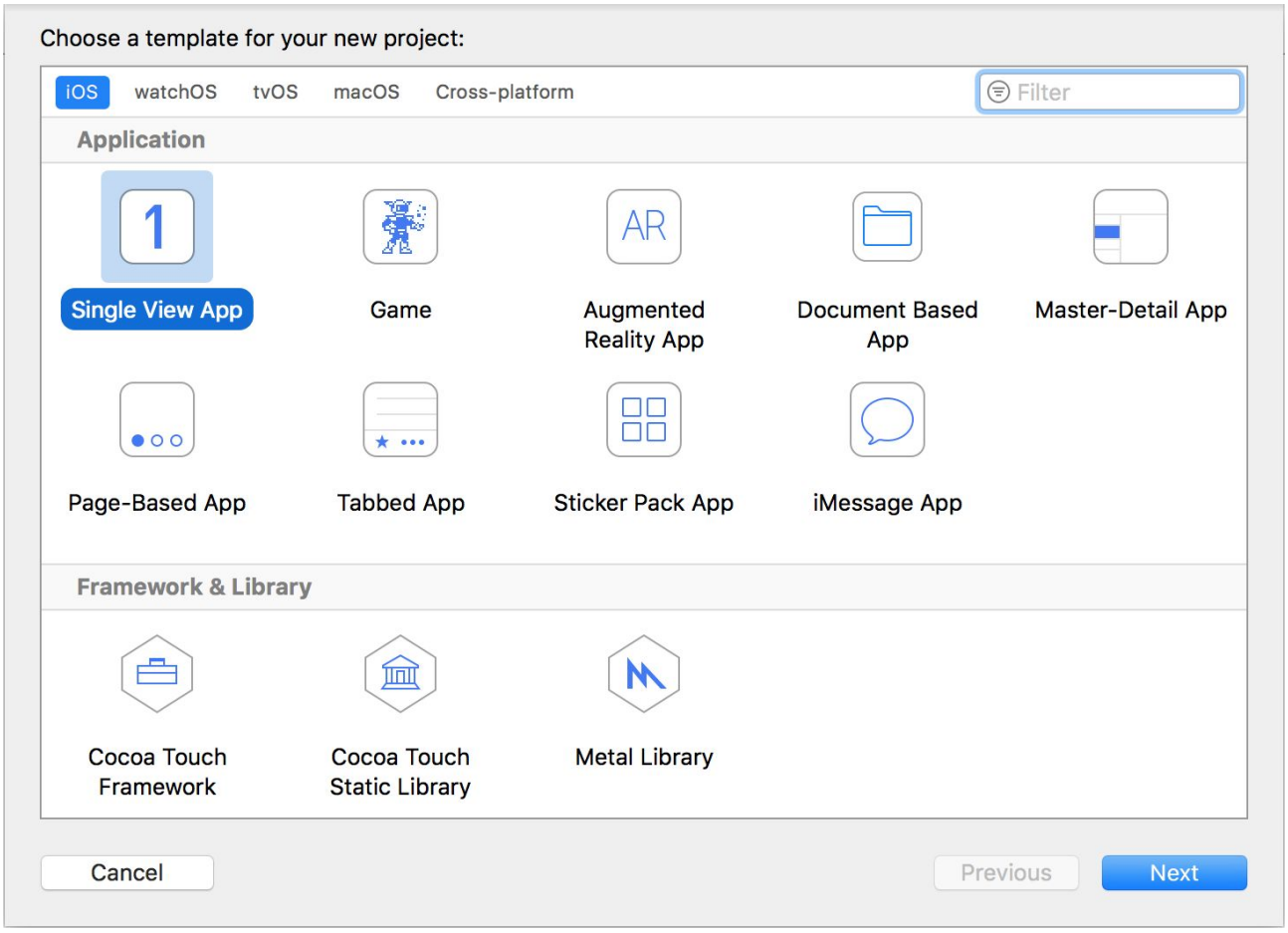
```
Person: {
    name = Steve;
    surname = Jobs;
}
```

Это один из простых примеров Objective-C Runtime – библиотеки, которая может значительно облегчить работу программиста.

Начало создания игры «Пинг-понг» на Objective-C

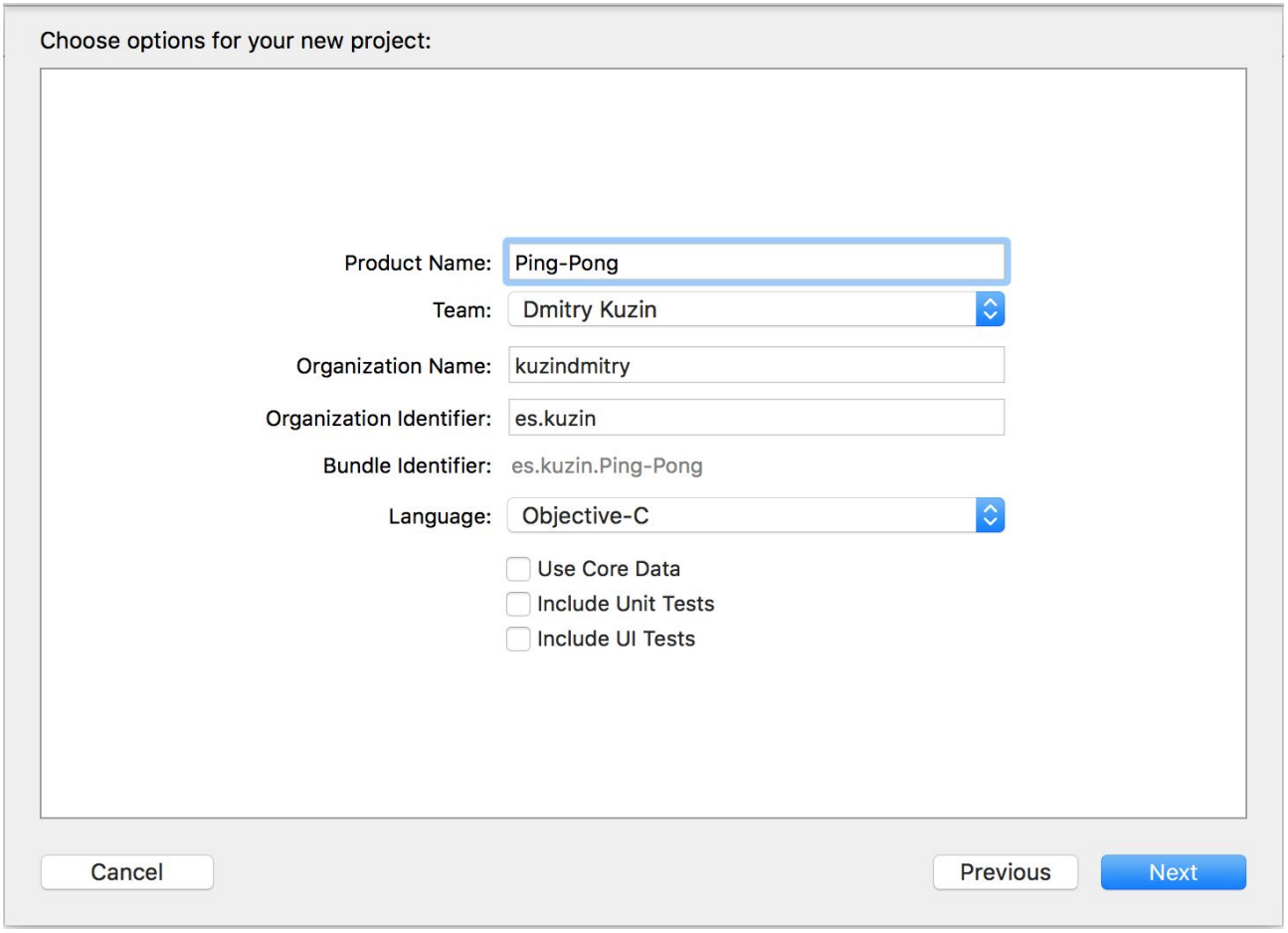
Создание проекта в Xcode

Чтобы начать разработку игры, создадим проект в Xcode. Открываем Xcode и нажимаем **Create New Project**. Выбираем один из типов приложения:

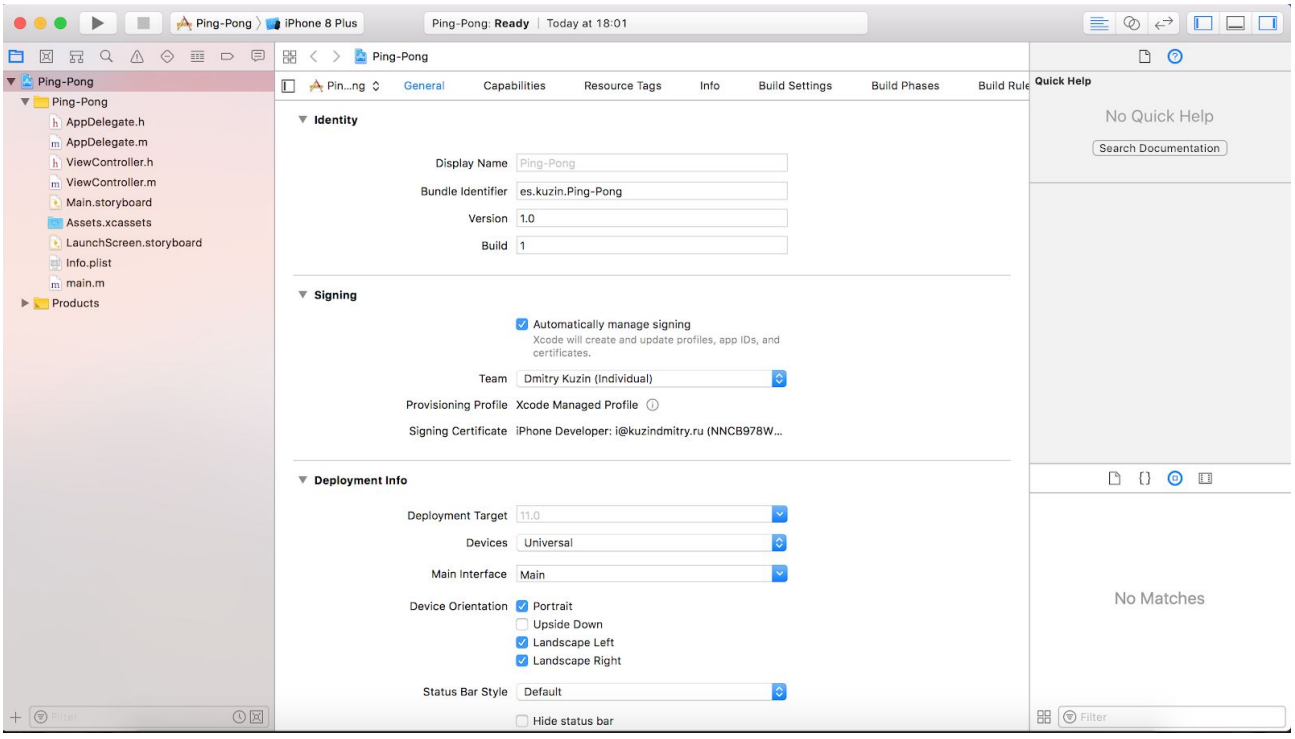


Так как мы создаем игру «с нуля», то выбираем самый простой шаблон – **Single View App**.

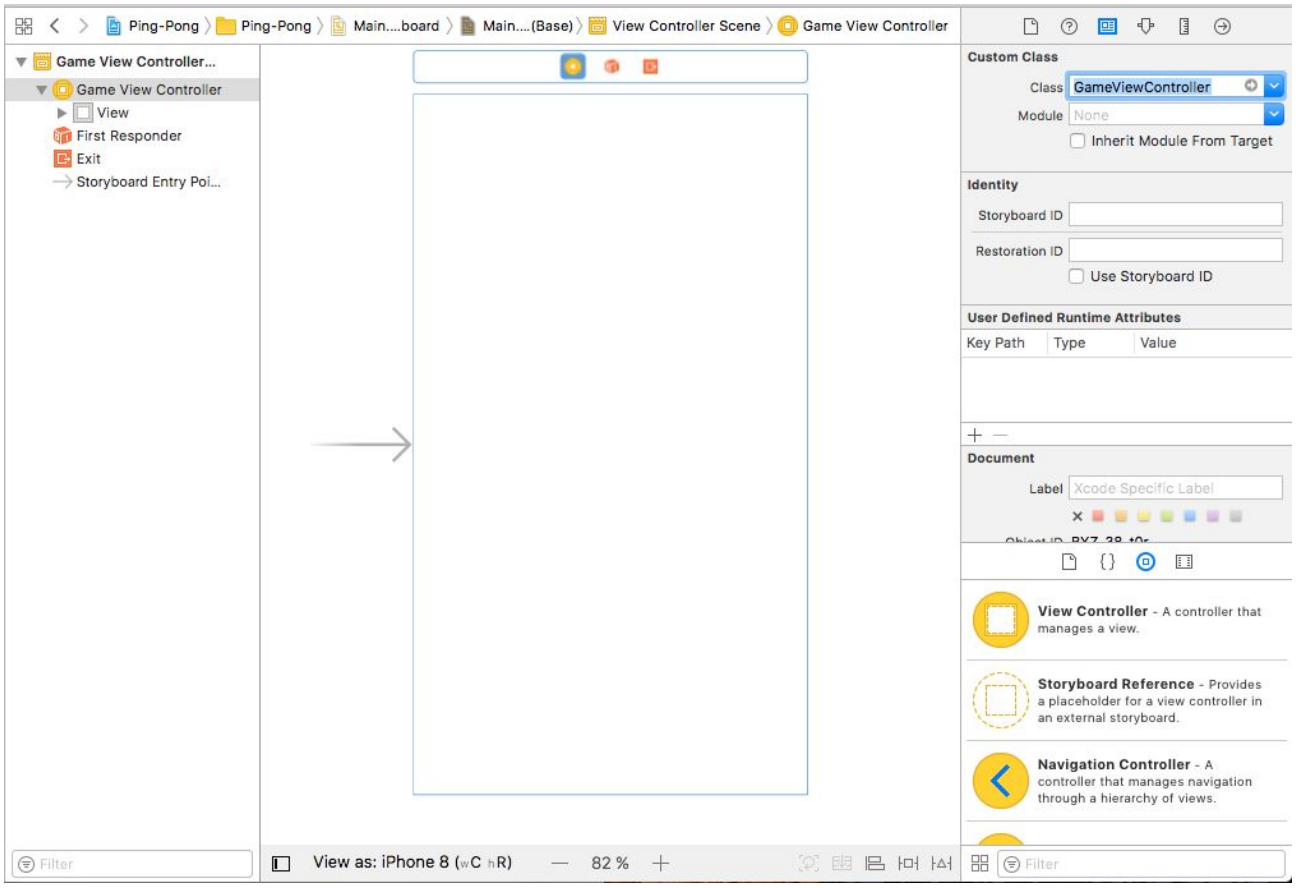
После этого указываем основную информацию о проекте: его название, команду разработчиков, название и идентификатор организации, язык.



После этого выбираем месторасположение проекта и подтверждаем создание проекта кнопкой **Create**. Откроется главное окно проекта:



Поменяем имя классу **ViewController** на **GameViewController**. Для этого понадобится изменить файлы объявления и реализации, а также название в **Main.storyboard**.



Нажимаем на первый значок над белым экраном и в правой панели изменяем класс на **GameViewController**.

Объявление переменных и конфигурирование

Откроем файл **GameViewController.m** и объявим переменные для расчета размеров:

```
#define SCREEN_WIDTH [UIScreen mainScreen].bounds.size.width
#define SCREEN_HEIGHT [UIScreen mainScreen].bounds.size.height
#define HALF_SCREEN_WIDTH SCREEN_WIDTH/2
#define HALF_SCREEN_HEIGHT SCREEN_HEIGHT/2
#define MAX_SCORE 6
```

Эти константы понадобятся в дальнейшем для более удобного доступа к размеру экрана, а также для определения максимального количества очков у игрока.

Теперь необходимо объявить свойства, которые будут у **GameViewController**:

```
@interface GameViewController ()

@property (strong, nonatomic) UIImageView *paddleTop;
@property (strong, nonatomic) UIImageView *paddleBottom;
@property (strong, nonatomic) UIView *gridView;
@property (strong, nonatomic) UIView *ball;
@property (strong, nonatomic) UITouch *topTouch;
@property (strong, nonatomic) UITouch *bottomTouch;
@property (strong, nonatomic) NSTimer * timer;
@property (nonatomic) float dx;
@property (nonatomic) float dy;
@property (nonatomic) float speed;
@property (strong, nonatomic) UILabel *scoreTop;
@property (strong, nonatomic) UILabel *scoreBottom;

@end
```

Далее создадим метод для первоначальной конфигурации экрана и основных игровых компонентов:

```
- (void)config {
    self.view.backgroundColor = [UIColor colorWithRed:100.0/255.0
green:135.0/255.0 blue:191.0/255.0 alpha:1.0];

    _gridView = [[UIView alloc] initWithFrame:CGRectMake(0, HALF_SCREEN_HEIGHT -
2, SCREEN_WIDTH, 4)];
    _gridView.backgroundColor = [[UIColor whiteColor]
colorWithAlphaComponent:0.5];
    [self.view addSubview:_gridView];

    _paddleTop = [[UIImageView alloc] initWithFrame:CGRectMake(30, 40, 90, 60)];
    _paddleTop.image = [UIImage imageNamed:@"paddleTop"];
    _paddleTop.contentMode = UIViewContentModeScaleAspectFit;
    [self.view addSubview:_paddleTop];

    _paddleBottom = [[UIImageView alloc] initWithFrame:CGRectMake(30,
SCREEN_HEIGHT - 90, 90, 60)];
    _paddleBottom.image = [UIImage imageNamed:@"paddleBottom"];
    _paddleBottom.contentMode = UIViewContentModeScaleAspectFit;
    [self.view addSubview:_paddleBottom];

    _ball = [[UIView alloc] initWithFrame:CGRectMake(self.view.center.x - 10,
self.view.center.y - 10, 20, 20)];
    _ball.backgroundColor = [UIColor whiteColor];
    _ball.layer.cornerRadius = 10;
    _ball.hidden = YES;
    [self.view addSubview:_ball];

    _scoreTop = [[UILabel alloc] initWithFrame:CGRectMake(SCREEN_WIDTH - 70,
HALF_SCREEN_HEIGHT - 70, 50, 50)];
    _scoreTop.textColor = [UIColor whiteColor];
    _scoreTop.text = @"0";
    _scoreTop.font = [UIFont systemFontOfSize:40.0 weight:UIFontWeightLight];
    _scoreTop.textAlignment = NSTextAlignmentCenter;
    [self.view addSubview:_scoreTop];

    _scoreBottom = [[UILabel alloc] initWithFrame:CGRectMake(SCREEN_WIDTH - 70,
HALF_SCREEN_HEIGHT + 70, 50, 50)];
    _scoreBottom.textColor = [UIColor whiteColor];
    _scoreBottom.text = @"0";
    _scoreBottom.font = [UIFont systemFontOfSize:40.0 weight:UIFontWeightLight];
    _scoreBottom.textAlignment = NSTextAlignmentCenter;
    [self.view addSubview:_scoreBottom];
}
```

Рассмотрим подробнее:

```
self.view.backgroundColor = [UIColor colorWithRed:100.0/255.0 green:135.0/255.0
blue:191.0/255.0 alpha:1.0];
```

Первоначально устанавливается цвет для заднего фона. Взаимодействуем с **view** – компонентом, отображающим представление. У наследников **UIViewController** объект view присутствует по умолчанию и представляет собой весь экран, который отображается для пользователя.

```
   _gridView = [[UIView alloc] initWithFrame:CGRectMake(0, HALF_SCREEN_HEIGHT -
2, SCREEN_WIDTH, 4)];
    _gridView.backgroundColor = [[UIColor whiteColor]
colorWithAlphaComponent:0.5];
    [self.view addSubview:_gridView];
```

Здесь мы устанавливаем размер и фон сетки на игровом поле, которая затем добавляется на экран. Сетка также является представлением, объектом класса **UIView**. Размер указывается в следующих величинах: положение от левого края (x), положение от верха экрана (y), ширина (width), высота (height). На следующей строке указывается фон для **view** с прозрачностью 50%. Затем сетка добавляется на экран.

```
    _paddleTop = [[UIImageView alloc] initWithFrame:CGRectMake(30, 40, 90, 60)];
    _paddleTop.image = [UIImage imageNamed:@"paddleTop"];
    _paddleTop.contentMode = UIViewContentModeScaleAspectFit;
    [self.view addSubview:_paddleTop];

    _paddleBottom = [[UIImageView alloc] initWithFrame:CGRectMake(30,
SCREEN_HEIGHT - 90, 90, 60)];
    _paddleBottom.image = [UIImage imageNamed:@"paddleBottom"];
    _paddleBottom.contentMode = UIViewContentModeScaleAspectFit;
    [self.view addSubview:_paddleBottom];
```

Создаем объекты ракеток – верхнюю и нижнюю. Они являются объектами **UIImageView**. Нам потребуется специальный компонент для отображения изображений. Так как **UIImageView** – наследник **UIView,** у него тоже есть конструктор с конфигурацией размера.

Затем устанавливаем изображение и указываем его название в **Assets.xcassets** (изображение добавим позже). Далее выбираем стиль отображения (растягивание, сжатие до натурального размера или другой). После этого ракетка добавляется на экран.

```
    _ball = [[UIView alloc] initWithFrame:CGRectMake(self.view.center.x - 10,
self.view.center.y - 10, 20, 20)];
    _ball.backgroundColor = [UIColor whiteColor];
    _ball.layer.cornerRadius = 10;
    _ball.hidden = YES;
    [self.view addSubview:_ball];
```

Создаем представление для мяча. У него есть позиция центра, благодаря которой можно разместить мяч посередине поля (позже мы поработаем с центром представления). Устанавливаем фон для представления, а затем – значение для закругления углов его слоя, чтобы мяч был круглым. Определяем значение видимости (до начала игры необходимо спрятать мяч). Добавляем на экран.

Далее настраиваем текстовые поля, чтобы показывать текущий счет (очки игрока):

```
_scoreTop = [[UILabel alloc] initWithFrame:CGRectMake(SCREEN_WIDTH - 70,
HALF_SCREEN_HEIGHT - 70, 50, 50)];
    _scoreTop.textColor = [UIColor whiteColor];
    _scoreTop.text = @"0";
    _scoreTop.font = [UIFont systemFontOfSize:40.0 weight:UIFontWeightLight];
    _scoreTop.textAlignment = NSTextAlignmentCenter;
    [self.view addSubview:_scoreTop];

    _scoreBottom = [[UILabel alloc] initWithFrame:CGRectMake(SCREEN_WIDTH - 70,
HALF_SCREEN_HEIGHT + 70, 50, 50)];
    _scoreBottom.textColor = [UIColor whiteColor];
    _scoreBottom.text = @"0";
    _scoreBottom.font = [UIFont systemFontOfSize:40.0 weight:UIFontWeightLight];
    _scoreBottom.textAlignment = NSTextAlignmentCenter;
    [self.view addSubview:_scoreBottom];
```

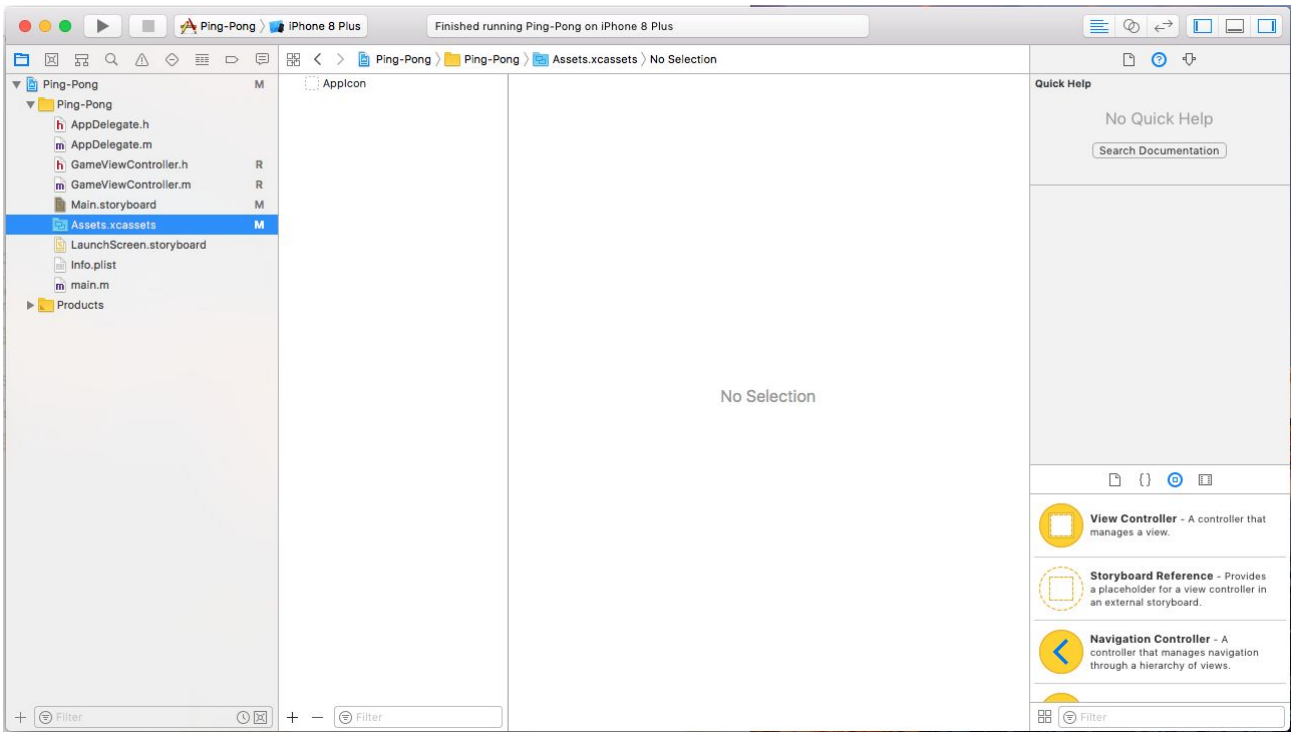
Объекты класса UILabel отвечают за вывод текстовой информации. При их создании устанавливаем положение и размер на экране. Затем выбираем цвет текста и его значение (по умолчанию – 0). Устанавливаем шрифт и его размер, расположение текста (слева, по центру или справа). Добавляем объект на экран.

Далее необходимо добавить стандартный метод, который показывает, что экран загружен, и включить в него выполнение метода конфигурации:

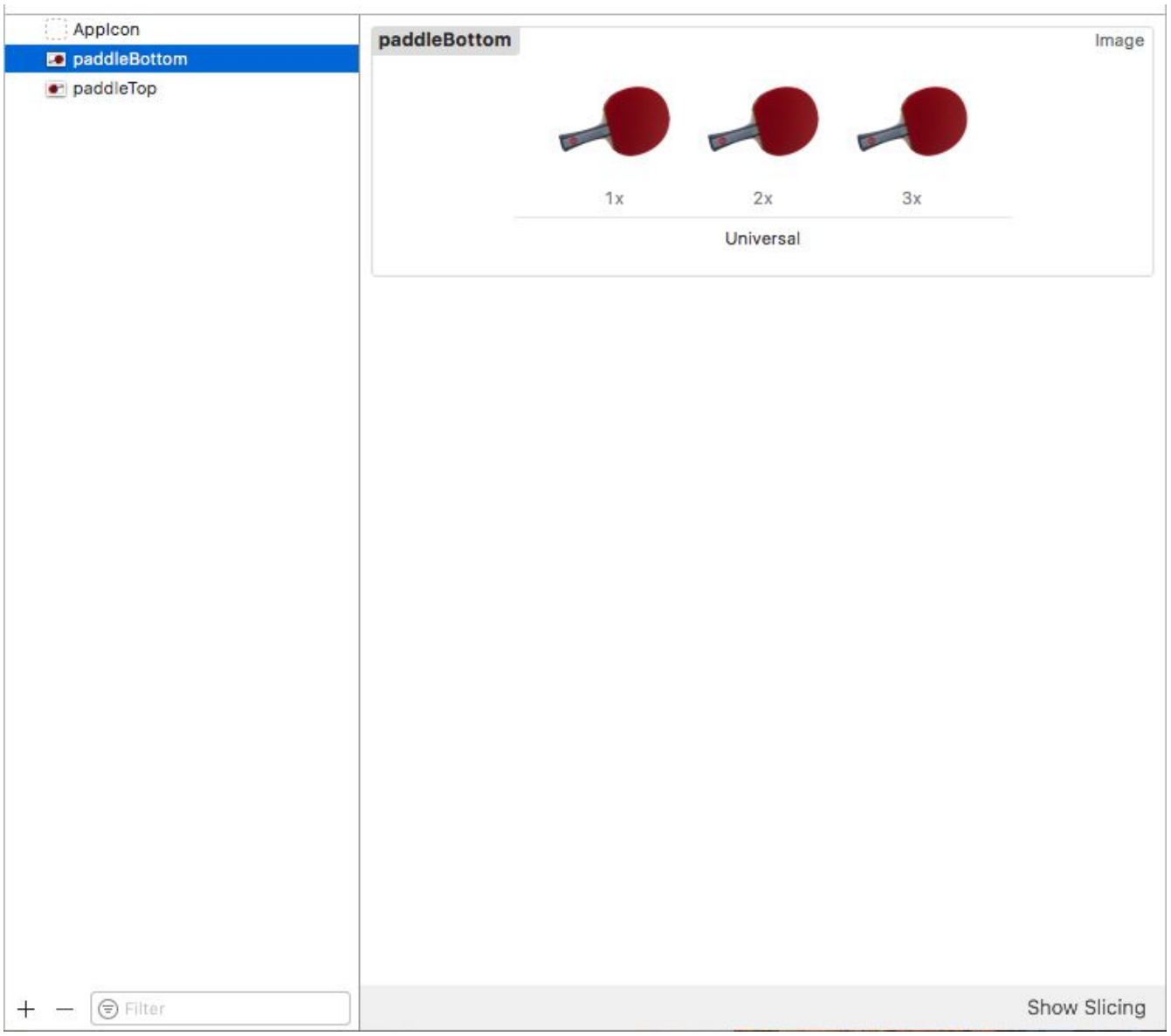
```
- (void)viewDidLoad {
    [super viewDidLoad];
    [self config];
}
```

Добавление изображений

Теперь добавим изображения ракеток. Для этого откроем файл **Assets.xcassets**:



Добавим изображения самих ракеток: перенесем их в левую половину.

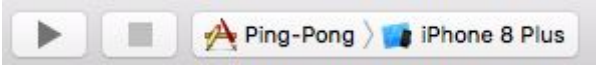


Присвоим изображениям нужные названия: **paddleTop** для верхней ракетки и **paddleBottom** – для нижней.

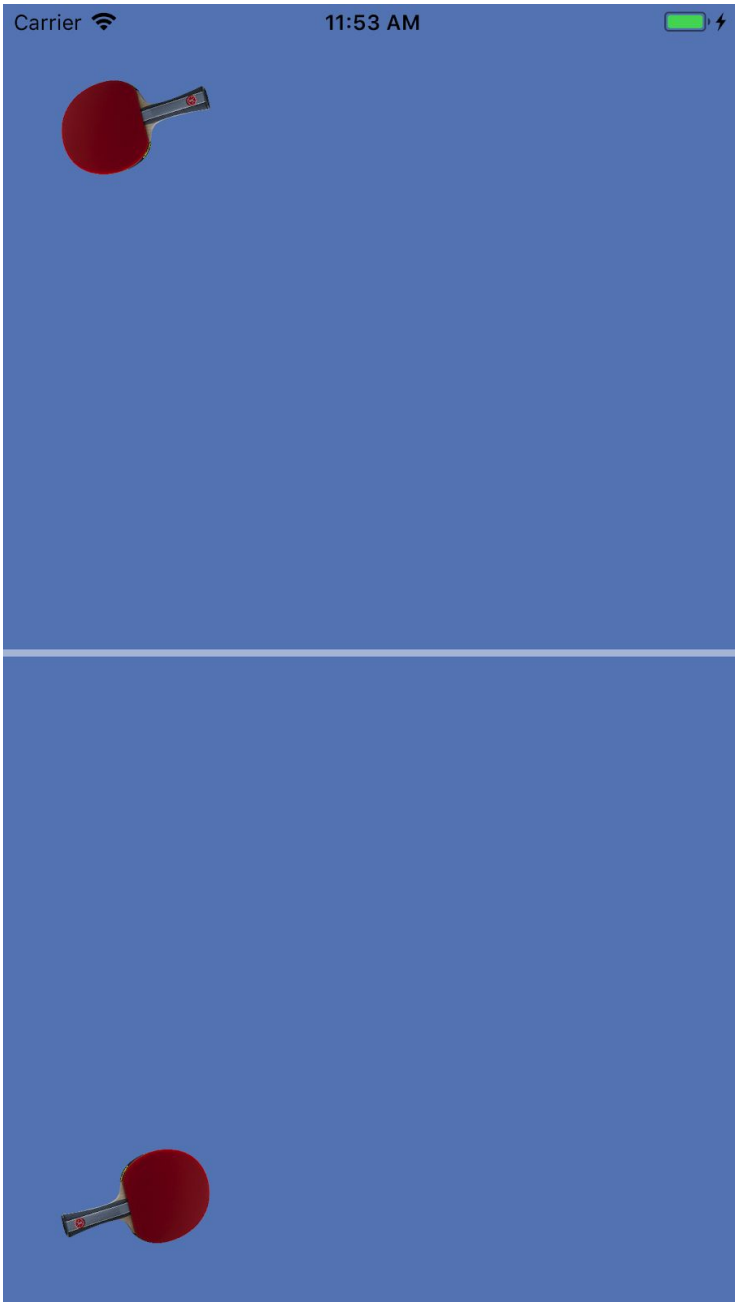
У одного объекта в Assets может быть 3 размера изображения – для корректного отображения на устройствах с разным разрешением.

Первый запуск приложения

Основные подготовительные работы закончены, создан код. Проверим, корректно ли отображаются все компоненты. Чтобы запустить приложение, используем специальную кнопку запуска на верхней панели:



После нажатия на нее приложение будет собрано и запущено на выбранном справа симуляторе (в нашем примере – iPhone 8 Plus).



Так выглядит запущенное приложение. В него добавлены сетка и ракетки, установлен задний фон для всего экрана, симулирующий стол.

Практическое задание

1. Сделать проект и создать в нем поле для игры в настольный теннис по примеру из методички.
2. Код из методички отрефакторить (привести в порядок).

Дополнительные материалы

1. https://habrahabr.ru/post/177421/;
2. http://c-objective.ru/objective-c-dlya-nachinayushih/operator-define-v-objective-c/;
3. Скотт Кнастер, Вакар Малик, Марк Далримпл. «Objective-C. Программирование для Mac OS X и iOS».

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Стивен Кочан. «Программирование на Objective-C»;
2. Скотт Кнастер, Вакар Малик, Марк Далримпл.«Objective-C. Программирование для Mac OS X и iOS».