Objective-C

Урок 6

Блоки и многопоточное программирование

Рассмотрение многопоточного программирования. Операции с очередями. Применение блоков.

Блоки

Блочные объекты

Блоки – это тип в Objective-C, который хранит в себе функцию. Они схожи со стандартными функциями языка C, но могут еще и содержать переменные, привязанные к автоматической и управляемой куче памяти. Блоки могут быть полезны для обратного вызова, потому что несут как код, который будет выполняться на обратном вызове, так и данные, необходимые в это время. Блоковые объекты могут принимать и возвращать параметры, как и обычные функции.

Блоковые объекты – дополнительные расширения в языке C – включают в себя не только код, который образует функции, но и переменные связи. Также блоки называют замыканиями.

Блоковые объекты и обычные функции языка C создаются похожим образом. Блоковые объекты могут возвращать значения и принимать параметры. Их можно определять как встраиваемые либо обрабатывать как отдельные блоки кода, наподобие функций языка C. При создании таких объектов разными способами (встраиваемым способом и реализации как отдельного блока кода) области видимости переменных, доступных блоковым объектам, будут существенно отличаться.

Блоки обладают внушительным потенциалом, поэтому широко применяются во многих библиотеках, которые предоставляются компанией Apple. Одним из примеров использования блоков в Cocoa является сортировка массива, которую мы рассмотрим далее.

Применение блоков

Рассмотрим пример реализации и применения блока, который будет рассчитывать сумму двух чисел:

```
int (^sum)(int,int) = ^(int first, int second) {
    return first + second;
};

int result = sum(3, 4);

NSLog(@"Result - %i", result);
```

В результате выполнения программы в консоль будет выведен результат: сумма 3 и 4.

Изначально указывается возвращаемый тип, затем в скобках – ключевой символ «^», после него – название блока, а затем в скобках – принимаемые параметры. Ему присваивается функция, схожая с C-функцией, только с добавлением специального символа «^» в начале.

Рассмотрим пример пустого блока:

```
void (^block)(void) = ^{
    NSLog(@"Run block");
};
block();
```

После определения и первого создания блоков можно перейти к передаче их в качестве параметра методам языка Objective-C:

```
typedef NSString * (^intToString)(int intValue); // Блок

- (NSString *)intToString:(int)intValue usingBlock:(intToString)block {
    return block(intValue);
}
```

Блоки в Objective-C

Рассмотрим пример сортировки массива с применением блоков:

```
        NSArray *array = [NSArray arrayWithObjects:@4, @1, @3, @5, @0, @2, nil];
        NSArray *sortedArray = [array
sortedArrayUsingComparator:^NSComparisonResult(id  _Nonnull obj1, id  _Nonnull
obj2) {
            NSInteger first = [obj1 integerValue];
            NSInteger second = [obj2 integerValue];
            if (first > second) {
                return NSOrderedDescending;
            }
            if (first < second) {
                return NSOrderedAscending;
            }
            return NSOrderedSame;
        }];
        NSLog(@"array %@", sortedArray);
```

При выполнении программы в блоке проверяется соответствие параметру и возвращается поведение. В результате массив будет отсортирован от меньшего значения к большему.

Чтобы использовать более подходящее имя для блока, применяют оператор **typedef**:

```
typedef int(^SquareBlock)(int);

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

        SquareBlock square = ^(int number) {
            return number * number;
        };
        int result = square(3);
        NSLog(@"Result - %i", result); // Result - 9

    }
    return 0;
}
```

Блоки и переменные

После объявления блок сохраняет состояние, которое было задано в точке создания. Также блоки имеют доступ к переменным:

- Глобальные переменные, включая локальные статические переменные во внешней области видимости;
- Глобальные функции;
- Параметры, переданные из внешней области видимости;
- Переменные **__block** на уровне функции;
- Нестатические переменные в охватывающей области захватываются как константы;
- Переменные экземпляра в языке Objective-C;
- Локальные переменные внутри блока.

Рассмотрим пример использования локальных переменных в блоке:

```
int first = 30;
int second = 20;

int (^subtraction)(void) = ^(void) {
    return first - second;
};

int firstResult = subtraction();

NSLog(@"First result - %i", firstResult); // 10

first = 100;
second = 50;

int secondResult = subtraction();

NSLog(@"Second result - %i", secondResult); // 10
```

С точки зрения логики, первый и второй результат должны отличаться: ведь перед вторым выполнением блока значения переменных были изменены. Но блок запомнил состояние в точке создания, и изменение значений этих переменных никак не повлияет на его выполнение.

Ключевое слово __block

Чтобы перехватить значение из блока, необходимо перед переменной, которой будет присваиваться переменная из блока, поставить ключевое слово **__block**.

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

        __block int result = 0;

        void (^square)(int) = ^(int number) {
            result = number * number;
        };

        square(3);

        NSLog(@"Result - %i", result); // Result - 9

    }
    return 0;
}
```

Но некоторые переменные все же нельзя объявить с применением ключевого слова **__block**. Ограничение распространяется на массивы переменной длины и структуры, содержащие массивы переменной длины.

Многопоточное программирование

Многопоточность – это свойство операционной системы, предоставляющее возможность выполнять процессы в различных потоках без определенного временного порядка.

Современные операционные системы позволяют нескольким задачам выполняться параллельно даже на одном процессоре. Многоядерные процессоры способны выполнять несколько задач одновременно.

Grand Central Dispatch

Grand Central Dispatch (GCD) – это низкоуровневый API, созданный компанией Apple на языке C. Он упрощает выполнение блоков кода в разных потоках. Разработчикам не приходится работать с очередями напрямую: они взаимодействуют только с диспетчерскими очередями, распределяя по ним конкретные задачи.

Существует несколько режимов работы с GCD: синхронный, асинхронный, с определенной задержкой и прочие.

Для работы с GCD не нужно импортировать дополнительные библиотеки, так как поддержка этого API уже добавлена в Foundation. Все существующие методы начинаются с ключевого слова **dispatch_**.

Рассмотрим виды диспетчерских очередей:

- **Главная очередь (Main queue)** – выполняет все ключевые задачи, а также те, что связаны с пользовательским интерфейсом (UI). Cocoa требует вызывать и выполнять все методы, которые относятся к пользовательскому интерфейсу, именно в главном потоке;
- **Параллельные очереди (Concurrency queue)** – позволяют выполнять синхронные и асинхронные задачи, не связанные с пользовательским интерфейсом;
- **Последовательные очереди (Serial queue)** – выполняют задачи по принципу «первый вошел – первым вышел». Для этих очередей не существует разницы между синхронным и асинхронным выполнением, так как все задачи выполняются последовательно.

В любой момент жизненного цикла приложения можно одновременно использовать несколько диспетчерских очередей. Можно создавать и собственные очереди. Диспетчерским очередям можно передавать задачи в виде блоковых объектов и функций C.

Взаимная блокировка (deadlock) – ситуация, при которой две или более конкурирующих задач ожидают завершения остальных.

Последовательные очереди

Встречаются задачи, требующие применять именно последовательные очереди. Например, если необходимо реализовать в приложении правило очереди, или выполнение следующей задачи зависит от предыдущей. Такие очереди исключают риск взаимных блокировок.

Рассмотрим объявление последовательной очереди:

```
dispatch_queue_t serial_queue = dispatch_queue_create("ru.example.serialQueue",
NULL);
```

- **dispatch_queue_t** – это тип для очередей;
- **serial_queue** – произвольное название очереди;
- **dispatch_queue_create()** – функция для создания очереди, которая принимает на вход два аргумента. Первый – это уникальное имя очереди, а второй – ее атрибуты;

Сразу после создания очереди в нее можно поставить задачи. Очереди также имеют счетчики ссылок, так что и с ними нельзя не забывать про память.

Параллельные очереди

Ключевая особенность параллельных очередей – возможность выполнять следующую задачу, не дожидаясь завершения предыдущей. Количество задач, которые могут выполняться одновременно, невозможно предсказать – это зависит от загруженности процессора в конкретный момент времени. Параллельные задачи имеют приоритет, задающий скорость их выполнения.

Для получения параллельных очередей используется метод **dispatch_queue_global_queue**:

```
dispatch_queue_t queue = dispatch_get_global_queue(QOS_CLASS_DEFAULT, 0);
```

Главная очередь

Для доступа к главной очереди воспользуемся методом **dispatch_get_main_queue**:

```
dispatch_queue_t queue = dispatch_get_main_queue();
```

Помним, что главная очередь отвечает за пользовательский интерфейс и множество других задач. Ее неграмотное использование может привести к блокированию приложения.

Добавление заданий

Добавить задание в очередь можно двумя способами:

- Синхронно – очередь будет ожидать завершения задания;
- Асинхронно – функция возвращает управление сразу после добавления задания в очередь, не дожидаясь ее завершения. Этот метод предпочтителен, так как он не блокирует другие задачи.

Выполнение задачи синхронно в параллельных очередях:

```
dispatch_queue_t globalQueue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

int a = 10;
int b = 20;
__block int c;

dispatch_sync(globalQueue, ^{
    c = a + b;
});
```

Используется метод **dispatch_sync()**. Первым аргументом у него является очередь, в которой необходимо выполнить задания, а второй аргумент – это блок с самим заданием.

Выполнение задачи асинхронно в главной очереди:

```
dispatch_queue_t mainQueue = dispatch_get_main_queue();

int a = 10;
int b = 20;
__block int c;

dispatch_async(mainQueue, ^{
    c = a + b;
});
```

Для этого применяется метод **dispatch_async()**, у которого первым аргументом тоже является очередь, а вторым – блок с заданием.

Управление очередью

Чтобы остановить выполнение очереди, воспользуемся методом **dispatch_suspend**, который принимает необходимую очередь.

```
dispatch_queue_t myQueue = dispatch_queue_create("ru.example.myQueue", NULL);
dispatch_suspend(myQueue);
```

По выполнении этого метода очередь будет остановлена, но все запущенные ранее задачи будут завершены.

Чтобы вернуть очередь к выполнению, вызывается специальный метод **dispatch_resume**.

```
dispatch_queue_t myQueue = dispatch_queue_create("ru.example.myQueue", NULL);
dispatch_resume(myQueue);
```

Глобальные очереди

Компания Apple предоставляет разработчикам 4 вида глобальных очередей с разным качеством обслуживания (**qos**) и приоритетами:

```
dispatch_queue_t user_interactive =
dispatch_get_global_queue(QOS_CLASS_USER_INTERACTIVE, 0);
dispatch_queue_t user_initiated =
dispatch_get_global_queue(QOS_CLASS_USER_INITIATED, 0);
dispatch_queue_t utility = dispatch_get_global_queue(QOS_CLASS_UTILITY, 0);
dispatch_queue_t back = dispatch_get_global_queue(QOS_CLASS_BACKGROUND, 0);
dispatch_queue_t defaultQueue = dispatch_get_global_queue(QOS_CLASS_DEFAULT, 0);
```

- **QOS_CLASS_USER_INTERACTIVE** – для заданий, которые взаимодействуют с пользователем в данный момент и занимают очень мало времени. Применяется, если нет необходимости использовать Main queue, но задача должна быть выполнена как можно быстрее (когда пользователь ждет результата). Имеет высокий приоритет, но ниже чем, у Main queue;
- **QOS_CLASS_USER_INITIATED** – для заданий, которые инициируются пользователем и требуют обратной связи, но не внутри интерактивного события (пользователь ждет обратной связи, чтобы продолжить взаимодействие). Имеет высокий приоритет, но ниже, чем у двух предыдущих;
- **QOS_CLASS_UTILITY** – для заданий, которые требуют времени для выполнения, а срочную обратную связь предоставлять не нужно. Это может быть загрузка данных, очистка базы данных. Эти задачи выполняются в тени от пользователя, но они важны для корректной работы. Приоритет ниже, чем у предыдущих очередей;
- **QOS_CLASS_BACKGROUND** – для заданий, не связанных с визуализацией и не критичных по времени выполнения. Как правило, это синхронизация с сервером. Обычно задачи, которые выполняются в фоне, занимают значительное время. Имеет самый низкий приоритет среди всех глобальных очередей.

Также существует очередь по умолчанию – **QOS_CLASS_DEFAULT**. В зависимости от контекста система определит, какую глобальную очередь использовать. Это может быть либо **QOS_CLASS_USER_INTERACTIVE**, либо **QOS_CLASS_USER_INITIATED**.

Собственные очереди являются глобальными и по умолчанию будут иметь **QOS_CLASS_DEFAULT**. При создании можно использовать атрибут целевой очереди:

```
dispatch_queue_t userInitiatedQueue =
dispatch_get_global_queue(QOS_CLASS_USER_INITIATED, 0);
dispatch_queue_t myQueue =
dispatch_queue_create_with_target("ru.example.myQueue", NULL,
userInitiatedQueue);
```

При использовании целевой очереди собственная будет работать на ней. Это позволяет выполнять задачи с приоритетом, который имеет данная очередь.

Важно помнить, что все глобальные очереди являются системными, и добавленные задачи – не единственные в них.

Синхронизация

Существуют такие задачи, после выполнения которых не в главном потоке необходимо обновить пользовательский интерфейс. Для этого подойдет синхронизация. Рассмотрим пример:

```
dispatch_async(queue, ^{
    [api loadImageWithCompletion: ^(UIImage *image){
        dispatch_async(dispatch_get_main_queue(), ^{
            imageView.image = image;
        });
    }];
});
```

Изначально загружается картинка с помощью метода **loadImageWithCompletion**. В результате загрузки картинка возвращается посредством блока, и ее необходимо разместить на экране. Но с пользовательским интерфейсом нельзя работать не в главном потоке. В этом случае можно добавить еще одну задачу в главный поток – для обновления картинки. После загрузки картинки она будет успешно установлена на экран.

Группы

Чтобы обрабатывать задачи в разных потоках, но по окончании их выполнения получить общий результат, используют группу. Она позволяет добавлять задачи, запускать их, а по окончании выполнения получать уведомление. Рассмотрим пример:

```
__block NSInteger a = 2;
__block NSInteger b = 4;

dispatch_group_t group = dispatch_group_create();
dispatch_queue_t queue = dispatch_get_global_queue(QOS_CLASS_UTILITY, 0);
dispatch_group_async(group, queue, ^{
    a = a + 3;
});
dispatch_group_async(group, queue, ^{
    b = b + 4;
});

dispatch_group_notify(group, queue, ^{
    NSLog(@"a - %d, b - %d", a, b); // a - 5, b - 8
});
```

Изначально создается группа и очередь, в которой будут выполняться задачи. Затем в группу добавляются задачи. После вызывается метод, который уведомит об окончании выполнения поставленных задач. Так как задачи были успешно выполнены, в уведомлении их значение будет уже 5 и 8 (a и b соответственно).

Барьеры

Барьеры GCD решают важную задачу: они ожидают момента, когда очередь будет полностью пуста, чтобы выполнить свой блок. С начала своего выполнения блок барьера обеспечивает, что очередь не выполняет никаких иных задач и в течение этого времени по существу работает как синхронная функция. По окончании блока барьера очередь восстанавливается и гарантирует, что никакая запись не будет проводиться одновременно с чтением или другой записью.

Рассмотрим пример:

```
dispatch_queue_t queue = dispatch_queue_create("ru.example.queue", NULL);
dispatch_barrier_async(queue, ^{
    NSLog(@"Barrier");
});

dispatch_async(queue, ^{
    NSLog(@"Another task");
});
```

Таким образом, вторая задача не будет начата до момента выполнения первой.

Семафор

Семафор позволяет одновременно выполнять участок кода только конкретному количеству потоков. В его основе лежит счетчик, который и определяет, можно ли выполнять участок кода текущему потоку или нет. Если счетчик больше нуля – поток выполняет код, в противном случае – нет.

Семафор в GCD представлен типом **dispatch_semaphore_t**. Для создания семафора существует функция **dispatch_semaphore_create**. Она принимает один аргумент – число потоков, которые могут одновременно выполнять участок кода. Выглядит это так:

```
dispatch_semaphore_t semaphore = dispatch_semaphore_create(1);
```

В данном случае семафор позволит только одному потоку выполнять код. Если мы больше не нуждаемся в созданном семафоре, нужно его уничтожить (при выключенном ARC):

```
dispatch_release(semaphore);
```

Операция ожидания семафора осуществляется с помощью функции **dispatch_semaphore_wait**. У нее два аргумента: первый – семафор, второй –время ожидания. Когда семафор станет больше нуля, функция **dispatch_semaphore_wait** «отпустит» поток, и нужный участок кода будет выполнен. После этого обязательно нужно вызвать функцию **dispatch_semaphore_signal** (третья операция) и передать ей семафор. Эта функция увеличивает значение семафора на единицу. Пример:

```
// ожидаем бесконечно
dispatch_semaphore_wait(semaphore, DISPATCH_TIME_FOREVER);

/* код */

// увеличиваем значение семафора
dispatch_semaphore_signal(semaphore);
```

Использование NSOperationQueue

Для добавления операций в очереди вместо **dispatch_queue_t** будет применяться **NSOperationQueue**. Это специальный класс, который позволяет добавлять операции в очереди и всегда выполняет их параллельно. Он учитывает зависимости одной операции от другой, так что они будут выполнены согласовано.

Чтобы использовать очередь, можно воспользоваться следующими способами:

```
// Текущая очередь
NSOperationQueue *currentQueue = [NSOperationQueue currentQueue];
// Главная очередь
NSOperationQueue *mainQueue = [NSOperationQueue mainQueue];

// Создание собственной очереди
NSOperationQueue *customQueue = [[NSOperationQueue alloc] init];
```

Для добавления операций в очередь применяются специальные методы. Один из них – это **создание подкласса для NSOperation** и добавление операции в очередь посредством метода **addOperation**.

Создадим наследника класса **NSOperation**:

```
@implementation Operation

- (void)start {
    NSLog(@"OPERATION START");
}

- (void)main {
    NSLog(@"RESULT");
}

- (BOOL)isCancelled {
    NSLog(@"isCancelled");
    return [super isCancelled];
}

- (BOOL)isFinished {
    NSLog(@"isFinished");
    return [super isFinished];
}

-(BOOL)isReady {
    NSLog(@"isReady");
    return [super isReady];
}

@end
```

В методе **main** находится сама задача. Метод **start** вызывается при ее запуске. Можно проверять состояние задачи: при создании она будет готова к исполнению («**isReady**»), в конце выполнения – будет завершена («**isFinished**»). Задачу можно отменить и проверить, не отменена ли она («**isCancelled**»). Для ее отмены необходимо самостоятельно реализовать специальный метод **cancel**.

Чтобы выполнить задачу, выбираем очередь, и, воспользовавшись методом, добавляем операцию:

```
NSOperationQueue *mainQueue = [NSOperationQueue mainQueue];

Operation *operation = [[Operation alloc] init];

[mainQueue addOperation:operation];
```

В результате в консоли будет выведено сообщение «**Print using NSOperation**». Для этого будет использована главная очередь.

Второй способ добавления задания в очередь – это **блоки.** Для этого применяется метод **addOperationWithBlock**, где в параметре указывается сам блок:

```
NSOperationQueue *mainQueue = [NSOperationQueue mainQueue];

[mainQueue addOperationWithBlock:^{
    NSLog(@"Print using NSOperationQueue");
}];
```

Будет выведено сообщение «**Print using NSOperationQueue**».

Зависимости

Зависимости – это способ сериализации выполнения отдельных объектов операции. Операция, которая зависит от других, не может начаться до их полного выполнения. Зависимости можно использовать для создания простых, один к одному, зависимостей между двумя объектами операции, или строить сложные графики зависимостей объекта.

Чтобы установить зависимость между двумя объектами операции, используют метод NSOperation **addDependency:**. Он создает одностороннюю зависимость от текущей операции объекта в целевую операцию, указанную в качестве параметра. Это означает, что текущий объект не может начать выполнение до завершения целевого объекта.

Зависимости не ограничиваются операцией в той же очереди. Объекты операции управляют своими зависимостями, и поэтому вполне приемлемо при создании зависимостей между операциями добавить их в различные очереди. Но циклические зависимости между операциями создавать нельзя.

Когда все зависимости между операциями установлены, объект обычно готов для выполнения (если вы настроите поведение метода **isReady**, и готовность работы будет определяться в соответствии с установленными вами критериями). Если объект операции находится в очереди, она может начать выполнение этой операции в любое время. Если вы планируете выполнять операции вручную, вызовите метод операции **start**.

Рассмотрим пример зависимости операций:

```
    NSOperationQueue *mainQueue = [NSOperationQueue mainQueue];

    Operation *operation = [[Operation alloc] init];
    Operation *operation2 = [[Operation alloc] init];
    [operation addDependency:operation2];

    [mainQueue addOperation:operation];
```

При исполнении данного кода не будет вызвана ни одна задача, так как для первой операции была добавлена зависимость от второй. Это означает, что пока не будет выполнена вторая, не реализуется и первая.

Практика

Создание калькулятора с применением блоков

Чтобы производить расчеты, создадим класс **Arithmetic**. Также добавим перечисление с возможными методами:

```
typedef int (^ArithmeticBlock) (int a, int b);

typedef enum ActionType {
    ActionTypeSum,
    ActionTypeSubstraction,
    ActionTypeMultiplication,
    ActionTypeDivision,
    ActionTypeRemainderOfTheDivision
} ActionType ;

@interface Arithmetic : NSObject

+ (int)beginWithAction:(ActionType)action firstNumber: (int)first secondNumber:
(int)second;

@end
```

Для расчетов будем использовать статический метод **beginWithAction**. Статический метод объявляется с помощью знака «+» перед ним. Он отличается от обычного тем, что для его использования не нужно инициализировать объект.

Теперь необходимо создать блоки для каждого метода:

```
ArithmeticBlock sum = ^(int a, int b) {
    return a + b;
};

ArithmeticBlock substraction = ^(int a, int b) {
    return a - b;
};

ArithmeticBlock multiplication = ^(int a, int b) {
    return a * b;
};

ArithmeticBlock division = ^(int a, int b) {
    return a / b;
};

ArithmeticBlock remainderOfTheDivision = ^(int a, int b) {
    return a % b;
};
```

Теперь создадим объявленную функцию для воспроизведения расчетов:

```
@implementation Arithmetic

+ (int)beginWithAction:(ActionType)action firstNumber:(int)first
secondNumber:(int)second {

    switch (action) {
        case ActionTypeSum:
            return sum(first, second);
            break;
        case ActionTypeSubstraction:
            return substraction(first, second);
            break;
        case ActionTypeMultiplication:
            return multiplication(first, second);
            break;
        case ActionTypeDivision:
            return division(first, second);
            break;
        case ActionTypeRemainderOfTheDivision:
            return remainderOfTheDivision(first, second);
            break;
    }

}

@end
```

После реализации можно использовать данный класс. Опробуем каждый метод и выведем результаты в консоль:

```
#import "Arithmetic.h"

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

        int first = 10;
        int second = 2;

        int resultSum = [Arithmetic beginWithAction:ActionTypeSum
firstNumber:first secondNumber:second];
        int resultSubstraction = [Arithmetic
beginWithAction:ActionTypeSubstraction firstNumber:first secondNumber:second];
        int resultMultiplication = [Arithmetic
beginWithAction:ActionTypeMultiplication firstNumber:first secondNumber:second];
        int resultDivision = [Arithmetic beginWithAction:ActionTypeDivision
firstNumber:first secondNumber:second];
        int resultRemainderOfTheDivision = [Arithmetic
beginWithAction:ActionTypeRemainderOfTheDivision firstNumber:first
secondNumber:second];

        NSLog(@"Result sum - %i", resultSum);
        NSLog(@"Result substraction - %i", resultSubstraction);
        NSLog(@"Result multiplication - %i", resultMultiplication);
        NSLog(@"Result division - %i", resultDivision);
        NSLog(@"Result remainder of the division - %i",
resultRemainderOfTheDivision);

    }
    return 0;
}
```

Результатом выполнения программы будет следующая запись в консоли:

```
Result sum - 12
Result substraction - 8
Result multiplication - 20
Result division - 5
Result remainder of the division - 0
```

Реализован калькулятор с применением блоков.

Практическое задание

1. Попрактиковаться с применением блоков: создать программу для вывода сообщений в консоль с использованием минимум 6 блоков.
2. Добавить выполнение блоков в различные очереди с применением GCD.

Дополнительные материалы

1. https://developer.apple.com/documentation/dispatch?language=objc;
2. https://habrahabr.ru/post/320152/;
3. Стивен Кочан. «Программирование на Objective-C»;
4. 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. https://habrahabr.ru/post/320152/
2. Стивен Кочан. «Программирование на Objective-C».
3. Скотт Кнастер, Вакар Малик, Марк Далримпл. «Objective-C. Программирование для Mac OS X и iOS».