GeekBrains

# Лекция 4. Блоки и многопоточное программирование в Objective-C.

Узнаем, что такое блок и как он работает в Objective – C. Рассмотрим работу в многопоточной среде, вспомним, что такое поток и изучим некоторые технологии от компании Apple, которые помогут работать с потоками на более высоком уровне, а именно Grand Center Dispatch и Operation.

# Оглавление

# Блоки

# Блочные объекты

Блоки – это тип в Objective-C, который хранит в себе функцию. Они схожи со стандартными функциями языка C или closure в Swift, 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
Изначально указывается возвращаемый тип, затем в скобках – ключевой символ «^», после него – название блока, а затем в скобках – принимаемые параметры. Ему присваивается функция, схожая с С-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);
}

```

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**:

```objc
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.

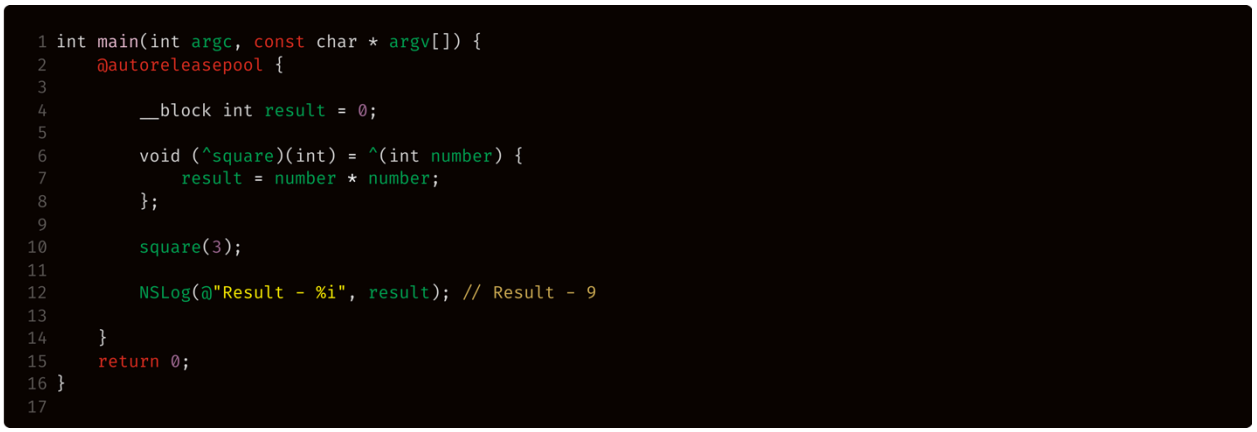
Рассмотрим пример использования локальных переменных в блоке:

```objc
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**.



```objc
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.

# Введение в параллелизм

Параллелизм — это понятие нескольких вещей, происходящих одновременно. С распространением многоядерных процессоров и осознанием того, что количество ядер в каждом процессоре будет только увеличиваться, разработчикам программного обеспечения нужны новые способы их использования. Хотя такие операционные системы, как OS X и iOS, могут запускать несколько программ параллельно, большинство этих программ работают в фоновом режиме и выполняют задачи, требующие небольшого непрерывного процессорного времени. Это текущее приложение переднего плана, которое одновременно привлекает внимание пользователя и занимает компьютер. Если у приложения много работы, но занята лишь часть доступных ядер, эти дополнительные вычислительные ресурсы тратятся впустую.

К сожалению, написание многопоточного кода является сложной задачей. Потоки — это низкоуровневый инструмент, которым нужно управлять вручную. Учитывая, что

оптимальное количество потоков для приложения может динамически изменяться в зависимости от текущей загрузки системы и базового оборудования, реализация правильного решения для многопоточности становится чрезвычайно сложной, если не невозможной. Кроме того, механизмы синхронизации, обычно используемые с потоками, усложняют и повышают риск разработки программного обеспечения без каких-либо гарантий повышения производительности.

Как в OS X, так и в iOS используется более асинхронный подход к выполнению параллельных задач, чем это традиционно используется в системах и приложениях на основе потоков. Вместо того, чтобы создавать потоки напрямую, приложениям нужно только определить конкретные задачи, а затем позволить системе выполнить их. Позволяя системе управлять потоками, приложения получают уровень масштабируемости, недостижимый при необработанных потоках. Разработчики приложений также получают более простую и эффективную модель программирования.

В данном разделе описываются приемы и технологии, которые вы должны использовать для реализации параллелизма в своих приложениях. Технологии, описанные в этой лекции, доступны как в OS X, так и в iOS. В данном разделе мы рассмотрим следующие технологии:

- Познакомимся с основами разработки асинхронных приложений и технологиями для асинхронного выполнения ваших пользовательских задач.
- Рассмотрим Operation Queue, данная технология показывает, как инкапсулировать и выполнять задачи с использованием объектов Objective-C.
- Рассмотрим Dispatch Queues, данная технология показывает, как выполнять задачи одновременно в приложениях на основе языка C.
- Dispatch Sources показывает, как асинхронно обрабатывать системные события

# Примечание о терминологии

Прежде, чем приступить к обсуждению параллелизма, необходимо определить некоторые соответствующие термины, чтобы избежать путаницы. Разработчики, более знакомые с системами UNIX или старыми технологиями OS X, могут обнаружить, что термины «задача», «процесс» и «поток» используются в этом документе несколько иначе. В данном разделе эти термины используются следующим образом:

- Термин *поток* используется для обозначения отдельного пути выполнения кода. Базовая реализация потоков в OS X основана на API потоков POSIX.
- Термин *процесс* используется для обозначения работающего исполняемого файла, который может охватывать несколько потоков.
- Термин «задача» используется для обозначения абстрактной концепции работы, которую необходимо выполнить.

# Параллелизм

На заре вычислительной техники максимальный объем работы в единицу времени, которую мог выполнить компьютер, определялся тактовой частотой процессора. Но по мере развития технологий и компактности конструкции процессоров тепловые и другие физические ограничения стали ограничивать максимальную тактовую частоту процессоров. И поэтому производители чипов искали другие способы увеличить общую производительность своих чипов. Решение, на котором они остановились, заключалось в увеличении количества процессорных ядер на каждом чипе. Увеличивая количество ядер, один чип может выполнять больше инструкций в секунду без увеличения скорости процессора, изменения размера чипа или тепловых характеристик. Единственная проблема заключалась в том, как использовать дополнительные ядра.

Чтобы использовать преимущества нескольких ядер, компьютеру необходимо программное обеспечение, которое может выполнять несколько задач одновременно. В современной многозадачной операционной системе, такой как OS X или iOS, в любой момент времени может выполняться сотня или более программ, поэтому должно быть возможно планирование каждой программы на отдельном ядре. Однако большинство этих программ являются либо системными демонами, либо фоновыми приложениями, которые потребляют очень мало реального времени обработки. Вместо этого, что действительно необходимо, так это способ более эффективного использования дополнительных ядер отдельными приложениями.

Традиционным способом использования нескольких ядер приложением является создание нескольких потоков. Однако по мере увеличения количества ядер возникают проблемы с многопоточными решениями. Самая большая проблема заключается в том, что многопоточный код не очень хорошо масштабируется до произвольного количества ядер. Вы не можете создать столько потоков, сколько ядер, и ожидать, что программа будет работать хорошо. Даже если вам удастся

получить правильные числа, все еще остается проблема программирования для такого количества потоков, обеспечения их эффективной работы и предотвращения их взаимодействия друг с другом.

Итак, подытоживая проблему, нужно, чтобы приложения могли использовать переменное количество вычислительных ядер. Объем работы, выполняемой одним приложением, также должен иметь возможность динамически масштабироваться, чтобы приспосабливаться к меняющимся системным условиям. И решение должно быть достаточно простым, чтобы не увеличивать объем работы, необходимой для использования этих ядер. Хорошей новостью является то, что операционные системы Apple обеспечивают решение всех этих проблем, и в этой главе рассматриваются технологии, составляющие это решение.

# Отход от потоков

Хотя потоки существуют уже много лет и продолжают использоваться, они не решают общую проблему масштабируемого выполнения нескольких задач. При использовании потоков бремя создания масштабируемого решения полностью ложится на ваши плечи, как на разработчика. Вы должны решить, сколько потоков создать, и динамически изменять это число по мере изменения состояния системы. Другая проблема заключается в том, что ваше приложение берет на себя большую часть затрат, связанных с созданием и обслуживанием любых используемых им потоков.

Вместо того, чтобы полагаться на потоки, OS X и iOS используют *асинхронный подход к проектированию*, который помогает решать проблемы параллелизма. Асинхронные функции присутствуют в операционных системах уже много лет и часто используются для инициирования задач, которые могут занять много времени, например чтение данных с диска. При вызове асинхронная функция выполняет некоторую работу за кулисами, чтобы запустить выполнение задачи, но возвращается до того, как эта задача может быть фактически завершена. Как правило, эта работа включает в себя получение фонового потока, запуск нужной задачи в этом потоке, а затем отправку уведомления вызывающей стороне (обычно через функцию обратного вызова) после выполнения задачи. Раньше, если для того, что вы хотите сделать, не существовало асинхронной функции, вам приходилось писать свою собственную асинхронную функцию и создавать собственные потоки.

Одна из технологий асинхронного запуска задач — *Grand Central Dispatch (GCD)*. Эта

технология берет код управления потоками, который вы обычно пишете в своих собственных приложениях, и перемещает этот код на системный уровень. Все, что вам нужно сделать, это определить задачи, которые вы хотите выполнить, и добавить их в соответствующую очередь отправки (dispatch queue). GCD позаботится о создании необходимых потоков и о планировании выполнения ваших задач в этих потоках. Поскольку управление потоками теперь является частью системы, GCD обеспечивает целостный подход к управлению задачами и их выполнению, обеспечивая более высокую эффективность, чем традиционные потоки.

Следующая технология – это, *Operation Queue* — объекты Objective-C, которые очень похожи на Dispatch Queue. Вы определяете задачи, которые хотите выполнить, а затем добавляете их в очередь операций, которая занимается планированием и выполнением этих задач. Как и GCD, очереди операций берут на себя все управление потоками за вас, обеспечивая максимально быстрое и эффективное выполнение задач в системе.

# Методы асинхронного проектирования

Прежде чем вы даже подумаете о перепроектировании своего кода для поддержки параллелизма, вы должны спросить себя, необходимо ли это делать. Параллелизм может повысить скорость отклика вашего кода, гарантируя, что ваш основной поток может свободно реагировать на пользовательские события. Это может даже повысить эффективность вашего кода, используя больше ядер для выполнения большего объема работы за то же время. Однако это также добавляет накладные расходы и увеличивает общую сложность вашего кода, что затрудняет его написание и отладку.

Параллелизм не является функцией, которую можно внедрить в приложение в конце производственного цикла, поскольку это усложняет его работу. Чтобы сделать это правильно, необходимо внимательно рассмотреть задачи, которые выполняет ваше приложение, и структуры данных, используемые для выполнения этих задач. Если вы сделаете это неправильно, вы можете обнаружить, что ваш код работает медленнее, чем раньше, и хуже реагирует на действия пользователя. Поэтому стоит уделить некоторое время в начале цикла проектирования, чтобы установить некоторые цели и подумать о подходе, который вам необходимо использовать.

Каждое приложение имеет разные требования и свой набор задач, которые оно выполняет. Документ не может точно сказать вам, как проектировать ваше приложение и связанные с ним задачи. Однако в следующих разделах мы попытаемся дать некоторые рекомендации, которые помогут вам сделать правильный выбор в процессе проектирования.

# Ожидаемое поведение вашего приложения

Прежде чем вы даже подумаете о добавлении параллелизма в свое приложение, вы всегда должны начинать с определения того, что вы считаете правильным поведением вашего приложения. Понимание ожидаемого поведения вашего приложения дает вам возможность позже проверить свой проект. Это также должно дать вам некоторое представление об ожидаемых преимуществах производительности, которые вы можете получить, внедрив параллелизм.

Первое, что вы должны сделать, это перечислить задачи, которые выполняет ваше приложение, и объекты или структуры данных, связанные с каждой задачей. Первоначально вы можете начать с задач, которые выполняются, когда пользователь выбирает пункт меню или нажимает кнопку. Эти задачи предлагают дискретное поведение и имеют четко определенные начальную и конечную точки. Вам также следует перечислить другие типы задач, которые ваше приложение может выполнять без взаимодействия с пользователем, например, задачи на основе таймера.

После того, как у вас есть список задач высокого уровня, начните разбивать каждую задачу на набор шагов, которые необходимо предпринять для успешного выполнения задачи. На этом уровне вас в первую очередь должны интересовать модификации, которые необходимо внести в любые структуры данных и объекты, и то, как эти модификации влияют на общее состояние вашего приложения. Вы также должны отметить любые зависимости между объектами и структурами данных. Например, если задача включает в себя внесение одинаковых изменений в массив объектов, стоит отметить, влияют ли изменения в одном объекте на какие-либо другие объекты. Если объекты могут быть изменены независимо друг от друга, это может быть место, где вы могли бы вносить эти изменения одновременно.

# Фактор из исполняемых единиц работы

Исходя из вашего понимания задач вашего приложения, вы уже должны быть в состоянии определить места, где ваш код может выиграть от параллелизма. Если изменение порядка одного или нескольких шагов в задаче меняет результаты, вам, вероятно, придется продолжать выполнять эти шаги последовательно. Однако, если изменение порядка не влияет на вывод, вам следует рассмотреть возможность одновременного выполнения этих шагов. В обоих случаях вы определяете исполняемую единицу работы, которая представляет выполняемый шаг или шаги. Затем эта единица работы становится тем, что вы инкапсулируете с помощью блока или объекта операции и отправляете в соответствующую очередь.

Для каждой выполняемой единицы работы, которую вы идентифицируете, не слишком беспокойтесь об объеме выполняемой работы, по крайней мере на начальном этапе. Хотя раскрутка потока всегда сопряжена с затратами, одно из преимуществ очередей диспетчеризации и очередей операций заключается в том, что во многих случаях эти затраты намного меньше, чем для традиционных потоков. Таким образом, вы можете выполнять более мелкие единицы работы более эффективно, используя очереди, чем потоки. Конечно, вы всегда должны измерять свою фактическую производительность и корректировать размер ваших задач по мере необходимости, но изначально ни одна задача не должна считаться слишком маленькой.

# Определите очереди, которые вам нужны

Теперь, когда ваши задачи разбиты на отдельные единицы работы и инкапсулированы с использованием блочных объектов или объектов операций, вам необходимо определить очереди, которые вы собираетесь использовать для выполнения этого кода. Для данной задачи проверьте созданные вами блоки или объекты операций и порядок, в котором они должны выполняться для правильного выполнения задачи.

Если вы реализовали свои задачи с помощью блоков, вы можете добавить свои

блоки либо в последовательную, либо в параллельную очередь отправки. Если требуется определенный порядок, вы всегда должны добавлять свои блоки в очередь последовательной отправки. Если определенный порядок не требуется, вы можете добавить блоки в параллельную очередь отправки или добавить их в несколько разных очередей отправки, в зависимости от ваших потребностей.

Если вы реализовали свои задачи с помощью объектов операций, выбор очереди часто менее интересен, чем конфигурация ваших объектов. Для последовательного выполнения объектов операций необходимо настроить зависимости между связанными объектами. Зависимости предотвращают выполнение одной операции до тех пор, пока объекты, от которых она зависит, не закончат свою работу.

# Советы по повышению эффективности

Помимо простого разбиения кода на более мелкие задачи и добавления их в очередь, есть и другие способы повысить общую эффективность кода с помощью очередей:

- **Рассмотрите возможность вычисления значений непосредственно в вашей задаче, если важным фактором является использование памяти.** Если ваше приложение уже привязано к памяти, прямое вычисление значений может быть быстрее, чем загрузка кэшированных значений из основной памяти. Вычисление значений напрямую использует регистры и кэши данного ядра процессора, которые намного быстрее, чем основная память. Конечно, вы должны делать это только в том случае, если тестирование показывает, что это выигрыш в производительности.

- **Определите серийные задачи заранее и сделайте все возможное, чтобы сделать их более параллельными.** Если задача должна выполняться последовательно, поскольку она зависит от некоторого общего ресурса, рассмотрите возможность изменения архитектуры, чтобы удалить этот общий ресурс. Вы можете рассмотреть возможность создания копий ресурса для каждого клиента, которому он нужен, или полностью исключить ресурс.

- **Избегайте использования замков (Lock).** Поддержка, обеспечиваемая очередями отправки и очередями операций, делает блокировки ненужными в большинстве ситуаций. Вместо использования блокировок для защиты какого-либо общего ресурса назначьте последовательную очередь (или

используйте зависимости объекта операции) для выполнения задач в правильном порядке.

- **По возможности полагайтесь на системные фреймворки.** Лучший способ добиться параллелизма — воспользоваться преимуществами встроенного параллелизма, предоставляемого системными фреймворками. Многие фреймворки используют потоки и другие технологии внутри для реализации параллельного поведения. При определении ваших задач посмотрите, определяет ли существующая структура функцию или метод, который делает именно то, что вы хотите, и делает это одновременно. Использование этого API может сэкономить ваши усилия и, скорее всего, даст вам максимально возможный параллелизм.

# Влияние на производительность

Очереди операций, очереди отправки и источники отправки предоставляются для облегчения одновременного выполнения большего количества кода. Однако эти технологии не гарантируют повышения эффективности или скорости отклика вашего приложения. Вы по-прежнему несете ответственность за использование очередей таким образом, чтобы они были эффективными для ваших нужд и не создавали чрезмерной нагрузки на другие ресурсы вашего приложения. Например, хотя вы можете создать 10 000 объектов операций и отправить их в очередь операций, это приведет к тому, что ваше приложение будет выделять потенциально нетривиальный объем памяти, что может привести к подкачке и снижению производительности.

Прежде чем внедрять какой-либо параллелизм в свой код — будь то с использованием очередей или потоков — вы всегда должны собрать набор базовых показателей, отражающих текущую производительность вашего приложения. После внесения изменений вы должны собрать дополнительные метрики и сравнить их с базовым уровнем, чтобы увидеть, улучшилась ли общая эффективность вашего приложения. Если введение параллелизма делает ваше приложение менее эффективным или отзывчивым, вам следует использовать доступные инструменты повышения производительности, чтобы проверить возможные причины. Общие сведения о производительности и доступных инструментах повышения производительности, а также ссылки на более сложные темы, связанные с производительностью.

# 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.

# Литература


1. https://developer.apple.com/library/archive/documentation/General/Conceptual/ConcurrencyProgrammingGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40008091-CH1-SW1
2. https://apple.github.io/swift-corelibs-libdispatch/tutorial/
3. https://developer.apple.com/documentation/dispatch/dispatch_source
4. Стивен Кочан. «Программирование на Objective-C»;
5. Скотт Кнастер, Вакар Малик, Марк Далримпл.«Objective-C. Программирование для Mac OS X и iOS».