The beautiful and maddening thing about programming is that there are so many ways to solve the same issue. Each solution has it's own trades offs and as developers it is up to us to choose the solution that has the most positives for our problem.

Let's take networking as an example.

Before `blocks` were introduced to iOS you had to use the `delegate` on `NSURLConnection` to gradually build up the response from a network request. This resulted in a lot of boilerplate code that more often than not we would abstract away to a base/parent class. This was a good approach and ensured that each class had a well defined purpose however when blocks arrived things began to change. First, great third-party libraries like `AFNetworking` began to appear that operated on top of `NSURLConnection` and then Apple jumped into this block based environment with it's new networking layer: `NSURLSession` (AFNetworking is now built on top of `NSURLSession`). Both of these approaches offered a more developer friendly interface by using blocks and removed the need for that base/parent class we spoke about above which made making network requests much easier.

Houston we have take off .

However not everything improved - due to ease of which we could request/respond to networking requests, you started to see a lot more networking code implemented in viewcontrollers . These bloated viewcontrollers made life much harder for developers as we began having viewcontrollers being responsible for multiple different domains - configuring views, responding to user actions, making network requests and parsing network responses. As we know, a well-designed class should have a single responsibility and this erosion of a distant networking layer was making our classes violate the SRP. Now this wasn't the fault of blocks (which are awesome, btw) but rather a fault with how we choose to use them. By remembering how we used to construct the networking layer with NSURLConnection we can take the best of a distant networking layer and couple it with blocks to produce a project that is more readable and predictable.

LET'S START QUEUING

With NSOperation and NSOperationQueue we have a great way of isolating the networking code in our projects as it allows us to think of both the network request and parsing of the response as one operation/task that is executed together that is encapsulated with an NSOperation subclass. Now, you may be thinking why not do this with an NSObject subclass and GCD - it could work however I always like to choose the highest level of abstraction (NSOperationQueue is built on top of GCD) and NSOperationQueue offers other fringe benefits such as being able to cancel the queue (when a user logs out), prioritise operations inside the queue (user triggered operations are more important than app triggered operations) and even pause the queue (if the device drops internet connection). If NSOperationQueue is is new to you, check out Apple's documentation on it.

With our NSOperation we are going to go down a slightly less well travelled path - typically in your subclass you would override the main method and perform work in there however instead we need to use the start method as this operation is going to be making network request. NSURLSession will push the actual network request onto a background thread which if we used the main method would cause our NSOperation subclass to finish as the queue would think that it's work had finished. So what we need to do is override the start method and manually control NSOperation state machine to control what it's state of execution. Let's look at the properties that we are going to be overriding:

```
@property (readonly, getter=isReady) BOOL ready;
@property (readonly, getter=isExecuting) BOOL executing;
@property (readonly, getter=isFinished) BOOL finished;
```

Ready indicates that our NSOperation subclass is good to go and if the queue wants tom it can execute this operation.

Executing indicates that our NSOperation subclass is actually doing work at the moment.

Finished indicates that our NSOperation subclass has completed it's task and should be removed from the queue.

As the operation progresses through it's lifecycle we will change the values for these properties. The other property that we haven't spoken about but that is at the heart of what we are attempting to do here is:

```
@property (readonly, getter=isAsynchronous) BOOL asynchronous
```

All of our networking operations are going to be asynchronous so we will override this property to always return this. Armed with this knowledge let's build a subclass of NSOperation

(I'm going to use NWM (**N**etworking **W**ing**m**an) as the prefix)

```
@interface NWMOperation : NSOperation

/**
 Finishes the execution of the operation.

 @note - This shouldn't be called externally as this is used internally by subclasses. To
 */
- (void)finish;



@implementation NWMOperation

/*
 We need to do old school synthesizing as the compiler has trouble creating the internal
 */
@synthesize ready = _ready;
@synthesize executing = _executing;
@synthesize finished = _finished;

#pragma mark - Init

- (instancetype)init
{
    self = [super init];

    if (self)
    {
        self.ready = YES;
    }

    return self;
}

#pragma mark - State

- (void)setReady:(BOOL)ready
{
    if (_ready != ready)
    {
        [self willChangeValueForKey:NSStringFromSelector(@selector(isReady))];
        _ready = ready;
        [self didChangeValueForKey:NSStringFromSelector(@selector(isReady))];
    }
}

- (BOOL)isReady
{
    return _ready;
}

- (void)setExecuting:(BOOL)executing
{
    if (_executing != executing)
    {
        [self willChangeValueForKey:NSStringFromSelector(@selector(isExecuting))];
        _executing = executing;
        [self didChangeValueForKey:NSStringFromSelector(@selector(isExecuting))];
    }
```

```
}

- (BOOL)isExecuting
{
    return _executing;
}

- (void)setFinished:(BOOL)finished
{
    if (_finished != finished)
    {
        [self willChangeValueForKey:NSStringFromSelector(@selector(isFinished))];
        _finished = finished;
        [self didChangeValueForKey:NSStringFromSelector(@selector(isFinished))];
    }
}

- (BOOL)isFinished
{
    return _finished;
}

- (BOOL)isAsynchronous
{
    return YES;
}

#pragma mark - Control

- (void)start
{
    if (!self.isExecuting)
    {
        self.ready = NO;
        self.executing = YES;
        self.finished = NO;

        NSLog(@"\"%@\" Operation Started.", self.name);
    }
}

- (void)finish
{
    if (self.executing)
    {
        NSLog(@"\"%@\" Operation Finished.", self.name);

        self.executing = NO;
        self.finished = YES;
    }
}

@end
```

There is a lot of code there but really it is overriding the state machine of `NSOperation` and allowing us to encapsulate the behaviour needed to create asynchronous operations.

So thats the parent class but let's begin fresh this out by actually making a network request. Stackoverflow has a wonderful open API that I'm going to use in the below example.

```
@interface NWMAnswersOperation ()

/**
```

```
 Completion block to be called once the the request and parsing is completed. Will returr
 */
@property (nonatomic, copy) void (^completion)(NSArray *answers);

@end

@implementation NWMAnswersOperation

#pragma mark - Init

- (instancetype)initWithCompletion:(void (^)(NSArray *answers))completion
{
    self = [super init];

    if (self)
    {
        self.completion = completion;
        self.name = @"Answers-Retrieval";
    }

    return self;
}

#pragma mark - Start

- (void)start
{
    [super start];

    NSURLSession *session = [NSURLSession sharedSession];

    NSURL *url = [NSURL URLWithString:@"https://api.stackexchange.com/2.2/answers?site=st

    NSURLSessionDataTask *task = [session dataTaskWithURL:url
                                        completionHandler:^(NSData * _Nullable data, NSUI
                                    {
                                        NSDictionary *answersJSON = [NSJSONSerialization JS



                                        NWMAnswerParser *parser = [[NWMAnswerParser alloc]
                                        NSArray *answers = [parser parseAnswers:answersJSON

                                        if (self.completion)
                                        {
                                            self.completion(answers);
                                        }

                                        [self finish];
                                    }];

    [task resume];
}

#pragma mark - Cancel

- (void)cancel
{
    [super cancel];

    [self finish];
}
@end
```

In the above example we are retrieving the data from the answers end point, parsing that data, calling the completion block and finishing the operation (so that the next operation can be executed).

GETTING THE MANAGERS

`NSOperation` and `NSURLSession` work perfectly together to allow you to treat network request and parsing of the response as one task while keeping the main thread free to care about displaying the data and responding to the users actions. However this is only part of the story as we don't want our viewcontrollers to have to access an `NSOperationQueue` to schedule these network requests.

What we want to do is create a dedicated class that is responsible for creating and scheduling our network requests.

```
/**
 Class to handle all API requests related to answers.
 */
@interface NWMAnswersAPIManager : NSObject

/**
 Retrieve answers from API.

 @param completion - block that will be called once network request has been completed. W
 */
+ (void)retrieveAnswersWithCompletion:(void (^)(NSArray *answers))completion;

@end
```

```
@implementation NWMAnswersAPIManager

#pragma mark - Retrieval

+ (void)retrieveAnswersWithCompletion:(void (^)(NSArray *answers))completion
{
    NWMAnswersRetrievalOperation *operation = [[NWMAnswersRetrievalOperation alloc] initW

    [[NWMOperationQueueManager sharedInstance] addOperation:operation];
}

@end
```

The above manager will schedule the operation to be executed on the relevant queue and allow the viewcontroller to interact with a class method interface (no need for messy instances of the manager to litter your viewcontroller code).

The sharper reader will have spotted another custom class `NWMOperationQueueManager` - this class is responsible for the queue (or queues) that our operation will be added to.

```
/**
 This class coordinates the operations.
 */
@interface NWMOperationQueueManager : NSObject

/**
 Returns the global NWMOperationQueueManager instance.
```

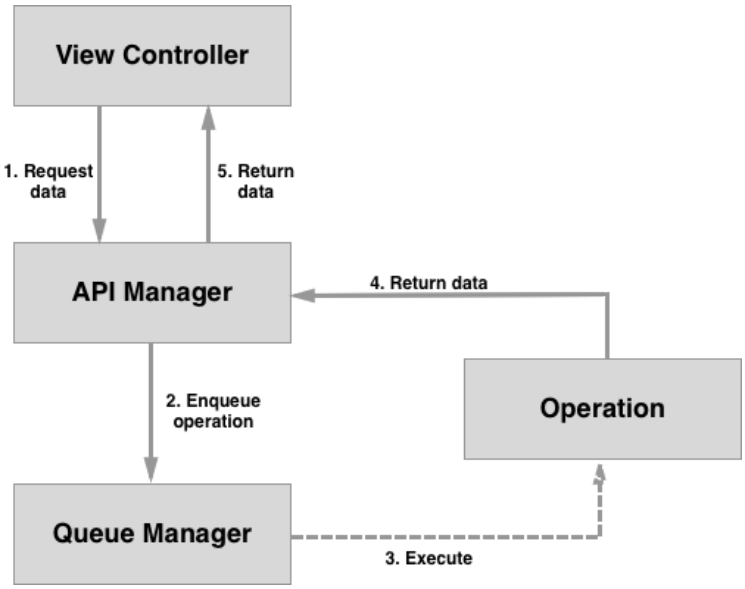
```
 @return NWMOperationQueueManager instance.
 */
+ (instancetype)sharedInstance;

/**
 Add an operation to an operation queue.

 @param operation - the new operation to be added.
 */
- (void)addOperation:(NSOperation *)operation;



@interface NWMOperationQueueManager ()

/**
 NSOperationQueue that operations will be added to.
 */
@property (nonatomic, strong) NSOperationQueue *queue;

@end

@implementation NWMOperationQueueManager

#pragma mark - SharedInstance

+ (instancetype)sharedInstance
{
    static NWMOperationQueueManager *sharedInstance = nil;

    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^
                  {
                      sharedInstance = [[NWMOperationQueueManager alloc] init];
                  });

    return sharedInstance;
}

#pragma mark - Init

- (instancetype)init
{
    self = [super init];

    if (self)
    {
        self.queue = [[NSOperationQueue alloc] init];
    }

    return self;
}

#pragma mark - AddOperation

- (void)addOperation:(NSOperation *)operation
{
    [self.queue addOperation:operation];
}

@end
```

The above manager kind of feels like it's overkill but as you expand your app you will probably find that you want more than one queue (perhaps one for user driven events and one for system events)

then isolating what queue an operation is added to becomes a very useful technique.

DIAGRAMS ALWAYS LIVEN THE PARTY

So what we end up with is the architecture described in the below diagram:



GOING HOME

So there is it one possible implementation of combining `NSOperation` with `NSURLSession` to move work off the main queue and simplify the request and parsing of data.

You can find the completed project by heading over to https://github.com/wibosco/NetworkingWingman-Example.