

Блоки и многопоточное программирование в Objective – C

Урок 4

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.





План курса





Что будет на уроке сегодня

- 📌 Поговорим о block в Objective – C
- 📌 Различия closure vs block
- 📌 Блоки и переменные
- 📌 Ключевое слово _block
- 📌 Введение в многопоточное программирование
- 📌 GCD
- 📌 Operation Queue



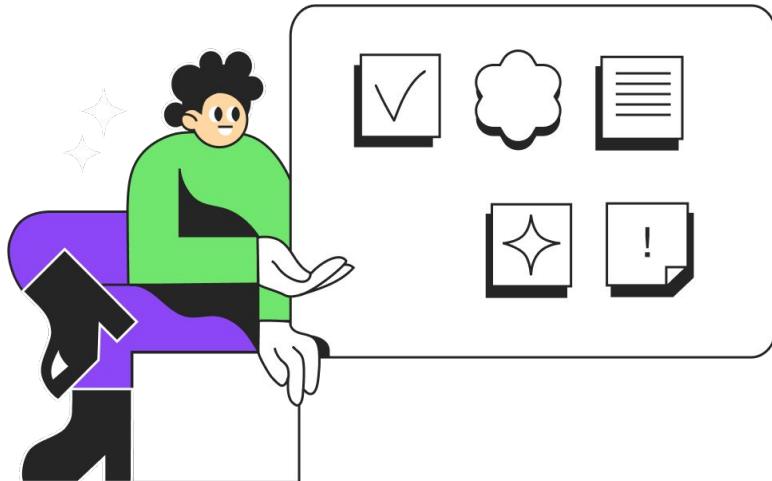


Блоки



Блоки

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.





📌 Closure expression

```
func executeOperation(completion: (Bool) -> Void) {  
    let success = // Some work  
    completion(success)  
}
```

📌 Obj-c block

```
- (void)executeOperationWithCompletion : (void (^)(BOOL success)) completion {  
    BOOL success = // Some work  
    completion(success);  
}
```



📌 Closure expression

The diagram illustrates the type signature of a closure and its definition. At the top, there are two columns: "Parameters" and "Return type". The "Parameters" column contains a single parameter box labeled "(Int)". The "Return type" column contains a single return type box labeled "String". Below this, the code defines a variable "closure" as a closure type: "var closure: (Int) -> String". This is followed by the closure's definition: "closure = { (num: Int) -> String in // Some work }". The code is annotated with green boxes and arrows indicating the type mapping: an arrow points from the parameter box to the "num" parameter in the closure definition; another arrow points from the return type box to the "String" return type in the closure definition.

Closure body

closure (5)

Execute closure

📌 Obj-c block

```
graph TD; block["block ="] --> caret["^"]; block --> nsString["NSString*"]; caret --> intParam["int num"]; caret --> intValue["int"]; nsString --> intParam2["int"]; nsString --> intValue2["int"];
```

The diagram illustrates the structure of the code:

```
block = ^ NSString* (^block)(int);  
block = ^ NSString* (int num){  
    // Some work  
};
```

Block body

```
block(5);
```

- Execute block



📌 Closure expression

```
var closure: (Int) -> Void  
  
closure = { (num: Int) in  
    // Some work  
}
```

```
var voidClosure: () -> Void  
  
voidClosure = {  
    // Some work  
}
```

```
var stringClosure: () -> String  
stringClosure = {  
    // return string  
}
```

📌 Obj-c block

```
void* (^block)(int);  
  
block = ^(int num) {  
    // Some work  
};
```

```
void* (^voidBlock)(int);  
  
voidBlock = ^{  
    // Some work  
};
```

```
NSString* (^stringBlock)();  
stringBlock = ^{  
    // Some work  
};
```



📌 Closure expression

```
func executeOperation(completion: (Bool) -> Void) {  
    let success = // Some work  
    completion(success)  
}
```

📌 Obj-c block

```
- (void)executeOperationWithCompletion : (void (^)(bool success)) completion {  
    BOOL success = // Some work  
    completion(success);  
}
```



📌 Closure expression

```
typealias CompletionClosure = (Bool) -> Void

func executeOperation(completion:
CompletionClosure) {
    let success = // Some
work completion(success)
}
```

📌 Obj-c block

```
typedef void(^CompletionBlock)( BOOL);

- (void)executeOperationWithCompletion : (CompletionBlock) completion {
    BOOL success = // Some work
completion(success);
}
```



📌 Closure expression

```
func testMethod() {  
    var multiplier = 7;  
  
    let closure: (Int) -> Int = { num in  
        multiplier = 10  
  
        return num * multiplier  
    }  
  
    print(closure(3)) // 30  
}
```

📌 Obj-c block

```
- (void)testMethod {  
    int multiplier = 7;  
  
    int (^myBlock)(int) = ^(int num) {  
        multiplier = 10; └───────── Compile error  
        return num * multiplier;  
    };  
  
    NSLog(@"%@", myBlock(3));  
}
```



📌 Closure expression

No **__block** modifier.

```
func testMethod() {  
    var multiplier = 7;  
  
    let closure: (Int) -> Int = { num in  
        multiplier = 10  
  
        return num * multiplier  
    }  
  
    print(closure(3)) // 30  
}
```

📌 Obj-c block

```
- (void)testMethod {  
    __block int multiplier = 7;  
  
    int (^myBlock)(int) = ^(int num) {  
        multiplier = 10;  
  
        return num * multiplier;  
    };  
  
    NSLog(@"%@", myBlock(3)); // 30  
}
```



📌 Closure expression

```
func testMethod() {  
    var multiplier = 7;  
  
    let closure: (Int) -> Int = { num in  
        return num * multiplier  
    }  
  
    multiplier = 10  
  
    print(closure(3)) // 30  
}
```

📌 Obj-c block

```
- (void)testMethod {  
    int multiplier = 7;  
  
    int (^myBlock)(int) = ^(int num) {  
        return num * multiplier;  
    };  
  
    multiplier = 10;  
  
    NSLog(@"%@", myBlock(3)); // 21  
}
```



📌 Closure expression

No **__block** modifier.

```
func testMethod() {  
    var multiplier = 7;  
  
    let closure: (Int) -> Int = { num in  
        return num * multiplier  
    }  
  
    multiplier = 10  
  
    print(closure(3)) // 30  
}
```

📌 Obj-c block

```
- (void)testMethod {  
    __block int multiplier = 7;  
  
    int (^myBlock)(int) = ^(int num) {  
        return num * multiplier;  
    };  
  
    multiplier = 10;  
  
    NSLog(@"%@", myBlock(3)); // 21  
}
```



Многопоточность?



Базовые концепции

- ✓ 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.





Преимущество нескольких потоков

Быстрое взаимодействие

Ответная реакция

Оптимизированное
потребление ресурсов



Преимущество нескольких потоков

Быстрое взаимодействие

Ответная реакция

Оптимизированное
потребление ресурсов



Преимущество нескольких потоков

Быстрое взаимодействие

Ответная реакция

Оптимизированное
потребление ресурсов



Преимущество нескольких потоков

Быстрое взаимодействие

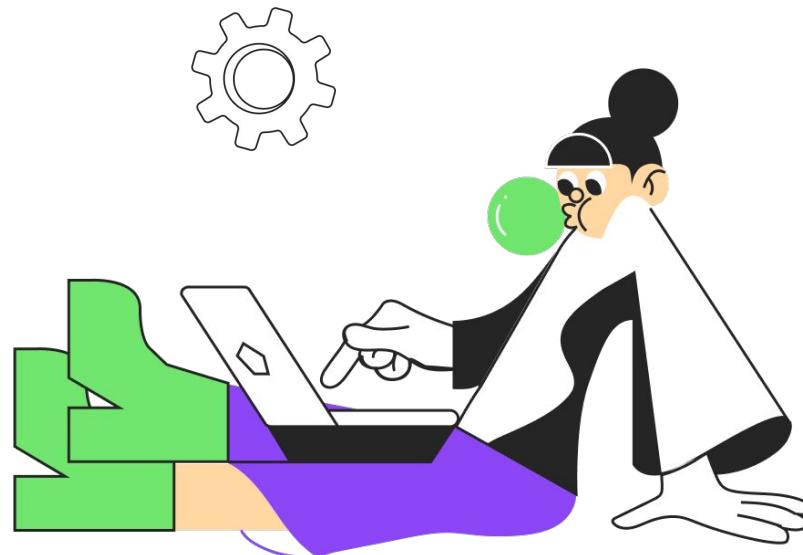
Ответная реакция

Оптимизированное
потребление ресурсов



Типы потоков в iOS

- ✓ Mach (or Kernel) threads
- ✓ POSIX threads
- ✓ NSThread





Mach (or Kernel) threads



Mach (or Kernel) threads



Никогда не используйте



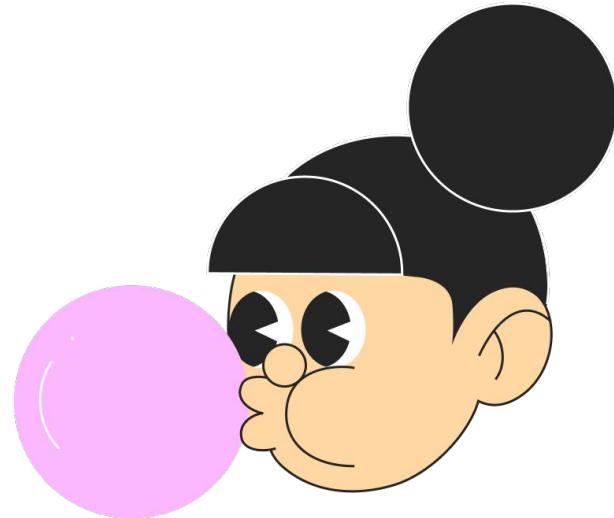


POSIX threads



POSIX threads

💡 Это лучший выбор для создания потоков...
если вы не пишете приложение Cocoa



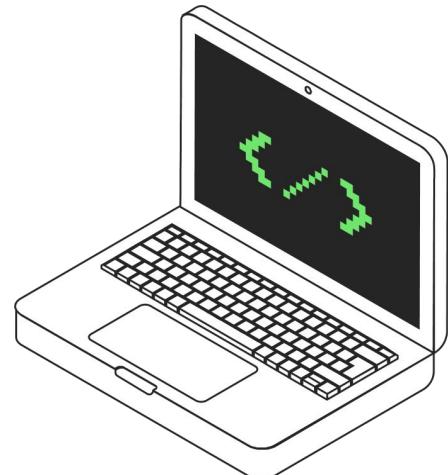


NSThread



NSThread

- ✓ Приложение создает потоки и управляет ими.
- ✓ Потоки могут иметь приоритет.
ОС использует это для планирования их выполнения.
- ✓ Нет прямого API для ожидания завершения потока.
Используйте мьютекс (например, NSLock) и собственный код.





Apple предоставляет API, необходимые
для управления потоками



Concurrency APIs

Grand Central Dispatch (GCD)

Operation queues





Concurrency APIs

Grand Central Dispatch (GCD)

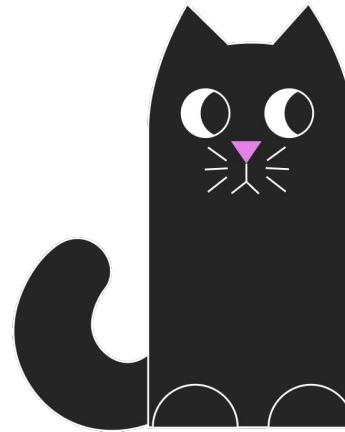
Operation queues





Grand Central Dispatch

- ✓ Цель состоит в том, чтобы поставить в очередь задачи — либо метод, либо блок — которые могут выполняться параллельно, в зависимости от доступности ресурсов.
- ✓ GCD управляет пулом потоков за кулисами.
- ✓ GCD решает, в каком именно потоке будут выполняться ваши блоки кода.





Dispatch queues

Создание очереди

```
dispatch_queue_t queue =  
dispatch_queue_create("com.rsschool.concurrency.demo", 0);
```

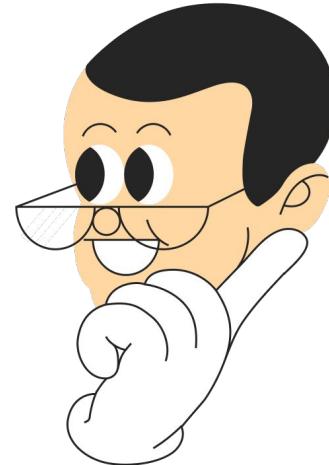


Dispatch queues

Типы очередей

Serial

Concurrent





Dispatch queues

Типы очередей

📌 Serial

```
dispatch_queue_t queue = dispatch_queue_create ("com.rsschool.concurrency.demo" , 0);
// or
dispatch_queue_t queue = dispatch_queue_create ("com.rsschool.concurrency.demo" ,
DISPATCH_QUEUE_SERIAL );
```

📌 Concurrent

```
dispatch_queue_t queue = dispatch_queue_create ("com.rsschool.concurrency.demo" ,
DISPATCH_QUEUE_CONCURRENT );
```

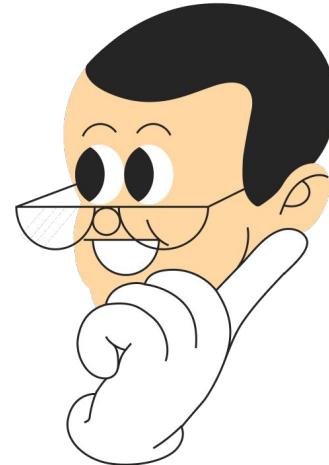


Dispatch queues

Типы очередей

Serial

Concurrent





Dispatch queues

Queue types

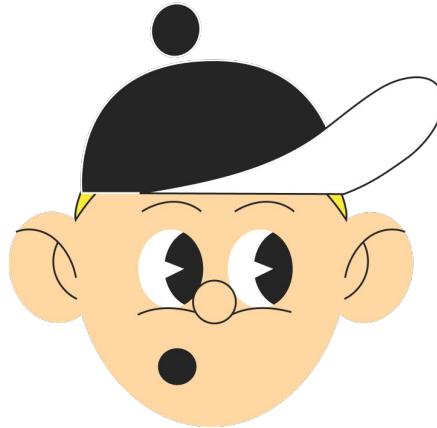
Serial

Concurrent



Serial

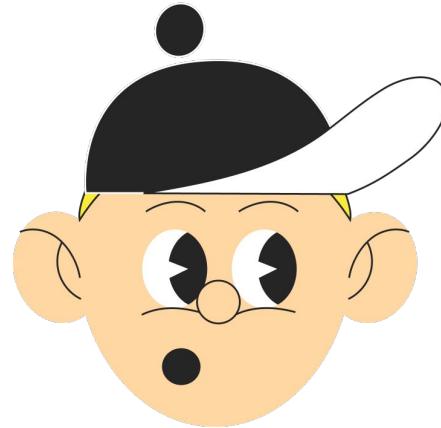
- ✓ С очередью связан один поток





Serial

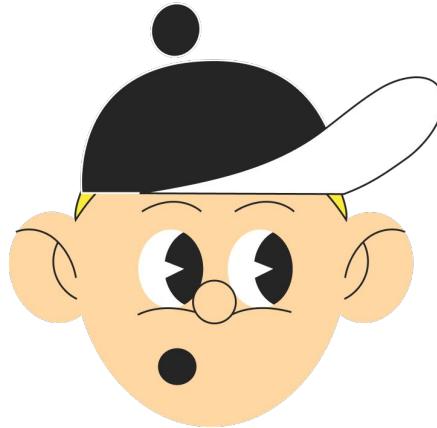
- ✓ С очередью связан один поток
- ✓ В любой момент времени может выполняться только одна задача





Serial

- ✓ С очередью связан один поток
- ✓ В любой момент времени может выполняться только одна задача
- ✓ Текущая задача должна быть завершена до начала следующей





Dispatch queues

Типы очередей

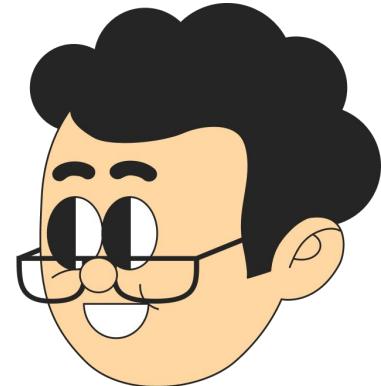
Serial

Concurrent



Concurrent

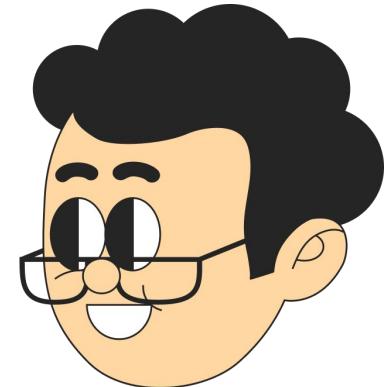
- ✓ Очередь может использовать столько потоков, сколько у системы есть ресурсов





Concurrent

- ✓ Очередь может использовать столько потоков, сколько у системы есть ресурсов
- ✓ Нет никаких гарантий, в каком порядке будут выполняться задачи





Dispatch queues

Типы очередей

Serial

- ✓ С очередью связан один поток
- ✓ В любой момент времени может выполняться только одна задача
- ✓ Текущая задача должна быть завершена до начала следующей

Concurrent

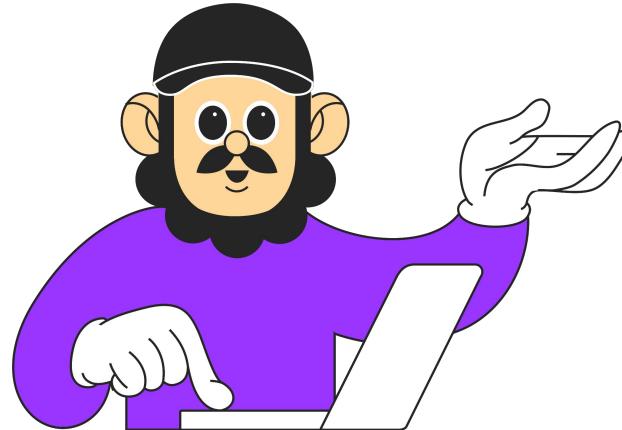
- ✓ Очередь может использовать столько потоков, сколько у системы есть ресурсов
- ✓ Нет никаких гарантий, в каком порядке будут выполняться задачи



Dispatch queues

Приоритеты очередей

- ✓ High
- ✓ Default
- ✓ Low
- ✓ Background





Dispatch queues

Concurrent queues priorities

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



Преимущество нескольких потоков

Quality of service (QoS)

userInteractive

background

userInitiated

default

utility

unspecified



Преимущество нескольких потоков

Quality of service (QoS)

userInteractive

background

userInitiated

default

utility

unspecified



Преимущество нескольких потоков

Quality of service (QoS)

userInteractive

background

userInitiated

default

utility

unspecified



Преимущество нескольких потоков

Quality of service (QoS)

userInteractive

background

userInitiated

default

utility

unspecified



Преимущество нескольких потоков

Quality of service (QoS)

userInteractive

background

userInitiated

default

utility

unspecified



Преимущество нескольких потоков

Quality of service (QoS)

userInteractive

background

userInitiated

default

utility

unspecified



Dispatch queues

Quality of service (QoS)

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



Dispatch queues

Добавление задачи в очередь

Synchronously

Asynchronously



Dispatch queues

Добавление задачи в очередь

Synchronously

Приложение будет ждать и блокировать текущую очередь, пока не завершится выполнение, прежде чем перейти к следующей задаче.

Asynchronously



Dispatch queues

Добавление задачи в очередь

Synchronously

Приложение будет ждать и блокировать текущую очередь, пока не завершится выполнение, прежде чем перейти к следующей задаче.

Asynchronously

Приложение запустит задачу и немедленно вернет выполнение



Асинхронный
не означает параллельный 🤝



Dispatch queues

Синхронизировать добавление задач в очереди

```
dispatch_queue_t queue = dispatch_get_global_queue (DISPATCH_QUEUE_PRIORITY_HIGH , 0);
dispatch_sync(queue, ^{
    // Some task
});
```



Dispatch queues

Асинхронное добавление задач в очереди

```
dispatch_queue_t queue = dispatch_get_global_queue (DISPATCH_QUEUE_PRIORITY_HIGH , 0);
dispatch_async(queue, ^{
    // Some task
});
```



Dispatch group — это механизм, который вы будете использовать, когда захотите отслеживать завершение группы.



Dispatch group

Создание группы

```
dispatch_group_t group =  
dispatch_group_create();
```



Dispatch group

Добавить задачу в группу рассылки

```
dispatch_group_t group = dispatch_group_create ();
dispatch_queue_t queue = dispatch_get_global_queue (DISPATCH_QUEUE_PRIORITY_HIGH , 0);

dispatch_group_async (group, queue, ^{
    // Some task
});
```



Dispatch group

Завершение групповой задачи асинхронного отслеживания

```
dispatch_group_t group = dispatch_group_create ();
dispatch_queue_t queue = dispatch_get_global_queue (DISPATCH_QUEUE_PRIORITY_HIGH , 0);

dispatch_group_async (group, queue, ^{
    // Some task
});

dispatch_group_notify (group, dispatch_get_main_queue (), ^{
    // Some completion task
});
```



Dispatch group

Синхронизация выполнения групповых задач отслеживания

```
dispatch_group_t group = dispatch_group_create ();
dispatch_queue_t queue = dispatch_get_global_queue (DISPATCH_QUEUE_PRIORITY_HIGH , 0);

dispatch_group_async (group, queue, ^{
    // Some task
});

dispatch_group_wait (group, DISPATCH_TIME_FOREVER );
```



Dispatch group

Обертывание асинхронных методов

```
dispatch_group_async (group, queue, ^{
    // Some task
    dispatch_group_enter (group);
    dispatch_async (queue, ^{
        // Perform some work
        dispatch_group_leave (group);
    });
});
```



NSOperation – абстрактный класс,
представляющий код и данные,
связанные с одной задачей.

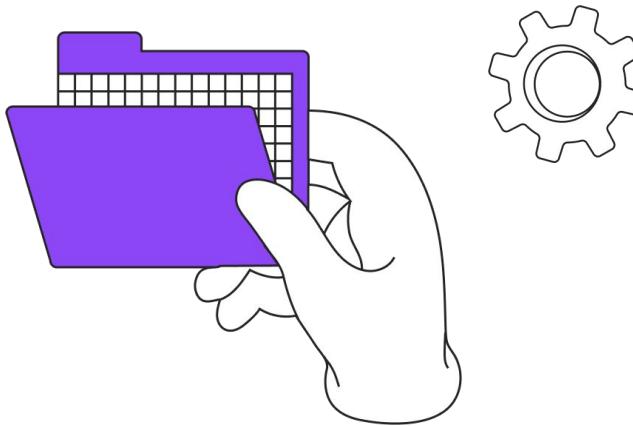


NSOperation

Dependencies

Easy cancellation
KVO-compliant

properties Inheritance





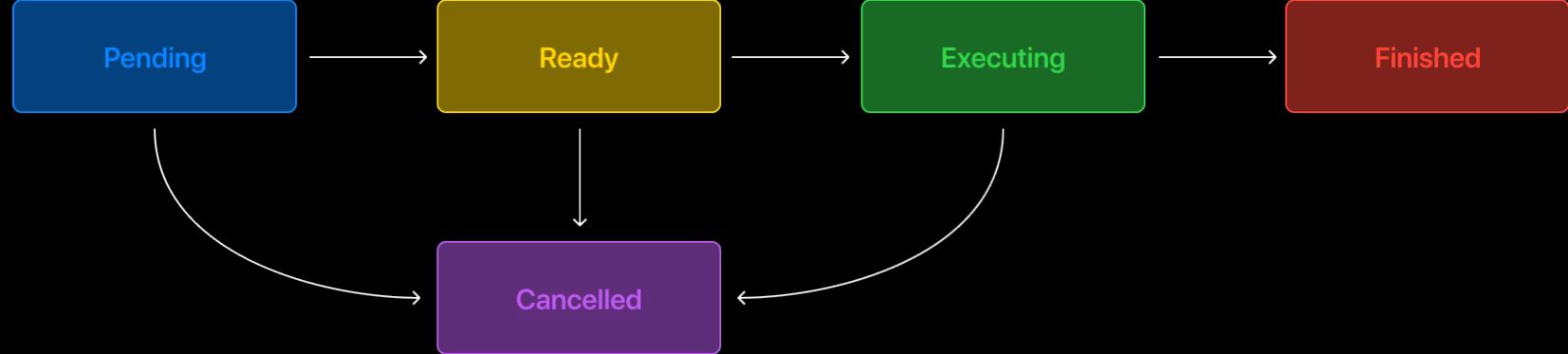
NSOperation states

- ✓ Pending
- ✓ Ready
- ✓ Executing
- ✓ Finished
- ✓ Cancelled



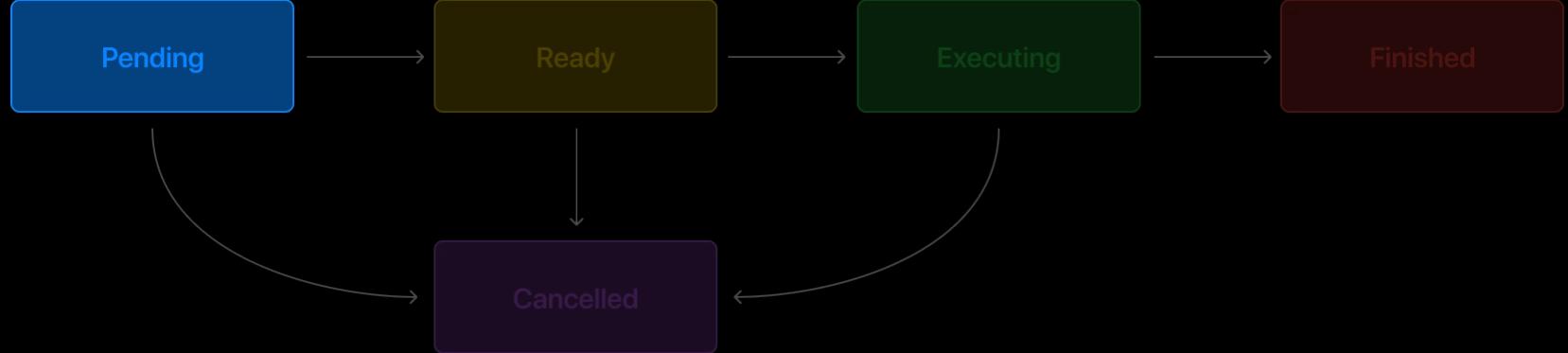


NSOperation states



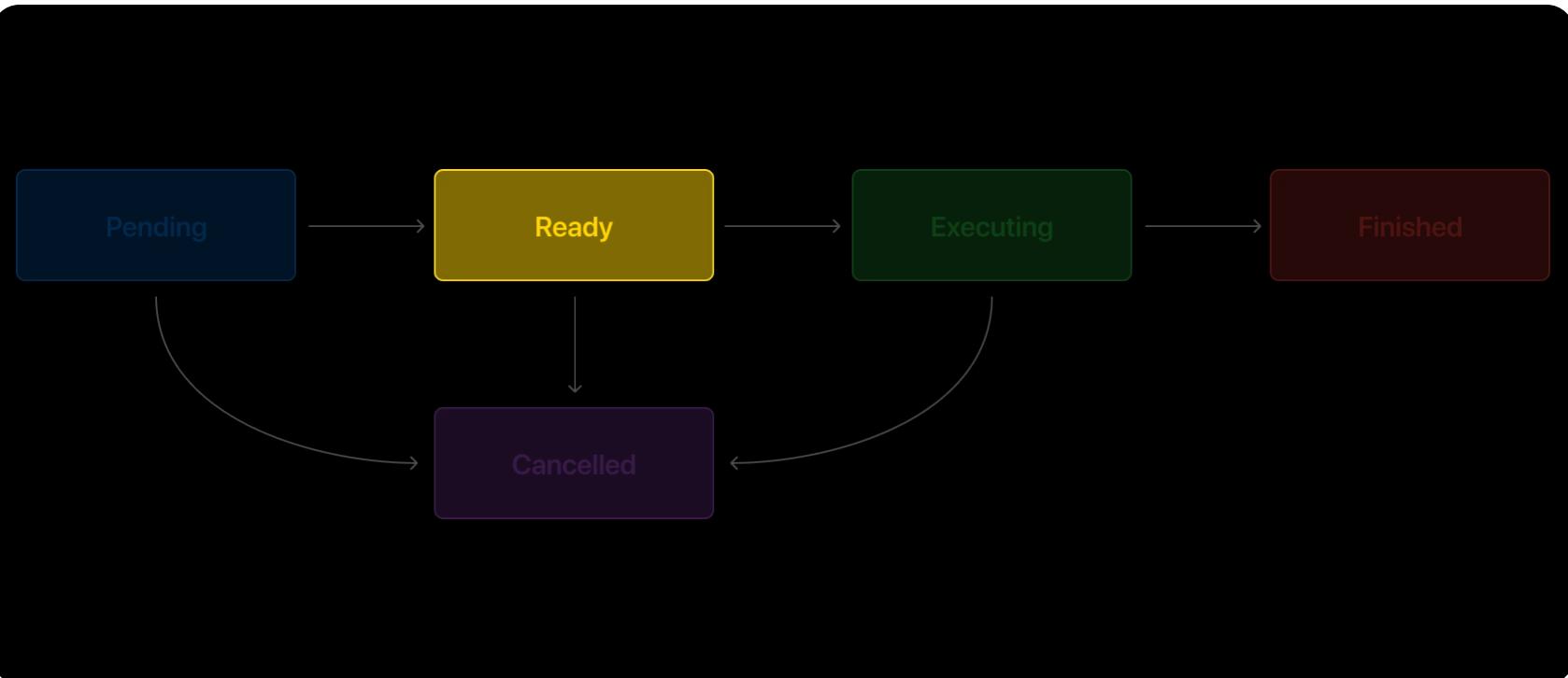


NSOperation states



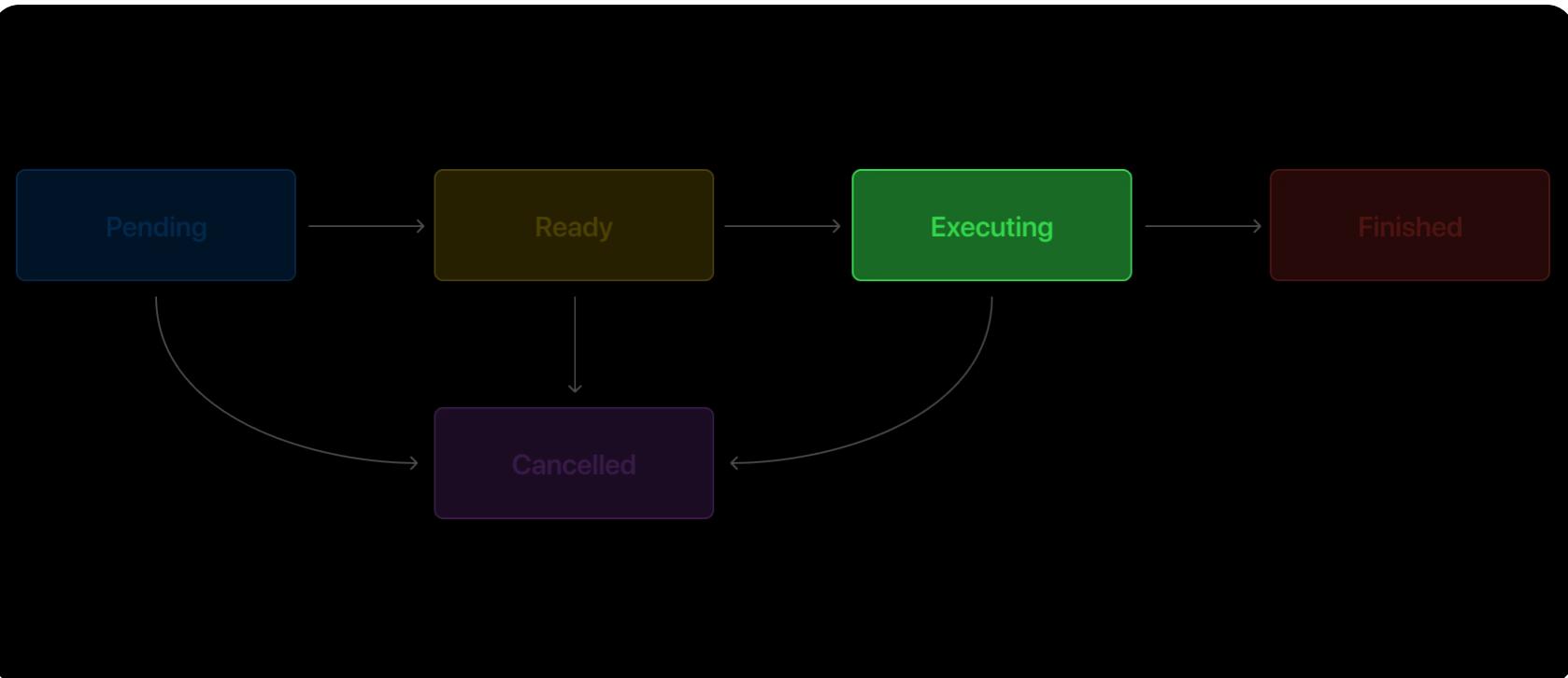


NSOperation states



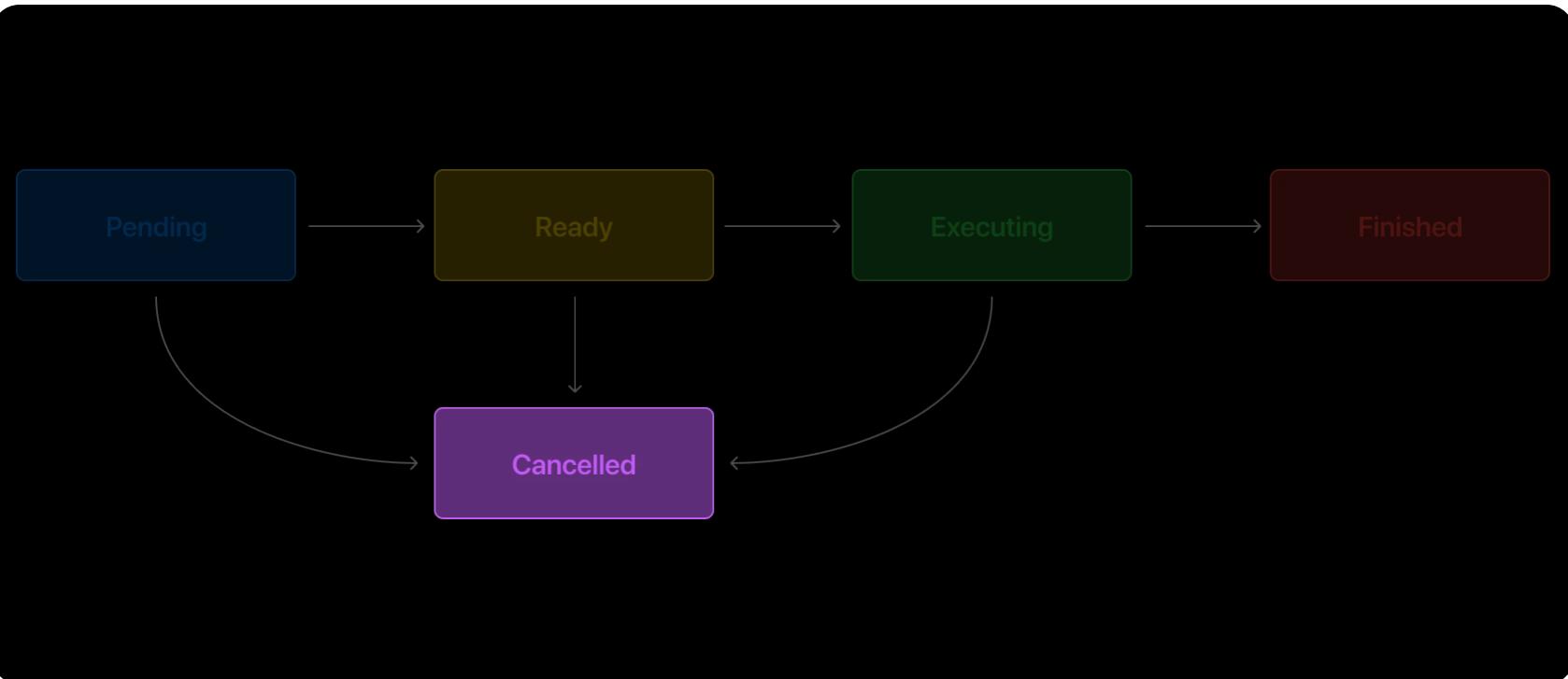


NSOperation states



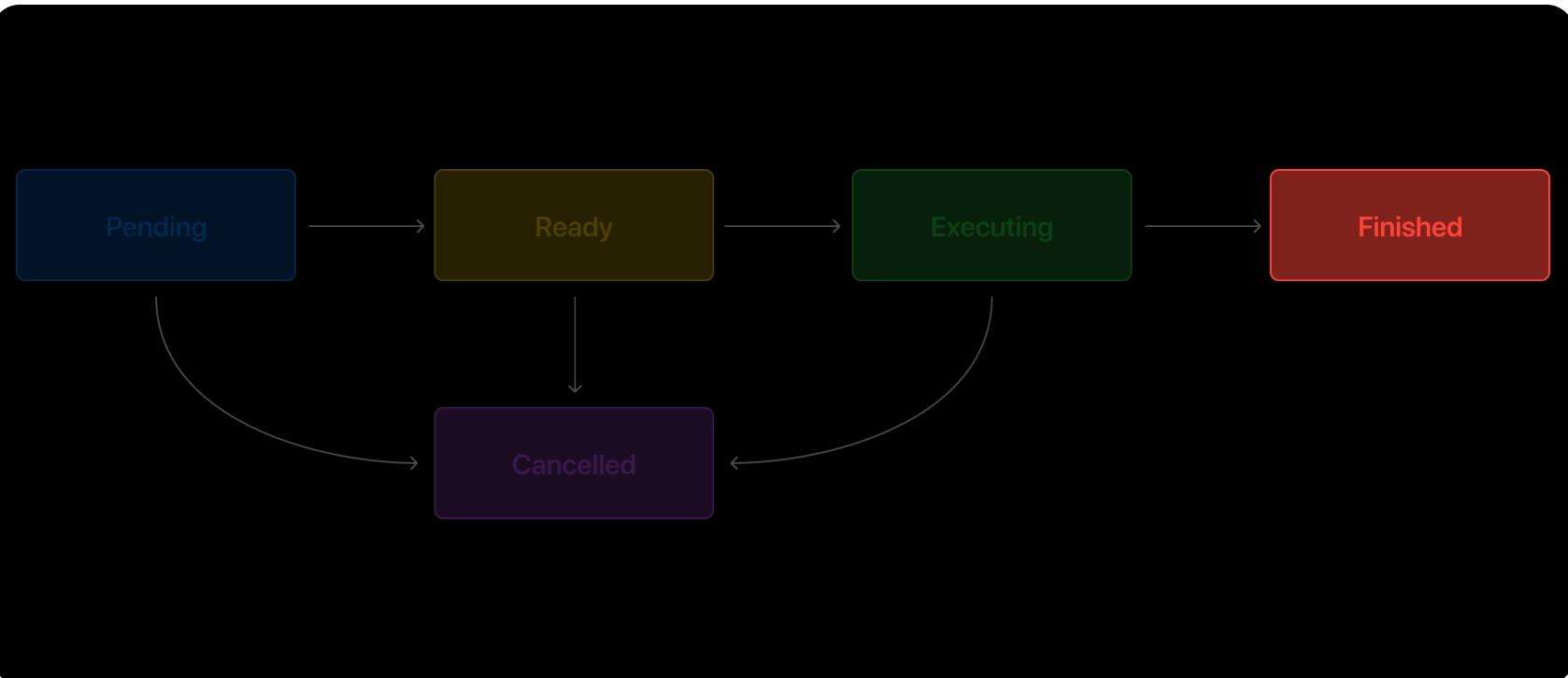


NSOperation states





NSOperation states





NSBlockOperation

Создание NSBlockOperation

```
NSBlockOperation *blockOperation = [ NSBlockOperation blockOperationWithBlock :^{
    // Some task
} ];
```



NSBlockOperation

Запуск NSBlockOperation

```
NSBlockOperation *blockOperation = [ NSBlockOperation blockOperationWithBlock :^{
    // Some task
} ];
[blockOperation start];
```



NSBlockOperation

Запуск NSBlockOperation

```
NSBlockOperation *blockOperation = [NSBlockOperation  
blockOperationWithBlock:^{  
    // Some task  
}];  
  
[blockOperation start];
```



NSBlockOperation запускает операцию в параллельной глобальной очереди по умолчанию



NSBlockOperation

Добавьте несколько блоков в экземпляр NSBlockOperation

```
NSBlockOperation *blockOperation = [ NSBlockOperation new ];  
  
[blockOperation addExecutionBlock:^{  
    // Task 1  
}];  
  
[blockOperation addExecutionBlock:^{  
    // Task 2  
}];  
  
[blockOperation start];
```



NSBlockOperation

Отслеживайте выполнение всех задач внутри экземпляра NSBlockOperation

```
NSBlockOperation *blockOperation = [ NSBlockOperation new ];  
  
[blockOperation addExecutionBlock:^{
    // Task 1
}];  
  
[blockOperation addExecutionBlock:^{
    // Task 2
}];  
  
blockOperation.completionBlock = ^{
    // Some completion action
};  
  
[blockOperation start];
```



NSOperation subclassing

```
@interface CustomOperation : NSOperation  
@end
```



NSOperation subclassing

Методы переопределения для синхронных операций

- `(void)main;`



NSOperation subclassing

Методы переопределения для синхронных операций

```
- (void)main {
    // Some work
}
```



NSOperation subclassing

Start operation

```
CustomOperation *operation = [CustomOperation new];  
[operation start];
```



NSOperation subclassing

Start operation

```
CustomOperation *operation = [CustomOperation new];  
[operation start];
```

 Когда вы вызываете метод `start` непосредственно в операции,
вы выполняете синхронный вызов в текущем потоке



NSOperation subclassing

Start operation

```
CustomOperation *operation = [CustomOperation new];  
[operation start];
```



Прямой вызов метода запуска может привести
к исключению, если операции еще не готовы к запуску



NSOperation subclassing

Start operation

```
CustomOperation *operation = [CustomOperation new];  
[operation start];
```



Чтобы решить синхронную проблему,
вы можете использовать очереди операций



NSOperationQueue — это очередь,
регулирующая выполнение операций.



NSOperationQueue по умолчанию
является параллельным.



Очередь операций выполняет
операции, которые готовы.



После добавления в очередь операций операция остается в ней до тех пор, пока не сообщит о завершении своей задачи или не будет отменена.



Очереди операций сохраняют операции до тех пор, пока они не будут завершены, а сами очереди сохраняются до завершения всех операций.

Приостановка очереди операций с незавершенными операциями может привести к утечке памяти





NSOperationQueue

Создание очереди

```
NSOperationQueue *queue = [NSOperationQueue new];
```



NSOperationQueue

Quality of Service

- 💡 Вы можете определить значения качества обслуживания как для очередей, так и для операций



NSOperationQueue

Quality of Service

```
NSOperationQueue *queue = [NSOperationQueue new];
queue.qualityOfService = NSQualityOfServiceUtility;

CustomOperation *operation = [CustomOperation new];
operation.qualityOfService = NSQualityOfServiceUtility;
```



NSOperationQueue

Quality of Service



Уровень качества обслуживания очереди операций по умолчанию: `background`





NSOperationQueue

Добавление операции в очередь

```
NSOperationQueue *queue = [NSOperationQueue new];
CustomOperation *operation = [CustomOperation new];
[queue addOperation:operation];
```



NSOperationQueue

Максимальное количество операций

```
NSOperationQueue *queue = [NSOperationQueue new];  
queue.maxConcurrentOperationCount = 3;
```



NSOperationQueue

Ожидания завершения

```
NSOperationQueue *queue = [NSOperationQueue new];
CustomOperation *operation = [CustomOperation new];
[queue addOperation:operation];

[queue waitUntilAllOperationsAreFinished];
```



Operation Dependencies

NSOperation предоставляет способ установить зависимости между операциями.

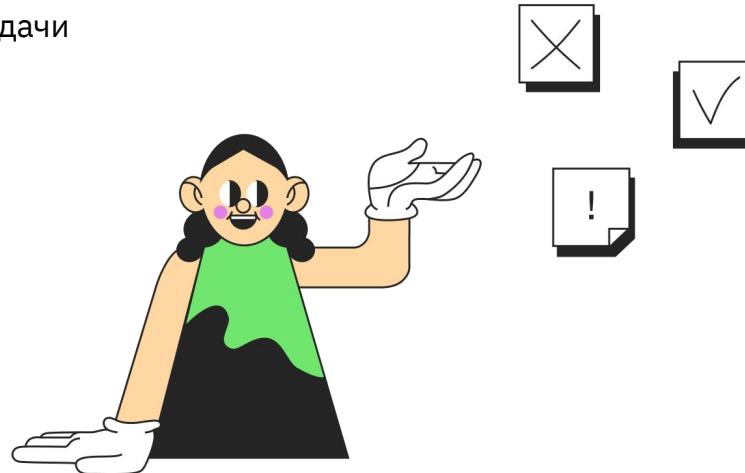




Operation Dependencies

Преимущества зависимостей

- ✓ Гарантирует, что зависимая операция не начнется до завершения предварительной операции.
- ✓ Обеспечивает чистый способ автоматической передачи данных из первой операции во вторую.





Operation Dependencies

Добавление зависимостей

```
NSOperationQueue *queue = [NSOperationQueue new];  
  
CustomOperation *operation1 = [CustomOperation new];  
CustomOperation *operation2 = [CustomOperation new];  
  
[operation2 addDependency:operation1];  
  
[queue addOperation:operation1]; [queue  
addOperation:operation2];
```



Operation Dependencies

Удаление зависимостей

```
NSOperationQueue *queue = [NSOperationQueue new];  
  
CustomOperation *operation1 = [CustomOperation new];  
CustomOperation *operation2 = [CustomOperation new];  
  
[operation2 addDependency:operation1];  
[operation2 removeDependency:operation1];  
  
[queue addOperation:operation1];  
[queue addOperation:operation2];
```



Operation Dependencies

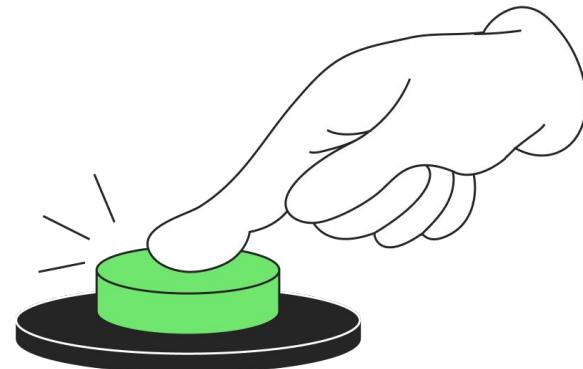
Доступ к зависимостям

```
// NSOperation property  
@property (readonly, copy) NSArray<NSOperation *> *dependencies;
```



Cancelling Operations

NSOperation предоставляет способ отменить текущую операцию





Cancelling Operations

```
NSOperationQueue *queue = [NSOperationQueue new];
CustomOperation *operation = [CustomOperation new];
[queue addOperation:operation];

[operation cancel];
```



Cancelling Operations

Implementing cancellation for custom operation

```
// Method main of our CustomOperation class
- (void)main {
    // Some hard work part 1

    if (self.isCancelled) { return; }

    // Some hard work part 2

    if (self.isCancelled) { return; }

    // Some hard work part 3
    // Task is completed
}
```



Cancelling Operations

Cancel all queue operations

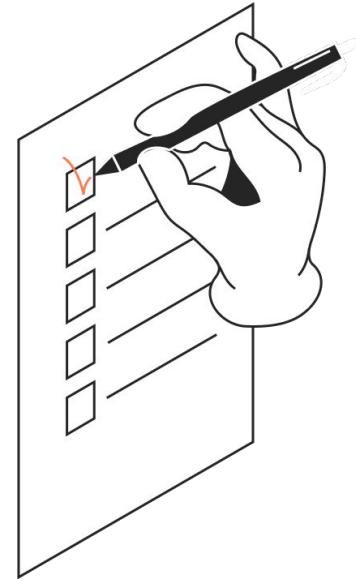
```
NSOperationQueue *queue = [NSOperationQueue new];
CustomOperation *operation1 = [CustomOperation new];
CustomOperation *operation2 = [CustomOperation new];

[queue addOperation:operation1];
[queue addOperation:operation2];
[queue cancelAllOperations];
```



Что почитать?

- 📌 Стивен Кочан. «Программирование на Objective-C».
- 📌 Скотт Кнастер, Вакар Малик, Марк Далримпл.«Objective-C. Программирование для Mac OS X и iOS».
- 📌 <https://www.objc.io/>
- 📌 <https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/>





Вопросы?

Вопросы?



Вопросы?

