Operation and OperationQueue

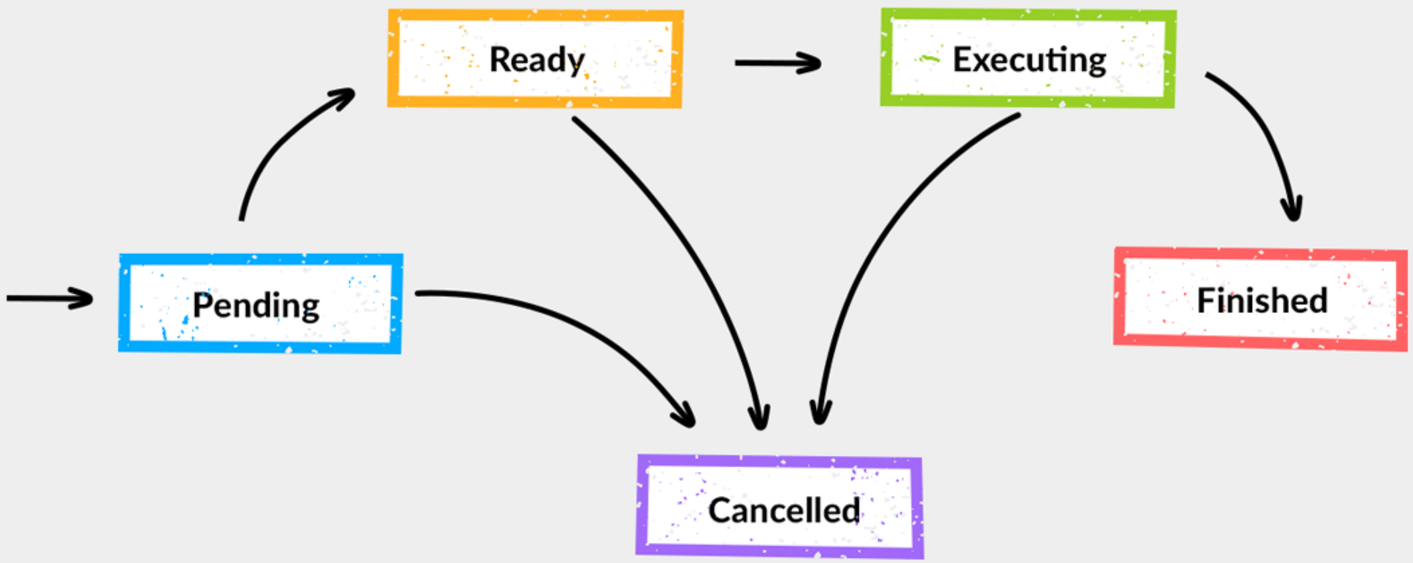
Lecture plan

- Operation
- Operation and Operation Queue
- Async operation
- maxConcurrentCountOperation
- Cancel
- Dependencies
- waitUntil
- completionBlock
- Suspend
- GCD vs Operation

Block operation

```
class BlockOperationTest {

    private let operationQueue = OperationQueue()

    func test() {

        let blockOperation = BlockOperation {
            print("test")
        }
        operationQueue.addOperation(blockOperation)
    }
}
```

Operation structure



Operation structure

- isReady
- isAsynchronous
- isExecuting
- isFinished
- isCanceled

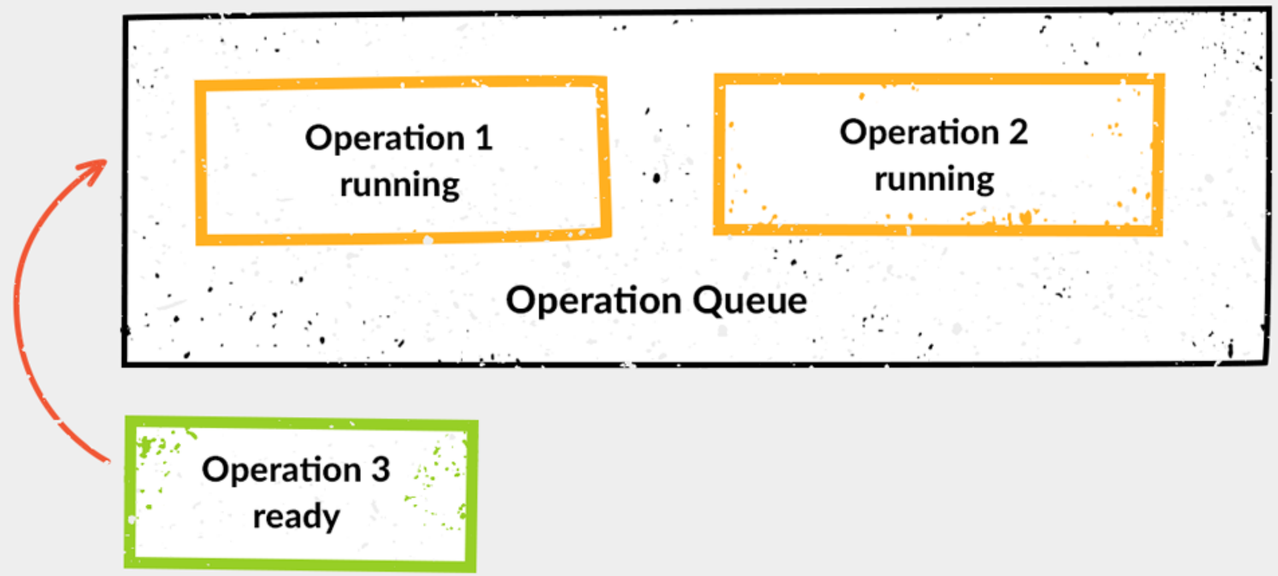
- main()
- start()

Operation KVO

```
class OperationKVOTest: NSObject {

    func test() {

        let operation = Operation()
        operation.addObserver(self, forKeyPath: "isCancelled", options:
NSKeyValueObservingOptions.new, context: nil)
    }

    override func observeValue(forKeyPath keyPath: String?, of object: Any?,
change: [NSKeyValueChangeKey : Any]?, context: UnsafeMutableRawPointer?) {

        if keyPath == "isCancelled" {
            //Handle
        }
    }
}
```

- isCancelled
- isAsynchronous
- isExecuting
- isFinished
- isReady
- dependencies
- completionBlock

Operation and operation queue



Operation and operation queue

```
class OperationTest2 {
    private let operationQueue = OperationQueue()

    func test() {

        operationQueue.addOperation {
            print("test2")
        }
    }
}
```

Operation and operation queue

```
class OperationTest {

    class OperationA: Operation {

        override func main() {
            print("test")
        }
    }

    private let operationQueue = OperationQueue()

    func test() {

        let testOperation = OperationA()
        operationQueue.addOperation(testOperation)
    }
}
```

Async operation

```
class AsyncOperation: Operation {
    private var finish = false
    private var execute = false
    private let queue = DispatchQueue(label: "AsyncOperation")

    override var isAsynchronous: Bool { return true }
    override var isFinished: Bool { return finish }
    override var isExecuting: Bool { return execute }

    override func start() {
        queue.async {
            self.main()
        }
        execute = true
    }

    override func main() {
        print("test")
        finish = true
        execute = false
    }
}
```

```
func test() {
    let operation = AsyncOperation()
    operation.start()
}
```

KVO + Async operation

```
class AsyncOperation: Operation {
    private var finish = false
    private var execute = false
    private let queue = DispatchQueue(label: "AsyncOperation")

    override var isAsynchronous: Bool { return true }
    override var isFinished: Bool { return finish }
    override var isExecuting: Bool { return execute }

    override func start() {
        willChangeValue(forKey: "isExecuting")
        queue.async {
            self.main()
        }
        execute = true
        didChangeValue(forKey: "isExecuting")
    }

    override func main() {
        print("test")
        willChangeValue(forKey: "isFinished")
        willChangeValue(forKey: "isExecuting")

        finish = true
        execute = false

        didChangeValue(forKey: "isFinished")
        didChangeValue(forKey: "isExecuting")
    }
}
```

```
func test() {
    let operation = AsyncOperation()
    operation.start()
}
```

Quality of service

```
class OperationQualityOfServiceTest {

    private let operationQueue = OperationQueue()

    func test1() {

        let operation = BlockOperation {
            print("test")
        }
        operation.qualityOfService = .userInteractive

        operationQueue.addOperation(operation)
    }

    func test2() {
        let operation = BlockOperation {
            print("test")
        }
        operation.qualityOfService = .userInteractive

        operationQueue.qualityOfService = .utility
        operationQueue.addOperation(operation)
    }
}
```

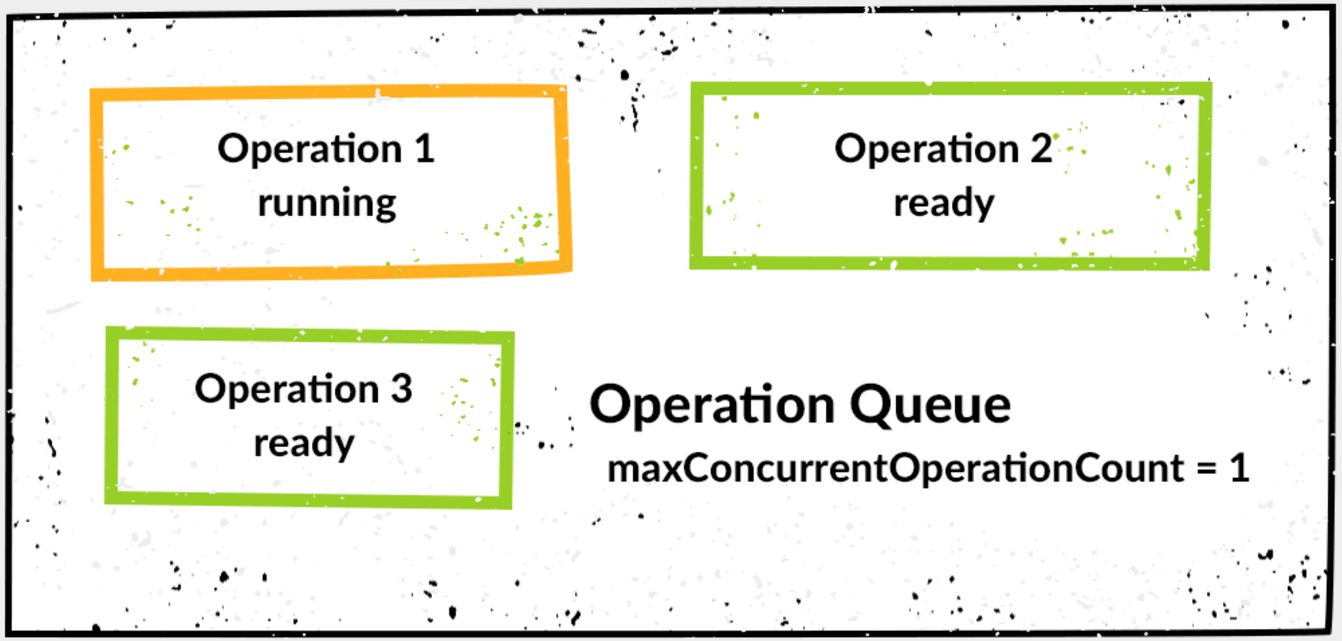
OperationQueue Qos

- В очередь без qos добавляется операция с qos => очередь и ее операции не изменяется
- В очередь с qos ставится операция с более высокой qos => у очереди поднимается qos; у всех операций поднимается qos; операции с низким qos, повышают свой qos при добавлении
- У очереди поднимают qos => операции очереди также поднимают свой приоритет; операции, которые будут добавлены с низким приоритетом также поднимают свой приоритет
- У очереди понижают qos => операции очереди не изменяются; операции, которые будут добавлены получат такой же qos, в случае qos не выше qos очереди

Operation Qos

- Операция без qos => При создании получает qos операции, очереди, блока или потока в котором была создана.
- Операция с qos была добавлена в очередь с большим qos => qos операции поднимается чтобы соответствовать qos queue.
- У очереди, содержащей операцию поднимается qos => у операции тоже поднимается qos, если qos очереди выше чем у операции.
- Другая операция (child) была добавлена в зависимости к операции (parent) => Parent повышает qos, если child qos выше.
- У операции повышают qos => у child операций поднимается qos если он ниже; другие операции в очереди перед текущей операции, повышают свой приоритет если он ниже.
- У операции понижают qos => у child операций qos не изменяется; qos очереди не изменяется.

maxConcurrentOperationCount



maxConcurrentOperationCount

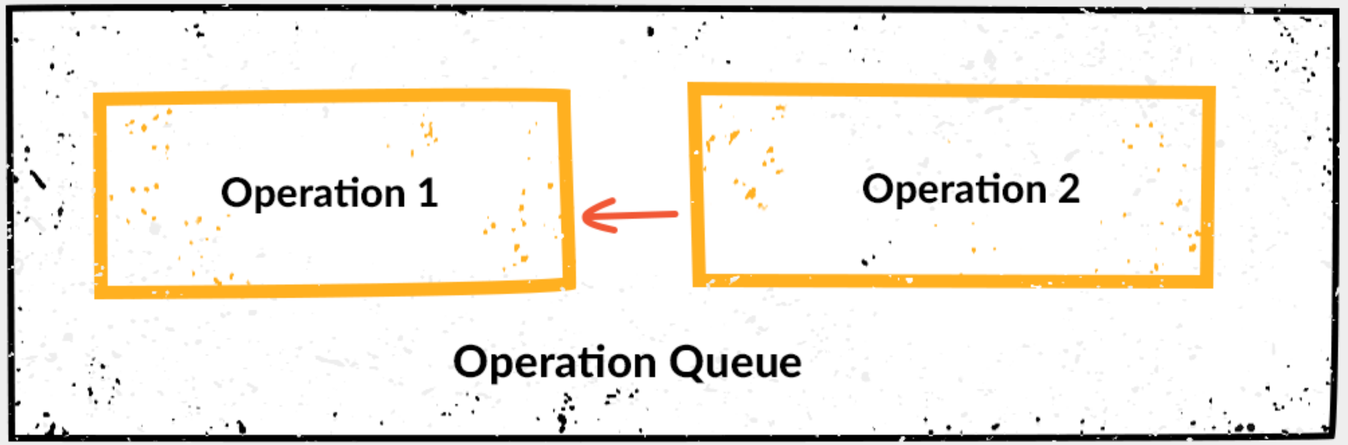
```
class OperationCountTest {

    private let operationQueue = OperationQueue()

    func test() {
        operationQueue.maxConcurrentOperationCount = 1
        operationQueue.addOperation {

            sleep(1)
            print("test1")
        }
        operationQueue.addOperation {

            sleep(1)
            print("test2")
        }
        operationQueue.addOperation {

            sleep(1)
            print("test3")
        }
    }
}
```

...
1 second
...
test1
...
1 second
...
test2
...
1 second
...
test3

Cancel

```
class CancelTest {
    private let operationQueue = OperationQueue()

    class OperationCancelTest: Operation {
        override func main() {
            if isCancelled {
                return
            }
            sleep(1)
            if isCancelled {
                return
            }
            print("test")
        }
    }

    func test() {
        let cancelOperation = OperationCancelTest()
        operationQueue.addOperation(cancelOperation)
        cancelOperation.cancel()
    }
}
```

Dependencies

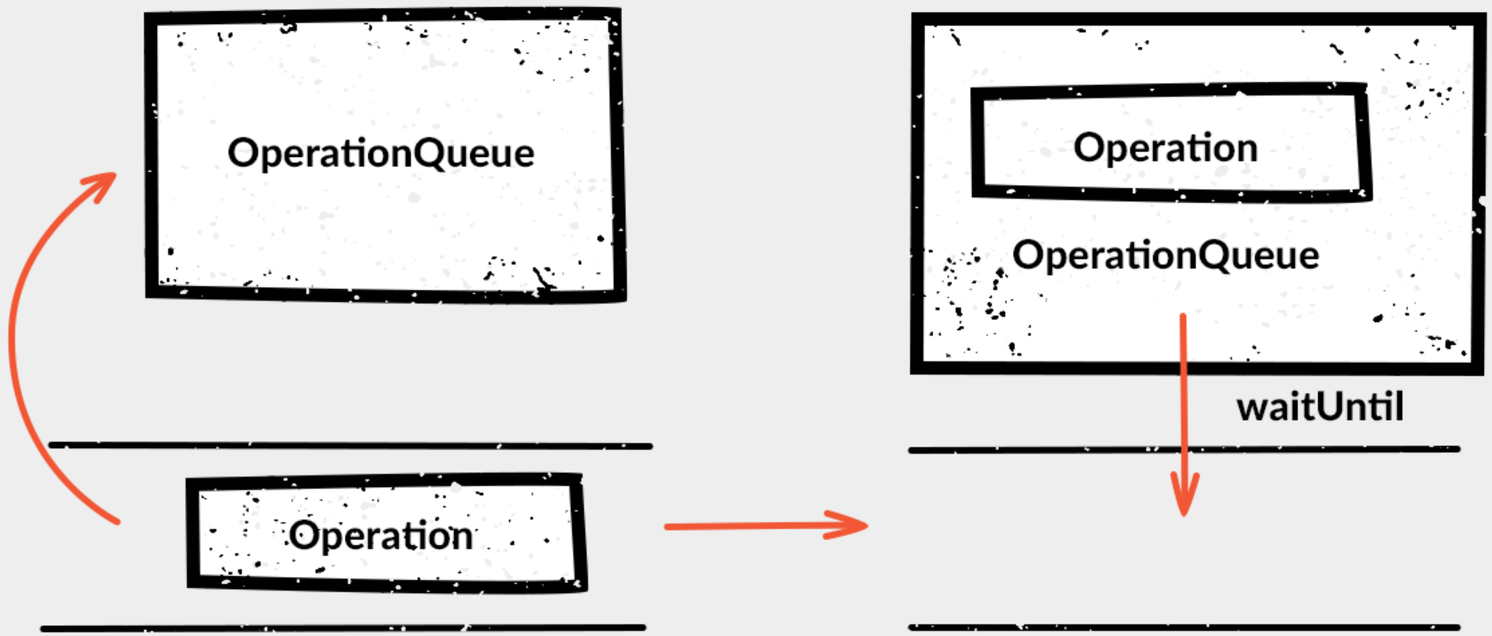


Dependencies

```
class DependenciesTest {

    private let operationQueue = OperationQueue()

    func test() {

        let operation1 = BlockOperation { print("test1") }
        let operation2 = BlockOperation { print("test2") }
        operation2.addDependency(operation1)

        operationQueue.addOperation(operation1)
        operationQueue.addOperation(operation2)
    }
}
```

test1
test2

waitUntil


OperationQueue
Operation
Operation
OperationQueue
waitUntil

waitUntil

```
class WaitOperationsTest1 {

    private let operationQueue = OperationQueue()

    func test() {
        operationQueue.addOperation {

            sleep(1)
            print("test1")
        }
        operationQueue.addOperation {

            sleep(2)
            print("test2")
        }
        operationQueue.waitUntilAllOperationsAreFinished()
    }
}
```

waitUntil

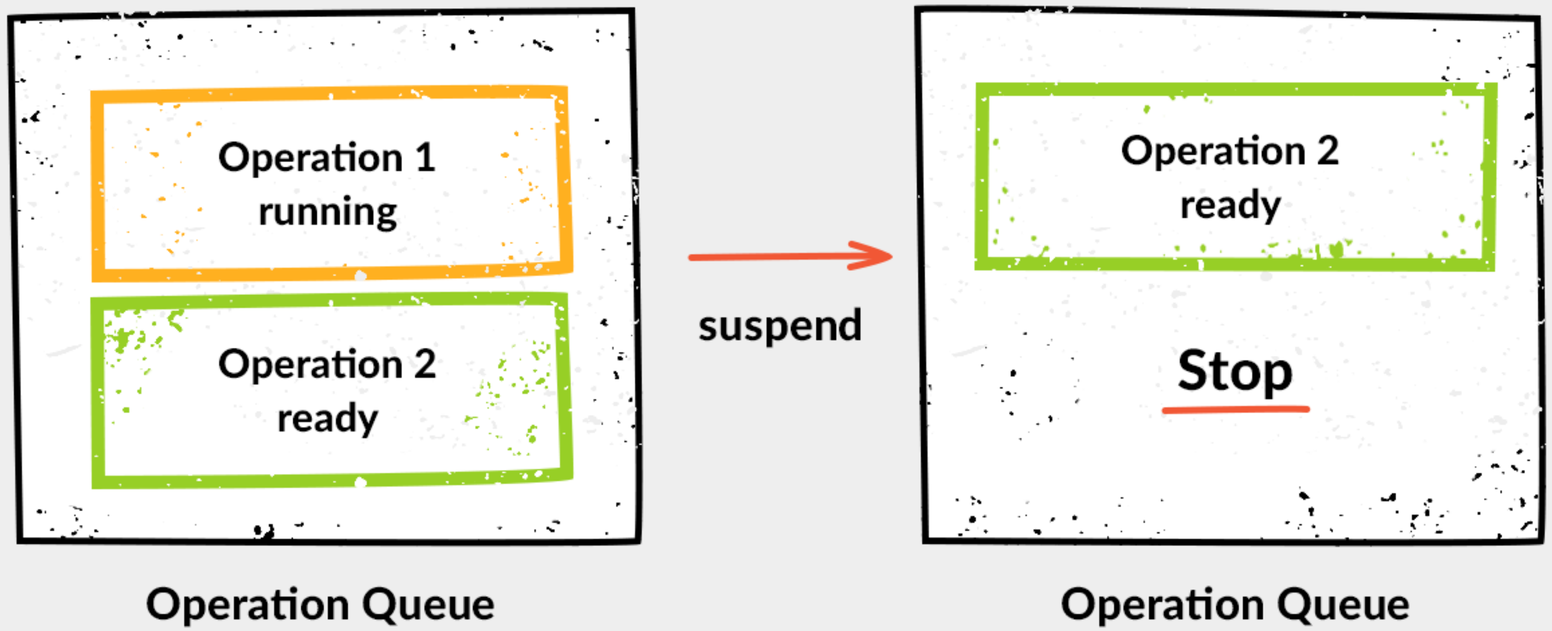
```
class WaitOperationsTest2 {

    private let operationQueue = OperationQueue()

    func test() {
        let operation1 = BlockOperation {

            sleep(1)
            print("test1")
        }
        let operation2 = BlockOperation {

            sleep(2)
            print("test2")
        }
        operationQueue.addOperations([operation1, operation2], waitUntilFinished: true)
    }
}
```

Completion block

```
class CompletionBlockTest {

    private let operationQueue = OperationQueue()

    func test() {

        let operation = BlockOperation {
            print("test")
        }
        operation.completionBlock = {
            print("finish")
        }
        operationQueue.addOperation(operation)
    }
}
```

test
finish

Suspend


Operation 1
running
Operation 2
ready
suspend
Operation 2
ready
Stop
Operation Queue
Operation Queue

Suspend

```
class OperationSuspendTest {

    private let operationQueue = OperationQueue()

    func test() {
        let operation1 = BlockOperation {
            sleep(1)
            print("test1")
        }

        let operation2 = BlockOperation {
            sleep(1)
            print("test2")
        }

        operationQueue.maxConcurrentOperationCount = 1
        operationQueue.addOperation(operation1)
        operationQueue.addOperation(operation2)

        operationQueue.isSuspended = true
    }
}
```

...
1 second
...
test1

GCD vs Operation

Operation:

- Cancelation
- Observable
- Dependencies

Gcd:

- Simplicity
- Low-level