Desarrollo de Apps para iOS

Concurrencia y Usabilidad

IWEB 2016-2017

Santiago Pavón

ver: 2016.11.14

© Santiago Pavón - UPM-DIT

Objetivos

- La interface de usuario:
  - que no se quede bloqueada mientras estamos realizando un cálculo muy largo, descargando recursos de la red, ...
  - que siempre responda ágilmente a las acciones del usuario.

© Santiago Pavón - UPM-DIT

UIKit y Main Thread

- Todas las aplicaciones tienen un thread principal (main thread) donde se ejecuta el main run loop.
  - Este thread procesa los eventos, ejecuta las acciones de nuestros controles (target-action), actualiza el interface de usuario.
  - Evitar ralentizarlo por ejecutar en él tareas muy largas.
    - Trocear estas tareas, usar concurrencia.
- Cuidado con la concurrencia:
  - La mayor parte del **UIKit no es thread-safe**.
    - por motivos de eficiencia.
  - Los accesos a la interface de usuario deben hacerse solo desde el main thread.

© Santiago Pavón - UPM-DIT

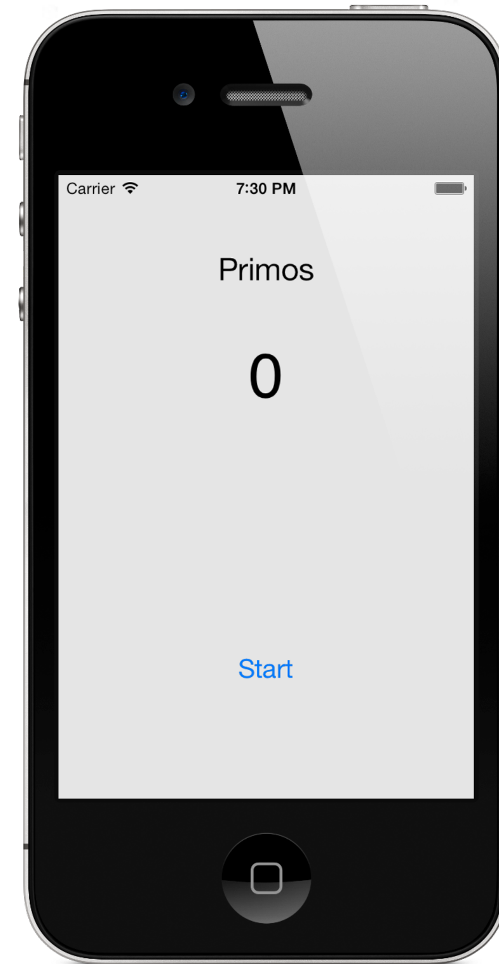
Ejemplo: Números Primos

- Este ejemplo ejecuta una acción en el main thread que tarda mucho en terminar (∞).
  - **Se congela la interface de usuario.**
- Tiene una label donde muestra el último primo encontrado:

```
@IBOutlet weak var primeLabel: UILabel!
```

- Y un botón para empezar a calcular primos:

```
@IBAction func findPrimes()
```



© Santiago Pavón - UPM-DIT

```
private func isPrime(_ n: Int) -> Bool {
    if n < 2 { return false }
    let r = Int(sqrt(Double(n)))
    for i in stride(from: 2, through: r, by: 1) {
        if n % i == 0 {
            return false
        }
    }
    return true
}

@IBAction func findPrimes() {
    for n in 2...Int.max {
        if isPrime(n) {
            print("Number \(n) is prime");
            primeLabel.text = "\(n)"
        }
    }
}
```

Esto es un bucle infinito, que provoca que el GUI se quede bloqueado.

© Santiago Pavón - UPM-DIT

Timer
Temporizadores

© Santiago Pavón - UPM-DIT

Timer

- Son objetos que planifican la ejecución de un método en ciertos instantes de tiempo.
  - Típicamente para retrasar una ejecución o para realizar ejecuciones repetitivas.
- El método planificado en el temporizador se ejecutará en el mismo thread del run loop donde se planificó.
  - Si es el main thread, la ejecución del método debe durar poco para no ralentizarlo.
    - Si el método requiere mucho tiempo de ejecución, conviene descomponerlo en varios métodos que requieran menos tiempo de ejecución cada uno de ellos.

© Santiago Pavón - UPM-DIT

- Crear un temporizador y lanzarlo (*planificarlo en el run loop*):

```
class func scheduledTimer(timeInterval ti: TimeInterval,
                          target aTarget: Any,
                          selector aSelector: Selector,
                          userInfo: Any?,
                          repeats yesOrNo: Bool) -> Timer
```

  - Ejecuta el selector del objeto target cuando ha pasado el tiempo (en segundos) especificado.
    - Y puede repetirse indefinidamente si se desea.
  - El método selector tiene que tener un parámetro, donde le pasa el objeto **Timer** que lo llamó.

- Parar el temporizador: Para no ejecutar más veces el selector.

```
func invalidate()
```

- Estado del temporizador: ¿Es válido el temporizador?

```
var isValid: Bool { get }
```

- Tolerancia: segundos que se puede retrasar el disparo del temporizador.
  - Recomendable porque mejora el gasto de batería.

```
var tolerance: TimeInterval
```

© Santiago Pavón - UPM-DIT

```
var counter = 1  // Ultimo primo encontrado

@IBAction func findPrimes() {

     // Busco un nuevo primo cada 0.1 segundos
     Timer.scheduledTimer(timeInterval: 0.1,
                          target: self,
                          selector: #selector(findNextPrime(_:)),
                          userInfo: nil,
                          repeats: true)
}

func findNextPrime(_ timer: Timer) {

    repeat {
        counter += 1
    } while !isPrime(counter)

    self.primeLabel.text = "\(counter)"
}
```

© Santiago Pavón - UPM-DIT

Notas

El método referenciado por el selector debe ser accesible desde el runtime de Objective-C.

Esto ocurre cuando el objeto target deriva de `NSObject`, pero si es un objeto Swift puro, entonces es necesario añadir el prefijo `@objc` delante de la declaración del método usado como selector.

Además, el método referenciado por el selector no puede tener visibilidad `private`, su visibilidad debe ser por lo menos `internal`.

© Santiago Pavón - UPM-DIT

Concurrencia

- El uso de temporizadores:
  - Es sencillo de utilizar si tenemos pocas tareas concurrentes.
    - Aunque partir un método que requiere mucho tiempo de ejecución en métodos que hagan tareas más cortas puede ser complicado.
  - Pero si necesitamos ejecutar bastantes tareas simultáneamente, la complejidad del código puede dispararse.
  - No es concurrencia de verdad.
    - Si la ejecución de los métodos dura mucho, el main thread estará muy ocupado y la interface de usuario no responderá con agilidad.

© Santiago Pavón - UPM-DIT

Threads

© Santiago Pavón - UPM-DIT

Threads

- El framework Foundation proporciona un recubrimiento (**`Thread`**) para el manejo de threads.
- Pero hay que tener en cuenta que realizar programas concurrentes usando directamente threads es bastante complicado.
  - Condiciones de carrera, acceso simultáneo a secciones críticas y datos compartidos, bloqueos, inversión de prioridades, etc.
    - Obtendremos resultados y comportamientos erróneos.
    - Muy difícil de depurar.
  - Además, el uso de muchos threads puede ralentizar un programa.

© Santiago Pavón - UPM-DIT

Thread

- Crear un objeto para controlar la ejecución de un thread.
- Varias formas de crearlos:
  - Invocando:

```
class func detachNewThreadSelector(_ selector: Selector,
                                   toTarget target: Any,
                                  with argument: Any?)
```

    - Su ejecución empieza inmediatamente.
  - Invocando:

```
convenience init(target: Any,
               selector: Selector,
                 object: Any?)
```

    - Su ejecución empieza al llamar al método **start**.
  - También se puede crear una clase derivada de Thread y sobrescribir el método **main**.
    - Hay que crear manualmente un **AutoreleasePool**.

```
func autoreleasepool<Result>(invoking: () throws -> Result)
                             rethrows -> Result
```

- Cada instancia de **Thread** se ejecuta en su propio run loop.

© Santiago Pavón - UPM-DIT

Cerrojos

- **NSLock**: Control de acceso a secciones críticas y a datos compartidos:

```
let lock = NSLock()

lock.lock()
// zona protegida
lock.unlock()
```

- **NSCondition**: Señalización entre threads:

```
let cond = NSCondition()

// En el Thread Productor:
  cond.lock()
  // Meter al saco
  cond.signal()
  cond.unlock()


// En el Thread Consumidor:
  cond.lock()
  while (saco esta vacio)
      cond.wait()
  // Sacar del saco
  cond.unlock()
```

© Santiago Pavón - UPM-DIT

Bloqueos

- Ocurre cuando un thread está bloqueado esperando por una condición que nunca va a ocurrir.
- Ejemplo:
  - Un thread tiene el cerrojo A y espera por el cerrojo B.
  - Otro thread tiene el cerrojo B y espera por el cerrojo A.
- Recomendación:
  - no llamar a un bloque de código protegido con un cerrojo desde un bloque de código protegido con otro cerrojo.

© Santiago Pavón - UPM-DIT

Dormir un rato

- Si el programa tiene muchos threads consumiendo mucha cpu, el programa puede congestionarse.
  - La interface de usuario dejará de responder ágilmente.
  - Esto se agrava más en un terminal con pocos recursos.
- Solución: Podemos hacer que los threads se duerman un rato para favorecer a los demás threads.

```
Thread.sleep(forTimeInterval: 0.1)

let date = Date(timeIntervalSinceNow:0.1)
Thread.sleep(until: date)
```

- Muy importante: no dormir nunca el main thread.

© Santiago Pavón - UPM-DIT

Recomendaciones

- En general el uso de threads no se recomienda por su dificultad.
  - Difícil determinar cuantos threads usar dependiendo de los recursos disponibles, y gestionarlos.
    - O ajustar su número dinámicamente según la carga actual del sistema.
  - Difícil conseguir que se ejecuten eficientemente.
  - Difícil sincronización entre threads.
- Las tecnologías recomendadas para implementar concurrencia en las aplicaciones son:

  `Grand Central Dispatch`

  `Operation Queues`

  - No nos preocupamos de la creación y gestión de los threads.
  - La idea es definir tareas concretas a realizar, y dejar que el sistema las realice.

© Santiago Pavón - UPM-DIT

Grand Central Dispatch

© Santiago Pavón - UPM-DIT

GCD

- GCD es un framework para manejar la ejecución asíncrona y concurrente de tareas.
  - Nos oculta los detalles de la multiprogramación.
  - No tenemos que preocuparnos de los recursos disponibles. GCD se encarga de gestionarlos.
  - Se programa muy fácilmente.
- GCD nos proporciona colas a las que enviaremos tareas para que se ejecuten en orden.
  - Una tarea es un objeto closure.
  - En algún momento GCD asignará un thread libre a la siguiente tarea de la cola para que se ejecute.
    - GCD decide cuántos threads crea, cuándo un thread ejecutará una tarea de la cola, y durante cuánto tiempo.
  - El encolado es **thread-safe**: Se puede acceder a las colas desde distintos threads de forma segura.

© Santiago Pavón - UPM-DIT

DispatchQueue

- Las colas pueden ser serie o concurrentes:
  - **Cola concurrente**:
    - El comienzo de la ejecución de las tareas es FIFO.
      - Las tareas empiezan a ejecutarse en el mismo orden en que se metieron en la cola.
    - Pero una vez que empieza la ejecución de una tarea, se puede comenzar con la ejecución de la siguiente tarea.
      - Es decir, las tareas se ejecutan concurrentemente.
    - Y pueden acabar en cualquier orden.
  - **Cola serie**:
    - Las tareas se ejecutan de una en una en modo FIFO, en el mismo orden en que se encolaron.
    - Hasta que una tarea no termina, no se empieza a ejecutar la siguiente.
- Hay varios tipos de colas disponibles:
  - **Cola main**: cola del sistema para ejecutar tareas en serie en el main thread.
  - **Colas globales**: colas del sistema para ejecutar tareas concurrentemente con distintas prioridades
  - **Colas personales**: podemos crear nuestras propias colas serie o concurrentes en cualquier momento.

© Santiago Pavón - UPM-DIT

DispatchQueue.**main**

- Esta la cola a usar para ejecutar tareas en **serie** en el **Main Thread**.
    - Esta cola la crea el sistema automáticamente.
- Esta cola se obtiene accediendo a la propiedad de clase **`main`** de **`DispatchQueue`**:

```
class var main: DispatchQueue {get}
```

© Santiago Pavón - UPM-DIT

DispatchQueue.**global**

- En el sistema ya existen varias colas globales que se usan para ejecutar tareas **concurrentemente** con diferentes calidades de servicio (prioridad).
  - El comienzo de la ejecución de las tareas es FIFO, pero una vez que empieza la ejecución de una tarea, ya puede comenzar con la ejecución de la siguiente tarea, pudiendo acabar en cualquier orden.
- GCD crea automáticamente **varias** colas con distintas propiedades.
- Estas colas se obtienen con el método de clase `global` de `DispatchQueue`:

```
class func global(qos: DispatchQoS.QoSClass = default) -> DispatchQueue
```

  - Valores para **qos** (indican la calidad o prioridad de las tareas):

```
.userInteractive // Tareas muy breves que tienen que
                 // hacerse con suma prioridad.
.userInitiated   // Tareas mas largas que también
                 // tienen prioridad alta.
.utility         // Tareas largas que no tienen prioridad máxima.
.background      // Tareas para cosas que no necesito
                 // ahora mismo y tienen menos prioridad.
.default         // Calidad de servicio por defecto.
.unspecified     // Sin calidad de servicio.
```

© Santiago Pavón - UPM-DIT

DispatchQueue: Colas Personalizadas

- Pueden crearse colas personalizadas nuevas en cualquier momento.
  - Se crean usando alguno de los constructores existentes:

```
init(label: String,
     qos: DispatchQoS = default,
     attributes: DispatchQueue.Attributes = default,
     autoreleaseFrequency: DispatchQueue.AutoreleaseFrequency = default,
     target: DispatchQueue? = default)
```

- Parámetros:
  - **label**: es un String que identifica la cola para ayudar en la depuración.
  - **qos**: es la calidad del servicio.
  - **attributes**: conjunto de valores

```
initiallyInactive   // no ejecutar tareas hasta que se active la cola
concurrent          // para crear una cola concurrente.
```

  - **autorelaeaseFrecuency**: indicar cuando se hace el autorelease..
  - **target**: las tareas de las colas personalizadas se envían a una cola global cuando tienen que ejecutarse. Podemos indicar a que cola global queremos que se envíen.
- Por defecto las colas creadas son serie, excepto si se pasa concurrent en los atributos.
  - Las colas serie ejecutan sus tareas de una en una y en orden FIFO.
    - Si una operación se bloquea, sólo se bloquea su cola. Las demás colas continúan ejecutando sus tareas.
    - Si la cola creada es concurrente, los bloques se desencolan en orden FIFO, y se ejecutan concurrentemente (si hay recursos disponibles para ello). Pueden terminar en cualquier orden.
- Usos: evitar que el main thread se bloquee, realizar tareas largas en otro thread, proteger zonas críticas, …

© Santiago Pavón - UPM-DIT

```swift
func demo(serie: Bool) {
    let attrs = serie ? [] : DispatchQueue.Attributes.concurrent
    let q = DispatchQueue(label: "demo",  attributes: attrs)

    for i in 1...3 {
       q.async {
            print("Empieza la tarea \(i)")
            Thread.sleep(forTimeInterval: 0.5)
            print("Termina la tarea \(i)")
        }
    }
}

demo(serie: true)
// Empieza la tares 1
// Termina la tarea 1
// Empieza la tarea 2
// Termina la tarea 2
// Empieza la tarea 3
// Termina la tarea 3

demo(serie: false)
// Empieza la tarea 2
// Empieza la tarea 3
// Empieza la tarea 1
// Termina la tarea 3
// Termina la tarea 1
// Termina la tarea 2
```

© Santiago Pavón - UPM-DIT

DispatchQueue: Encolar Tareas

- Existen muchos métodos para enviar una tarea a una cola:

```
func async(group: DispatchGroup? = default,
           qos: DispatchQoS = default,
           flags: DispatchWorkItemFlags = default,
           execute work: @escaping () -> Void)

func async(execute workItem: DispatchWorkItem)

func sync(execute: () -> Void)

func sync<T>(flags: DispatchWorkItemFlags,
             execute work: () throws -> T) rethrows -> T
```

- donde:
  - **async** no es bloqueante.
  - **sync** es bloqueante.
    - Se espera hasta que el closure ha terminado, y devuelve su valor.
  - DispatchWorkItem encapsula un trabajo a ejecutar (un closure).
  - DispatchGroup se usa para agregar trabajos.

© Santiago Pavón - UPM-DIT

DispatchWorkItem

- Este tipo encapsula una tarea a ejecutar.
- Se crea pasando un closure en el constructor.

```
init(qos: DispatchQoS = default,
      flags: DispatchWorkItemFlags = default,
      block: @escaping () -> ())
```

- Existen métodos para esperar hasta un DispatchWorkItem ha terminado su ejecución (**wait**), notificar cuando ha terminado (**notify**), cancelarlo (**cancel**), …

© Santiago Pavón - UPM-DIT

Ejemplo: Primos-Cola Global

```
class ViewController: UIViewController {

    @IBOutlet weak var label1: UILabel!
    @IBOutlet weak var label2: UILabel!


    override func viewDidLoad() {
        super.viewDidLoad()

        // Buscar el siguiente primo del valor dado y
        // ponerlo en label dada.
        displayNextPrimes(into: label1, startingAt: 1)
        displayNextPrimes(into: label2, startingAt: 1000000)
    }
```

© Santiago Pavón - UPM-DIT

```swift
    private func displayNextPrimes(into label: UILabel, startingAt v: Int) {
        var v = v

        // Cola Global con QOS Utility
        let queue = DispatchQueue.global(qos: DispatchQoS.QoSClass.utility)

        // Encolar asincrono
        queue.async {
            repeat {
                v += 1
            } while !self.isPrime(v)

            // Actualizar el GUI en el Main Thread
            DispatchQueue.main.async {

                label.text = "\(v)"

                // Pasados 0.01 segundos busco el siguiente
                let t = DispatchTime.now() + 0.01
                queue.asyncAfter(deadline: t) {
                    self.displayNextPrimes(into: label, startingAt: v)
                }
            }
        }
    }

    private func isPrime(n: Int) -> Bool {. . .}
}
```

© Santiago Pavón - UPM-DIT

Ejemplo: Bajarse una Foto

```
let imgUrl = "http://www.etsit.upm.es/fileadmin/user_upload/banner_portada_escuela.jpg"

// Construir un URL
if let url = URL(string: imgUrl) {
    // Envio la tarea a un thread
    let queue = DispatchQueue(label: "Download Queue")
    queue.async {
        // Bajar los datos del sitio Web
        if let data = try? Data(contentsOf: url),
            // Construir una imagen con los datos bajados
            let img = UIImage(data: data) {

            // El GUI se actualiza en el Main Thread
            DispatchQueue.main.async {
                self.imageView.image = img
            }
        }
    }
}
```

Uso GCD y envío la tarea a un thread

El GUI solo se actualiza en el Main Thread

© Santiago Pavón - UPM-DIT

```swift
let imgUrl = "http://www.etsit.upm.es/fileadmin/user_upload/banner_portada_escuela.jpg"

// Construir un URL
if let url = URL(string: imgUrl) {

    // Envio la tarea a un thread
    let queue = DispatchQueue(label: "Download Queue")
    queue.async {

        // El GUI se actualiza en el Main Thread
        DispatchQueue.main.async {
            // Mostrar indicador de actividad de red
            UIApplication.shared.isNetworkActivityIndicatorVisible = true
        }
        defer {
            DispatchQueue.main.async {
                // Ocultar indicador de actividad de red
                UIApplication.shared.isNetworkActivityIndicatorVisible = false
            }
        }

        // Bajar los datos del sitio Web
        if let data = try? Data(contentsOf: url),
            // Construir una imagen con los datos bajados
            let img = UIImage(data: data) {

            // El GUI se actualiza en el Main Thread
            DispatchQueue.main.async {
                self.imageView.image = img
            }
        }
    }
}
```

Uso GCD y envío la tarea a un thread

El GUI solo se actualiza en el Main Thread

© Santiago Pavón - UPM-DIT

Ejemplo: Zona Crítica

- Podemos utilizar colas serie para proteger el acceso a zonas críticas:
  - Hay que hacer que la única forma de acceder a la zona crítica sea enviando una tarea (closure) a través de una cola serie.
  - Como esas tareas se ejecutarán de una en una, se elimina la posibilidad de que varias tareas se ejecuten concurrentemente sobre la zona crítica.

© Santiago Pavón - UPM-DIT

```swift
// Zona critica
private var total = 0

// Cola de acceso a la zona critica
private var queue = DispatchQueue("Cola de acceso")

func meter(_ n: Int) {
    queue.async {

        self.total += n;

        DispatchQueue.main.async {
            self.totalLabel?.text = "\(self.total)"
        }
    }
}

func sacar(_ n: Int) {
    queue.async {

        self.total -= n;

        DispatchQueue.main.async {
            self.totalLabel?.text = "\(self.total)"
        }
    }
}
```

Zona crítica

Crear cola serie para serializar tareas

Así es Thread Safe

Zona crítica

Actualizo el GUI en el main thread

Así es Thread Safe

Zona crítica

Actualizo el GUI en el main thread

© Santiago Pavón - UPM-DIT

Retrasar una Tarea

- Esperar hasta el momento especificado, y entonces enviar asíncronamente una tarea a la cola especificada.

```
let queue = DispatchQueue.global()

// Enviar dentro de 5 segundos
let t = DispatchTime.now() + 5
queue.asyncAfter(deadline: t) {
    // Las sentencias a ejecutar
}
```

- Existen varios métodos:

```
func asyncAfter(deadline: DispatchTime,
                execute: DispatchWorkItem)
func asyncAfter(deadline: DispatchTime,
                qos: DispatchQoS,
                flags: DispatchWorkItemFlags,
                execute: @escaping () -> Void)
func asyncAfter(wallDeadline: DispatchWallTime,
                execute: DispatchWorkItem)
func asyncAfter(wallDeadline: DispatchWallTime,
                qos: DispatchQoS,
                flags: DispatchWorkItemFlags,
                execute: @escaping () -> Void)
```

© Santiago Pavón - UPM-DIT

Ejecutar solo una vez

- En las versiones anteriores a Swift 3 se usaba la función **dispatch_once** para forzar a que ciertas sentencias solo se ejecutaran una sola vez incluso en un entorno multithread.
  - Casos típicos la creación de singletons, la configuración inicial de algún elemento.
- Esta función no es segura en ciertos escenarios, y se ha eliminado del nuevo API.
- En Swift 3 pueden usarse variables globales y propiedades estáticas para realizar ejecuciones que solo pueden ocurrir una vez.
  - Se inicializan de forma perezosa la primera vez que se usan desde cualquier thread.

```
let a = UnaClase()     // Inicializacion perezosa.
var b: UnaClase = { let r = UnaClase()
                    r.unaPropiedad = "valor"
                    return r
                  }() // Lo mismo ejecutando un closure.
class Singleton {
    static let shared = Singleton() // Inicializacion perezosa.
    private init() {}  // Inicializador privado.
}
```

© Santiago Pavón - UPM-DIT

DispatchWorkItemFlags

- Pueden configurarse el comportamiento de una tarea usando las opciones proporcionadas por este tipo.
- Ejemplo:
  - Crear una barrera con la opción DispatchWorkItemFlags.**barrier**.
    - Una barrera es un punto de sincronización en una cola concurrente.
    - Una cola no ejecuta una tarea añadida con esta opción hasta que no han terminado todas las tareas añadidas anteriormente.
    - Cuando las tareas añadidas anteriormente han terminado, se ejecuta la tarea barrera.
    - Cuando termina la tarea barrera, la cola sigue funcionando de forma concurrente.

© Santiago Pavón - UPM-DIT

```
let q = DispatchQueue(label: "cola",  attributes: .concurrent)

q.async { self.producir() }
q.async { self.producir() }
q.async { self.producir() }

q.async(flags: .barrier) { self.barrera() }

q.async { self.consumir() }
q.async { self.consumir() }
q.async { self.consumir() }


func producir() { print("P") }

func barrera() { print("B") }

func consumir() { print("C") }


// PPPBCCC
```

© Santiago Pavón - UPM-DIT

API más ...

- `DispatchTime`
- `DispatchGroup`
- `DispatchSemaphore`
- `DispatchSource`
- `DispatchData`
- `DispatchIO`
- `DispatchSpecificKey`
- etc. . .

© Santiago Pavón - UPM-DIT

Operations

© Santiago Pavón - UPM-DIT

Operation Queue y Operations

- Abstracción de alto nivel para no tener que usar threads.
- Una operación (`Operation`) es un objeto que representa una unidad de trabajo a realizar.
  - con dependencias y prioridades
- Las colas de operaciones (`OperationQueue`):
  - Ejecutan **simultáneamente** todas las operaciones que se añaden a la cola, pero respetando las restricciones que indiquemos (**prioridades** y **dependencias**)
  - Ejecutan las operaciones usando eficientemente los recursos disponibles.
    - Cuando queda un thread libre se saca una operación de la cola.

© Santiago Pavón - UPM-DIT

Operation

- Es la clase que encapsula el código de una tarea a realizar.
- Es una clase base **abstracta**:
  - No se crean instancias de esta clase.
- Opciones para crear instancias de esta clase:
  - Usar la subclase **BlockOperation** que ya nos proporcionan hecha,
    - Al inicializador se le pasa un closure con las sentencias a ejecutar.
  - Crear nuestra propia clase personalizada que derive de Operation.
    - En estas subclases hay que sobrescribir el método **main**, controlar las excepciones, crear un autorelease pool, etc.

© Santiago Pavón - UPM-DIT

BlockOperation

- Es una **subclase** de **`Operation`** que ya nos proporciona hecha.
- La operación ejecutará el block que se pasa en el inicializador o los blocks que se añadas después.

```
init(block: @escaping () -> Void)
func addExecutionBlock(_: @escaping () -> Void)
```

© Santiago Pavón - UPM-DIT

Demo: BlockOperation

- Demo:
  - Calcular simultáneamente dos secuencias de primos y mostrarlos en dos labels.
  - Crear un método que busca un primo y lo muestra en una label.
    - Crea un Operation con un closure que
      - calcula el siguiente primo,
      - actualiza la label para mostrar el primo encontrado,
      - se duerme un rato
      - y llama otra vez al mismo método que hemos creado para buscar el siguiente primo.

© Santiago Pavón - UPM-DIT

```
class ViewController: UIViewController {

    @IBOutlet weak var label1: UILabel!
    @IBOutlet weak var label2: UILabel!



    // Cola de las operaciones
    var queue: OperationQueue!


    override func viewDidLoad() {
        super.viewDidLoad()

        queue = NSOperationQueue()

        // Crea operaciones que busquen el siguiente primo despues
        // del valor dado y lo pongan en las labels dadas.
        displayNextPrimes(into: label1, startingAt: 1)
        displayNextPrimes(into: label2, startingAt: 1000000)
    }
```

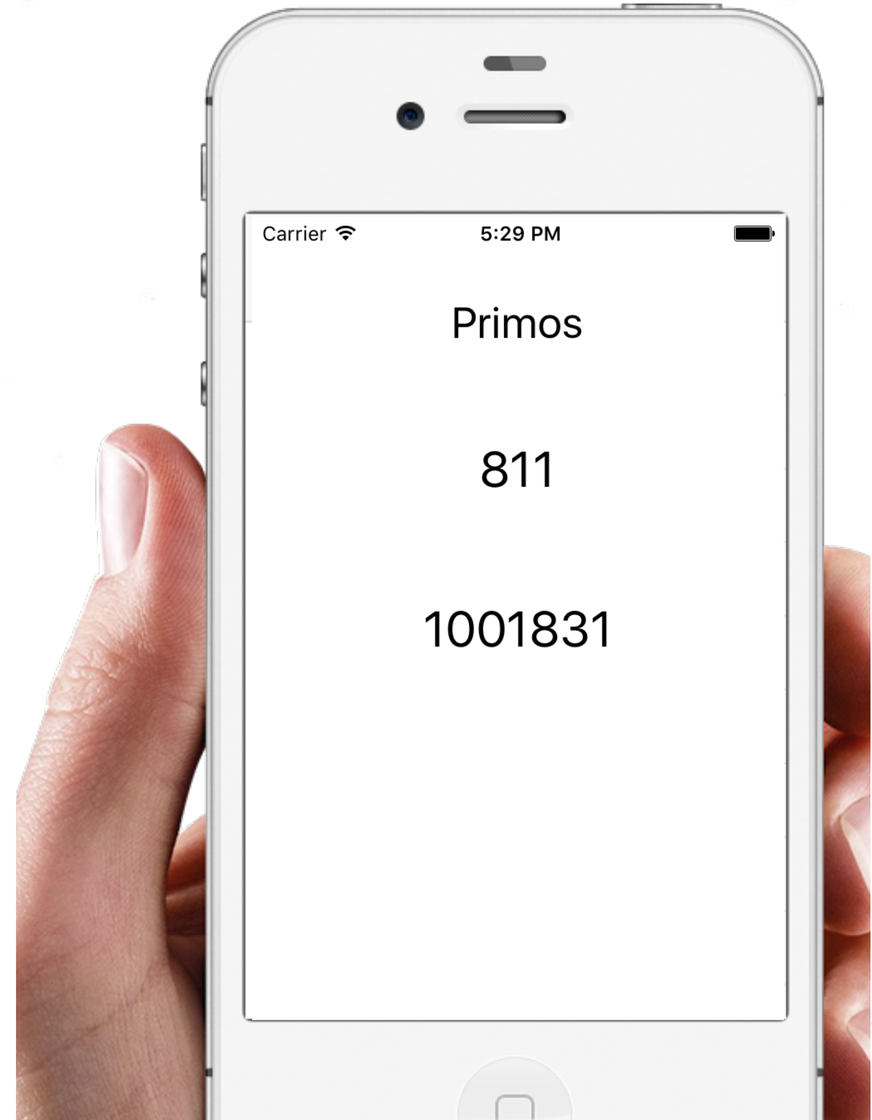
© Santiago Pavón - UPM-DIT

```
private func displayNextPrimes(into label: UILabel, startingAt v: Int) {

    // Creo un objeto Operacion
    let ope = BlockOperation(block: {

        repeat {
            v += 1
        } while !self.isPrime(v)

        // Actualizar la label (el GUI) en el Main Thread
        OperationQueue.main.addOperation({
            label.text = "\(v)"
        })

        Thread.sleep(forTimeInterval: 0.01)  // Me duermo un rato

        // Creo una nueva operacion
        self.displayNextPrimes(into: label, startingAt: v)
    })

    // Meter la operacion en el Operation Queue
    queue.addOperation(ope)
}

private func isPrime(n: Int) -> Bool {. . .}
}
```

© Santiago Pavón - UPM-DIT

Demo: Subclase Personalizada

- Demo:
  - Calcular simultáneamente dos secuencias de primos y mostrarlos en dos labels.
  - Crear un método que busca un primo y lo muestra en una label.
    - Crea una subclase personalizada que deriva de Operation, y se sobrescribe el método main para que calcule el siguiente primo.
    - Crea un completion block que actualiza la label y llama al mismo método para buscar el siguiente primo.



© Santiago Pavón - UPM-DIT

```
class NextPrimeOperation : Operation {

    // Valor inicial desde el que buscar el siguiente primo,
    // y donde se deja el siguiente primo encontrado.
    var value: Int

    init(startingAt value: Int) {
        self.value = value
    }

    override func main() {
        autoreleasepool {
            if self.cancelled { return }
            repeat {
                value += 1
            } while !self.isPrime(value)
        }
    }

    fileprivate func isPrime(_ n: Int) -> Bool {
        if n < 2 { return false }
        let r = Int(sqrt(Double(n)))
        for i in stride(from: 2, through: r, by: 1) {
            if n % i == 0 {
                return false
            }
        }
        return true
    }
}
```

© Santiago Pavón - UPM-DIT

```swift
class ViewController: UIViewController {

    @IBOutlet weak var label1: UILabel!
    @IBOutlet weak var label2: UILabel!

    // Cola de las operaciones
    var queue: OperationQueue!

    override func viewDidLoad() {
        super.viewDidLoad()

        queue = OperationQueue()

        // Crea operaciones que busquen el siguiente primo despues
        // del valor dado y lo pongan en las labels dadas.
        displayNextPrimes(into: label1, startingAt: 1)
        displayNextPrimes(into: label2, startingAt: 1000000)
    }
```

© Santiago Pavón - UPM-DIT

```swift
    fileprivate func displayNextPrimes(into label: UILabel, startingAt v: Int) {

        // Creo un objeto Operacion
        let ope = NextPrimeOperation(startingAt: v)

        // Closure que se ejecuta cuando termina la operación.
        // Nota sobre gestion de memoria:
        //   - ope retiene al closure asignado en el completionBlock.
        //   - el closure retiene a ope ya que lo apunta.
        //   - TENGO UN BUCLE DE RETENCIONES ☹️
        //     Lo soluciono con la lista de capturas unowned.
        ope.completionBlock = { [unowned ope] in

            let v = ope.value

            // Actualizar la label (el GUI) en el Main Thread
            OperationQueue.main.addOperation({
                label.text = "\(v)"
            })

            Thread.sleepFor(timeInterval: 0.01)  // Me duermo un rato

            // Creo una nueva operacion
            self.displayNextPrimes(into: label, startingAt: v)
        }

        // Meter la operacion en el Operation Queue
        queue.addOperation(ope)
    }
}
```

© Santiago Pavón - UPM-DIT

Completion Block

- A un objeto Operation puede asignársele un completion block.
  - Se invoca cuando la operación ha terminado de ejecutarse.
    - Tanto si termina con éxito o es cancelada.
      - Consultar el valor de la propiedad **cancelled**.

```
var completionBlock: (() -> Void)?
```

- Cuidado: El contexto de ejecución del bloque será algún thread secundario.
  - No es el Main Thread.

© Santiago Pavón - UPM-DIT

Dependencias entre Operaciones

- Podemos indicar que una operación B no puede ejecutarse hasta que no haya terminado la ejecución de otra operación A.

```
func addDependency(_ op: Operation)
```

- Consultar dependencias:

```
var dependencies: [Operation] { get }
```

- Eliminar dependencias:

```
func removeDependency(_ op: Operation)
```

© Santiago Pavón - UPM-DIT

Prioridad de las Operaciones

- Las operaciones tienen una prioridad
  - La usan las colas para decidir que operación ejecutar y cuanto tiempo se ejecutará.

```
var queuePriority: Operation.QueuePriority
```

- Valores:

```
Operation.QueuePriority.veryLow
Operation.QueuePriority.low
Operation.QueuePriority.normal
Operation.QueuePriority.high
Operation.QueuePriority.veryHigh
```

© Santiago Pavón - UPM-DIT

Calidad de Servicio de las Operaciones

- Las operaciones tienen una Calidad de Servicio
  - Indica la importancia de la operación para acceder a los recursos del sistema (CPU, red, memoria, . . .)

```
var qualityOfService: QualityOfService
```

- Valores:

```
QualityOfService.userInteractive
QualityOfService.userInitiated
QualityOfService.utility
QualityOfService.background
QualityOfService.default
```

© Santiago Pavón - UPM-DIT

Cancelar una Operación

- La ejecución de una operación puede cancelarse:

```
func cancel()
```

- Cancelar una operación sólo hace que la propiedad **isCancelled** valga **true**.
- Debemos consultar esta propiedad con frecuencia en el método **main** o los bloques de las operaciones, y terminar si vale **true**.
- Cancelar una operación pendiente en la cola no la saca de esta.
  - Hay que esperar hasta que empiece a ejecutarse, y comprobar inicialmente en el método **main** o en los bloques si la operación fue cancelada, y si es así, terminar. Las operaciones sólo se eliminan de la cola cuando termina su ejecución.
- En los completionBlock también hay que mirar si la operación fue cancelada.

© Santiago Pavón - UPM-DIT

Propiedades de las Operaciones

- `var isReady: Bool { get }`
  - **true** si la operación está lista para ejecutarse, es decir, no hay dependencias por las que esperar.
- `var isCancelled: Bool { get }`
  - **true** si la operación fue cancelada.
- `var isExecuting: Bool { get }`
  - **true** si la operación es está ejecutando (ha empezado y no ha acabado su ejecución).
- `var isFinished: Bool { get }`
  - **true** si terminó la ejecución de la operación.
- `var isAsynchronous:  Bool { get }`
  - Indica si la operación se ejecuta asíncronamente.
- . . .

© Santiago Pavón - UPM-DIT

Esperar a que Termine una Operación

- Detener el thread actual hasta que termine una operación.

```
func waitUntilFinished()
```

- Una operación no debe invocar este método con operaciones que están en su misma cola, ya que pueden producirse bloqueos.

- Normalmente se usa con operaciones añadidas a otras colas para esperar a que terminen.

© Santiago Pavón - UPM-DIT

OperationQueue

- Son los objetos que gestionan la ejecución de las operaciones.
- Para crear una cola:

```
let queue = OperationQueue()
```

- Para añadir una operación a la cola:

```
queue.addOperation(opA)
```

  - La operación se ejecutará cuanto exista un thread disponible y la operación esté lista (**isReady** indicará que no hay dependencias pendientes).
- Por defecto, las colas deciden cuantos threads usarán dependiendo del hardware disponible.
  - Podemos modificar el número de threads a usar con la propiedad:

```
var maxConcurrentOperationCount: Int
```

    - Tendremos una cola serie si sólo usamos un thread.
- Una cola puede suspenderse para que no ejecute ninguna operación más.

```
var isSuspended: Bool
```

© Santiago Pavón - UPM-DIT

```
var operations: [Operation] { get }

func addOperations(_ ops: [Operation],
     waitUntilFinished wait: Bool)

func addOperation(_ block: @escaping () -> Void)

func waitUntilAllOperationsAreFinished()

func cancelAllOperations()

class var current: OperationQueue? { get }
class var mainQueue: OperationQueue { get }

. . .
```

© Santiago Pavón - UPM-DIT

¿Qué uso?
¿Operation o GCD?

© Santiago Pavón - UPM-DIT

¿Qué Elijo?

- Operation y Operation Queue están construidas sobre GCD.
  - Representan un nivel de abstracción más alto, lo cual es recomendable,
  - pero su uso introduce algo de sobrecarga respecto de usar GCD,
  - pero proporcionan algunas facilidades que GCD no tiene (hay que programárselas):
    - Definir dependencias entre tareas, cancelar o suspender tareas, etc.
  - . . .
- Hay que elegir teniendo en cuenta estos puntos.

© Santiago Pavón - UPM-DIT

© Santiago Pavón - UPM-DIT