



POLITÉCNICA

ETSIT  
UPM

*dit*  
UPM

# Desarrollo de Apps para iOS Concurrencia y Usabilidad

IWEB,LSWC 2014-2015

Santiago Pavón

ver: 2015.05.14

# 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.

# Main Run Loop

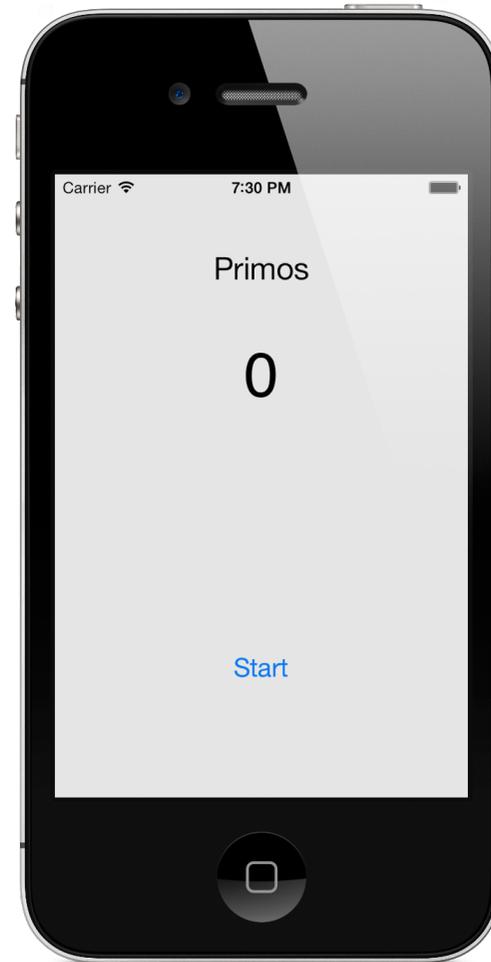
- Todas las aplicaciones tienen un thread (main thread) donde se ejecuta el main run loop.
  - Procesa los eventos, ejecuta las acciones de nuestros controles (target-action), actualiza el interface de usuario.

# 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()
```



```
private func isPrime(n: Int) -> Bool {  
    let r = Int(sqrt(Double(n)))  
    for (var i = 2 ; i <= r ; i++) {  
        if n % i == 0 {  
            return false  
        }  
    }  
    return true  
}
```

```
@IBAction func findPrimes() {  
    for var n = 2 ; ; n++ {  
        if isPrime(n) {  
            println("Number \n(n) is prime");  
            primeLabel.text = "\n(n)"  
        }  
    }  
}
```

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

# NSTimer

## Temporizadores

# NSTimer

- 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 parar el main run loop.
    - Si el método requiere mucho tiempo de ejecución, conviene descomponerlo en varios métodos que requieran menos tiempo de ejecución cada uno de ellos.

- Crear un temporizador y planificarlo en el run loop:

```
class func scheduledTimerWithTimeInterval(  
    _ seconds: NSTimeInterval,  
    target target: AnyObject,  
    selector aSelector: Selector,  
    userInfo userInfo: AnyObject?,  
    repeats repeats: Bool) -> NSTimer
```

- Ejecuta el selector del objeto target cuando han pasado el tiempo (en segundos) especificado.
  - Y puede repetirse indefinidamente si se desea.
- El método selector tiene que tener un parámetro, donde le pasan el objeto **NSTimer** que lo llamó.
- Parar el temporizador: Para no ejecutar más veces el selector.

```
func invalidate()
```

- Estado del temporizador: ¿Es válido el temporizador?

```
var valid: Bool { get }
```

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

@IBAction func findPrimes() {

    // Busco un nuevo primo cada 0.1 segundos
    NSTimer.scheduledTimerWithTimeInterval(0.1,
                                           target: self,
                                           selector: "findNextPrime:",
                                           userInfo: nil,
                                           repeats: true)
}

func findNextPrime(timer: NSTimer) {

    while !isPrime(++counter) {}

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

# Concurrencia

# 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 run loop estará muy ocupado y la interface de usuario no responderá con agilidad.

# UIKit no es Thread-safe

- Cuidado con la concurrencia:
  - La mayor parte del UIKit no es thread-safe.
    - por motivos de eficiencia.
  - Todo el trabajo de la interface de usuario debe hacerse en el main thread.

# Threads

# Threads

- El framework Foundation proporciona un recubrimiento (**NSThread**) 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.

# NSThread

- Crear un objeto para controlar la ejecución de un thread.

- Varias formas de crearlos:

- Invocando:

```
class func detachNewThreadSelector(_ aSelector: Selector,  
                                  toTarget aTarget: AnyObject,  
                                  withObject anArgument: AnyObject?)
```

- Su ejecución empieza inmediatamente.

- Invocando:

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

- Su ejecución empieza al llamar al método **start**.

- También se puede crear una clase derivada de NSThread y sobrescribir el método **main**.

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

- hay que crear manualmente un **NSAutoreleasePool**.

# Cerrojos

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

`NSLock`

`@synchronized(objetolock)`

- Señalización entre threads:

`NSCondition`

```
NSLock * myLock = [[NSLock alloc] init];
```

```
[myLock lock];  
// zona protegida  
[myLock unlock];
```

```
@synchronized(self) {  
    // zona protegida  
}
```

```
NSCondition * cond = [[NSCondition alloc] init];
```

```
// 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];
```

# 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.

# 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.

```
NSThread.sleepForTimeInterval(0.1)
```

```
let date = NSDate(timeIntervalSinceNow:0.1)  
NSThread.sleepUntilDate(date)
```

- Muy importante: no dormir nunca el main thread.

# 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.

# Operations

# Operation Queue y Operations

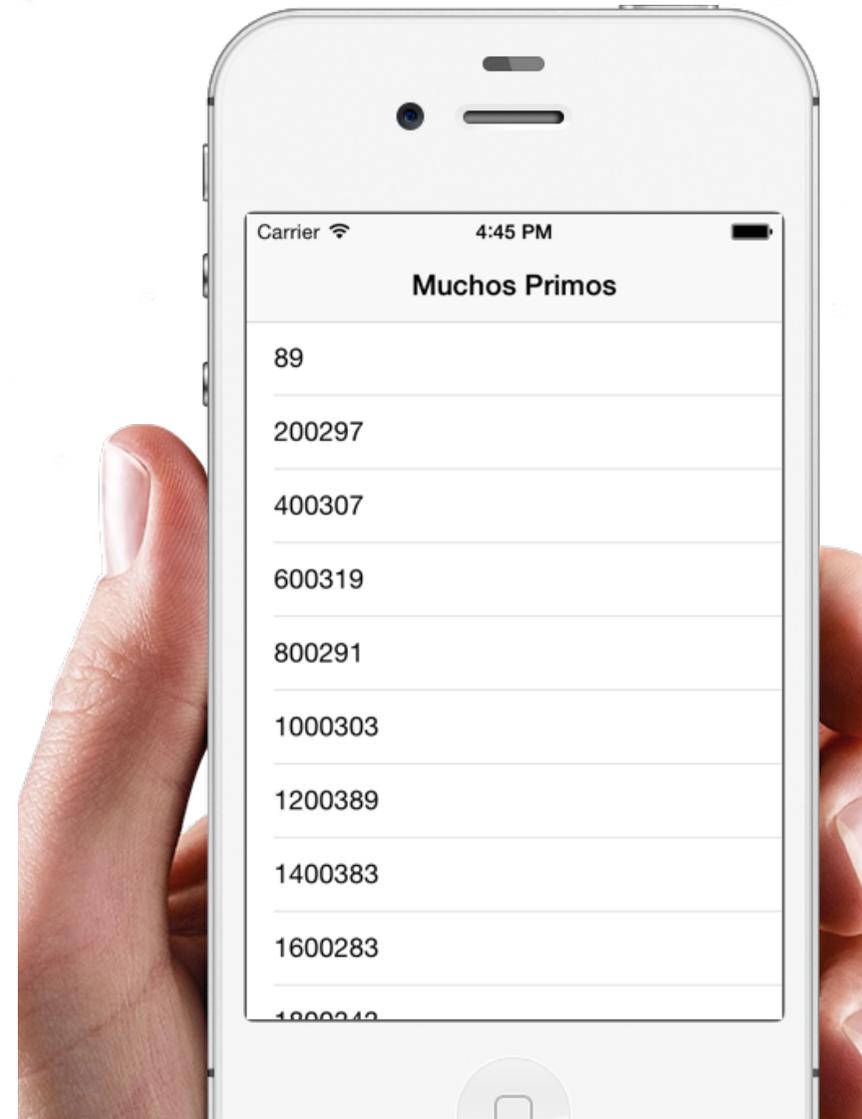
- Abstracción de alto nivel para no tener que usar threads.
- Una operación (**NSOperation**) es un objeto que representa una unidad de trabajo a realizar.
  - con dependencias y prioridades
- Las colas de operaciones (**NSOperationQueue**):
  - Ejecutan simultáneamente todas las operaciones que se añaden 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.

# NSOperation

- Es la clase que encapsula el código de una tarea a realizar.
- Es una clase **abstracta**:
  - No se crean instancias de esta clase.
- Para crear instancias de esta clase:
  - Tenemos que crear primero nuestras propias subclases de NSOperation.
    - En las subclases hay que sobrescribir el método **main**, controlar las excepciones, crear un autorelease poll, etc.
  - O usar una subclase que ya nos proporcionan hecha:  
**NSBlockOperation.**

# Una Subclase Personalizada

- Demo:
  - Calcular muchos primos simultáneamente y mostrarlos en una tabla.
  - Creo una subclase de NSOperation, sobrescribiendo el método main, que calcula el siguiente primo.
  - La tabla se actualiza con el completion block de cada operación.
    - Y se crea otra operación.



```

class PrimeOperation : NSOperation {

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

    init(startValue: Int) {
        value = startValue
    }

    override func main() {
        autoreleasepool {
            if self.cancelled { return }
            while !self.isPrime(++self.value) {}
            if self.cancelled { return }
            NSThread.sleepForTimeInterval(0.1) // Me duermo un rato
        }
    }

    private func isPrime(n: Int) -> Bool {
        let r = Int(sqrt(Double(n)))
        for (var i = 2 ; i <= r ; i++) {
            if n % i == 0 {
                return false
            }
        }
        return true
    }
}

```

```

private let DATA_SIZE = 100

class PrimosTableViewController: UITableViewController {

    // Array donde guardo el ultimo primo encontrado en cada fila.
    var data = [Int]()

    // Cola de las operaciones
    var queue: NSOperationQueue!

    override func viewDidLoad() {
        super.viewDidLoad()

        queue = NSOperationQueue()

        // Rellenar el array de datos y crear las operaciones
        for var i = 0 ; i < DATA_SIZE ; i++ {
            let startValue = i * 200000
            data.insert(startValue, atIndex: i)

            // Crea una operacion que busque el siguiente primo despues
            // del valor dado y lo ponga en la fila dada.
            createOperationForRow(i, startValue: startValue)
        }
    }
}

```

```

private func createOperationForRow(row: Int, startValue v: Int) {

    // Creo un objeto Operacion
    var ope = PrimeOperation(startValue: v)

    // Closure que se ejecuta cuando termina la operación
    ope.completionBlock = {

        let ip = NSIndexPath(forRow: row, inSection: 0)

        // Actualizar el GUI en el Main Thread
        NSOperationQueue.mainQueue().addOperationWithBlock({
            self.data[row] = ope.value

            self.tableView.reloadRowsAtIndexPaths([ip],
                withRowAnimation: .None)
        })

        // Creo una nueva operacion
        self.createOperationForRow(row, startValue: ope.value)
    }

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

```

```

override func numberOfSectionsInTableView(tableView: UITableView)
                                                    -> Int {
    return 1
}

override func tableView(tableView: UITableView,
                        numberOfRowsInSection section: Int) -> Int {
    return DATA_SIZE
}

override func tableView(tableView: UITableView,
                        cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {

    let cell = tableView.dequeueReusableCellWithIdentifier("Prime Cell",
                                                    forIndexPath: indexPath) as UITableViewCell

    cell.textLabel?.text = "\(data[indexPath.row])"

    return cell
}

```

# NSBlockOperation

- Es una **subclase** de **NSOperation** ya hecha.
- La operación ejecutará el block o los blocks dados.

convenience **init(block** block: () -> Void)

func **addExecutionBlock**(\_ block: () -> Void)

```

private let DATA_SIZE = 100

class PrimosTableViewController: UITableViewController {

    // Array donde guardo el ultimo primo encontrado en cada fila.
    var data = [Int]()

    // Cola de las operaciones
    var queue: NSOperationQueue!

    override func viewDidLoad() {
        super.viewDidLoad()

        queue = NSOperationQueue()

        // Rellenar el array de datos y crear las operaciones
        for var i = 0 ; i < DATA_SIZE ; i++ {
            let startValue = i * 200000
            data.insert(startValue, atIndex: i)

            // Crea una operacion que busque el siguiente primo despues
            // del valor dado y lo ponga en la fila dada.
            createOperationForRow(i, startValue: startValue)
        }
    }
}

```

```

private func createOperationForRow(row: Int, var startValue v: Int) {

    // Creo un objeto Operacion
    let ope = NSBlockOperation()

    // Creo el bloque a ejecutar
    let block: () -> Void = {
        if ope.cancelled { return }
        while !self.isPrime(++v) {}
        if ope.cancelled { return }

        // Actualizar el GUI en el Main Thread
        NSOperationQueue.mainQueue().addOperationWithBlock({
            self.data[row] = v
            let ip = NSIndexPath(forRow: row, inSection: 0)
            self.tableView.reloadRowsAtIndexPaths([ip],
                withRowAnimation: .None)

        })
        NSThread.sleepForTimeInterval(0.1) // Me duermo un rato

        // Creo una nueva operacion
        self.createOperationForRow(row, startValue: v)
    }

    // Añado el bloque a ejecutar a la operacion
    ope.addExecutionBlock(block)
    // Meter la operacion en el Operation Queue
    queue.addOperation(ope)
}

```

```

override func numberOfSectionsInTableView(tableView: UITableView)
                                                    -> Int {
    return 1
}

override func tableView(tableView: UITableView,
                        numberOfRowsInSection section: Int) -> Int {
    return DATA_SIZE
}

override func tableView(tableView: UITableView,
                        cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {

    let cell = tableView.dequeueReusableCellWithIdentifier("Prime Cell",
                                                    forIndexPath: indexPath) as UITableViewCell

    cell.textLabel?.text = "\(data[indexPath.row])"

    return cell
}

```

# Completion Block

- A un objeto `NSOperation` 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.

# Dependencias entre Operaciones

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

```
func addDependency(_ operation: NSOperation)
```

- Consultar dependencias:

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

- Eliminar dependencias:

```
func removeDependency(_ operation: NSOperation)
```

# 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: NSOperationQueuePriority
```

- Valores:

```
NSOperationQueuePriority.VeryLow
```

```
NSOperationQueuePriority.Low
```

```
NSOperationQueuePriority.Normal
```

```
NSOperationQueuePriority.High
```

```
NSOperationQueuePriority.VeryHigh
```

# Cancelar una Operación

- La ejecución de una operación puede cancelarse:

```
func cancel()
```

- Cancelar una operación sólo hace que la propiedad **cancelled** 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.

# Propiedad de las Operaciones

- **var ready: Bool { get }**
  - **true** si la operación está lista para ejecutarse, es decir, no hay dependencias por las que esperar.
- **var cancelled: Bool { get }**
  - **true** si la operación fue cancelada.
- **var executing: Bool { get }**
  - **true** si la operación es está ejecutando (ha empezado y no ha acabado su ejecución).
- **var finished: Bool { get }**
  - **true** si terminó la ejecución de la operación.
- **var concurrent: Bool { get }**
  - Sobreescibir para devolver **true** si la operación creará su propio thread para ejecutarse.
  - Hay que sobreescibir también otros métodos: **start()**, **executing**, **finished**, **asynchronous**.
- **var asynchronous: Bool { get }**
  - Indica si la operación se ejecuta asíncronamente.

# 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.

# NSOperationQueue

- Son los objetos que gestionan la ejecución de las operaciones.

- Para crear una cola:

```
let queue = NSOperationQueue()
```

- Para añadir una operación a la cola:

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

- La operación se ejecutará cuanto exista un thread disponible y la operación esté lista (**ready** 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 suspended: Bool
```

# NSOperationQueue: Más Métodos

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

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

func addOperationWithBlock(_ block: () -> Void)

func waitUntilAllOperationsAreFinished()

func cancelAllOperations()

class func currentQueue() -> NSOperationQueue?
class func mainQueue() -> NSOperationQueue

. . .
```

# Grand Central Dispatch

# GCD

- GCD es el nombre de libdispatch: librería (API C) para manejar el multithreading.
  - Nos oculta los detalles de la multiprogramación.
  - No tenemos que preocuparnos de los recursos disponibles. GCD se encarga de gestionarlos.
  - Se programa muy fácilmente.
  - Desde iOS 7 los objetos del GCD son objetos Objective-C.
- GCD nos proporciona colas a las que enviaremos tareas.
  - Una tarea es un objeto block (closure).
    - También pueden usarse punteros a funciones.
  - En algún momento GCD asignará un thread libre a esa tarea para que se ejecute.
    - GCD decide cuántos threads crea, cuándo un thread se encarga de ejecutar una tarea, y durante cuánto tiempo.
  - El encolado es **thread-safe**: Se puede acceder a las colas desde distintos threads de forma segura.

- 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.

# Cola: main

- Ejecuta tareas en **serie** en el **Main Thread**.
  - Esta cola la crea el sistema automáticamente.
- Para obtener esta cola:

```
func dispatch_get_main_queue() -> dispatch_queue_t!
```

# Cola: Global

- Es una cola global ya creada en el sistema que se usa para ejecutar tareas **concurrentemente** según la calidad de servicio (prioridad) indicada
  - El comienzo de la ejecución de las tareas es FIFO.
- GCD crea automáticamente **varias** colas con distintas propiedades.
- Para obtener estas colas:

```
func dispatch_get_global_queue(_ identifier: Int,  
                                _ flags: UInt) -> dispatch_queue_t!
```

- Valores para **identifier** (indican prioridad de las tareas):

```
QOS_CLASS_USER_INTERACTIVE  
QOS_CLASS_USER_INITIATED  
QOS_CLASS_UTILITY  
QOS_CLASS_BACKGROUND
```

- **flags**: es 0 (reservado para su uso en el futuro).

# Colas: serie o concurrentes

- Pueden crearse en cualquier momento nuevas colas serie o concurrentes.

- Las aplicaciones deben crearlas explícitamente.

```
func dispatch_queue_create(_ label: UnsafePointer<Int8>,  
                           attr: dispatch_queue_attr_t!) -> dispatch_queue_t!
```

- Parámetros:

- **label**: es un string (de C) que identifica la cola para ayuda en la depuración.

- Valores de **attr**:

```
DISPATCH_QUEUE_SERIAL      // para crear una cola serie.  
DISPATCH_QUEUE_CONCURRENT // para crear una cola concurrente.
```

- Si la cola creada es serie, ejecuta 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: sacar del main thread tareas que pueden bloquearse, realizar tareas muy largas en otro thread, proteger zonas críticas, ...

# Encolar Tareas

- Enviar una tarea a una cola:

```
func dispatch_async(_ queue: dispatch_queue_t!,  
                    _ block: dispatch_block_t!)
```

```
func dispatch_sync(_ queue: dispatch_queue_t!,  
                   _ block: dispatch_block_t!)
```

- donde **dispatch\_block\_t** es un bloque (closure) de tipo `() -> Void`.
- **dispatch\_async** no es bloqueante.
- **dispatch\_sync** es bloqueante. Se espera hasta que el bloque ha terminado

# Ejemplo: Primos-Cola Global

```
private let DATA_SIZE = 100

class PrimosTableViewController: UITableViewController {

    // Array donde guardo el ultimo primo encontrado en cada fila.
    var data = [Int]()

    // Cola Global
    var queue: dispatch_queue_t!

    override func viewDidLoad() {
        super.viewDidLoad()

        queue = dispatch_get_global_queue(QOS_CLASS_BACKGROUND, 0)

        // Rellenar el array de datos y crear las operaciones
        for var i = 0 ; i < DATA_SIZE ; i++ {
            let startValue = i * 200000
            data.insert(startValue, atIndex: i)

            // Crea una operacion que busque el siguiente primo despues
            // del valor dado y lo ponga en la fila dada.
            createOperationForRow(i, startValue: startValue)
        }
    }
}
```

```

private func createOperationForRow(row: Int, var startValue v: Int) {

    dispatch_async(queue, {

        while !self.isPrime(++v) {}

        // Actualizar el GUI en el Main Thread
        dispatch_async(dispatch_get_main_queue(), {
            let ip = NSIndexPath(forRow: row, inSection: 0)
            self.data[row] = v
            self.tableView.reloadRowsAtIndexPaths([ip],
                                                    withRowAnimation: .None)

            // Pasados 0.1 segundo creo otra tarea
            let t = dispatch_time(DISPATCH_TIME_NOW, Int64(NSEC_PER_SEC/10))
            dispatch_after(t, self.queue, {
                self.createOperationForRow(row, startValue: v)
            })
        })
    })
}

```

# Ejemplo: Bajarse una Foto

```
let imgUrl = "http://www.etsit.upm.es/images/portada/logoetsitupm.png"
// Escapar caracteres conflictivos de la URL
let escapeUrl = imgUrl.stringByAddingPercentEscapesUsingEncoding(NSUTF8StringEncoding)!
// Construir un NSURL
let url = NSURL(string: escapeUrl)!

// Envio la tarea a un thread
let queue = dispatch_queue_create("Download", DISPATCH_QUEUE_SERIAL)
dispatch_async(queue, {

    // Bajar los datos del sitio Web
    if let data = NSData(contentsOfURL: url) {
        // Construir una imagen con los datos bajados
        if let img = UIImage(data: data) {

            // El GUI se actualiza en el Main Thread
            dispatch_async(dispatch_get_main_queue(), {
                self.imageView.image = img
            })

        } else { println("Error construyendo la imagen") }
    } else { println("Error descargando") }
})
```

# 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.

```
// Cola de acceso a la zona critica
```

```
var queue = dispatch_queue_create("Cola de acceso",  
                                DISPATCH_QUEUE_SERIAL)
```

```
func meter(n: Int) {
```

```
    dispatch_async(queue, {
```

Así es Thread Safe

Crear cola serie para  
serializar tareas

```
        self.total += n;
```

Zona crítica

```
        dispatch_async(dispatch_get_main_queue(), {  
            self.totalLabel?.text = "\(self.total)"
```

Actualizo el GUI en  
el main thread

```
        })
```

```
    })
```

```
}
```

```
func sacar(n: Int) {
```

```
    dispatch_async(queue, {
```

Así es Thread Safe

```
        self.total -= n;
```

Zona crítica

```
        dispatch_async(dispatch_get_main_queue(), {  
            self.totalLabel?.text = "\(self.total)"
```

Actualizo el GUI en  
el main thread

```
        })
```

```
    })
```

```
}
```

# Retrasar una Tarea

- **dispatch\_after** espera hasta el momento especificado, y entonces envía asíncronamente una tarea a la cola especificada.

```
// Disparar un segue dentro de 5 segundos.  
let t = dispatch_time(DISPATCH_TIME_NOW,  
                      Int64(5 * NSEC_PER_SEC))  
dispatch_after(t, dispatch_get_main_queue(), {  
    self.performSegueWithIdentifier("ID", sender: self)  
})
```

# Ejecutar solo una vez

- **dispatch\_once** ejecuta una tarea una sola vez durante toda la vida de la app.
  - Esta función no hace nada si se intenta ejecutar el bloque una segunda vez.
- La ejecución es síncrona: se espera hasta que la función ha terminado.
  - Si varios thread llaman simultáneamente a esta función para realizar una tarea, las llamadas esperan hasta que la tarea se ha sido realizada por uno de ellos.
- Esta función toma como primer parámetro una referencia a un objeto **dispatch\_once\_t** para identificar la tarea y controlar que solo se ejecute una vez.
  - El objeto `dispatch_once_t` debe crearse y conservarse en un ámbito global o estático.

```

class Singleton {

    // Propiedad calculada que devuelve la unica instancia
    class var shared : Singleton {

        // Tipo anidado: una estructura para poder tener valores
        // de clase/estaticos
        struct Statics {
            static var pred : dispatch_once_t = 0
            static var instance : Singleton?
        }

        /*
        // Esto no es Thread-Safe
        if Statics.instance == nil {
            Statics.instance = Singleton()
        }
        */

        dispatch_once(&Statics.pred) {
            Statics.instance = Singleton()
        }

        return Statics.instance!
    }
}

```

# Barreras

- **dispatch\_barrier\_sync** y **dispatch\_barrier\_async** crean un punto de sincronización en una cola concurrente.
  - La cola no ejecuta la tarea añadida con estas funciones 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.

```
var q = dispatch_queue_create("queue", DISPATCH_QUEUE_CONCURRENT)

dispatch_async(q, { self.producir() })
dispatch_async(q, { self.producir() })
dispatch_async(q, { self.producir() })

dispatch_barrier_async(q, { self.barrera() })

dispatch_async(q, { self.consumir() })
dispatch_async(q, { self.consumir() })
dispatch_async(q, { self.consumir() })

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

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

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

// PPPBCCC
```

# API más ...

- `dispatch_resume`
- `dispatch_suspend`
- `dispatch_time`
- `dispatch_set_context`
- `dispatch_get_context`
- `dispatch_group_create`
- `dispatch_group_enter`
- `dispatch_group_leave`
- `dispatch_group_async`
- `dispatch_group_wait`
- `dispatch_semaphore_create`
- `dispatch_semaphore_signal`
- `dispatch_semaphore_wait`
- `dispatch_source_*`
- etc...

¿Qué uso?  
¿NSOperation o GCD?

# ¿Qué Elijo?

- NSOperation y NSOperation 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.

