Tecnologías para el desarrollo de aplicaciones en dispositivos móviles

Sesión 4: Comunicación entre objetos

```objc
//Supongamos que este es nuestro modelo,
//que necesita comunicarse con el controlador
@interface MiModelo
  //Podríamos tener una referencia al controlador
  @property MiControlador *controlador;
  @property int cotizacion;
  //Este método llamaría al controlador para "avisarle"
  //de un cambio en la cotización
  - (void) cambioEnLaCotizacion();
@end
```

Problema: el modelo está fuertemente acoplado al controlador. Necesitamos mecanismos alternativos que nos permitan **comunicar dos objetos sin introducir tanto acoplamiento**

Puntos a tratar

- Target-action
- Delegates y protocols
- Key-Value Observing
- Notificaciones

Target-action

- Ya hemos visto que es el mecanismo típico para comunicar eventos de la vista al controlador
- Repasando: cuando se produzca un evento sobre un objeto (vista) queremos avisar a otro objeto (controlador) llamando a un método determinado (el *action*)

Target-action por código

Hasta ahora hemos enlazado vista y controlador gráficamente con Xcode pero también se puede hacer por código

```
[unBoton addTarget:self action:@selector(botonPulsado)
         forControlEvents:UIControlEventTouchUpInside];
...
(void)botonPulsado {
     NSLog(@"Pulsado!");
}
```

Target-action por código (2)

También le podemos pasar al *action* la fuente del evento, o la fuente y el propio evento

```
[unBoton addTarget:self action:@selector(botonPulsado:paraEvento:)
         forControlEvents:UIControlEventTouchUpInside];
...
(void)botonPulsado:(id) sender paraEvento:(UIEvent *)evento{
     NSLog(@"Objeto: %@", [sender debugDescription]);
     NSLog(@"Evento: %@", [evento debugDescription]);
}
```

Limitación básica de target-action

No podemos usar una signatura arbitraria para el *action*, y por tanto **no podemos usar este mecanismo cuando necesitemos pasar información "personalizada"**

Queremos comunicar a otro objeto que suceden ciertos eventos, y además queremos pasar información arbitraria en cada evento.

- Posible solución: **especificar *formalmente* qué signatura tienen los métodos que informarán que se ha producido cada evento** (establecer un *protocolo* de comunicación).
- Debemos estar seguros de que el objeto receptor implementa dichos métodos si no queremos un error en tiempo de ejecución (algo como los `interface` de Java)

- **Protocol**: especifica los métodos que debe implementar un objeto que lo cumpla

```
//Esto se define en su propio .h: UACalificable.h
@protocol UACalificable
- (void)setNota: (CGFloat) nota;
- (CGFloat)nota;
@end
```

- Para especificar que una clase cumple el protocolo:

```
@interface UAAsignatura : NSObject<UACalificable>
- (void)setNota: (CGFloat) nota;
- (CGFloat)nota;
@end
```

Clases "genéricas"

- Podemos definir una variable de la que no sabemos todavía el tipo concreto (`id``) pero sí sabemos que implementa un determinado *protocol*:

```
id<UACalificable> algoCalificable;
[algoCalificable setNota:10.0];
```

Los métodos de un protocol no son estrictamente obligatorios

- Si no implementamos algún método el compilador generará ***warnings*, no errores**. podemos marcar métodos como opcionales para eliminar los *warnings*

```
@protocol MiProtocol
//Si no se pone nada, se considera "obligatorio"
- (void)metodoObligatorio;
@optional
- (void)metodoOpcional;
- (void)otroMetodoOpcional;
//Si aquí no ponemos nada seguiría en "modo @optional". Lo cambiamos
@required
- (void)otroMetodoObligatorio;
@end
```

Nombrar un protocol

- Cuestión de estilo: se recomienda que los nombres de los protocolos sean *adjetivos* (como en el ejemplo anterior) o *gerundios* (por ejemplo, Cocoa tiene un protocolo `NSCopying`, que indica que se puede hacer una copia del objeto llamando a `copyWithZone:`)

Delegates

- Una vez definido un *protocol* y un objeto conforme con él, ya sabemos que hay una serie de métodos para comunicarnos con el objeto.
- Al objeto receptor lo llamaremos **delegate**, ya que **en él delegamos la responsabilidad de procesar los eventos**.
- El *delegate* es un patrón de diseño ampliamente usado en la plataforma. Aparece en iOS en múltiples ocasiones (por ejemplo el `AppDelegate`).

Delegates y acoplamiento

Con el *delegate* no eliminamos el acoplamiento pero pasamos a **depender de un interfaz** (protocolo en el *argot* iOS) en vez de una clase concreta

> *"Program to an 'interface', not an 'implementation'.", un principio básico del diseño OO*

Un punto engorroso de los *delegates* es que hay que llamar **explícitamente** a los métodos del protocolo para avisar al objeto receptor. ¿Podríamos hacer **que el aviso fuera automático** cuando pasara “algo interesante”?

Key-Value Observing

- Nos permite **avisar automáticamente al receptor** cuando cambie una propiedad del emisor. El receptor es el que se "suscribe" a los cambios
- Para que se pueda usar KVO, es necesario que las propiedades a observar sean "KVC-compliant"

Suscribirse a los cambios

```
//El objeto "receptor" desea observar cambios en la propiedad "nombre" del "emisor"
[emisor addObserver:receptor
  forKeyPath:@"nombre"
  //Luego veremos para qué se usan estos parámetros
  //Por el momento los dejamos a 0 y nil, respectivamente
  options:0
  context: nil];
```

"Escuchar" los cambios

- Cuando hay un cambio se llama automáticamente al método **`observeValueForKeyPath`**

```
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
         change:(NSDictionary *)change context:(void *)context {
   NSLog(@"La propiedad %@ cambia a %@", keyPath, [object valueForKey:keyPath];
 }
```

- KVO es **síncrono**

```
emisor.nombre = @"Pepe";
//Antes de que se ejecute el NSLog ya se habrá llamado al observeValueForKeyPath
NSLog(@"Ya se ha llamado al setter");
```

Opciones

- Podemos indicar que nos pase el nuevo valor, o que nos pase también el antiguo, o algo más sofisticado, como que nos avise antes y después del cambio,... Para esto se usa el parámetro **`options`**, que es una máscara de bits de opciones. Por ejemplo:

```
[emisor addObserver:receptor
  forKeyPath:@"nombre"
  //queremos tanto el nuevo valor como el antiguo
  options: (NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld)
  context: nil];
```

opciones (2)

- en el **`observeValueForKeyPath`**, el parámetro **`change`** es un diccionario que nos da acceso a las opciones

```
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
         change:(NSDictionary *)change context:(void *)context {
   NSLog(@"Cambia la propiedad %@ de %@ a %@", keyPath,
       change[NSKeyValueChangeOldKey],
       change[NSKeyValueChangeNewKey]);
}
```

Contexto

El parámetro `context` puede usarse si necesitamos observar una misma propiedad por varios motivos distintos. Llamaríamos al `addObserver` varias veces con distintos valores de `context`, que se recibe tal cual cuando se llama al `observeValueForKeyPath`

Dejar de observar una propiedad

Si un objeto se libera y está usando KVO, el programa abortará al intentar enviar el cambio en la propiedad.

```
- (void) dealloc {
    NSLog(@"Des-registrando observador in extremis...");
    [emisor removeObserver:receptor forKeyPath:@"nombre"];
  }
```

KVO y el acoplamiento emisor/receptor

Con KVO cuando nos suscribimos a un cambio en una propiedad **debemos especificar el objeto concreto** que estamos escuchando

```
//Estamos atados al objeto concreto llamado "receptor"
[emisor addObserver:receptor
  forKeyPath:@"nombre"
  options: 0;
  context: nil];
```

Notificaciones

- Cada aplicación iOS tiene un **centro de notificaciones**, es lo que en otras plataformas se llama un sistema *publish/subscribe* o una *cola de mensajes*
- Desacoplamiento entre emisor/receptor
  - El receptor se suscribe a un tipo de notificación, no a un emisor ni propiedad en concreto
  - El emisor envía una notificación al centro de notificaciones
  - El centro de notificaciones es el que remite la notificación a los suscritos

API de notificaciones: emisores

- El centro de notificaciones es un *singleton* al que podemos acceder con

```
[NSNotificationCenter defaultCenter]
```

- Enviar una notificación

```
//Las notificaciones tienen un nombre y un contenido (un diccionario)
NSDictionary *contenido = @{@"empresa": @"AAPL", @"valor": @95.55};
[[NSNotificationCenter defaultCenter] postNotificationName:@"cotizacion"
    //object es el objeto que figura como remitente de la notificación
    object:self
    userInfo:contenido];
```

API de notificaciones: receptores

- suscribirse a una notificación

```
[[NSNotificationCenter defaultCenter] addObserver:self
    //método que se ejecutará cuando se reciba la notificación
    selector: @selector(nuevaCotizacion:)
    //nombre de la notificación que nos interesa
    name:@"cotizacion"
    //Si estamos interesados en notificaciones de un objeto concreto, lo pondríamos a
    object:nil]
```

- recibir la notificación (en el `selector`)

```
- (void) nuevaCotizacion: (NSNotification *) notificacion {
      NSLog(@"Recibida notificación: %@, con contenido: %@", notificacion.name, notifi
  }
```

Crear el "manejador" de la notificación con un bloque

- También podemos implementar el manejador de la notificación como un bloque, nos da una sintaxis más concisa

```
[[NSNotificationCenter defaultCenter] addObserverForName:@"cotizacion"
     object:nil
     //Podemos recibir la notificación en una cola de operaciones
     queue:nil
     //Bloque a ejecutar cuando se reciba la notificación
     usingBlock:^(NSNotification *notificacion) {
         NSLog(@"Notificación %@, payload: %@",
               notificacion.name, notificacion.userInfo);
}];
```

Eliminar una suscripción

- Si un objeto desaparece de la memoria y está suscrito a una notificación, cuando se intente el envío el programa abortará. Hay que eliminar la suscripción antes.

```
//Cuando se libera un objeto se llama a su "dealloc"
- (void) dealloc {
    NSLog(@"Eliminando todas las suscripciones in extremis...");
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```

Eliminar una suscripción con "manejador" de bloque

```
//Con este "formato" el observador es un objeto creado dinámicamente por Cocoa
id observador = [[NSNotificationCenter defaultCenter]
                 addObserverForName:@"cotizacion"
                 object:nil
                 queue:nil
                 usingBlock:^(NSNotification *notificacion) {
                    ...
                }];

...

- (void) dealloc {
    NSLog(@"Eliminando todas las suscripciones in extremis...");
    [[NSNotificationCenter defaultCenter] removeObserver:observador];
}
```

Notificaciones del sistema

- Muchos objetos de Cocoa pueden enviar notificaciones. Por ejemplo al pulsar el botón de inicio se genera `UIApplicationDidEnterBackgroundNotification`.

```
[[NSNotificationCenter defaultCenter]
   addObserverForName:UIApplicationDidEnterBackgroundNotification
   object:nil queue:nil
   usingBlock:^(NSNotification *notificacion){
            NSLog(@"Me han dicho que entramos en background");
   }];
```

- Por eficiencia, no todas las notificaciones del sistema se envían por defecto, muchas hay que activarlas