Vistas

Índice

1. Patrón Modelo-Vista-Controlador (MVC)

La API de Cocoa está claramente basada en este patrón, por lo que resulta apropiado hacer que nuestras aplicaciones también lo adopten. Este patrón se basa en dividir nuestras funcionalidades en tres grandes bloques: el modelo, la vista, y el controlador.

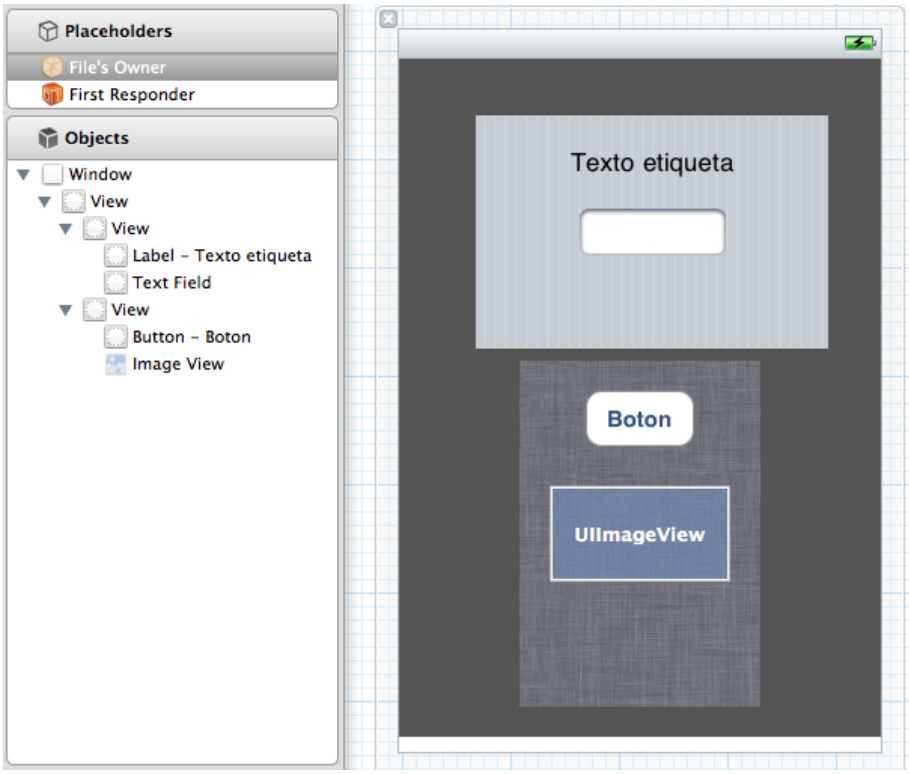
- **Modelo**: Aquí reside toda la lógica de negocio y de acceso a datos de nuestra aplicación. En esta parte encontramos los objetos del dominio de nuestra aplicación, como por ejemplo `UAAsignatura`, y todos aquellos que nos permitan acceder y manipular estos datos.
- **Vista**: En la vista tenemos los componentes que se muestran en la interfaz. Estos componentes serán las clases del *framework UIKit* cuyo nombre tiene el sufijo `View`, y que derivan de la clase `UIView`. Normalmente crearemos estos objetos de forma visual mediante la herramienta Interface Builder incluida en Xcode, aunque también se podrán crear de forma programática. A parte de los tipos de vistas que podemos encontrar en la API de Cocoa Touch, podremos crear vistas propias definiendo nuevas subclases de `UIView`.
- **Controlador**: El controlador es el que se encarga de coordinar modelo y vista. Aquí es donde definimos realmente el comportamiendo de nuestra aplicación. Si bien modelo y vista deben diseñarse de forma que sean lo más reutilizables posible, el controlador está fuertemente vinculado a nuestra aplicación concreta.

Por el momento hemos comenzado viendo cómo implementar las clases del dominio de nuestra aplicación, que formarán parte del modelo. Ahora vamos a centrarnos en la creación de la vista.

2. Jerarquía de vistas, ventanas, y controladores

Como hemos comentado, las clases de la vista derivan todas de `UIView`, y en ellas se especifica cómo debe mostrarse dicha vista en la pantalla. Una vista define una región rectangular (*frame*) que se mostrará en la pantalla del dispositivo, y en el código de su clase se indica cómo debe dibujarse el contenido en dicha región. Además de encargarse de mostrar el contenido, la vista se encarga también de interactuar con el usuario, es lo que se conoce como un *responder*. La clase `UIView` deriva de `UIResponder`, es decir, la vista también se encarga de responder ante los eventos (contactos en la pantalla tácil) que el usuario realice en su región. Podemos encontrar una serie de vistas ya predefinidas en la API de Cocoa Touch que podremos utilizar en nuestras aplicaciones (bien añadiéndolas mediante *Interface Builder* o bien añadiéndolas de forma programática), como son por ejemplo `UIImageView`, `UITextView`, `UILabel`, `UIButton` o `UIWebView`, entre otras muchas. Se suele tener como convención poner a las vistas el sufijo `View`, aunque vemos que no siempre es así. También podremos crearnos nuestra propia subclase de `UIView` para definir una vista personalizada, en la que nosotros decidamos cómo se dibuja el contenido de su región y cómo responde ante los eventos del usuario.

Las vistas se organizan de forma jerárquica. Una vista puede contener subvistas, de forma que cualquier modificación que se haga sobre una vista afectará también a todas sus subvistas. En la raíz de la jerarquía tendremos un tipo especial de vista conocida como ventana, que se define en la clase `UIWindow`. Normalmente nuestra aplicación tendrá sólo una ventana, que ocupará toda la pantalla (podremos tener más de una ventana si por ejemplo nuestra aplicación soporta tener una pantalla externa). En la aplicación veremos aquellas vistas que sean subvistas de la ventana principal.



Jerarquía de vistas

Nota

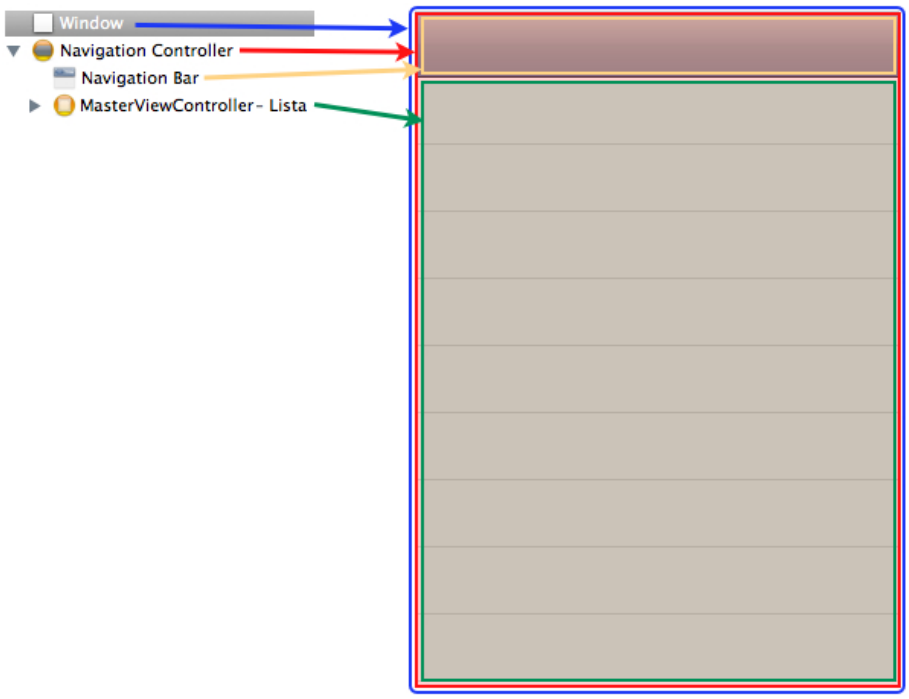
Una subvista no está restringida a la región que ocupa su vista padre, sino que puede mostrarse fuera de ella. Por lo tanto, no se puede determinar la jerarquía de vistas a partir del contenido que veamos en la pantalla. Sin embargo, si que afecta al orden en el que se dibujan (una vista se dibuja antes que sus subvistas, por lo que las subvistas taparán el contenido de la vista padre).

Normalmente en la ventana principal tendremos como hija una vista que abarcará toda la ventana, y dentro de ella encontraremos los diferentes objetos de la interfaz. Habitualmente no tendremos directamente un objeto de tipo `UIView`, sino que lo que encontraremos dentro de la ventana es un controlador. Los controladores se encargan de gestionar el ciclo de vida de las vistas y los cambios que se puedan producir en ellas (como por ejemplo ajustarlas al cambiar la orientación del dispositivo). No deberemos

crear un controlador por cada vista individual que se muestre en la pantalla, sino por cada pantalla de la aplicación (que se compondrá de una jerarquía de vistas). Lo que podemos encontrar habitualmente es un controlador que hace de contenedor de otros controladores (por ejemplo las aplicaciones basadas en navegación o en pestañas se componen de varias pantallas, tendremos un controlador general que se encarga de gestionar la pila de navegación o las pestañas, y a parte un controlador específico para cada pantalla).

Los controladores derivan de la clase `UIViewController`, y tienen una propiedad `view` que corresponde a la vista que mostrarán. Cuando añadamos un controlador a la ventana, se mostrará en ella la vista referenciada por dicha propiedad. Modificando su valor cambiaremos la vista que muestra el controlador.

Si hemos utilizado la plantilla *>Master-Detail Application* a la hora de crear nuestro proyecto, veremos que nos ha creado dos controladores para la pantalla principal: *Navigation controller*, que se crea de forma programática en nuestra `UIApplicationDelegate`, está implementado en la API de Cocoa Touch y es el que nos permite tener aplicaciones con una barra de navegación y una pila de pantallas; y `MasterViewController` que corresponde a la pantalla principal en la navegación y se implementa dentro de nuestra aplicación. También tenemos un tercer controlador (`DetailViewController`) que corresponde a la pantalla secundaria de la navegación en la que entraremos cuando se pulse un item de la pantalla principal.

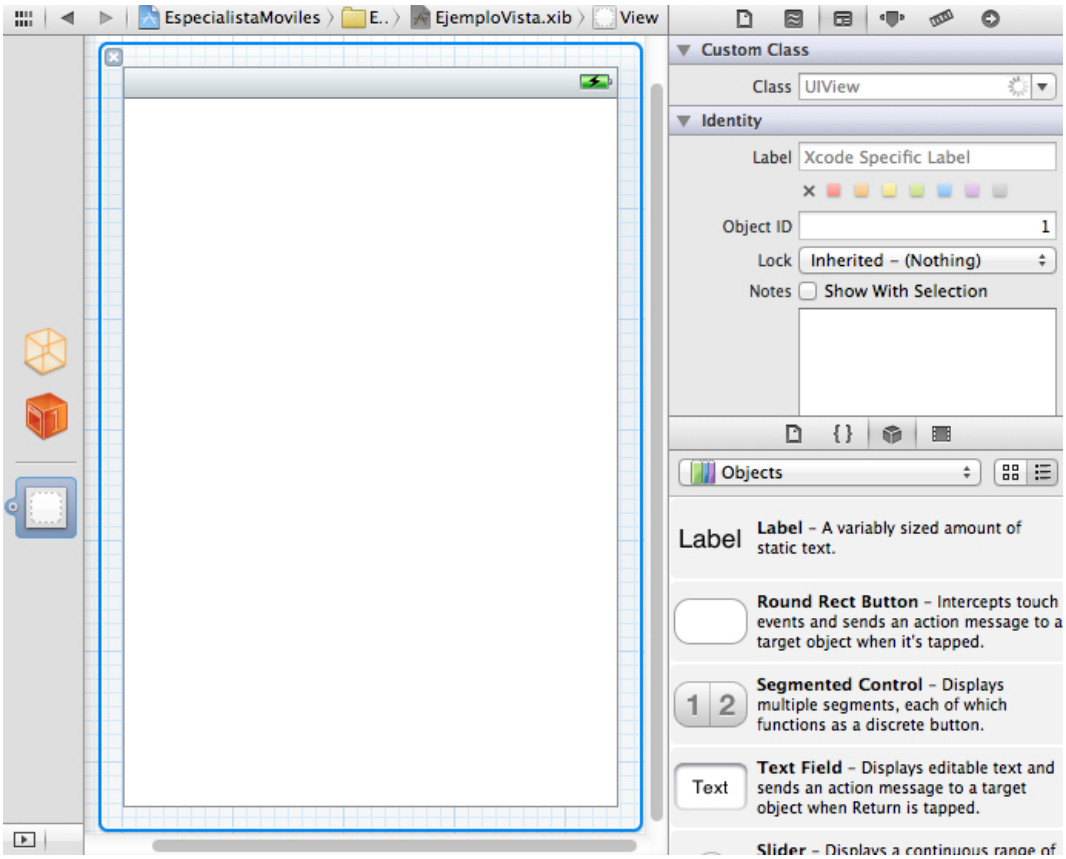


Controlador de navegación

3. Creación de vistas con Interface Builder

Lo más habitual será crear la interfaz de nuestra aplicación de forma visual con Interface Builder. En versiones anteriores de Xcode, Interface Builder se proporcionaba como una herramienta independiente, pero a partir de Xcode 4 está totalmente integrada en el entorno. Los ficheros en los que se define la interfaz se denominan ficheros NIB (o XIB). En las plantillas que hemos creado anteriormente los hemos visto como fichero `.xib`, aunque si construimos la aplicación y miramos el contenido del *bundle* generado, veremos que se empaqueten como `.nib`, ya que esta última es la versión compilada del fichero `.xib` anterior.

El fichero NIB simplemente es un contenedor de objetos de la interfaz, que podrá cargarse en el código de nuestra aplicación y así tener disponibles dichos objetos para mostrar. Si pulsamos sobre un fichero `.xib` en el navegador de Xcode, veremos que en el editor se abre Interface Builder. Vamos a estudiar los elementos que encontramos en dicha herramienta.



Aspecto de Interface Builder

- *Dock de objetos*: A la izquierda del editor veremos los iconos de los objetos que se definen (o se referencian) en el fichero NIB. Seleccionando uno de ellos podremos visualizarlo y modificar sus propiedades.

- *Editor visual*: En la parte central se presentan de forma visual los componentes definidos en el fichero NIB.
- *Inspector y librería*: A la izquierda tenemos el inspector, donde podemos ver y modificar las propiedades del objeto seleccionado en un momento dado. Bajo el inspector tenemos la librería de objetos que podemos añadir al NIB, simplemente arrastrándolos sobre el *dock* o sobre el editor.

Podemos crear un nuevo NIB en nuestro proyecto seleccionando *File* > *New* > *New File ...* > *iOS* > *User Interface* y dentro de esta sección seleccionaremos una de las plantillas disponibles. Podemos crear un NIB vacío, o bien crearlo con alguno de los elementos básicos ya añadidos (una vista, una ventana, o un *Application Delegate*). Lo habitual será crear un NIB junto a un controlador, como veremos en la próxima sesión, pero también puede interesarnos crear únicamente el fichero con los objetos de la interfaz.

3.1. Dock de objetos

Dentro del *dock* encontramos dos tipos de objetos, separados por una línea horizontal:

- **Objetos referenciados por el NIB**: Son objetos que ya existen en nuestra aplicación sin que tengan que ser creados por el NIB, simplemente se incluyen como referencia para establecer comunicaciones con ellos, y así poder comunicar la vista con el código de nuestra aplicación.
- **Objetos instanciados por el NIB**: Estos son los objetos que se crearán cuando carguemos el NIB desde nuestro código. Aquí es donde encontramos los componentes de la interfaz que se crearán.

Pinchando sobre cualquier objeto del *dock* podremos ver sus propiedades en el panel de la izquierda. De los objetos referenciados que encontramos en el *dock* podemos destacar:

- *File's Owner*: Referencia el objeto propietario de los objetos del NIB. Cuando se carga un NIB, siempre hay que especificar un objeto como propietario de los contenidos del NIB. El propietario debe existir antes de que se cargue el NIB.
- *First responder*: El *first responder* es el objeto que responderá en primera instancia a los eventos del usuario, si no se ha especificado otra forma de tratar los eventos. Realmente el elemento *First responder* del NIB es un objeto ficticio que nos permite indicar que los eventos los trate el *first responder* por defecto que tengamos en cada caso. Más adelante veremos esto con mayor detenimiento.

Respecto a los objetos que son instanciados por el NIB, podemos distinguir los siguientes:

- **Vistas y ventanas**: Objetos a mostrar en la interfaz, todos ellos subtipos de `UIView`. Se organizan de forma jerárquica. En el *dock* veremos los objetos en la cima de la jerarquía. Si desplegamos el *dock* podremos ver la jerarquía completa de objetos.
- **Objetos delegados**: Objetos pertenecientes a clases definidas por nosotros en los que delegamos para tratar los eventos de los componentes de la interfaz. Definiendo aquí estos objetos, conseguiremos que se instancien de forma automática al cargar el NIB.

- **Controladores**: Objetos que se encargan de gestionar el comportamiento de las vistas. Podemos utilizar controladores ya implementados en Cocoa Touch para implementar estilos de navegación ya predefinidos, o bien controladores propios creados como una subclase de `UIViewController` dentro de nuestra aplicación.

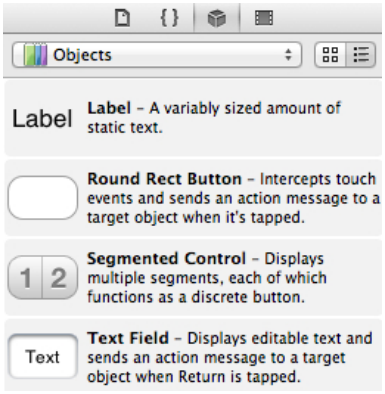
> **Nota**
>
> Podemos expandir o contraer el *dock* con el botón en forma de triángulo que encontramos en su esquina inferior izquierda. En la vista normal veremos sólo los objetos de nivel superior, pero en la vista expandida podremos desplegar toda la jerarquía de objetos.

3.2. Inspector y librería

Cuando tengamos abierto en el editor un fichero NIB en el panel de utilidades (lateral derecho) veremos algunas pestañas adicionales que no estaban mientras editábamos código fuente. En la parte superior (inspector), a parte de *File Inspector* (nombre del fichero `xib`, tipo, ruta, localización, pertenencia a *targets*, etc) y *Quick Help inspector* (ayuda rápida sobre los objetos seleccionados en el *dock* o en el editor), tenemos tres inspectores adicionales, que nos van a dar información sobre el objeto que tengamos seleccionado en un momento dado:

- *Identity inspector*: Identidad del objeto. Nos permite indicar su clase (atributo *Class*). También podemos asignarle una etiqueta para identificarlo dentro de Xcode.
- *Attributes inspector*: Atributos del objeto seleccionado. Aquí es donde podemos editar los atributos de la vista seleccionada (color de fondo, fuente, estilo, etc).
- *Size inspector*: Información sobre el tamaño y la posición del objeto en pantalla. Nos permite definir estos atributos de forma que se ajusten de forma automática según la disposición de la pantalla.
- *Connections inspector*: Nos permite definir conexiones entre los diferentes objetos del *dock*. Por ejemplo, podremos vincular una vista definida en el NIB con un campo del *File's Owner*, de forma que desde el código de la aplicación podremos tener acceso a dicha vista.

Bajo el inspector encontramos la librería de objetos. Esta librería se puede utilizar también para añadir código, pero cobra mayor importancia cuando trabajemos con el editor visual. De las distintas pestañas de la librería, nos interesará *Object library*. En ella encontraremos los diferentes tipos de objetos que podremos añadir a nuestro NIB (vistas, ventanas, controles, controladores, objetos y objetos externos). Podemos añadir cualquiera de estos objetos simplemente arrastrándolo al *dock* o al editor.



Librería de objetos

3.3. Objeto propietario del NIB

Si creamos una vista en el NIB, nos puede interesar tener acceso a ella desde nuestro código para mostrarla en pantalla o modificar su contenido. El lugar desde donde accederemos a dichas propiedades será habitualmente el objeto que definamos como *File's Owner*, que será un objeto que ya existe en nuestra aplicación (puede ser de cualquier tipo).

Al cargar un NIB, especificaremos como parámetros, además del nombre del NIB a cargar, el objeto que se va a comportar como propietario:

```
[[NSBundle mainBundle] loadNibNamed: @"NombreNib"
                              owner: self
                            options: nil];
```

Nota

Esta no es la forma más habitual de cargar un fichero NIB. Normalmente crearemos un controlador asociado al fichero NIB, y el NIB será cargado automáticamente desde el controlador, como veremos en la siguiente sesión. Sin embargo, por ahora vamos a cargar manualmente el fichero NIB para así entender mejor la forma en la que se realiza la carga.

Podemos observar que estamos cargando el NIB entre los recursos empaquetados con la aplicación (*main bundle*). Para tener acceso a dicho paquete de recursos incluidos con la aplicación tenemos el *singleton* `[NSBundle mainBundle]`.

En el ejemplo hemos puesto como objeto propietario el objeto desde el que estamos realizando la llamada (`self`), pero podríamos especificar cualquier otro. Lo que vemos es que la carga del fichero NIB no nos está devolviendo nada, entonces, ¿cómo podemos acceder a los objetos cargados? Para acceder a ellos necesitaremos vincular dichos objetos con propiedades de la clase propietaria, para que así tras cargarlo podamos acceder a ellos a través de esas propiedades. Esta vinculación se establecerá a través de lo que se conocen como *outlets*.

3.4. Conexiones con outlets

Los *outlets* nos permiten crear conexiones entre distintos objetos definidos o referenciados dentro de un fichero NIB. Podemos por ejemplo vincular distintos objetos instanciados en el NIB con las propiedades del *File's Owner*.

En primer lugar, deberemos asegurarnos de que en el NIB la clase del *File's Owner* está configurada correctamente. Esto lo tendremos en el atributo *Class* del *Identity inspector*. Por ejemplo supongamos que tenemos una clase a la que llamamos `EjemploVista` que será la que asignemos como propietaria al cargar el NIB. En ese caso, en el NIB tendremos que indicar que la identidad del *File's Owner* es de clase `EjemploVista`.

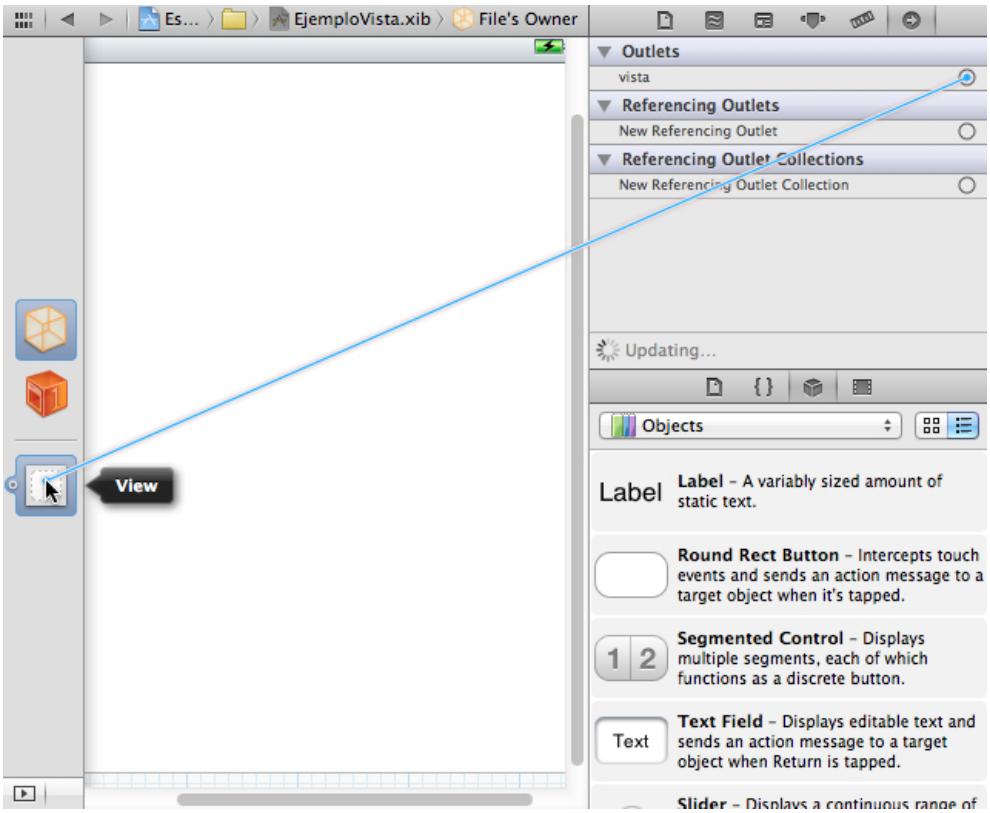
Ahora podemos declarar una serie de *outlets* en dicha clase. Los *outlets* son propiedades de la clase que *Interface Builder* reconocerá y que nos permitirá vincular con los diferentes objetos del NIB. Por ejemplo, si queremos vincular un objeto de tipo `UIView`, deberemos declarar en nuestra clase `EjemploVista` una propiedad de dicho tipo, y añadir el tipo `IBOutlet` a la declaración:

```
@interface EjemploVista : NSObject

@property(nonatomic,retain) IBOutlet UIView *vista;

@end
```

El tipo `IBOutlet` no tiene ningún efecto sobre la compilación, de hecho es eliminado en tiempo de preprocesamiento. Simplemente servirá para indicar a *Interface Builder* que dicha propiedad es un *outlet*. De esta forma, si ahora al editar el NIB seleccionamos *File's Owner* y *Connections inspector*, veremos que nos aparece en la lista el *outlet* `vista`. Podemos pinchar en el círculo junto al *outlet* y arrastrar hacia el objeto con el que queramos vincularlo. Deberemos conectarlo con un objeto de tipo `UIView`, ya que es así como lo hemos declarado.



Conexiones con outlets

Una vez hemos establecido dicha conexión, cuando carguemos el NIB como hemos visto anteriormente, especificando como propietario un objeto de clase `EjemploVista`, la vista (`UIView`) creada por el NIB será asignada a la propiedad `vista` de nuestro objeto `EjemploVista`.

Una consideración importante a la hora de cargar el NIB es cómo se realiza la gestión de la memoria de los objetos instanciados en dicho fichero. Siguiendo la regla de liberar quien reserva, si el NIB instancia, el NIB deberá liberar, por lo que tras cargar el NIB habremos obtenido los objetos de nivel superior con un *autorelease* pendiente que los borrará de la memoria si nadie los retiene (de igual forma que cuando utilizamos un método factoría). Por lo tanto, será responsabilidad del fichero propietario retener dichos objetos si quiere poder utilizarlos más adelante. Esto lo haremos declarando las propiedades asociadas a estos objetos con el modificador `strong`.

Deberemos retener todos los objetos de nivel superior. Sin embargo, los objetos de niveles inferiores de la jerarquía será retenidos por sus padres, por lo que no será necesario retenerlos en el objeto propietario. Con retener los objetos de nivel superior será suficiente.

Para evitar tener referencias cíclicas deberemos seguir la siguiente regla: las referencias a los *outlets* serán fuertes sólo en el *File's Owner*, en cualquier otro objeto estas referencias

siempre deberán ser débiles.

Nota

Hemos visto cómo establecer conexiones entre propiedades de *File's Owner* y objetos definidos en el NIB. Sin embargo, los *outlets* se pueden aplicar a cualquier objeto definido o referenciado en el NIB.

3.5. Tratamiento de eventos mediante acciones

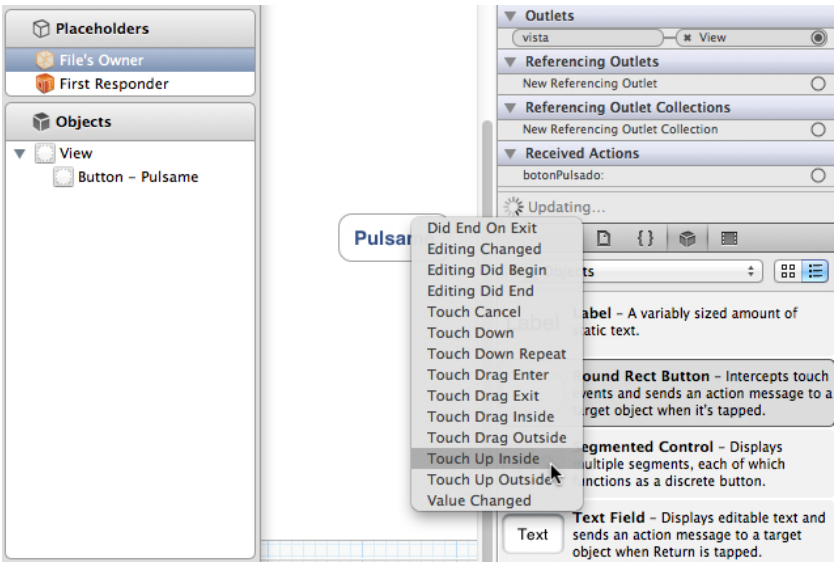
Con los *outlets* podemos vincular campos a objetos de la interfaz. Sin embargo, puede que nos interese que al realizar alguna acción sobre uno de dichos objetos de la interfaz se ejecute un método de nuestro objeto para darle respuesta al evento. Para hacer esto podemos utilizar las acciones. Los métodos declarados con el tipo `IBAction` podrán ser conectados con diferentes acciones de los objetos de la interfaz desde *Interface Bulder*. Por ejemplo, podemos declarar un método de este tipo de la siguiente forma:

```
- (IBAction)botonPulsado: (id)sender;
```

Normalmente estos métodos recibirán un parámetro de tipo `id` en el que recibirán una referencia al objeto que generó el evento (por ejemplo, en el caso de conectar con el evento de pulsación de un botón, el objeto `UIButton` correspondiente al botón que fue pulsado). El tipo `IBAction` es sustituido en tiempo de compilación por `void`, pero servirá para que *Interface Builder* lo reconozca como una acción.

Vamos a suponer que añadimos el método anterior a la clase `EjemploVista` definida previamente. Si ahora añadimos dentro de la vista del fichero NIB un botón (`UIButton`), arrastrando un objeto de este tipo desde la librería de objetos, podremos conectar la acción de pulsar dicho botón con el método anterior. Esto podremos hacerlo de dos formas alternativas:

- Seleccionando *File's Owner*. En su inspector de conexiones veremos la acción `botonPulsado:` que podemos arrastrar sobre el botón para conectarla con él. Al soltar, nos aparecerá una lista de los posibles eventos que genera el botón. Cogeremos *Touch Up Inside*, que es el que se ejecuta cuando soltamos el botón estándo dentro de él.
- Seleccionando el botón. En su inspector de conexiones vemos todos los eventos que puede enviar. Arrastramos desde el evento *Touch Up Inside* hasta *File's Owner* en el *dock*. Al soltar, nos saldrá una lista de las acciones definidas en dicho fichero (métodos que hemos declarado como `IBAction`). Seleccionaremos `botonPulsado:`



Conexión de eventos con acciones

Alternativa

También podemos establecer conexiones con acciones y *outlets* pulsando sobre el botón derecho (o *Ctrl-Click*) sobre los objetos del *dock* o del editor. La forma más rápida de crear estas conexiones consiste en utilizar la vista de asistente, para ver al mismo tiempo el fichero `.xib` y el fichero `.h` de su *File's Owner*. Si hacemos *Ctrl-Click* sobre la vista (en el *dock* o en el editor visual) y arrastramos sobre el código del fichero `.h`, nos permitirá crear automáticamente la declaración del método o de la propiedad correspondiente a la acción o al *outlet* respectivamente, y en el `.m` sintetizará la propiedad del *outlet* o creará el esqueleto de la función de la acción de forma automática.

3.6. Fichero de interfaz principal

Hemos visto que en las propiedades principales del proyecto (sección *Summary*) tenemos la opción de especificar un fichero `.xib` principal que se cargará cuando arranque la aplicación (*Main Interface*). En la plantilla creada esta opción no se utiliza, ya que la vista se está cargando de forma programática:

- En `main.m` a la función `UIApplicationMain` se le pasa como cuarto parámetro el nombre de la clase que hará de delegada de la aplicación. Al cargar la aplicación utilizará esta clase como delegada, y llamará a sus métodos cuando ocurran eventos del ciclo de vida de la aplicación
- En la clase delegada de la aplicación generada, en el evento `application:didFinishLaunchingWithOptions:` se crea la interfaz de forma programática y se muestra en pantalla.

Sin embargo, podríamos utilizar una forma alternativa de arrancar la aplicación:

- Creamos un fichero `.xib` utilizando la plantilla *Application*. Este plantilla contiene

todos los elementos necesarios para poder arrancar la aplicación a partir de dicho fichero de interfaz.
- Especificamos el fichero `.xib` creado en la propiedad *Main Interface* del proyecto.
- En `main.m`, dejamos el cuarto parámetro de `UIApplicationMain` a `nil`, para que no arranque directamente a partir del delegado de la aplicación, sino que coja la interfaz principal que hemos configurado.

El fichero de la interfaz principal que hemos creado con la plantilla *Application* contiene los siguientes elementos:

- El *File's Owner* es de tipo `UIApplication`, en este caso no es una clase de nuestra aplicación, sino que es una clase de la API de Cocoa Touch.
- Uno de los objetos instanciados por el fichero de interfaz es el tipo *App Delegate*. Deberemos hacer que este objeto sea del tipo concreto de nuestro objeto delegado de la aplicación.
- La clase `UIApplication` tiene un *outlet* llamado `delegate` que podremos vincular con el objeto *App Delegate*.
- Se instancia también la ventana principal de la aplicación. Esta ventana se conecta con un *outlet* en el *App Delegate*.

Podemos ver que los *outlets* que referencian los objetos de la interfaz en este caso no estarán definidos en el *File's Owner*, sino en el *App Delegate*, que además se trata de un objeto creado por el NIB (no referenciado). De esta forma, cuando la aplicación arranque `UIApplicationMain` cargará el contenido del NIB especificado, que instanciará al delegado de la aplicación y lo guardará en la propiedad `delegate` de `UIApplication`. A su vez, instanciará también la ventana principal que será guardada en un *outlet* del objeto delegado de la aplicación. El delegado deberá mostrar esta ventana en pantalla al iniciarse la aplicación.

4. Creación de vistas de forma programática

Hemos visto como crear la interfaz con Interface Builder, pero todo lo que se puede hacer con dicha herramienta se puede hacer también de forma programática, ya que lo único que hace Interface Builder es crear objetos de la API de Cocoa Touch (o definidos por nosotros) y establecer algunas de sus propiedades.

4.1. Ventanas

En primer lugar, podemos ver cómo podríamos crear la ventana principal desde nuestro código, instanciando un objeto de tipo `UIWindow` e indicando el tamaño del marco de la ventana:

```
UIWindow *window = [[UIWindow alloc]
            initWithFrame: [[UIScreen mainScreen] bounds]];
```

Lo habitual será inicializar la ventana principal con un marco del tamaño de la pantalla, tal como hemos hecho en el ejemplo anterior.

Para mostrar la ventana en pantalla debemos llamar a su método `makeKeyAndVisible` (esto se hará normalmente en el método `application:didFinishLaunchingWithOptions:` del delegado de `UIApplication`). Esto se hará tanto cuando la ventana ha sido cargada de un NIB como cuando la ventana se ha creado de forma programática.

```
[self.window makeKeyAndVisible];
```

Es posible que necesitemos acceder a esta ventana, por ejemplo para cambiar la vista que se muestra en ella. Hacer esto desde el delegado de `UIApplication` es sencillo, porque normalmente contamos con la propiedad `window` que hace referencia a la pantalla principal, pero acceder a ella desde otros lugares del código podría complicarse. Para facilitar el acceso a dicha ventana principal, podemos obtenerla a través del *singleton* `UIApplication`:

```
UIWindow *window  = [[UIApplication sharedApplication] keyWindow];
```

Si podemos acceder a una vista que se esté mostrando actualmente dentro de la ventana principal, también podemos obtener dicha ventana directamente a través de la vista:

```
UIWindow *window = [vista window];
```

4.2. Vistas

También podemos construir una vista creando un objeto de tipo `UIView`. En el inicializador deberemos proporcionar el marco que ocupará dicha vista. El marco se define mediante el tipo `CGRect` (se trata de una estructura, no de un objeto). Podemos crear un nuevo rectángulo con la macro `CGRectMake`. Por ejemplo, podríamos inicializar una vista de la siguiente forma:

```
UIView *vista = [[UIView alloc]
            initWithFrame:CGRectMake(0,0,100,100)];
```

Como alternativa, podríamos obtener el marco a partir del tamaño de la pantalla o de la aplicación:

```
// Marco sin contar la barra de estado
CGRect marcoVista = [[UIScreen mainScreen] applicationFrame];

// Limites de la pantalla, con barra de estado incluida
CGRect marcoVentana = [[UIScreen mainScreen] bounds]
```

Normalmente utilizaremos el primero para las vistas que queramos que ocupen todo el espacio disponible en la pantalla, y el segundo para definir la ventana principal. Hemos de destacar que `UIWindow` es una subclase de `UIView`, por lo que todas las operaciones disponibles para las vistas están disponibles también para las ventanas. Las ventanas no son más que un tipo especial de vista.

Las vistas (`UIView`) también nos proporcionan una serie de métodos para consultar y modificar la jerarquía. El método básico que necesitaremos es `addSubview`, que nos permitirá añadir una subvista a una vista determinada (o a la ventana principal):

```
[self.window addSubview: vista];
```

Con esto haremos que la vista aparezca en la pantalla. Podemos también eliminar una vista enviándole el mensaje `removeFromSuperview` (se le envía a la vista hija que queremos eliminar). Podemos también consultar la jerarquía con los siguientes métodos:

- `superview`: Nos da la vista padre de la vista destinataria del mensaje.
- `subviews`: Nos da la lista de subvistas de una vista dada.
- `isDescendantOfView`: Comprueba si una vista es descendiente de otra.

Como vemos, una vista tiene una lista de vistas hijas. Cada vista hija tiene un índice, que determinará el orden en el que se dibujan. El índice `0` es el más cercano al observador, y por lo tanto tapará a los índices superiores. Podemos insertar una vista en un índice determinado de la lista de subvistas con `insertSubview:atIndex:`.

Puede que tengamos una jerarquía compleja y necesitemos acceder desde el código a una determinada vista por ejemplo para inicializar su valor. Una opción es hacer un *outlet* para cada vista que queramos modificar, pero esto podría sobrecargar nuestro objeto de *outlets*. También puede ser complejo y poco fiable el buscar la vista en la jerarquía. En estos casos, lo más sencillo es darle a las vistas que buscamos una etiqueta (*tag*) mediante la propiedad *Tag* del inspector de atributos, o asignando la propiedad `tag` de forma programática. Podremos localizar en nuestro código una vista a partir de su etiqueta mediante `viewWithTag`. Llamando a este método sobre una vista, buscará entre todas las subvistas aquella con la etiqueta indicada:

```
UIView *texto = [self.window viewWithTag: 1];
```

Podemos también obtener la posición y dimensiones de una vista. Tenemos varias opciones:

```
// Limites en coordenadas locales
// Su origen siempre es (0,0)
CGRect area = [vista bounds];

// Posición del centro de la vista en coordenadas de su supervista
CGPoint centro = [vista center];

// Marco en coordenadas de la supervista
CGRect marco = [vista frame]
```

A partir de `bounds` y `center` podremos obtener `frame`.

4.3. Acciones y eventos

Cuando creamos vistas de forma programática ya no hace falta definir *outlets* para ellas, ya que la función de los *outlets* era permitirnos acceder a los contenidos del NIB desde

código. Con las acciones pasará algo similar, ya no es necesario declarar los métodos como `IBAction`, ya que dicho tipo sólo servía para que Interface Builder lo reconociese como una acción, pero si que tendremos que indicar a las vistas en código el *target* y el *selector* al que queremos que llame cuando suceda un evento.

Aunque no declaremos los *callbacks* con `IBAction`, deberemos hacer que devuelvan `void` (equivalente a `IBAction`), y que tomen como parámetro el objeto que generó el evento:

```
- (void)botonPulsado: (id) sender {
    ...
}
```

Esta es la forma más habitual del *callback*, aunque en su forma completa incluye también un segundo parámetro de tipo `UIEvent` con los datos del evento que se ha producido.

```
- (void)botonPulsado: (id) sender event: (UIEvent *) evento {
    ...
}
```

Ahora debemos conectar este método de forma programática con el objeto que genera los eventos. Esto lo haremos con `addTarget:action:forControlEvents:` de la clase `UIControl`, que es una subclase de `UIView` de la que derivan todos los controles (botones, campos de texto, *sliders*, etc). Como podemos ver, se utiliza el patrón *target*-acción.

```
[boton addTarget: self action: @selector(botonPulsado:)
    forControlEvents: UIControlEventTouchUpInside];
```

Podemos programar eventos poniendo como *target* `nil`:

```
[boton addTarget: nil action: @selector(botonPulsado:)
    forControlEvents: UIControlEventTouchUpInside];
```

En este caso, estamos indicando que cuando se produzca el evento se llame al selector indicado en el objeto que se comporte como *first responder* del control. Un *responder* es un objeto que deriva de la clase `UIResponder`, y que por lo tanto sabe como responder ante eventos de la interfaz. La clase `UIView` hereda de `UIResponder`, por lo que cualquier vista puede actual como *responder*.

Cuando tenemos una jerarquía de vistas se forma una cadena de *responders*. La cadena comienza desde la vista en la que se produce el evento, tras ello sube a su vista padre, y así sucesivamente hasta llegar a la ventana principal y finalmente a la aplicación (`UIApplication`), que también es un *responder*.

Cuando definamos una acción cuyo *target* es `nil`, se buscará el *selector* indicado en la cadena de *responders* en el orden indicado. Si en ninguno de ellos se encuentra dicho *selector*, el evento quedará sin atender.

Este tipo de acciones dirigidas a `nil` también pueden ser definidas con *Interface Builder*. Para ello encontramos entre los objetos referenciados uno llamado *First Responder*. Realmente, dicha referencia es `nil`, de forma que cuando conectemos un evento a dicho

objeto, realmente estaremos definiendo una acción a `nil`.

5. Propiedades de las vistas

A continuación vamos a repasar las propiedades básicas de las vistas, que podremos modificar tanto desde Interface Builder como de forma programática.

5.1. Disposición

Entre las propiedades más importantes en las vistas encontramos aquellas referentes a su disposición en pantalla. Hemos visto que tanto cuando creamos la vista con Interface Builder como cuando la inicializamos de forma programática hay que especificar el marco que ocupará la vista en la pantalla. Cuando se crea de forma visual el marco se puede definir pulsando con el ratón sobre los márgenes de la vista y arrastrando para así mover sus límites. En el código estos límites se especifican mediante el tipo `CGRect`, en el que se especifica posición *(x,y)* de inicio, y el ancho y el alto que ocupa la vista. Estos datos se especifican en el sistema de coordenadas de la supervista.

Sin embargo, en muchas ocasiones nos interesa que el tamaño no sea fijo sino que se adapte al área disponible. De esta forma nuestra interfaz podría adaptarse de forma sencilla a distintas orientaciones del dispositivo (horizontal o vertical).

Para que la disposición de los elementos de la pantalla se ajuste automáticamente al espacio disponible, deberemos activar la propiedad `autoresizedSubviews` de la supervista que contiene dichos elementos. Esta propiedad se puede encontrar también en el inspector de atributos de Interface Builder con el nombre *Autoresize Subviews*.

Ahora necesitamos especificar la forma en la que las diferentes subvistas van a ajustar su tamaño. Para ello definimos aquellos ejes que pueden tener tamaño flexible y aquellos que son de tamaño fijo. Esto lo podemos ajustar de forma visual desde el *size inspector*. Tendremos ejes internos de la vista (horizontal y vertical), y ejes externos (superior, inferior, derecho e izquierdo).

Los ejes internos marcados en rojo serán flexibles, mientras que los que no lo estén serán de tamaño fijo. Para los ejes externos el comportamiento es el contrario, los ejes marcados en rojo son de tamaño fijo, mientras que los que no estén marcados serán de tamaño flexible.

Esto se puede hacer también desde el código, mediante la propiedad `autoresizingMask` de la vista.

5.2. Transformaciones

Podemos también aplicar una transformación a las vistas, mediante su propiedad `transform`. Por defecto las vistas tendrán aplicada la transformación identidad

`CGAffineTransformIdentity`.

La transformación se define mediante una matriz de transformación 2D de dimensión 3x3. Podemos crear transformaciones de forma sencilla con macros como `CGAffineTransformMakeRotation` o `CGAffineTransformMakeScale`.

Atención

Si nuestra vista tiene aplicada una transformación diferente a la identidad, su propiedad `frame` no será significativa. En este caso sólo deberemos utilizar `center` y `bounds`.

5.3. Otras propiedades

En las vistas encontramos otras propiedades que nos permiten determinar su color o su opacidad. En primer lugar tenemos `backgroundColor`, con la que podemos dar el color de fondo de una vista. En el inspector de atributos (sección *View*) podemos verlo como propiedad *Background*. El color de fondo puede ser transparente, o puede utilizarse como fondo un determinado patrón basado en una imagen.

De forma programática, el color se especifica mediante un objeto de clase `UIColor`. En esta clase podemos crear un color personalizado a partir de sus componentes (rojo, verde, azul, alpha), o a partir de un patrón.

Por otro lado, también podemos hacer que una vista tenga un cierto grado de transparencia, o esté oculta. A diferencia de `backgroundColor`, que sólo afectaba al fondo de la vista, con la propiedad `alpha`, de tipo `CGFloat`, podemos controlar el nivel de transparencia de la vista completa con todo su contenido y sus subvistas. Si una vista no tiene transparencia, podemos poner su propiedad `opaque` a `YES` para así optimizar la forma de dibujarla. Esta propiedad sólo debe establecerse a `YES` si la vista llena todo su contendo y no deja ver nada del fondo. De no ser así, el resultado es impredecible. Debemos llevar cuidado con esto, ya que por defecto dicha propiedad es `YES`.

Por último, también podemos ocultar una vista con la propiedad `hidden`. Cuando hagamos que una vista se oculte, aunque seguirá ocupando su correspondiente espacio en pantalla, no será visible ni recibirá eventos.

6. Controles básicos

En Cocoa Touch podemos encontrar una serie de vistas y controles que podemos utilizar en nuestras aplicaciones. Todos estos elementos están disponibles en la librería de objetos, y podemos añadirlos a la interfaz simplemente arrastrándolos sobre el NIB. Vamos a ver algunos de ellos.

6.1. Etiquetas de texto

Una de las vistas más sencillas es la etiqueta de texto. Con ella podemos mostrar texto estático en la interfaz. En esta vista fundamentalmente configuraremos el texto a mostrar y sus propiedades (las encontraremos en la sección *Label* del inspector de atributos). También podemos crear esta vista de forma programática creando una instancia de `UILabel`.

```
UILabel *etiqueta = [[UILabel alloc]
    initWithFrame: CGRectMake(0,0,200,30)];
```

Nota

Podemos ver que todas las vistas se crean de forma programática utilizando el inicializador designado de su superclase `UIView`, tal como hemos visto anteriormente.

Es posible que desde el código necesitemos establecer el texto a mostrar en la etiqueta. Para ello utilizaremos su propiedad `text`.

```
[etiqueta setText: @"Titulo"];
```

6.2. Campo de texto

Un campo de texto nos proporciona un espacio donde el usuario puede introducir y editar texto. Se define en la clase `UITextField`, y pertenece a un grupo de vistas denominados controles, junto a otros componentes como por ejemplo los botones. Esto es así porque permiten al usuario interactuar con la aplicación. No heredan directamente de `UIView`, sino de su subclase `UIControl`, que incorpora los métodos para tratar eventos de la interfaz mediante el patrón *target*-acción como hemos visto anteriormente.

Sus propiedades se pueden encontrar en la sección *Text Field* del inspector de atributos. Podremos especificar un texto por defecto (*Text*), o bien un texto a mostrar sombreado en caso de que el usuario no haya introducido nada (*Placeholder Text*). Esto será útil por ejemplo para dar una pista al usuario sobre lo que debe introducir en dicho campo.

Si nos fijamos en el inspector de conexiones del campos de texto, veremos la lista de eventos que podemos conectar a nuestra acciones. Esta lista de eventos es común para cualquier control. En el caso de un campo de texto por ejemplo nos puede interesar el evento *Value Changed*.

6.3. Botones

Al igual que los campos de texto, los botones son otro tipo de control (heredan de `UIControl`). Se definen en la clase `UIButton`, que puede ser inicializada de la misma forma que el resto de vistas.

Si nos fijamos en el inspector de atributos de un botón (en la sección *Button*), vemos que podemos elegir el tipo de botón (atributo *Type*). Podemos seleccionar una serie de estilos

prefedinidos para los botones, o bien darle un estilo propio (*Custom*).

El texto que aparece en el botón se especifica en la propiedad *Title*, y podemos configurar también su color, sombreado, o añadir una imagen como icono.

En el inspector de conexiones, el evento que utilizaremos más comúnmente en los botones es *Touch Up Inside*, que se producirá cuando levantemos el dedo tras pulsar dentro del botón. Este será el momento en el que se realizará la acción asociada al botón.

6.4. Imágenes

Para mostrar imágenes en la interfaz de la aplicación lo más sencillo es utilizar la vista `UIImageView`, que se encarga de mostrar una imagen estática. Deberemos indicar la imagen en su atributo *Image*.

Si creamos esta vista de forma programática, deberemos especificar la imagen como un objeto de clase `UIImage`. La forma más sencilla de instanciar un objeto de este tipo es mediante su método factoría `imageNamed:`, en el que debemos proporcionar el nombre del fichero de la imagen, que buscará en el *bundle* principal de la aplicación (esto es equivalente a especificar la imagen en el inspector de atributos).

```
UIImage *imagen = [UIImage imageNamed: @"imagen.png"];
UIImageView *imageView = [[UIImageView alloc] initWithImage: imagen];
```

En este caso los límites de la vista se ajustarán al tamaño de la imagen.

Deberemos añadir las imágenes como recursos al proyecto, y dentro del *target* deberán aparecer dentro de la fase *Copy Bundle Resources*. Cuando añadamos un recurso de tipo imagen lo normal será que automáticamente sea añadido a dicha fase. La clase `UIImage` soporta numerosos formatos de imagen: TIFF, JPEG, GIF, PNG, DIB, ICO, CUR, XBM.

Pantalla retina

Si queremos que nuestra aplicación aproveche la resolución de la pantalla retina, sin malgastar memoria en dispositivos de resolución inferior, podemos proporcionar dos versiones de la imagen, una de ellas con sufijo `@2x`. Por ejemplo, podríamos tener `logo.png` y `logo@2x.png`. En el código (o en Interface Builder) especificaremos siempre como imagen `logo.png`, pero la API de Cocoa Touch cargará una versión u otra según si el dispositivo cuenta con pantalla retina o no.