

Introducción a Xcode y Objective-C

Índice

1 El entorno Xcode.....	4
1.1 Estructura del proyecto.....	8
1.2 Propiedades del proyecto.....	11
1.3 Configuraciones.....	15
1.4 Localización.....	16
1.5 Esquemas y acciones.....	18
1.6 Recursos y grupos.....	20
1.7 Interface Builder.....	22
1.8 Organizer.....	23
1.9 Repositorios SCM.....	25
1.10 Snapshots.....	34
1.11 Ejecución y firma.....	35
2 Ejercicios de Xcode.....	42
2.1 Creación de un proyecto con Xcode (1 punto).....	42
2.2 Repositorios remotos (0 puntos).....	42
2.3 Iconos y recursos (1 punto).....	43
2.4 Localización (1 punto).....	43
3 Introducción a Objective-C.....	44
3.1 Tipos de datos.....	44
3.2 Directivas.....	46
3.3 Paso de mensajes.....	50
3.4 Creación e inicialización.....	51
3.5 Algunas clases básicas de Cocoa Touch.....	52
3.6 Colecciones de datos.....	61
4 Ejercicios de Objective-C.....	67
4.1 Manejo de cadenas (1 punto).....	67

4.2 Manejo de fechas (1 punto).....	67
4.3 Gestión de errores (1 punto).....	68
5 Objetos y propiedades.....	69
5.1 Clases y objetos.....	69
5.2 Propiedades de los objetos.....	81
5.3 Key-Value-Coding (KVC).....	90
5.4 Protocolos.....	92
5.5 Categorías y extensiones.....	93
6 Ejercicios de objetos, propiedades y colecciones.....	95
6.1 Creación de objetos (1 punto).....	95
6.2 Propiedades (1 punto).....	96
6.3 Listas (1 punto).....	96
6.4 Gestión de memoria con ARC (0 puntos).....	97
7 Programación de eventos.....	98
7.1 Patrón target-selector.....	98
7.2 Notificaciones.....	99
7.3 Key Value Observing (KVO).....	101
7.4 Objetos delegados y protocolos.....	101
7.5 Bloques.....	102
7.6 Introspección.....	103
7.7 Ciclo de vida de las aplicaciones.....	106
8 Ejercicios de gestión de eventos.....	108
8.1 Temporizadores (1 punto).....	108
8.2 Notificaciones (1 punto).....	108
8.3 Delegados (1 punto).....	109
9 Depuración y pruebas.....	110
9.1 Depuración clásica: Uso de directivas NSLog y Asserts.....	110
9.2 Usando el depurador de XCode.....	112
9.3 Usando Instruments para detectar problemas de memoria.....	117
9.4 Pruebas de unidad.....	126
10 Depuración y pruebas - Ejercicios.....	132
10.1 Usando NSLog con distintos tipos de datos (0.5 puntos).....	132

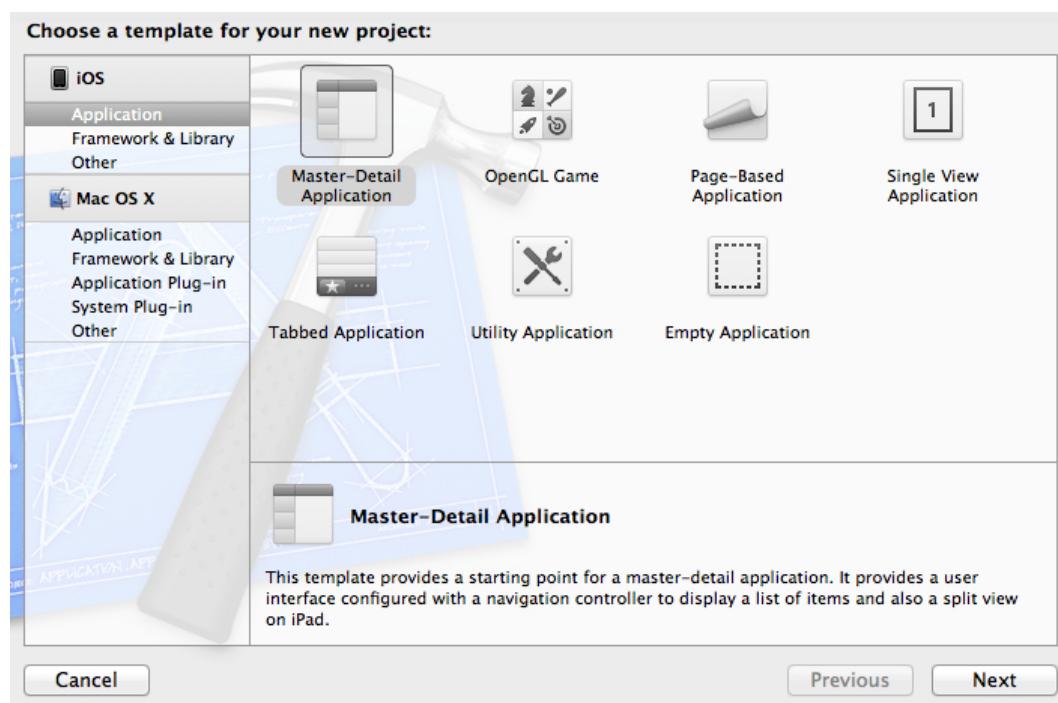
10.2 Breakpoints y análisis de variables (1 punto).....	132
10.3 NSZombie: Detectando las excepciones de memoria (0.5 puntos).....	132
10.4 Instruments: detectando memory leaks (1 puntos).....	132

1. El entorno Xcode

Para desarrollar aplicaciones iOS (iPod touch, iPhone, iPad) deberemos trabajar con una serie de tecnologías y herramientas determinadas. Por un lado, deberemos trabajar con el IDE Xcode dentro de una máquina con el sistema operativo MacOS instalado, cosa que únicamente podremos hacer en ordenadores Mac. Por otro lado, el lenguaje que deberemos utilizar es Objective-C, una extensión orientada a objetos del lenguaje C, pero muy diferente a C++. Además, deberemos utilizar la librería Cocoa Touch para crear una interfaz para las aplicaciones que pueda ser visualizada en un dispositivo iOS. Vamos a dedicar las primeras sesiones del módulo a estudiar fundamentalmente el entorno Xcode y el lenguaje Objective-C, pero introduciremos también algunos elementos básicos de Cocoa Touch para así poder practicar desde un primer momento con aplicaciones iOS.

Comenzaremos viendo cómo empezar a trabajar con el entorno Xcode. Vamos a centrarnos en la versión 4.2 de dicho IDE, ya que es la que se está utilizando actualmente y cuenta con grandes diferencias respecto a sus predecesoras.

Al abrir el entorno, lo primero que deberemos hacer es crear un nuevo proyecto con *File > New > New Project....*. En el asistente para la creación del proyecto nos dará a elegir una serie de plantillas de las que partir. En el lateral izquierdo vemos una serie de categorías de posibles plantillas, y en la parte central vemos las plantillas de la categoría seleccionada. Vemos además que las categorías se dividen en iOS y Mac OS X. Como es evidente, nos interesarán aquellas de la sección iOS.



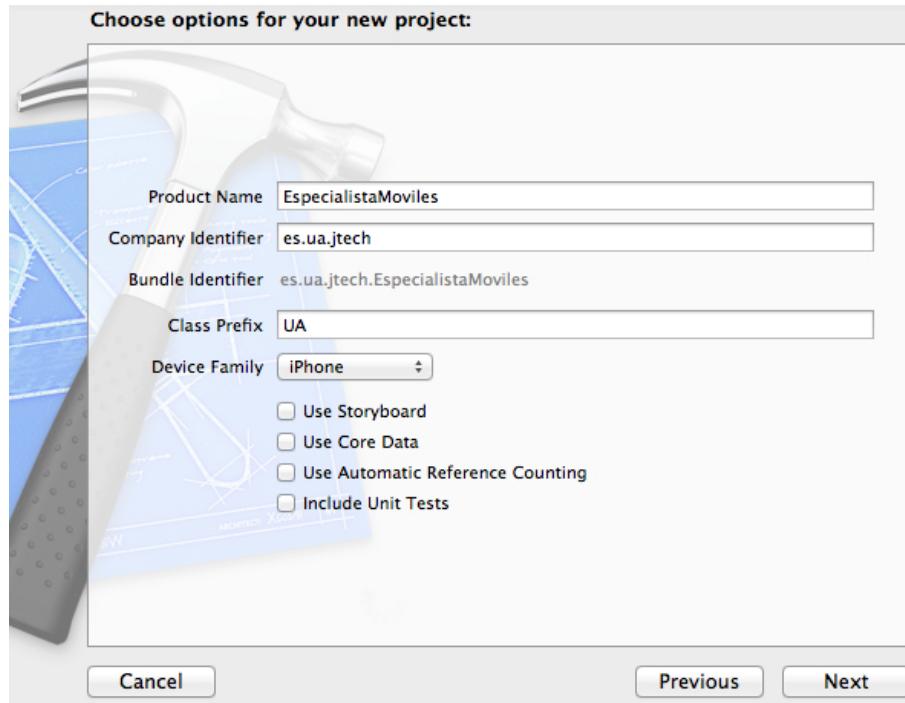
Tipos de proyectos

Las categorías que encontramos son:

- *Application*: Estos son los elementos que nos interesarán. Con ellos podemos crear diferentes plantillas de aplicaciones iOS. Encontramos plantillas para los estilos de navegación más comunes en aplicaciones iOS, que estudiaremos con mayor detenimiento más adelante, y también alguna plantilla básica como *Empty Application*, que no crea más componentes que una ventana vacía. Deberemos elegir aquella plantilla que más se ajuste al tipo de aplicación que vayamos a realizar. La más común en aplicaciones iOS es *Master-Detail Application* (en versiones anteriores de Xcode el tipo equivalente se llamaba *Navigation-based Application*), por lo que será la que utilicemos durante las primeras sesiones.
- *Framework & Library*: Nos permitirá crear librerías o *frameworks* que podamos utilizar en nuestras aplicaciones iOS. No se podrán ejecutar directamente como una aplicación, pero pueden ser introducidos en diferentes aplicaciones.
- *Other*: Aquí encontramos la opción de crear un nuevo proyecto vacío, sin utilizar ninguna plantilla.

Tras seleccionar un tipo de proyecto, nos pedirá el nombre del producto y el identificador de la compañía. El nombre del producto será el nombre que queramos darle al proyecto y a nuestra aplicación, aunque más adelante veremos que podemos modificar el nombre que se muestra al usuario. El identificador de la compañía se compone de una serie de cadenas en minúscula separadas por puntos que nos identificarán como desarrolladores. Normalmente pondremos aquí algo similar a nuestra URL escrita en orden inverso (como se hace con los nombres de paquetes en Java). Por ejemplo, si nuestra web es `jtech.ua.es`, pondremos como identificador de la compañía `es.ua.jtech`. Con el nombre del producto junto al identificador de la compañía compondrá el identificador del paquete (*bundle*) de nuestra aplicación. Por ejemplo, una aplicación nuestra con nombre `EspecialistaMoviles`, tendría el identificador `es.ua.jtech.EspecialistaMoviles`.

Podremos también especificar un prefijo para las clases de nuestra aplicación. En Objective-C no existen los paquetes como en lenguaje Java, por lo que como espacio de nombres para las clases es habitual utilizar un determinado prefijo. Por ejemplo, si estamos desarrollando una aplicación de la Universidad de Alicante, podríamos especificar como prefijo `UA`, de forma que todas las clases de nuestra aplicación llevarán ese prefijo en su nombre y nos permitirá distinguirlas de clases de librerías que estemos utilizando, y que podrían llamarse de la misma forma que las nuestras si no llevasen prefijo.



Datos del proyecto

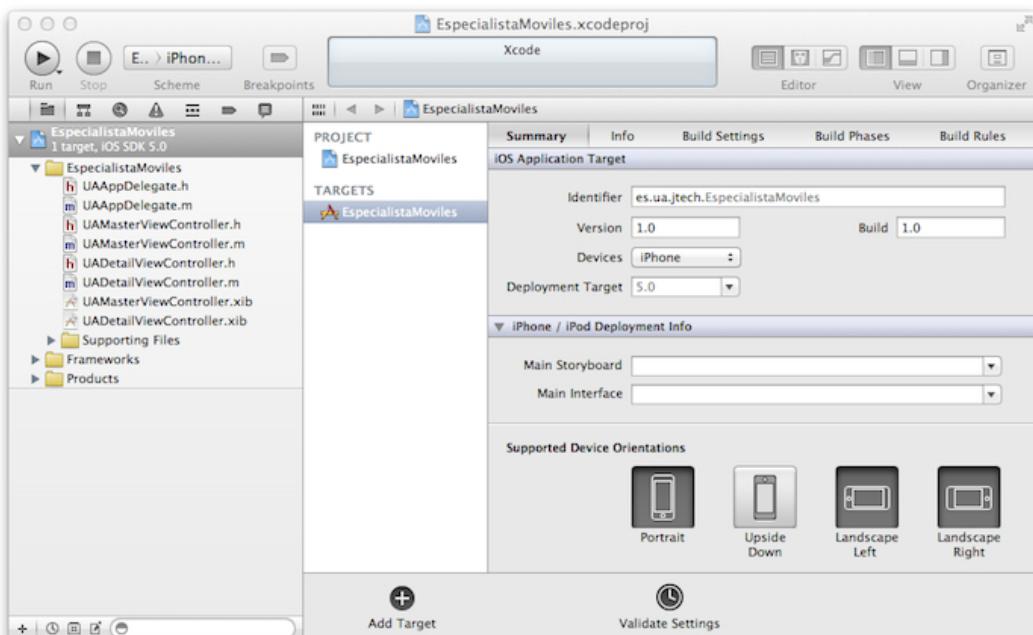
Bajo los campos anteriores, aparece un desplegable que nos permite elegir a qué tipo de dispositivos vamos a destinar la aplicación: iPhone, iPad, o Universal. Las aplicaciones de iPhone se pueden ver en el iPad emuladas, pero en un recuadro del tamaño del iPhone. Las de iPad sólo son compatibles con el iPad, mientras que las aplicaciones Universales son compatibles con ambos dispositivos y además su interfaz de adapta a cada uno de ellos. Por ahora comenzaremos realizando aplicaciones únicamente para iPhone, y más adelante veremos cómo desarrollar aplicaciones para iPad y aplicaciones Universales.

También tendremos cuatro *checkboxes* que para añadir *Storyboards*, *Core Data*, *Automatic Reference Counting* (ARC) y pruebas de unidad a nuestra aplicación. Por ahora vamos a desmarcarlos. Más adelante estudiaremos dichos elementos.

Tras llenar estos datos, pulsaremos *Next* y para finalizar tendremos que seleccionar la ubicación del sistema de archivos donde guardar el proyecto. Se creará un directorio con el nombre de nuestro proyecto en la localización que indiquemos, y dicho directorio contendrá todos los componentes del proyecto.

Nota

En la ventana de selección de la ubicación en la que guardar el proyecto nos dará también la opción de crear automáticamente un repositorio SCM local de tipo Git para dicho proyecto. Por el momento vamos a utilizar dicho repositorio, ya que lo único que tendremos que hacer es dejar marcada la casilla y Xcode se encargará de todo lo demás. Más adelante veremos como subir el proyecto a un repositorio remoto propio.



Aspecto del entorno Xcode

Una vez creado el proyecto, veremos la ventana principal del entorno Xcode. Podemos distinguir las siguientes secciones:

- **Navegador:** A la derecha vemos los artefactos de los que se compone el proyecto, organizados en una serie de grupos. Esta sección se compone de varias pestañas, que nos permitirán navegar por otros tipos de elementos, como por ejemplo por los resultados de una búsqueda, o por los resultados de la construcción del proyecto (errores y warnings).
- **Editor:** En la parte central de la pantalla vemos el editor en el que podremos trabajar con el fichero que esté actualmente seleccionado en el navegador. Dependiendo del tipo de fichero, editaremos código fuente, un fichero de propiedades, o la interfaz de la aplicación de forma visual.
- **Botones de construcción:** En la parte superior izquierda vemos una serie de botones que nos servirán para construir y ejecutar la aplicación.
- **Estado:** En la parte central de la parte superior veremos la información de estado del entorno, donde se mostrarán las acciones que se están realizando en un momento dado (como compilar o ejecutar el proyecto).
- **Opciones de la interfaz:** En la parte derecha de la barra superior tenemos una serie de botones que nos permitirán alterar el aspecto de la interfaz, mostrando u ocultando algunos elementos, como puede ser la vista de navegación (a la izquierda), la consola de depuración y búsqueda (abajo), y la barra de utilidades y ayuda rápida (derecha). También podemos abrir una segunda ventana de Xcode conocida como *Organizer*, que nos permitirá, entre otras cosas, gestionar los dispositivos con los que contamos o navegar por la documentación. Más adelante veremos dicha ventana con más detalle.

Recomendación

Resulta bastante útil mantener abierta la vista de ayuda rápida a la derecha de la pantalla, ya que conforme editamos el código aquí nos aparecerá la documentación de los elementos de la API sobre los que se encuentre el cursor.

En este momento podemos probar a ejecutar nuestro proyecto. Simplemente tendremos que pulsar el botón de ejecutar (en la parte superior izquierda), asegurándonos que junto a él en el cuadro desplegable tenemos seleccionado que se ejecute en el simulador del iPhone. Al pulsar el botón de ejecución, se mostrará el progreso de la construcción del proyecto en la parte superior central, y una vez finalizada se abrirá una ventana con un emulador de iPhone y se ejecutará nuestra aplicación en él, que actualmente será únicamente una pantalla con una barra de navegación y una lista vacía. En la barra de menús del simulador podemos encontrar opciones para simular diferentes condiciones del hardware, como por ejemplo los giros del dispositivo.



Simulador de iPhone

1.1. Estructura del proyecto

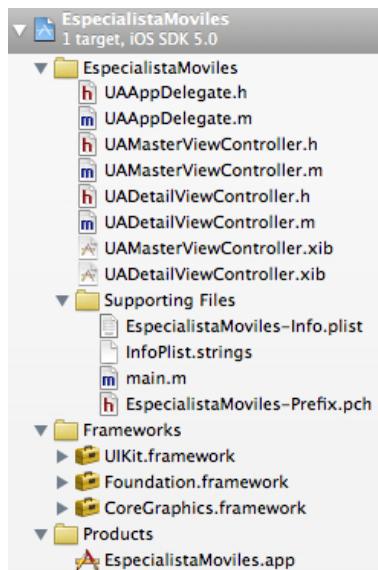
En la zona del navegador podemos ver los diferentes elementos que componen nuestro proyecto. Los tipos de ficheros que encontraremos normalmente en un proyecto Objective-C son los siguientes:

- .m: El código fuente Objective-C se guarda en ficheros con extensión .m, por lo que

ese será el formato que encontraremos normalmente, aunque también se podrá utilizar código C (.c) estándar, C++ (.cpp, .cc), u Objective-C combinado con C++ (.mm).

- .h: También tenemos los ficheros de cabeceras (.h). Las clases de Objective-C se componen de un fichero .m y un fichero .h con el mismo nombre.
- .xib: También encontramos .xib (también conocidos como nib, ya que este es su formato una vez compilados) que son los ficheros que contienen el código de la interfaz, y que en el entorno se editarán de forma visual.
- .plist: Formato de fichero de propiedades que estudiaremos más adelante con mayor detenimiento, y que se utiliza para almacenar una lista de propiedades. Resultan muy útiles para guardar los datos o la configuración de nuestra aplicación, ya que resultan muy fáciles de editar desde el entorno, y muy fáciles de leer y de escribir desde el programa.
- .strings: El formato .strings define un fichero en el que se definen una serie de cadenas de texto, cada una de ellas asociada a un identificador, de forma que creando varias versiones del mismo fichero .strings tendremos la aplicación localizada a diferentes idiomas.
- .app: Es el resultado de construir y empaquetar la aplicación. Este fichero es el que deberemos subir a iTunes para su publicación en la App Store.

Estos diferentes tipos de artefactos se pueden introducir en nuestro proyecto organizados por grupos. Es importante destacar que los grupos no corresponden a ninguna organización en el disco, ni tienen ninguna repercusión en la construcción del proyecto. Simplemente son una forma de tener los artefactos de nuestro proyecto organizados en el entorno, independientemente del lugar del disco donde residan realmente. Los grupos se muestran en el navegador con el icono de una carpeta amarilla.

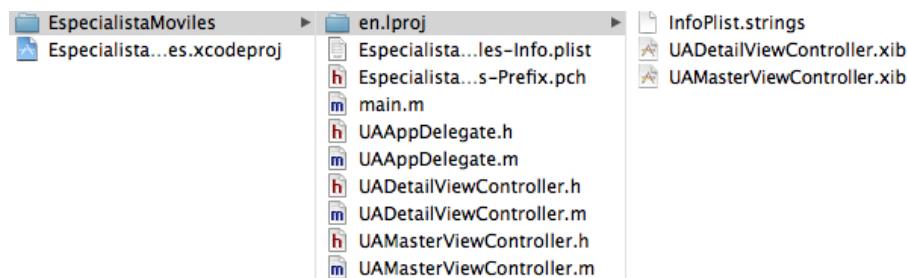


Configuración del proyecto creado

En la plantilla que nos ha creado encontramos los siguientes grupos de artefactos:

- **Fuentes** (están en un grupo con el mismo nombre del proyecto, en nuestro caso `EspecialistaMoviles`). Aquí se guardan todos los ficheros de código fuente que componen la aplicación (código fuente `.m`, cabeceras `.h` y ficheros de la interfaz `.xib`).
- **Soporte** (dentro del grupo de fuentes, en un subgrupo *Supporting files*). Aquí encontramos algunos recursos adicionales, como el fichero `main.m`, que es el punto de arranque de la aplicación, y que nosotros normalmente no deberemos tocar, el fichero `Projeto-Prefix.pch`, que define un prefijo con una serie de *imports* que se aplicarán a todos los ficheros del proyecto (normalmente para importar la API de Cocoa Touch), `Projeto-Info.plist`, que define una serie de propiedades de la aplicación (nombre, identificador, versión, ícono, tipos de dispositivos soportados, etc), y por último `InfoPlist.strings`, donde se externalizan las cadenas de texto que se necesitan utilizar en el fichero `.plist` anterior, para así poder localizar la aplicación fácilmente.
- **Frameworks**: Aquí se muestran los *frameworks* que la aplicación está utilizando actualmente. Por defecto nos habrá incluido `UIKit`, con la API de la interfaz de Cocoa Touch, `Foundation`, con la API básica de propósito general (cadenas, *arrays*, fechas, URLs, etc), y `CoreGraphics`, con la API básica de gráficos (fuentes, imágenes, geometría).
- **Products**: Aquí vemos los resultados de construir la aplicación. Veremos un fichero `EspecialistaMoviles.app` en rojo, ya que dicho fichero todavía no existe, es el producto que se generará cuando se construya la aplicación. Este fichero es el paquete (*bundle*) que contendrá la aplicación compilada y todos los recursos y ficheros de configuración necesarios para su ejecución. Esto es lo que deberemos enviar a Apple para su publicación en la App Store una vez finalizada la aplicación.

Sin embargo, si nos fijamos en la estructura de ficheros que hay almacenada realmente en el disco podemos ver que dichos grupos no existen:



Estructura de ficheros del proyecto

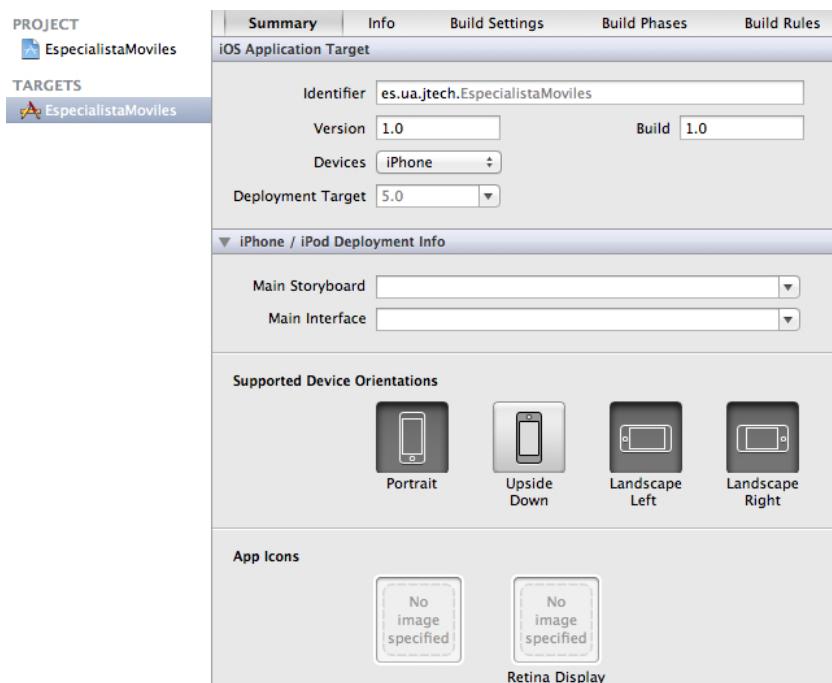
Vemos que tenemos en el directorio principal un fichero con extensión `xcodeproj`. Este es el fichero con la configuración de nuestro proyecto, y es el que deberemos seleccionar para abrir nuestro proyecto la próxima vez que vayamos a trabajar con él (haciendo doble-click sobre él se abrirá Xcode con nuestro proyecto). A parte de este fichero, tenemos un directorio con el nombre del proyecto que contiene todos los ficheros al mismo nivel. La única excepción son los ficheros que hay en un directorio de nombre

en.lproj. Esto se debe a que dichos ficheros están localizados, concretamente a idioma inglés como indica su directorio (en). Tendremos un subdirectorio como este por cada idioma que soporte nuestra aplicación (es.lproj, ca.lproj, etc). Es la forma en la que se trata la localización en las aplicaciones iOS, y lo veremos con más detalle a continuación.

1.2. Propiedades del proyecto

Si seleccionamos en el navegador el nodo raíz de nuestro proyecto, en el editor veremos una ficha con sus propiedades. Podemos distinguir dos apartados: *Project* y *Targets*. En *Project* tenemos la configuración general de nuestro proyecto, mientras que *Target* hace referencia a la construcción de un producto concreto. En nuestro caso sólo tenemos un *target*, que es la construcción de nuestra aplicación, pero podría haber varios. Por ejemplo, podríamos tener un target que construya nuestra aplicación para iPhone y otro que lo haga para iPad, empaquetando en cada una de ellas distintos recursos y compilando con diferentes propiedades. También tendremos dos *targets* si hemos creado pruebas de unidad, ya que tendremos un producto en el que se incluirán las pruebas, y otro en el que sólo se incluirá el código de producción que será el que publicaremos.

Si pinchamos sobre el *target*, en la pantalla principal veremos una serie de propiedades de nuestra aplicación que resultan bastante intuitivas, como la versión, dispositivos a los que está destinada, y versión mínima de iOS que necesita para funcionar. También podemos añadir aquí tanto los iconos como las imágenes para la pantalla de carga de nuestra aplicación, simplemente arrastrando dichas imágenes sobre los huecos correspondientes.



Configuración básica del target

Lo que no resulta tan claro es por ejemplo la diferencia entre versión y *build*. La primera de ellas indica el número de una *release* pública, y es lo que verá el usuario en la App Store, con el formato de tres enteros separados por puntos (por ejemplo, 1.2.1). Sin embargo, la segunda se utiliza para cada *build* interna. No es necesario seguir ningún formato dado, y puede coincidir o no con el número de versión. En muchas ocasiones se utiliza un número entero de *build* que vamos incrementando continuamente con cada iteración del proyecto, por ejemplo 1524. Otra diferencia existente entre ambos números es que podemos tener un número diferente de versión para distintos *targets*, pero la *build* siempre será la misma para todos ellos.

También vemos unos campos llamados *Main Storyboard* y *Main Interface*. Aquí se puede definir el punto de entrada de nuestra aplicación de forma declarativa, pudiendo especificar qué interfaz (fichero *xib*) o que *storyboard* se va a mostrar al arrancar la aplicación. En la plantilla crear ninguno de estos campos tiene valor, ya que la interfaz se crea de forma programática. En realidad el punto de entrada está en *main.m*, y podríamos modificarlo para tener mayor control sobre lo que la aplicación carga al arrancar, pero normalmente lo dejaremos tal como se crea por defecto. Más adelante veremos más detalles sobre el proceso de carga de la aplicación.

En la pestaña *Info* del *target* vemos los elementos de configuración anteriores, y algunos más, pero en forma de lista de propiedades. Realmente esta información es la que se almacena en el fichero *Info.plist*, si seleccionamos dicho fichero veremos los mismos datos también presentados como lista de propiedades. Por ejemplo la propiedad *Bundle display name* no estaba en la pantalla anterior, y nos permite especificar el nombre que aparecerá bajo el ícono de nuestra aplicación en el iPhone. Por defecto tenemos el nombre del proyecto, pero podemos modificarlo por ejemplo para acortarlo y así evitar que el iPhone recorte las letras que no quepan. Si pinchamos con el botón derecho sobre el fichero *Info.plist* y seleccionamos *Open As > Source Code*, veremos el contenido de este fichero en el formato XML en el que se definen los ficheros de listas de propiedades, y los nombres internos que tiene cada una de las propiedades de configuración.

Introducción a Xcode y Objective-C

PROJECT	Summary	Info	Build Settings	Build Phases	Build Rules
EspecialistaMovies					
TARGETS					
EspecialistaMovies					
Custom iOS Target Properties					
	Key	Type	Value		
	Bundle name	String	\$(PRODUCT_NAME)		
	Bundle identifier	String	es.ua.tech.\$(PRODUCT_NAME)identifier		
	InfoDictionary version	String	6.0		
	► Required device capabilities	Array	(1 item)		
	Bundle version	String	1.0		
	Executable file	String	\$(EXECUTABLE_NAME)		
	Application requires iPhone environment	Boolean	YES		
	► Icon files	Array	(0 items)		
	► Supported interface orientations	Array	(3 items)		
	Bundle display name	String	\$(PRODUCT_NAME)		
	Bundle OS Type code	String	APPL		
	Bundle creator OS Type code	String	????		
	Localization native development region	String	en		
	Bundle versions string, short	String	1.0		
	► Document Types (0)				
	► Exported UTIs (0)				
	► Imported UTIs (0)				
	► URL Types (0)				

Configuración completa del target

A continuación tenemos la pestaña *Build Settings*. Aquí se define una completa lista con todos los parámetros utilizados para la construcción del producto. Estos parámetros se pueden definir a diferentes niveles: valores por defecto para todos los proyectos iOS, valores generales para nuestro proyecto, y valores concretos para el *target* seleccionado. Se pueden mostrar los valores especificados en los diferentes niveles, y el valor que se utilizará (se coge el valor más a la izquierda que se haya introducido):

PROJECT	Summary	Info	Build Settings	Build Phases	Build Rules
EspecialistaMovies					
TARGETS					
EspecialistaMovies					
Basic All Combined Levels					
Setting	Resolved	Especialista... Especialista... iOS Default			
Architectures					
Additional SDKs					
Architectures	Standard	Standard	armv7		
Base SDK	Latest iOS (i...)	Latest iOS (i...)	iOS 5.0		
Build Active Architecture Only	No		No		
Supported Platforms	iphonesimulat...		iphonesimulat...		
Valid Architectures	armv6 armv7		armv6 armv7		
Build Locations					
Build Products Path	/Users/maloza...		/Users/maloza...		
Intermediate Build Files Path	/Users/maloza...		/Users/maloza...		
► Per-configuration Build Products Path	<Multiple val...		<Multiple val...		
► Per-configuration Intermediate Build Fi...	<Multiple val...		<Multiple val...		
Precompiled Headers Cache Path	/var/folders/p...		/var/folders/p...		
Build Options					
Build Variants	normal		normal		
Compiler for C/C++/Objective-C	Apple LLVM...	Apple LLVM...	DWARF with...		
Debug Information Format	DWARF with...		No		
Enable OpenMP Support	No		No		
Generate Profiling Code	No		No		
Precompiled Header Uses Files From B...	Yes		Yes		
Run Static Analyzer	No		No		
Scan All Source Files for Includes	No		No		
► Validate Built Product	<Multiple val...	<Multiple val...	No		

Parámetros de construcción del target

Hay un gran número de parámetros, por lo que es complicado saber lo que hace

exactamente cada uno, y la mayor parte de ellos no será necesario que los modifiquemos en ningún momento. Normalmente ya se les habrá asignado por defecto un valor adecuado, por lo que aquí la regla de oro es **si no sabes lo que hace, ¡no lo toques!**. En esta interfaz podremos cambiar los valores para el *target*. Si queremos cambiar los valores a nivel del proyecto, deberemos hacerlo seleccionando el proyecto en el lateral izquierdo del editor y accediendo a esta misma pestaña.

La información más importante que podemos encontrar aquí es:

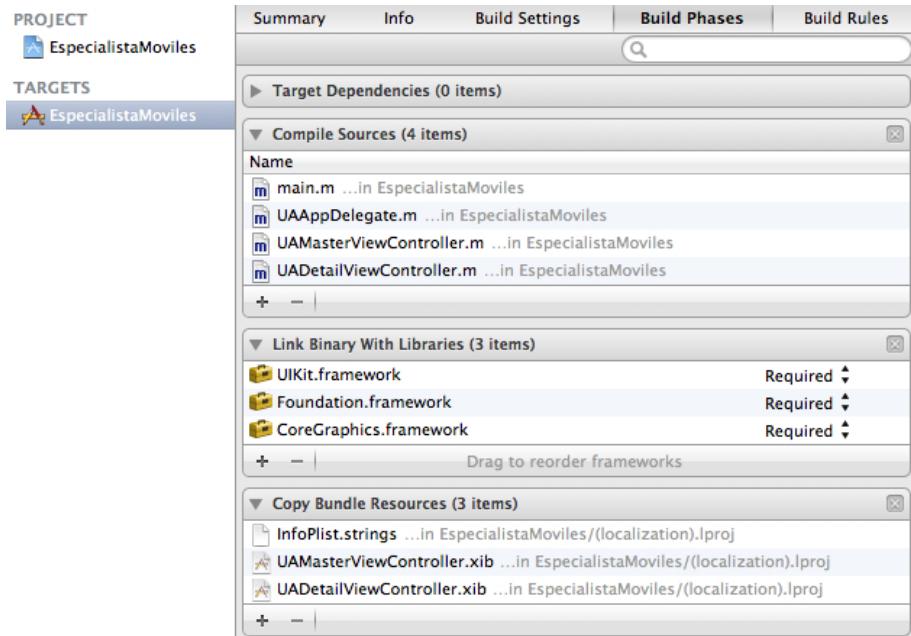
- *Base SDK*: Versión del SDK que se utiliza para compilar. Se diferencia de *Deployment Target* en que *Base SDK* indica la versión del SDK con la que se compila el proyecto, por lo que no podremos utilizar ninguna característica que se hubiese incluido en versiones posteriores a la seleccionada, mientras que *Deployment Target* indica la versión mínima para que nuestra aplicación funcione, independientemente de la versión con la que se haya compilado.
- *Code Signing*: Aquí podemos especificar el certificado con el que se firma la aplicación para probarla en dispositivos reales o para distribuirla. Más adelante veremos con más detalle la forma de obtener dicho certificado. Hasta que no lo tengamos, no podremos probar las aplicaciones más que en el emulador.

Advertencia

Si seleccionamos un *Deployment Target* menor que el *Base SDK*, deberemos llevar mucho cuidado de no utilizar ninguna característica que se haya incluido en versiones posteriores al *Deployment Target*. Si no llevásemos cuidado con esto, un comprador con una versión antigua de iOS podría adquirir nuestra aplicación pero no le funcionaría. Lo que si que podemos hacer es detectar en tiempo de ejecución si una determinada característica está disponible, y sólo utilizarla en ese caso.

Vemos también que para algunas opciones tenemos dos valores, uno para *Debug* y otro para *Release*. Estas son las configuraciones del proyecto, que veremos más adelante. Por ejemplo, podemos observar que para *Debug* se añade una macro DEBUG y se elimina la optimización del compilador para agilizar el proceso de construcción y hacer el código más fácilmente depurable, mientras que para *Release* se eliminan los símbolos de depuración para reducir el tamaño del binario.

La siguiente pestaña (*Build Phases*) es la que define realmente cómo se construye el *target* seleccionado. En ella figuran las diferentes fases de construcción del proyecto, y para cada una de ellas indica los ficheros que se utilizarán en ella:



Fases de construcción del target

Podemos ver aquí los ficheros que se van a compilar, las librerías con las que se va a enlazar, y los recursos que se van a copiar dentro del *bundle*. Por defecto, cuando añadamos elementos al proyecto se incluirán en la fase adecuada (cuando sea código se añadirá a la compilación, los *frameworks* a *linkado*, y el resto de recursos a copia). Sin embargo, en algunas ocasiones puede que algún fichero no sea añadido al lugar correcto, o que tengamos varios *targets* y nos interese controlar qué recursos se incluyen en cada uno de ellos. En esos casos tendremos que editar a mano esta configuración. Es importante destacar que el contenido y estructura final del paquete (*bundle*) generado se define en esta pantalla, y es totalmente independiente a la organización del proyecto que vemos en el navegador.

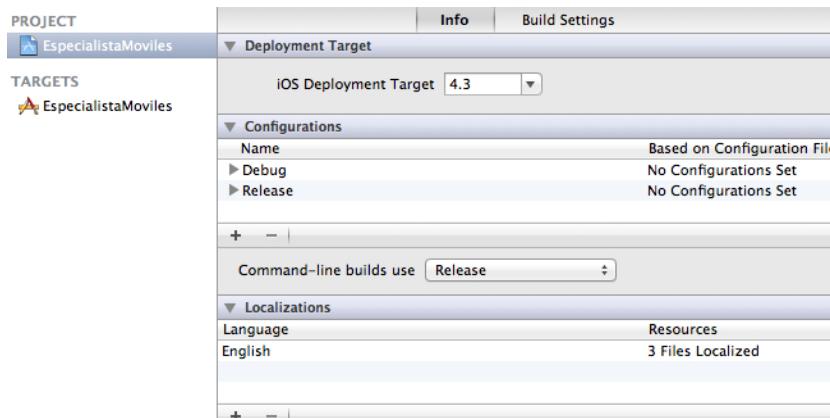
Por último, en *Build Rules*, se definen las reglas para la construcción de cada tipo de recurso. Por ejemplo, aquí se indica que las imágenes o ficheros de propiedades simplemente se copiarán, mientras que los ficheros .xib deberán compilarse con el compilador de Interface Builder. Normalmente no tendremos que modificar esta configuración.

1.3. Configuraciones

Hemos visto que en la pantalla *Build Settings* en algunos de los parámetros para la construcción de la aplicación teníamos dos valores: *Debug* y *Release*. Éstas son las denominadas configuraciones para la construcción de la aplicación. Por defecto se crean estas dos, pero podemos añadir configuraciones adicionales si lo consideramos oportuno (normalmente siempre se añade una tercera configuración para la distribución de la

aplicación, que veremos más adelante).

Podemos gestionar las configuraciones en la pestaña *Info* de las propiedades generales del proyecto:



Configuración general del proyecto

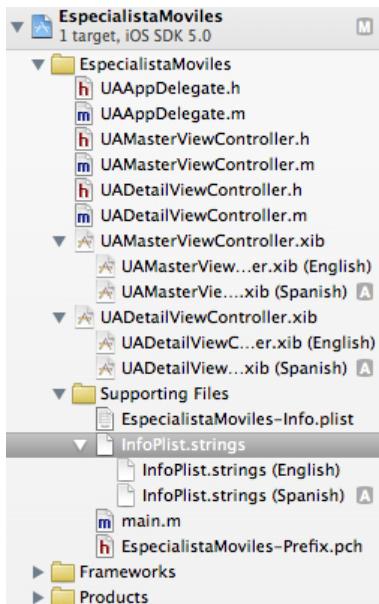
Podemos ver en la sección *Configurations* las dos configuraciones creadas por defecto. Bajo la lista, encontramos los botones (+) y (-) que nos permitirán añadir y eliminar configuraciones. Cuando añadimos una nueva configuración lo hacemos duplicando una de las existentes. Una práctica habitual es duplicar la configuración *Release* y crear una nueva configuración *Distribution* en la que cambiamos el certificado con el que se firma la aplicación.

Es importante diferenciar entre *targets* y *configuraciones*. Los *targets* nos permiten construir productos distintos, con diferentes nombre, propiedades, y conjuntos de fuentes y de recursos. Las configuraciones son diferentes formas de construir un mismo producto, cambiando las opciones del compilador, o utilizando diferentes firmas.

1.4. Localización

En la pantalla de propiedades generales del proyecto también vemos la posibilidad de gestionar las localizaciones de nuestra aplicación, es decir, los distintos idiomas en los que estará disponible. Por defecto sólo está en inglés, aunque con los botones (+) y (-) que hay bajo la lista podemos añadir fácilmente todas aquellas que queramos. Por ejemplo, podemos localizar también nuestra aplicación al español.

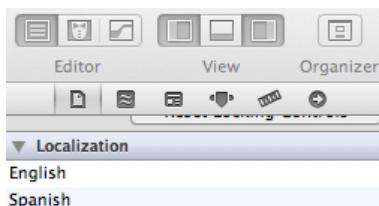
Por defecto localiza automáticamente los ficheros *xib* de la interfaz y los ficheros *strings* en los que se externalizan las cadenas de la aplicación. En el navegador veremos dos versiones de estos ficheros:



Ficheros localizados

En el disco veremos que realmente lo que se ha hecho es guardar dos copias de estos ficheros en los directorios `en.lproj` y `es.lproj`, indicando así la localización a la que pertenece cada uno.

Es posible, por ejemplo, que una interfaz no necesite ser localizada, bien por que no tiene textos, o porque los textos se van a poner desde el código. En tal caso, no es buena idea tener el fichero de interfaz duplicado, ya que cada cambio que hagamos en ella implicará modificar las dos copias. La gestión de la localización para cada fichero independiente se puede hacer desde el panel de utilidades (el panel de la derecha, que deberemos abrir en caso de que no esté abierto), seleccionando en el navegador el fichero para el que queremos modificar la localización:



Panel de utilidades de localización

De la misma forma, podemos también añadir localizaciones para ficheros que no estuviesen localizados. Por ejemplo, si tenemos una imagen en la que aparece algún texto, desde el panel de utilidades anterior podremos añadir una nueva localización y así tener una versión de la imagen en cada idioma.

Vemos también que por defecto nos ha creado (y localizado) el fichero

`InfoPlist.strings` que nos permite localizar las propiedades de `Info.plist`. Por ejemplo imaginemos que queremos que el nombre bajo el icono de la aplicación cambie según el idioma. Esta propiedad aparece en `Info.plist` como *Bundle display name*, pero realmente lo que necesitamos es el nombre interno de la propiedad, no la descripción que muestra Xcode de ella. Para ver esto, una vez abierto el fichero `Info.plist` en el editor, podemos pulsar sobre él con el botón derecho y seleccionar *Show Raw Keys/Values*. De esa forma veremos que la propiedad que buscamos se llama `CFBundleDisplayName`. Podemos localizarla en cada una de las versiones del fichero `InfoPlist.strings` de la siguiente forma:

```
// Versión en castellano del fichero InfoPlist.strings
"CFBundleDisplayName" = "MovilesUA";

// Versión en inglés del fichero InfoPlist.strings
"CFBundleDisplayName" = "MobileUA";
```

Si ejecutamos esta aplicación en el simulador, y pulsamos el botón *home* para volver a la pantalla principal, veremos que como nombre de nuestra aplicación aparece *MobileUA*. Si ahora desde la pantalla principal entramos en *Settings > General > International > Language* y cambiamos el idioma del simulador a español, veremos que el nombre de nuestra aplicación cambia a *MovilesUA*.

Con esto hemos localizado las cadenas de las propiedades del proyecto, pero ese fichero sólo se aplica al contenido de `Info.plist`. Si queremos localizar las cadenas de nuestra aplicación, por defecto se utilizará un fichero de nombre `Localizable.strings`, que seguirá el mismo formato (cada cadena se pone en una línea del tipo de las mostradas en el ejemplo anterior: `"identificador" = "cadena a mostrar" ;`).

Podemos crear ese fichero `strings` pulsando sobre el botón derecho del ratón sobre el grupo del navegador en el que lo queramos incluir y seleccionando *New File ...*. Como tipo de fichero seleccionaremos *iOS > Resource > Strings File*, y le llamaremos `Localizable`. No deberemos olvidar añadir las localizaciones desde el panel de utilizadas. Más adelante veremos cómo acceder desde el código de la aplicación a estas cadenas localizadas.

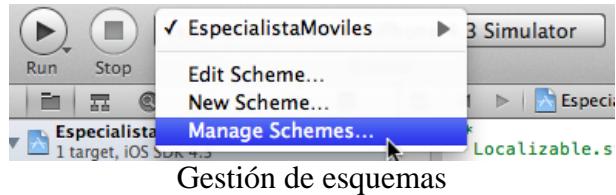
1.5. Esquemas y acciones

Otro elemento que encontramos para definir la forma en la que se construye y ejecuta la aplicación son los esquemas. Los esquemas son los encargados de vincular los *targets* a las configuraciones para cada una de las diferentes acciones que podemos realizar con el producto (construcción, ejecución, pruebas, análisis estático, análisis dinámico, o distribución).

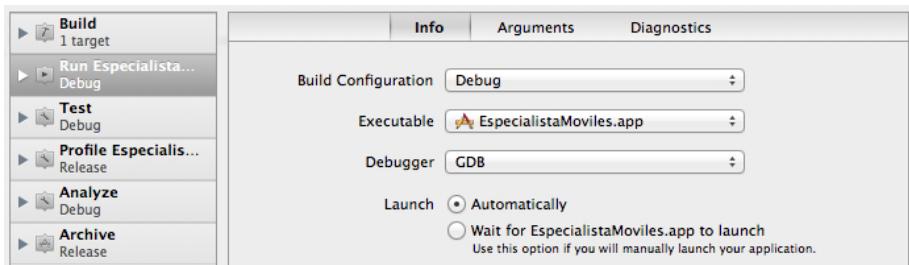
Por defecto nos habrá creado un único esquema con el nombre de nuestro proyecto, y normalmente no será necesario modificarlo ni crear esquemas alternativos, aunque vamos a ver qué información contiene para comprender mejor el proceso de construcción de la

aplicación.

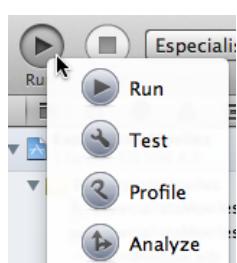
Para gestionar los esquemas debemos pulsar en el cuadro desplegable junto a los botones de ejecución de la aplicación:



Si queremos editar el esquema actual podemos pulsar en *Edit Scheme...*:



Vemos que el esquema contiene la configuración para cada posible acción que realicemos con el producto. La acción *Build* es un caso especial, ya que para el resto de acciones siempre será necesario que antes se construya el producto. En esta acción se especifica el *target* que se va a construir. En el resto de acciones se especificará la configuración que se utilizará para la construcción del target en cada caso. Podemos ejecutar estas diferentes acciones manteniendo pulsado el botón izquierdo del ratón sobre el botón *Run*:



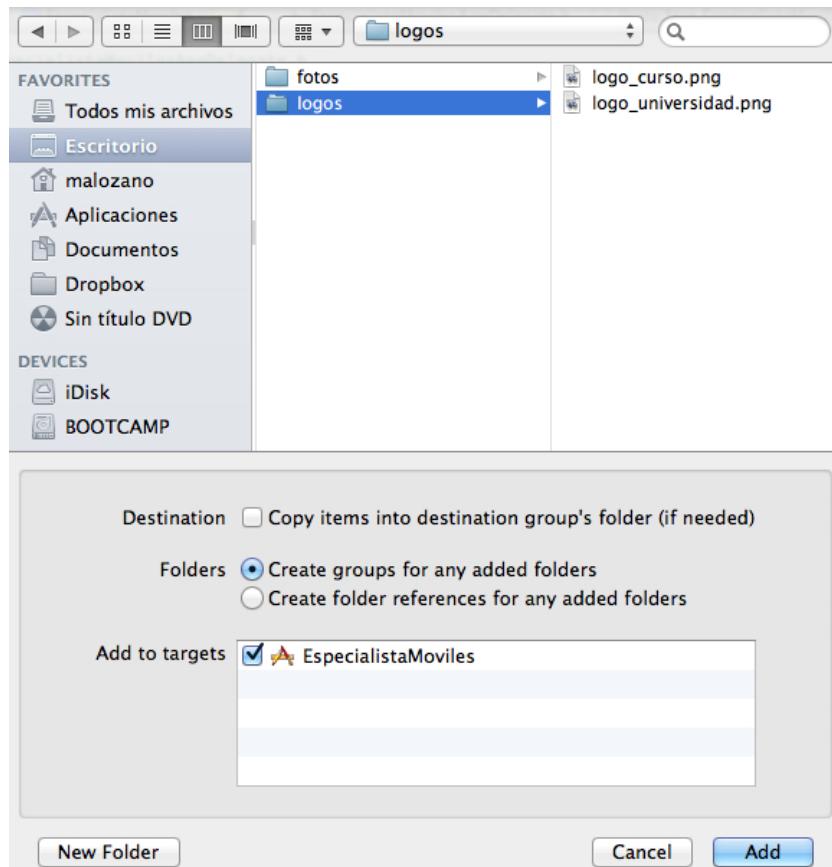
También pueden ser ejecutadas desde el menú *Product*, donde encontramos también la acción *Archive*, y la posibilidad de simplemente construir el producto sin realizar ninguna de las acciones anteriores. Las acciones disponibles son:

- *Build*: Construye el producto del *target* seleccionado en el esquema actual.
- *Run*: Ejecuta la aplicación en el dispositivo seleccionado (simulador o dispositivo real). Por defecto se construye con la configuración *Debug*.

- *Test*: Ejecuta las pruebas de unidad definidas en el producto. No funcionará si no hemos creado pruebas de unidad para el *target* seleccionado. Por defecto se construye con la configuración *Debug*.
- *Profile*: Ejecuta la aplicación para su análisis dinámico con la herramienta *Instruments*, que nos permitirá controlar en tiempo de ejecución elementos tales como el uso de la memoria, para poder así optimizarla y detectar posibles fugas de memoria. Más adelante estudiaremos esta herramienta con más detalles. Por defecto se construye con la configuración *Release*, ya que nos interesa optimizarla en tiempo de ejecución.
- *Analyze*: Realiza un análisis estático del código. Nos permite detectar construcciones de código incorrectas, como sentencias no alcanzables o posibles fugas de memoria desde el punto de vista estático. Por defecto se utiliza la configuración *Debug*.
- *Archive*: Genera un archivo para la distribución de nuestra aplicación. Para poder ejecutarla deberemos seleccionar como dispositivo destino *iOS Device*, en lugar de los simuladores. Utiliza por defecto la configuración *Release*, ya que es la versión que será publicada.

1.6. Recursos y grupos

Si queremos añadir recursos a nuestro proyecto, podemos pulsar con el botón derecho del ratón (*Ctrl-Click*) sobre el grupo en el que los queramos añadir, y seleccionar la opción *Add Files To "NombreProyecto"* Tendremos que seleccionar los recursos del disco que queramos añadir al proyecto. Puede ser un único fichero, o un conjunto de ficheros, carpetas y subcarpetas.



Añadir un nuevo recurso

Bajo el cuadro de selección de fichero vemos diferentes opciones. La primera de ellas es un *checkbox* con la opción *Copy items into destination group's folder (if needed)*. Si lo seleccionamos, copiará los recursos que añadamos el directorio en el que está el proyecto (en caso de que no estén ya dentro). En caso contrario, simplemente añadirá una referencia a ellos. Resulta conveniente copiarlos para que así el proyecto sea autocontenido, y podamos transportarlo fácilmente sin perder referencias.

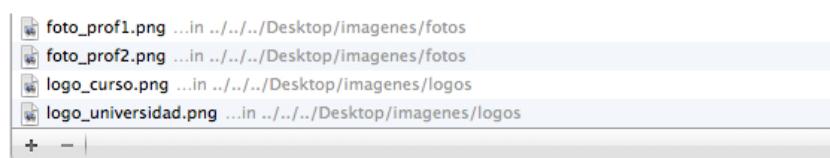
Bajo esta opción, nos da dos alternativas para incluir los ficheros seleccionados en el proyecto:

- *Create groups for any added folders*: Transforma las carpetas que se hayan seleccionado en grupos. Tendremos la misma estructura de grupos que la estructura de carpetas seleccionadas, pero los grupos sólo nos servirán para tener los recursos organizados dentro del entorno. Los grupos aparecerán como iconos de carpetas amarillas en el navegador.



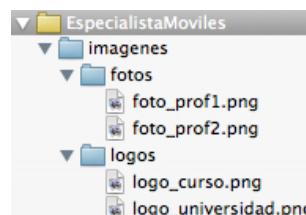
Grupo de recursos

Realmente todos los ficheros se incluirán en la raíz del paquete generado. Esto podemos comprobarlo si vamos a la sección *Build Phases* del *target* en el que hayamos incluido el recurso y consultamos la sección *Copy Bundle Resources*.



Copia de recursos en grupos

- *Create folder references for any added folders*: Con esta segunda opción las carpetas seleccionadas se incluyen como carpetas físicas en nuestro proyecto, no como grupos. Las carpetas físicas aparecen con el icono de carpeta de color azul.



Carpeta de recursos

En este caso la propia carpeta (con todo su contenido) será incluida en el paquete (*bundle*) generado. Si consultamos *Build Phases* veremos que lo que se copia al paquete en este caso es la carpeta completa.



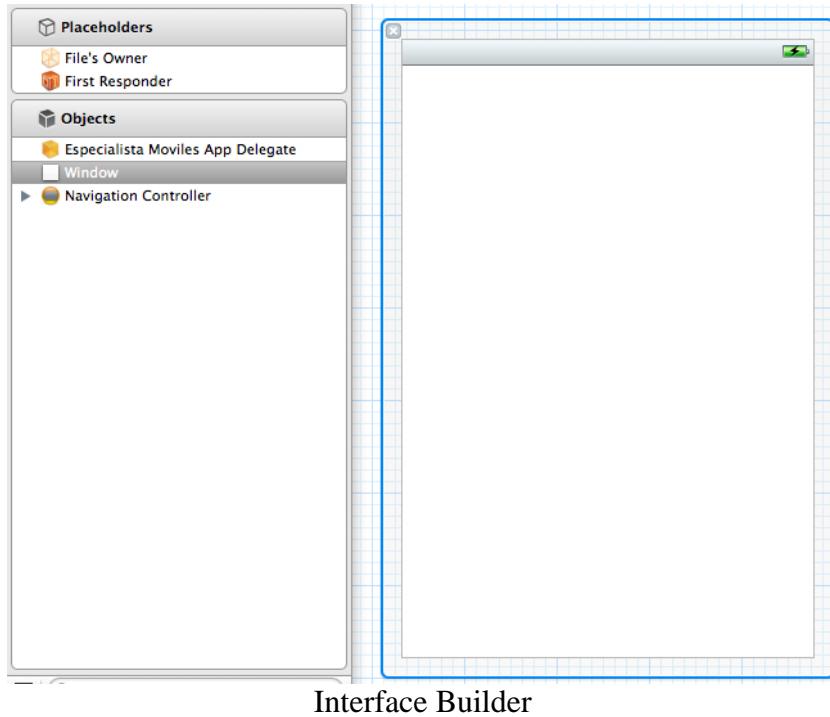
Copia de recursos en carpetas

Todos los recursos añadidos pueden ser localizados también como hemos visto anteriormente.

1.7. Interface Builder

Si en el navegador seleccionamos un fichero de tipo *xib*, en los que se define la interfaz, en el editor se abrirá la herramienta Interface Builder, que nos permitirá editar dicha

interfaz de forma visual. En versiones anteriores de Xcode Interface Builder era una herramienta independiente, pero a partir de Xcode 4 se encuentra integrado en el mismo entorno.

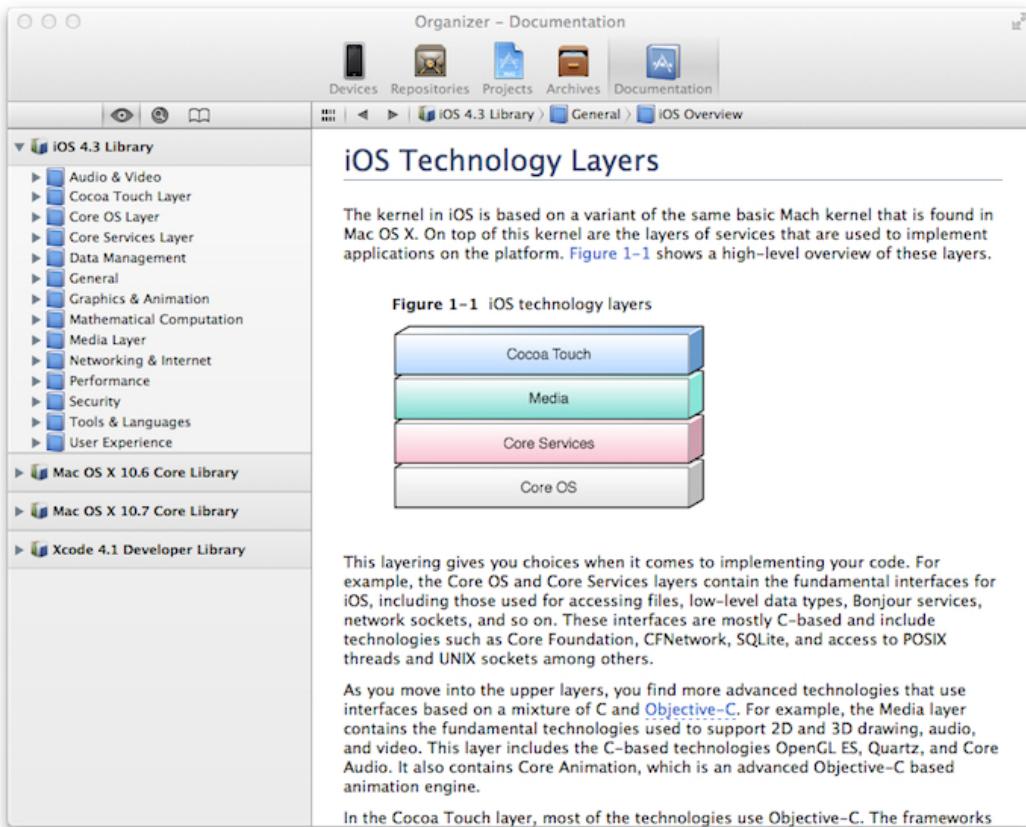


Veremos esta herramienta con más detalle cuando estudiemos la creación de la interfaz de las aplicaciones iOS.

1.8. Organizer

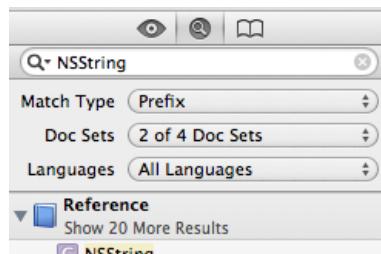
El *organizer* es una ventana independiente de la interfaz principal de Xcode desde donde podemos gestionar diferentes elementos como los dispositivos reales de los que disponemos para probar la aplicación, los repositorios de control de fuentes (SCM) que utilizamos en los proyectos, los archivos generados para la publicación de aplicaciones, y la documentación sobre las herramientas y las APIs con las que contamos en la plataforma iOS.

Podemos abrir *organizer* pulsando sobre el botón en la esquina superior derecha de la ventana principal de Xcode.



Ventana del Organizer

El uso más frecuente que le daremos el *organizer* será el de consultar la documentación de la plataforma iOS. La navegación por la documentación se hará de forma parecida a como se haría en un navegador. Normalmente nos interesarán buscar un determinado ítem. Para ello, en la barra lateral izquierda podemos especificar los criterios de búsqueda:



Búsqueda en la documentación

Es recomendable abrir los criterios avanzados de búsqueda y en *Doc Sets* eliminar los conjuntos de documentación de Mac OS X, ya que no nos interesarán para el desarrollo en iOS y así agilizaremos la búsqueda.

1.9. Repositorios SCM

Anteriormente hemos visto que Xcode nos pone muy fácil crear un repositorio Git para nuestro proyecto, ya que basta con marcar una casilla durante la creación del proyecto. Sin embargo, si queremos utilizar un repositorio remoto deberemos realizar una configuración adicional. Las opciones para trabajar con repositorios en Xcode son bastante limitadas, y en muchas ocasiones resulta poco intuitiva la forma de trabajar con ellas. Vamos a ver a continuación cómo trabajar con repositorios Git y SVN remotos.

Para gestionar los repositorios deberemos ir a la pestaña *Repositories* del *Organizer*. Ahí veremos en el lateral izquierdo un listado de los repositorios con los que estamos trabajando. Si hemos marcado la casilla de utilizar Git al crear nuestros proyectos, podremos ver aquí la información de dichos repositorios y el historial de revisiones de cada uno de ellos.

1.9.1. Repositorios Git

Si hemos creado un repositorio Git local para nuestro proyecto, será sencillo replicarlo en un repositorio Git remoto. En primer lugar, necesitaremos crear un repositorio remoto. Vamos a ver cómo crear un repositorio privado en BitBucket (bitbucket.org).

- En primer lugar, deberemos crearnos una cuenta en BitBucket, si no disponemos ya de una.
- Creamos desde nuestra cuenta de BitBucket un repositorio (*Repositories > Create repository*).
- Deberemos darle un nombre al repositorio. Será de tipo Git y como lenguaje especificaremos Objective-C.

The screenshot shows the BitBucket interface for creating a new repository. At the top, there's a navigation bar with the BitBucket logo, the word 'Atlassian', and tabs for 'Explore', 'Dashboard', and 'Repositories'. Below this is a large blue header bar with the text 'Create new repository' and 'Start from scratch.' In the main form area, there are several input fields and options:

- Owner:** A dropdown menu showing 'malozano'.
- Name (required):** An input field containing 'Prueba'. To its right are icons for 'Issue tracking' (a red square with green dots) and 'Private' (a lock icon). A checked checkbox next to 'Private' indicates it is selected.
- Repository type:** A radio button group where 'Git' is selected (indicated by a blue circle) and 'Mercurial' is unselected (indicated by a grey circle).
- Project management:** A group of checkboxes where 'Issue tracking' and 'Wiki' are unselected, and 'Private' is checked.
- Language:** A dropdown menu showing 'Objective-C'.
- Description:** A large text area for entering a description, which is currently empty.
- Website:** A text area for entering a website URL, which is currently empty.
- Create repository:** A button at the bottom left of the form.

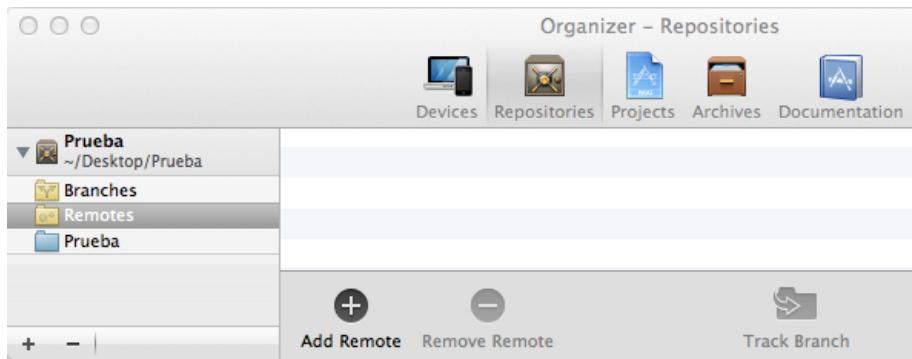
Creación de un repositorio BitBucket

- Una vez hecho esto, veremos el repositorio ya creado, en cuya ficha podremos encontrar la ruta que nos dará acceso a él.

The screenshot shows the details page for the 'Prueba' repository. At the top, there's a navigation bar with tabs: 'Overview' (which is active), 'Downloads (0)', 'Pull requests (0)', 'Source', 'Commits', and 'Admin'. Below the tabs, the repository owner is listed as 'malozano / Prueba'. Underneath, there's a section for cloning the repository with the text 'Clone this repository (size: 580 bytes): [HTTPS](https://malozano@bitbucket.org/malozano/prueba.git) / [SSH](ssh://git@bitbucket.org/malozano/prueba.git) / [SourceTree](#)' and the command '\$ git clone https://malozano@bitbucket.org/malozano/prueba.git'.

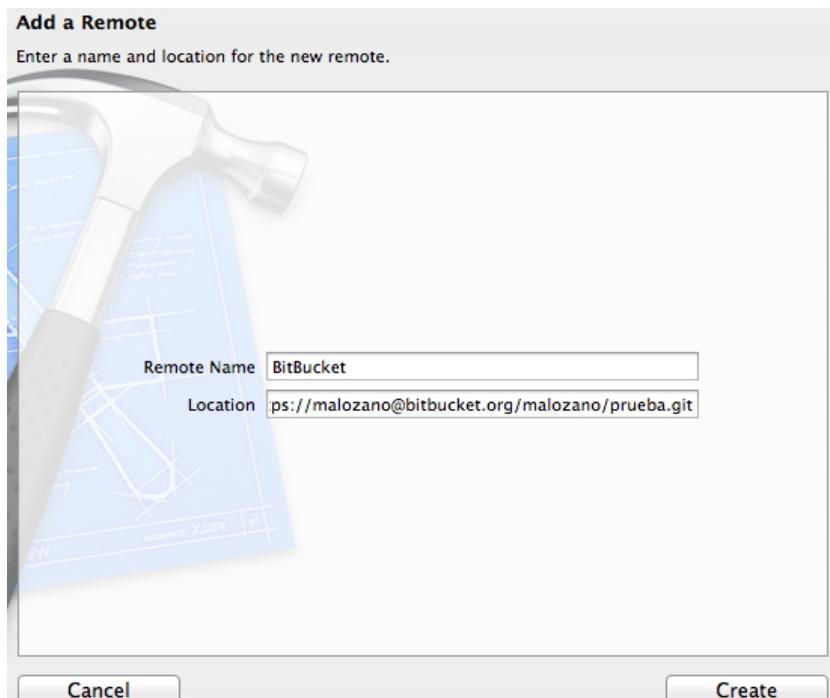
Ficha del repositorio en BitBucket

A continuación volvemos a Xcode, donde tenemos un proyecto ya creado con un repositorio Git local. Para vincular este repositorio local con el repositorio remoto que acabamos de crear, deberemos ir a la pestaña *Repositories* de la ventana del *Organizer*, donde veremos el repositorio local de nuestro proyecto.



Repositorio en Organizer

Dentro de este repositorio local, veremos una carpeta *Remotes*. Entraremos en ella, y pulsaremos el botón *Add Remotes* de la parte inferior de la pantalla. Ahora nos pedirá un nombre para el repositorio remoto y la ruta en la que se puede localizar. Utilizaremos aquí como ruta la ruta de tipo HTTPS que vimos en la ficha del repositorio de BitBucket:



Añadir repositorio remoto

Una vez añadido el repositorio remoto, deberemos introducir el login y password que nos den acceso a él. Con esto ya podemos volver al proyecto en Xcode, y enviar los cambios al servidor. En primer lugar deberemos hacer un *Commit* de los cambios en el repositorio local, si los hubiera (*File > Source Control > Commit...*). Tras esto, ya podremos subir estos cambios al repositorio remoto (*Push*). Para ello, seleccionamos la opción *File > Source Control > Push...*, tras lo cual nos pedirá que indiquemos el repositorio remoto en el que subir los cambios. Seleccionaremos el repositorio que acabamos de configurar.



Push en repositorio remoto

Tras realizar esta operación en la web de BitBucket podremos ver el código que hemos subido al repositorio:

Código en BitBucket

1.9.1.1. Compartir un proyecto ya existente

Hemos visto que es muy fácil crear un repositorio Git para el proyecto simplemente marcando una casilla en el asistente de creación del proyecto. Pero, ¿qué ocurre si ya tenemos creado un proyecto Xcode y no está vinculado a ningún repositorio? En este caso el entorno Xcode no nos permitirá crear un repositorio para este proyecto, pero podemos hacerlo sin ningún problema en línea de comando. En primer lugar, desde el directorio principal de nuestro proyecto (donde se encuentra el fichero `.xcodeproj`) inicializaremos el repositorio Git local con:

```
git init
```

Una vez hecho esto, añadiremos al repositorio los elementos del proyecto. El proyecto se compone de un fichero `.xcodeproj`, que realmente es un paquete con distintos elementos para la configuración del proyecto, y un directorio con el nombre del proyecto, que es donde se encontrarán los componentes creados por nosotros. Deberemos añadir el repositorio el directorio con nuestros contenidos, y del fichero `.xcodeproj` sólo nos quedaremos con el elemento `project.pbxproj`, ya que el resto de elementos de configuración están continuamente sometidos a cambios y sólo hacer referencia a nuestra instalación local, por lo que no es recomendable subirlos a un sistema de gestión de versiones. Por ejemplo, si nuestro proyecto se llama `Especialista` utilizaríamos el siguiente comando:

```
git add Especialista Especialista.xcodeproj/project.pbxproj
```

Tras añadir los ficheros al repositorio, podemos hacer el primer commit al repositorio local, especificando el mensaje de introducir en la revisión con el parámetro `-m`:

```
git commit -m "Initial commit"
```

Este procedimiento es el mismo que hace Xcode cuando marcamos la casilla para utilizar un repositorio Git al crear el proyecto. Ahora podemos abrir de nuevo el proyecto con Xcode, y veremos que el entorno ya reconoce el sistema de gestión de versiones. En Organizer veremos el repositorio que acabamos de crear.

Ahora podremos configurar el repositorio remoto tal como se ha comentado anteriormente, pero también se puede hacer desde línea de comando. Podemos vincular nuestro proyecto a un repositorio Git remoto de la siguiente forma:

```
git remote add origin https://malozano@bitbucket.org/malozano/prueba.git
```

Con esto añadimos un repositorio remoto con nombre `origin` y con la URL de nuestro repositorio BitBucket. Si ahora queremos subir los cambios locales al repositorio remoto podemos lanzar el comando `push`, especificando el nombre del repositorio remoto (`origin` en nuestro caso) y el nombre de la rama (`master`) a los que vamos a subir el proyecto:

```
git push origin master
```

Al configurar el repositorio remoto en línea de comando también veremos que aparece configurado en el entorno Xcode, y podremos subir los cambios mediante este entorno.

Alternativa. Si tenemos creado un repositorio remoto en BitBucket, en lugar de inicializar el repositorio como local con `git init`, podemos directamente clonar el repositorio remoto en nuestro directorio:

```
git clone https://malozano@bitbucket.org/malozano/prueba.git
```

Con esto tendremos inicializado el repositorio Git, y al mismo tiempo estará conectado ya con el repositorio remoto indicado (que por defecto tendrá como nombre `origin`). Por lo

tanto, no será necesario llamar posteriormente a `git remote add`.

1.9.1.2. Repositorio multiproyecto

Si queremos subir varios proyectos a un mismo repositorio deberemos utilizar la línea de comando para configurarlos. En este caso podemos crear un directorio donde introduciremos todos nuestros proyectos, e inicializaremos como repositorio Git dicho directorio (no cada proyecto individualmente). Por ejemplo, podemos crearnos un directorio `intro-ios` y dentro de él introducir todos los proyectos del módulo. Desde dicho directorio lanzaremos el comando para inicializarlo como repositorio (o bien también podríamos clonar nuestro repositorio remoto):

```
git init
```

A partir de este momento, cualquier proyecto que se cree dentro de este directorio pasará a formar parte del repositorio, sin tener que marcar la casilla para inicializarlo como repositorio Git (veremos que ni siquiera nos deja hacerlo).

```
intro-ios/
    Proyecto1
    Proyecto2
    ...
    ProyectoN
```

Al abrir un proyecto contenido en `intro-ios` desde Xcode veremos que en Organizer aparece un repositorio `intro-ios`. Todos los proyectos que estén dentro de dicho directorio compartirán repositorio. Podemos hacer un *commit* del proyecto, y entonces los cambios se guardarán en el repositorio. Al hacer esto es importante asegurarse de que no estén marcados los ficheros `project.xcworkspace` y `xcuserdata`, ya que como hemos comentado antes estos son directorios de configuración local con cambios continuos que no deben estar gestionados por Git.

Si queremos subir nuestro repositorio multiproyecto local a uno remoto deberemos configurarlo desde línea de comando tal como vimos anteriormente. En este caso, desde el directorio `intro-ios` lanzaremos:

```
git remote add origin https://malozano@bitbucket.org/malozano/prueba.git
```

Nota

Esto no será necesario si el repositorio local fue inicializado clonando el repositorio remoto con `git clone`.

Ahora desde el entorno podremos hacer *push* y *pull* desde cualquier proyecto creado o copiado dentro de `intro-ios`.

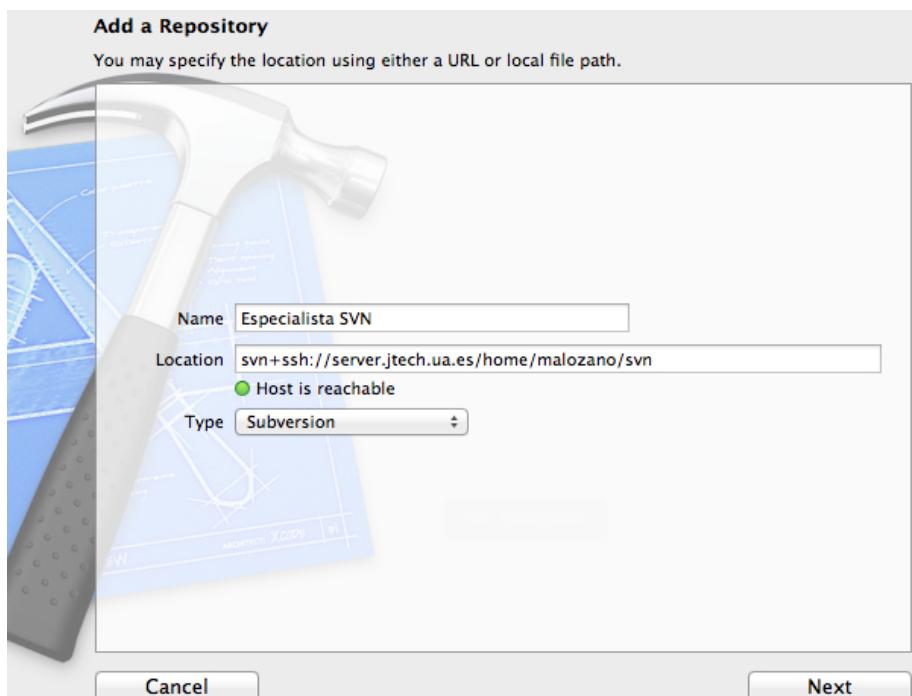
Advertencia

Hay que tener en cuenta de que si varios proyectos comparten repositorio, si en cualquiera de ellos tenemos cambios pendiente de hacer *commit*, no se podrá hacer *push* para ningún proyecto.

1.9.2. Repositorios SVN

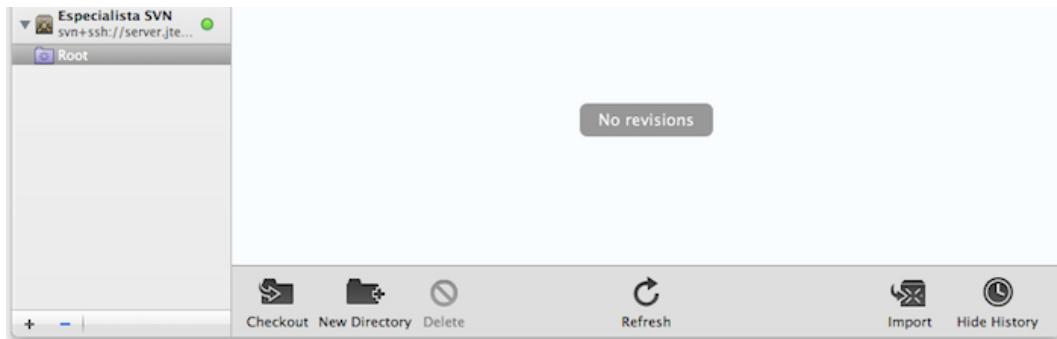
Vamos a ver la forma de acceder a un repositorio SVN remoto. Si queremos añadir un nuevo repositorio de forma manual deberemos pulsar el botón (+) que hay en la esquina inferior izquierda de la pantalla de repositorios de *Organizer*. Una vez pulsado el botón (+) y seleccionada la opción *Add Repository ...*, nos aparecerá la siguiente ventana donde introducir los datos de nuestro repositorio:

Le daremos un nombre y pondremos la ruta en la que se encuentra. Tras esto, pulsamos *Next*, y nos pedirá el login y password para acceder a dicho repositorio. Una vez introducidos, nos llevará a una segunda pantalla donde nos pedirá las rutas de `trunk`, `branches`, y `tags` en el repositorio. Si el repositorio está todavía vacío, dejaremos estos campos vacíos y pulsaremos *Add* para añadir el nuevo repositorio.



Configurar un nuevo repositorio SVN

Una vez creado, lo veremos en el lateral izquierdo con un elemento *Root* que corresponderá a la raíz del repositorio. Si seleccionamos esta ruta raíz, en la zona central de la pantalla veremos su contenido y podremos crear directorios o importar ficheros mediante los botones de la barra inferior:



Acceso al repositorio SVN

Aquí podemos crear el *layout* del repositorio. Para repositorios SVN se recomienda que en el raíz de nuestro proyecto tengamos tres subdirectorios: `trunk`, `branches`, y `tags`. Podemos crearlos con el botón *New Directory*.

Una vez creados los directorios, pinchamos sobre el ítem con el nombre de nuestro repositorio en la barra de la izquierda e indicamos la localización de los directorios que acabamos de crear. Si ponemos bien las rutas junto a ellas aparecerá una luz verde:



Configuración del layout del repositorio

Además de `Root`, ahora veremos en nuestro repositorio también las carpetas de los elementos que acabamos de crear. El código en desarrollo se guardará en `trunk`, por lo que es ahí donde deberemos subir nuestro proyecto.

Por desgracia, no hay ninguna forma de compartir un proyecto de forma automática desde el entorno Xcode, como si que ocurre con Eclipse. De hecho, las opciones para trabajar con repositorios SCM en Xcode son muy limitadas y en muchas ocasiones en la documentación oficial nos indican que debemos trabajar desde línea de comando. Una de las situaciones en las que debe hacerse así es para subir nuestro proyecto por primera vez.

Vamos a ver una forma alternativa de hacerlo utilizando el *Organizer*, en lugar de línea de comando.

- Lo primero que necesitaremos es tener un proyecto creado con Xcode en el disco (cerraremos la ventana de Xcode del proyecto, ya que no vamos a seguir trabajando)

con esa copia).

- En el *Organizer*, entramos en el directorio `trunk` de nuestro repositorio y aquí seleccionamos *Import* en la barra inferior.
- Nos abrirá un explorador de ficheros para seleccionar los ficheros que queremos subir al repositorio, seleccionaremos el directorio raíz de nuestro proyecto (el directorio que contiene el fichero `xcodeproj`).
- Con esto ya tenemos el proyecto en nuestro repositorio SVN, pero si ahora abrimos el proyecto del directorio de nuestro disco local en el que está guardado, Xcode no reconocerá la conexión con el repositorio, ya que no es lo que se conoce como una *working copy* (no guarda la configuración de conexión con el repositorio SVN).
- Para obtener una *working copy* deberemos hacer un *checkout* del proyecto. Para ello desde el *Organizer*, seleccionamos la carpeta *Trunk* de nuestro repositorio, dentro de ella nuestro proyecto, y pulsamos el botón *Checkout* de la barra inferior. Nos pedirá un lugar del sistema de archivos donde guardar el proyecto, y una vez descargado nos preguntará si queremos abrirlo con Xcode.

En los elementos del repositorio ahora veremos una carpeta azul con el nombre del proyecto descargado. Esta carpeta azul simboliza una *working copy* del proyecto. Al tener esta *working copy* reconocida, ya podremos realizar desde el mismo entorno Xcode operaciones de SVN en dicho proyecto.

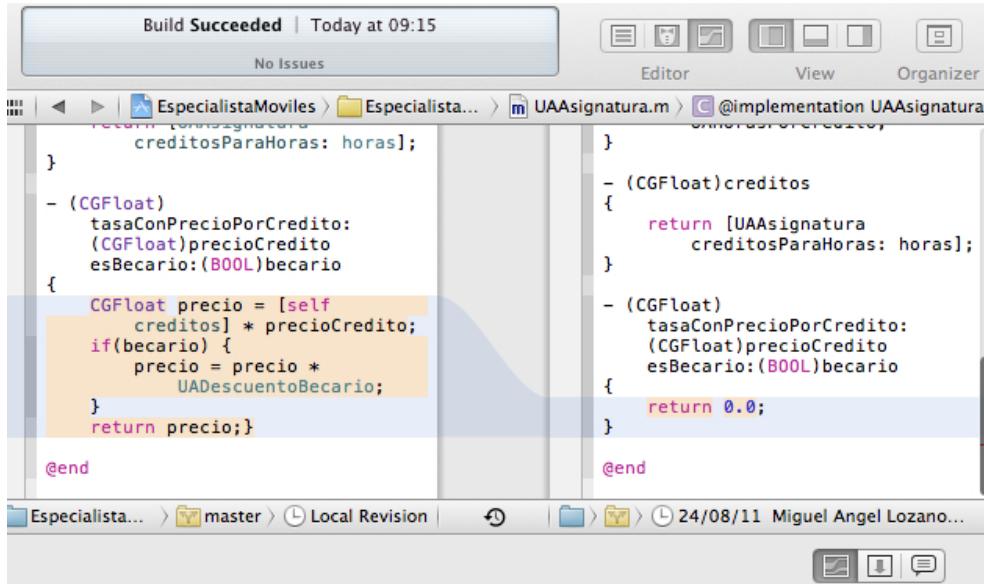


Cada vez que hagamos un cambio en un fichero, en el navegador de Xcode aparecerá una **M** a su lado indicando que hay modificaciones pendientes de ser enviadas, y de la misma forma, cuando añadamos un fichero nuevo, junto a él aparecerá una **A** para indicar que está pendiente de ser añadido al repositorio. Para enviar estos cambios al repositorio, desde Xcode seleccionaremos *File > Source Control > Commit*.

Si ya tuviesemos una *working copy* válida de un proyecto en el disco (es decir, una copia que contenga los directorios `.svn` con la configuración de acceso al repositorio), podemos añadirla al *Organizer* directamente desde la vista de repositorios, pulsando el botón (+) de la esquina inferior izquierda y seleccionando *Add Working Copy....* Nos pedirá la ruta local en la que tenemos guardada la *working copy* del proyecto.

A parte de las opciones para el control de versiones que encontramos al pulsar con el botón derecho sobre nuestro proyecto en el apartado *Source Control*, en Xcode también tenemos el editor de versiones. Se trata de una vista del editor que nos permite comparar diferentes versiones de un mismo fichero, y volver atrás si fuese necesario. Para abrir este editor utilizaremos los botones de la esquina superior derecha de la interfaz, concretamente el tercer botón del grupo *Editor*. En el editor de versiones podremos comparar la revisión local actual de cada fichero con cualquiera de las revisiones

anteriores.



```

Build Succeeded | Today at 09:15
No Issues
Editor View Organizer
EspecialistaMoviles > Especialista... > UAAsignatura.m @implementation UAAsignatura
    return [UAAsignatura
        creditosParaHoras: horas];
}
- (CGFloat)
    tasaConPrecioPorCredito:
    (CGFloat)precioCredito
    esBecario:(BOOL)becario
{
    CGFloat precio = [self
        creditos] * precioCredito;
    if(becario) {
        precio = precio *
        UADescuentoBecario;
    }
    return precio;
}
@end
- (CGFloat)
    creditosParaHoras:
    (CGFloat)creditos
{
    return [UAAsignatura
        creditosParaHoras: horas];
}
- (CGFloat)
    tasaConPrecioPorCredito:
    (CGFloat)precioCredito
    esBecario:(BOOL)becario
{
    return 0.0;
}
@end

```

Especialista... > master > Local Revision 24/08/11 Miguel Angel Lozano...

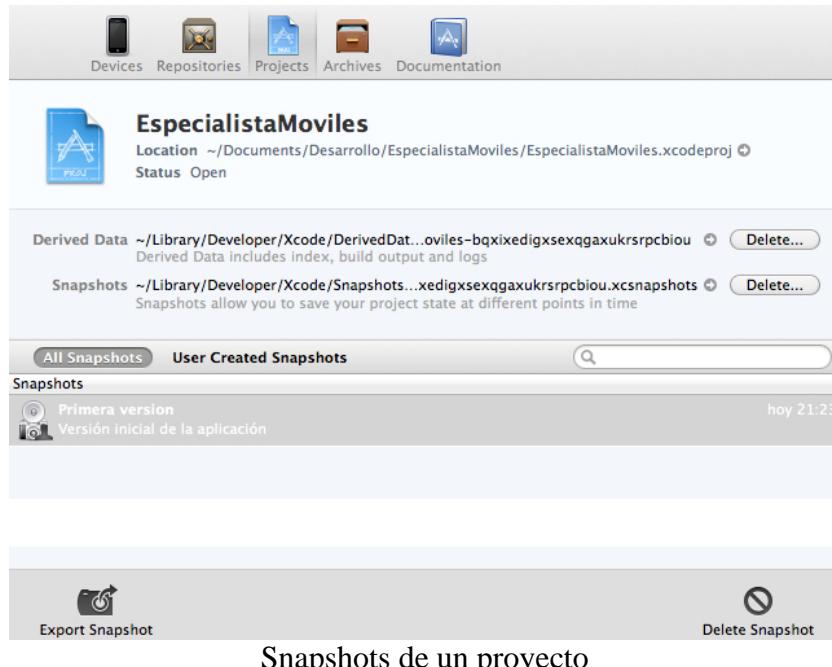
Editor de versiones

1.10. Snapshots

Una forma más rudimentaria de guardar copias de seguridad de los proyectos es la generación de *snapshots*. Un *snapshot* no es más que la copia de nuestro proyecto completo en un momento dado. Xcode se encarga de archivar y organizar estas copias.

Para crear un *snapshot* debemos seleccionar en Xcode *File > Create Snapshot ...*, para el cual nos pedirá un nombre y una descripción.

En la vista *Projects* del *Organizer* veremos para cada proyecto la lista de *snapshots* creados.



Desde aquí podremos exportar un *snapshot* para guardar el proyecto en el lugar del disco que especifiquemos tal como estaba en el momento en el que se tomó la instantánea, y así poder abrirlo con Xcode. En esta pantalla también podemos limpiar los directorios de trabajo del proyecto y borrar los *snapshots*.

Desde Xcode, también se puede restaurar un *snapshot* con la opción *File > Restore snapshot....* Tendremos que seleccionar el *snapshot* que queremos restaurar.

1.11. Ejecución y firma

Como hemos comentado anteriormente, para probar las aplicaciones en dispositivos reales necesitaremos un certificado con el que firmarlas, y para obtener dicho certificado tendremos que ser miembros de pago del *iOS Developer Program*, o bien pertenecer al *iOS University Program*. Para acceder a este segundo programa, se deberá contar con una cuenta Apple, y solicitar al profesor responsable una invitación para acceder al programa con dicha cuenta. El programa es gratuito, pero tiene la limitación de que no podremos publicar las aplicaciones en la App Store, sólo probarlas en nuestros propios dispositivos.

Vamos a ver a continuación cómo obtener dichos certificados y ejecutar las aplicaciones en dispositivos reales.

1.11.1. Creación del par de claves

El primer paso para la obtención del certificado es crear nuestras claves privada y pública. Para ello entraremos en el portal de desarrolladores de Apple:

<http://developer.apple.com>

Accedemos con nuestro usuario, que debe estar dado de alta en alguno de los programas comentados anteriormente.

The screenshot shows the iOS Dev Center homepage. At the top, there's a navigation bar with links for Technologies, Resources, Programs, Support, and Member Center. A search bar is also present. Below the navigation bar, there's a banner for 'iOS SDK 4.3' and a section for 'Resources for iOS 4.3' which includes 'Downloads' and 'Getting Started Videos'. To the right, there's a 'Featured Content' section with links to 'New Subscription Service for iOS Apps' and 'Getting Ready for iOS 4.3'. On the far right, there's a sidebar for the 'iOS Developer Program' with links to 'iOS Provisioning Portal', 'Apple Developer Forums', and 'Developer Support Center'. The user 'Hi, Miguel Angel Lozano Ortega' is logged in, along with 'My Profile' and 'Log out' options. The overall title of the page is 'Developer Center'.

Desde este portal, accedemos al *iOS Provisioning Portal*.

The screenshot shows the 'Provisioning Portal : Universidad de Alicante (Dpto. de Ciencia de la Computacion e I.A.)' homepage. On the left, there's a sidebar with links for Home, Certificates, Devices, App IDs, and Provisioning. The main content area has a heading 'Welcome to the iOS Provisioning Portal' and a message stating that the portal is designed to take you through the necessary steps to test your applications on iOS devices and prepare them for distribution. Below this, there's a callout box with an exclamation mark icon that says 'Visit the Member Center for Team, Account, and Program info'. It explains that the new Member Center is now the destination for sending invitations, requesting or purchasing technical support, and viewing account information. There's a link 'Visit the Member Center now'. At the bottom, there's another box with an iPhone icon and the text 'Get your application on an iOS with the Development Provisioning Assistant'. It explains that as a Program Admin, you can use the Development Provisioning Assistant to create and install a Provisioning Profile and iOS Development Certificate needed to build and install applications for iOS devices. There's a button 'Launch Assistant'.

Provisioning Portal

Desde aquí podemos gestionar nuestros certificados y los dispositivos registrados para poder utilizarlos en el desarrollo de aplicaciones. Vemos también que tenemos disponible

un asistente para crear los certificados por primera vez. El asistente se nos irá dando instrucciones detalladas y se encargará de configurar todo lo necesario para la obtención de los certificados y el registro de los dispositivos. Sin embargo, con Xcode 4 el registro de los dispositivos se puede hacer automáticamente desde el *Organizer*, así que nos resultará más sencillo si en lugar del asistente simplemente utilizamos el *iOS Provisioning Portal* para obtener el certificado del desarrollador, y dejamos el resto de tareas a *Organizer*.

The screenshot shows the iOS Provisioning Portal interface. At the top, it says "Welcome, Miguel Angel Lozano Ortega | Edit Profile | Log out". Below that, it says "Provisioning Portal : Universidad de Alicante (Dpto. de Ciencia de la Computacion e I.A.) | Go to iOS Dev Center". On the left, there's a sidebar with links: Home, Certificates (which is selected and highlighted in blue), Devices, App IDs, and Provisioning. In the main area, there are tabs: Development (selected), History, and How To. Under "Development", it says "Current Development Certificates". There's a section titled "Your Certificate" with a small icon. A message says "You currently do not have a valid certificate" and has a "Request Certificate" button. Below this, a note says "*If you do not have the WWDR Intermediate certificate installed, click here to download now.".

Gestión de certificados de desarrollador

Entraremos por lo tanto en *Certificates*, y dentro de la pestaña *Development* veremos los certificados con los que contamos actualmente (si es la primera vez que entramos no tendremos ninguno), y nos permitirá solicitarlos o gestionarlos. Antes de solicitar un nuevo certificado, descargaremos desde esta página el certificado WWDR, y haremos doble *click* sobre él para instalarlo en nuestros llaveros (*keychain*). Este certificado es necesario porque es el que valida los certificados de desarrollador que emite Apple. Sin él, el certificado que generemos no sería de confianza.

Una vez descargado e instalado dicho certificado intermedio, pulsaremos sobre el botón *Request Certificate* para comenzar con la solicitud de un nuevo certificado de desarrollador.

Nos aparecerán las instrucciones detalladas para solicitar el certificado. Deberemos utilizar la herramienta *Acceso a Llaveros (Keychain Access)* para realizar dicha solicitud, tal como explican las instrucciones. Abrimos la herramienta y accedemos a la opción del menú *Acceso a Llaveros > Asistente para Certificados > Solicitar un certificado de una autoridad de certificación*

Información del certificado

Introduzca información para el certificado que está solicitando.
Haga clic en continuar para solicitar un certificado de la CA.

Dirección de correo del usuario:	<input type="text" value="malozano@ua.es"/>
Nombre común:	<input type="text" value="Miguel Angel Lozano"/>
Dirección de correo de la CA:	<input type="text"/>
La palabra clave:	<input type="radio"/> Se envía por correo electrónico a la CA <input checked="" type="radio"/> Se guarda en el disco <input type="checkbox"/> Permitirme especificar la información del par de llaves

Continuar

Solicitud de un certificado de desarrollador

Aquí tendremos que poner nuestro *e-mail*, nuestro nombre, e indicar que se guarde en el disco. Una vez finalizado, pulsamos *Continuar* y nos guardará un fichero *.certSigningRequest* en la ubicación que seleccionemos.

Con esto habremos generado una clave privada, almacenada en los llaveros, y una clave pública que se incluye en el fichero de solicitud de certificado. Volvemos ahora al *iOS Provisioning Portal*, y bajo las instrucciones para la solicitud del certificado vemos un campo para enviar el fichero. Enviaremos a través de dicho campo el fichero de solicitud (*.certSigningRequest*) generado, y en la página *Certificates > Development* aparecerá nuestro certificado como pendiente. Cuando Apple emita el certificado podremos descargarlo a través de esta misma página (podemos recargarla pasados unos segundos hasta que el certificado esté disponible).

[Development](#) [History](#) [How To](#)

Current Development Certificates

 **Your Certificate**

Name	Provisioning Profiles	Expiration Date	Status	Action
Miguel Angel Lozano Ortega		Aug 23, 2012	Issued	Download Revoke

*If you do not have the WWDR intermediate certificate installed, click here to download now.

Certificado listo para descargar

Una vez esté disponible, pulsaremos el botón *Download* para descargarlo y haremos doble *click* sobre él para instalarlo en nuestros llaveros. Una vez hecho esto, en *Acceso a*

Llaveros > Inicio de sesión > Mis certificados deberemos ver nuestro certificado con la clave privada asociada.



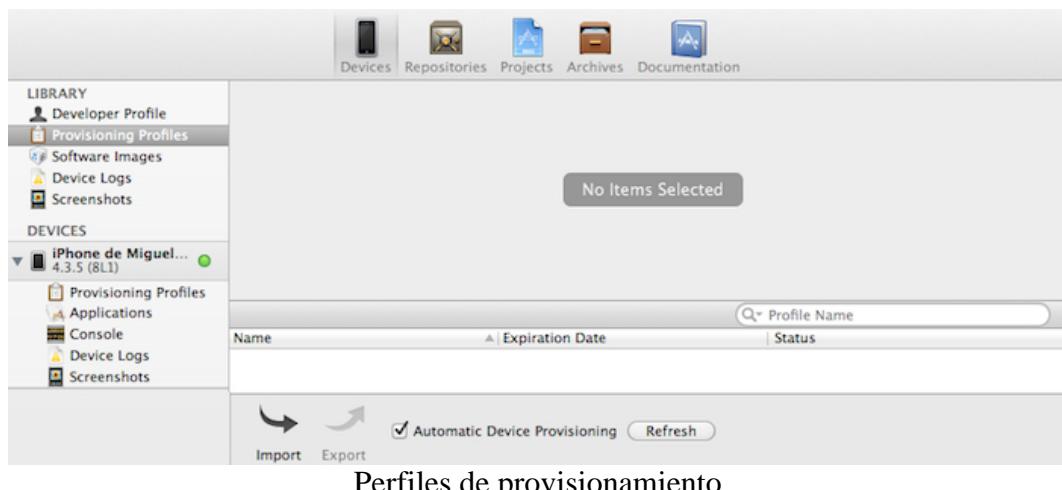
Importante

El certificado generado no podrá utilizarse en otros ordenadores para firmar la aplicación, ya que la clave privada está incluida en nuestros llaveros, pero no en el certificado, y será necesaria para poder firmar. Si queremos trabajar en otro ordenador tendremos que exportar la clave privada desde *Acceso a Llaveros*.

1.11.2. Perfil de provisionamiento

Una vez contamos con nuestro certificado de desarrollador instalado y con su clave privada asociada en nuestro llavero, podemos crear un perfil de provisionamiento para poder probar nuestras aplicaciones en nuestros dispositivos.

Para ello abriremos el *Organizer* e iremos a la sección *Devices*. Aquí entraremos en *Developer Profiles* y comprobaremos que ha reconocido correctamente nuestro perfil de desarrollador tras instalar el certificado. Ahora lo que necesitamos son perfiles de provisionamiento para poder instalar la aplicación en dispositivos concretos. Si entramos en *Provisioning Profiles* veremos que no tenemos ninguno por el momento, y nos aseguraremos que en la parte inferior tengamos marcado *Automatic Device Provisioning*.



Ahora conectaremos mediante el cable USB el dispositivo que queramos registrar y veremos que aparece en la barra de la izquierda. Pulsamos sobre el nombre del dispositivo en dicha barra y abrirá una pantalla con toda su información, entre ella, la versión de iOS y el identificador que se utilizará para registrarla en el *iOS Provisioning*

Portal.



Gestión de los dispositivos

Vemos que no tiene ningún perfil de provisionamiento disponible, pero abajo vemos un botón *Add to Portal* que nos permite registrar automáticamente nuestro dispositivo para desarrollo. Al pulsarlo nos pedirá nuestro *login* y *password* para acceder al portal de desarrolladores, y automáticamente se encargará de registrar el dispositivo, crear los perfiles de provisionamiento necesarios, descargarlos e instalarlos en Xcode.

Nota

Si estamos usando el *iOS University Program* al final del proceso, cuando esté obteniendo el perfil de distribución, nos saldrá un mensaje de error indicando que nuestro programa no permite realizar dicha acción. Esto es normal, ya que el perfil universitario no permite distribuir aplicaciones, por lo que la generación del perfil de distribución deberá fallar. Pulsaremos *Aceptar* y el resto del proceso se realizará correctamente.

Si ahora volvemos al *iOS Developer Portal* veremos que nuestro dispositivo se encuentra registrado ahí (sección *Devices*), y que contamos con un perfil de provisionamiento (sección *Provisioning*).

Ahora podemos volver a Xcode y ejecutar nuestra aplicación en un dispositivo real. Para ello seleccionamos nuestro dispositivo en el cuadro junto a los botones de ejecución, y pulsamos el botón *Run*.



Ejecutar en un dispositivo real

2. Ejercicios de Xcode

2.1. Creación de un proyecto con Xcode (1 punto)

a) Vamos a crear un nuevo proyecto de tipo *Master-Detail Application*. El nombre de producto será `Filmoteca` y nuestra organización `es.ua.jtech`. Utilizaremos como prefijo para todas las clases `UA`, y sólo destinaremos la aplicación al iPhone por el momento. Dejaremos marcada la casilla para crear **pruebas de unidad**, pero desmarcaremos el resto (Core Data, ARC, Storyboards), ya que no vamos a utilizar por el momento ninguna de esas características .

Importante

Deja marcada la casilla para utilizar un **repositorio Git local**, ya que esto será necesario para el siguiente ejercicio.

b) ¿Qué versión de iOS requiere la aplicación? Modifica la configuración necesaria para que funcione en dispositivos con iOS 4.0.

c) Ejecuta la aplicación en el simulador de iPhone. Comprueba que funciona correctamente.

2.2. Repositorios remotos (0 puntos)

Vamos ahora a conectar nuestro proyecto con un repositorio remoto en BitBucket, y a subir los cambios a dicho repositorio. Esto lo haremos directamente desde el entorno Xcode. Para ello se pide:

a) Crear un repositorio en BitBucket para el módulo. Recuerda que debe llevar como nombre el nombre corto del módulo, en este caso `objc`. El lenguaje será Objective-C.

b) Desde la ventana del *Organizer*, en la pestaña *Repositories*, seleccionaremos el repositorio Git local que se ha creado para nuestro proyecto `Filmoteca`. Dentro de él veremos una carpeta *Remotes*. Pulsamos sobre ella, y en la parte inferior de la pantalla veremos un botón *Add Remote*. Pulsaremos sobre este botón para introducir nuestro repositorio de BitBucket. Como nombre utilizaremos `origin`, y la URL podemos obtenerla desde la web del repositorio en BitBucket.

c) Una vez dado de alta el repositorio, haremos un *Push* del proyecto. Deberemos ver ahora nuestro proyecto en BitBucket. A partir de este momento deberemos hacer *commits* locales cada vez que se finalice un apartado de un ejercicio, y *push* al terminar la clase o terminar los ejercicios.

Cuidado

Xcode no nos permite hacer *push* si tenemos cambios pendientes de ser confirmados (*commit*) en el repositorio local. Deberemos hacer *commit* siempre antes de realizar *push*.

2.3. Iconos y recursos (1 punto)

Vamos a añadir ahora un ícono al proyecto, y una imagen a mostrar durante la carga de la aplicación. Estas imágenes se proporcionan en el fichero de plantillas de la sesión. Se pide:

- a) Añadir la imagen `icono.png` como ícono de la aplicación. Para hacer esto podemos seleccionar en el navegador el nodo principal del proyecto, y en la pestaña *Summary* del *target* principal veremos una serie de huecos para indicar los íconos. Bastará con arrastrar la imagen al hueco correspondiente.
- b) Añade la imagen `icono@2x.png` como ícono para los dispositivos con pantalla retina. Puedes hacer que el simulador tenga pantalla retina, mediante la opción *Hardware > Dispositivo > iPhone (Retina)*. Comprueba que la imagen se ve con la definición adecuada a esta versión del simulador.
- c) Añade las imágenes `Default.png` y `Default@2x.png` como imágenes a mostrar durante el tiempo de carga de la aplicación (*Launch Images*). Comprueba que aparecen correctamente.

2.4. Localización (1 punto)

Vamos ahora a localizar los textos de nuestra aplicación. Se pide:

- a) Añade a la aplicación las localizaciones a inglés y español.
- b) Haz que el nombre que se muestra bajo el ícono de la aplicación en el iPhone cambie según el idioma del dispositivo (`Filmoteca`, `FilmLibrary`). Puedes probar que los distintos idiomas se muestran correctamente cambiando el idioma del simulador (*Settings > General > International > Language*).
- c) No queremos localizar los ficheros `.xib`. Elimina la localización de dichos ficheros.
- d) Crea un fichero `Localizable.strings`, y en él una cadena para el título de la aplicación que tenga como identificador `"AppName"`, y como valor `Filmoteca` o `FilmLibrary`, según el idioma del dispositivo.

3. Introducción a Objective-C

Prácticamente toda la programación de aplicaciones iOS se realizará en lenguaje Objective-C, utilizando la API Cocoa Touch. Este lenguaje es una extensión de C, por lo que podremos utilizar en cualquier lugar código C estándar, aunque normalmente utilizaremos los elementos equivalentes definidos en Objective-C para así mantener una coherencia con la API Cocoa, que está definida en dicho lenguaje.

Vamos a ver los elementos básicos que aporta Objective-C sobre los elementos del lenguaje con los que contamos en lenguaje C estándar. También es posible combinar Objective-C y C++, dando lugar a Objective-C++, aunque esto será menos común.

3.1. Tipos de datos

3.1.1. Tipos de datos básicos

A parte de los tipos de datos básicos que conocemos de C (char, short, int, long, float, double, unsigned int, etc), en Cocoa se definen algunos tipos numéricos equivalentes a ellos que encontraremos frecuentemente en dicha API:

- `NSInteger` (int o long)
- `NSUInteger` (unsigned int o unsigned long)
- `CGFloat` (float o double)

Podemos asignar sin problemas estos tipos de Cocoa a sus tipos C estándar equivalentes, y al contrario.

Contamos también con el tipo booleano definido como `BOOL`, y que puede tomar como valores las constantes YES (1) y NO (0):

```
BOOL activo = YES;
```

3.1.2. Enumeraciones

Las enumeraciones resultan útiles cuando una variable puede tomar su valor de un conjunto limitado de posibles opciones. Dentro de la API de Cocoa es habitual encontrar enumeraciones, y se definen de la misma forma que en C estándar:

```
typedef enum {
    UATipoAsignaturaOptativa,
    UATipoAsignaturaObligatoria,
    UATipoAsignaturaTroncal
} UATipoAsignatura;
```

A cada elemento de la enumeración se le asigna un valor entero, empezando desde 0, y de forma incremental siguiendo el orden en el que está definida la enumeración. Podemos

también especificar de forma manual el número asignado a cada elemento.

```
typedef enum {
    UATipoAsignaturaOptativa = 0,
    UATipoAsignaturaObligatoria = 1,
    UATipoAsignaturaTroncal = 2
} UATipoAsignatura;
```

3.1.3. Estructuras

También encontramos y utilizamos estructuras de C estándar dentro de la API de Cocoa.

```
struct CGPoint {
    CGFloat x;
    CGFloat y;
};
typedef struct CGPoint CGPoint;
```

Muchas veces encontramos librerías de funciones para inicializarlas o realizar operaciones con ellas.

```
CGPoint punto = CGPointMake(x,y);
```

3.1.4. Cadenas

En C normalmente definimos una cadena entre comillas, por ejemplo "cadena". Con esto estamos definiendo un *array* de caracteres terminado en `null`. Esto es lo que se conoce como cadena C, pero en Objective-C normalmente no utilizaremos dicha representación. Las cadenas en Objective-C se representarán mediante la clase `NSString`, y los literales de este tipo en el código se definirán anteponiendo el símbolo @ a la cadena:

```
NSString* cadena = @"cadena";
```

Nota

Todas las variables para acceder a objetos de Objective-C tendrán que definirse como punteros, y por lo tanto en la declaración de la variable tendremos que poner el símbolo *. Los únicos tipos de datos que no se definirán como punteros serán los tipos básicos y las estructuras.

Más adelante veremos las operaciones que podemos realizar con la clase `NSString`, entre ellas el convertir una cadena C estándar a un `NSString`.

3.1.5. Objetos

Las cadenas en Objective-C son una clase concreta de objetos, a diferencia de las cadenas C estándar, pero las hemos visto como un caso especial por la forma en la que podemos definir en el código literales de ese tipo.

En la API de Objective-C encontramos una extensa librería de clases, y normalmente deberemos también crear clases propias en nuestras aplicaciones. Para referenciar una

instancia de una clase siempre lo haremos mediante un puntero.

```
MiClase* objeto;
```

Sin embargo, como veremos más adelante, la forma de trabajar con dicho puntero diferirá mucho de la forma en la que se hacía en C, ya que Objective-C se encarga de ocultar toda esa complejidad subyacente y nos ofrece una forma sencilla de manipular los objetos, más parecida a la forma con la que trabajamos con la API de Java.

No obstante, podemos utilizar punteros al estilo de C. Por ejemplo, podemos crearnos punteros de tipos básicos o de estructuras, pero lo habitual será trabajar con objetos de Objective-C.

Otra forma de hacer referencia a un objeto de Objective-C es mediante el tipo `id`. Cuando tengamos una variable de este tipo podremos utilizarla para referenciar cualquier objeto, independientemente de la clase a la que pertenezca. El compilador no realizará ningún control sobre los métodos a los que llamemos, por lo que deberemos llevar cuidado al utilizarlo para no obtener ningún error en tiempo de ejecución.

```
MiClase* mc = // Inicializa MiClase;
MiOtraClase* moc = // Inicializa MiOtraClase;
...
id referencia = nil;
referencia = mc;
referencia = moc;
```

Como vemos, `id` puede referenciar instancias de dos clases distintas sin que exista ninguna relación entre ellas. Hay que destacar también que las referencias a objetos con `id` no llevan el símbolo `*`.

También observamos que para indicar un puntero de objeto a nulo utilizamos `nil`. También contamos con `NULL`, y ambos se resuelven de la misma forma, pero conceptualmente se aplican a casos distintos. En caso de `nil`, nos referimos a un puntero nulo a un objeto de Objective-C, mientras que utilizamos `NULL` para otros tipos de punteros.

3.2. Directivas

También podemos incluir en el código una serie de directivas de preprocesamiento que resultarán de utilidad y que encontraremos habitualmente en aplicaciones iOS.

3.2.1. La directiva `#import`

Con esta directiva podemos importar ficheros de cabecera de librerías que utilicemos en nuestro código. Se diferencia de `#include` en que con `#import` se evita que un mismo fichero sea incluido más de una vez cuando encontremos inclusiones recursivas.

Encontramos dos versiones de esta directiva: `#import<...>` e `#import "..."`. La primera de ellas buscará los ficheros en la ruta de inclusión del compilador, por lo que utilizaremos esta forma cuando estemos incluyendo las librerías de Cocoa. Con la segunda, se incluirá también nuestro propio directorio de fuentes, por lo que se utilizará para importar el código de nuestra propia aplicación.

3.2.2. La directiva `#define`

Este directiva toma un nombre (símbolo) y un valor, y sustituye en el código todas las ocurrencias de dicho nombre por el valor indicado antes de realizar la compilación. Como valor podremos poner cualquier expresión, ya que la sustitución se hace como preprocesamiento antes de compilar.

También podemos definir símbolos mediante parámetros del compilador. Por ejemplo, si nos fijamos en las *Build Settings* del proyecto, en el apartado *Preprocessing* vemos que para la configuración *Debug* se le pasa un símbolo `DEBUG` con valor 1.

Tenemos también las directivas `#ifdef` (y `#ifndef`) que nos permiten incluir un bloque de código en la compilación sólo si un determinado símbolo está definido (o no). Por ejemplo, podemos hacer que sólo se escriban logs si estamos en la configuración *Debug* de la siguiente forma:

```
#ifdef DEBUG  
    NSLog(@"%@", @"Texto del log");  
#endif
```

Los nombres de las constantes (para ser más exactos macros) definidas de esta forma normalmente se utilizan letras mayúsculas separando las distintas palabras con el carácter subrayado '`_`' (`UPPER_CASE_UNDERSCORE`).

3.2.3. La directiva `#pragma mark`

Se trata de una directiva muy utilizada en los ficheros de fuentes de Objective-C, ya que es reconocida y utilizada por Xcode para organizar nuestro código en secciones. Con dicha directiva marcamos el principio de cada sección de código y le damos un nombre. Con la barra de navegación de Xcode podemos saltar directamente a cualquier de las secciones:

```
#pragma mark Constructores  
// Código de los constructores  
  
#pragma mark Eventos del ciclo de vida  
// Código de los manejadores de eventos  
  
#pragma mark Fuente de datos  
// Métodos para la obtención de datos  
  
#pragma mark Gestión de la memoria
```

```
// Código de gestión de memoria
```

3.2.4. Modificadores

Tenemos disponibles también modificadores que podemos utilizar en la declaración de las variables.

3.2.4.1. Modificador const

Indica que el valor de una variable no va a poder ser modificado. Se le debe asignar un valor en la declaración, y este valor no podrá cambiar posteriormente. Se diferencia de `#define` en que en este caso tenemos una variable en tiempo de compilación, y no una sustitución en preprocesamiento. En general, no es recomendable utilizar `#define` para definir las constantes. En su lugar utilizaremos normalmente `enum` o `const`.

Hay que llevar cuidado con el lugar en el que se declara `const` cuando se trate de punteros. Siempre afecta al elemento que tenga inmediatamente a la izquierda, excepto en el caso en el que esté al principio, que afectará al elemento de la derecha:

```
// Puntero variable a objeto NSString constante (MAL)
const NSString * UATitulo = @"Menu";

// Equivalente al anterior (MAL)
NSString const * UATitulo = @"Menu";

// Puntero constante a objeto NSString (BIEN)
NSString * const UATitulo = @"Menu";
```

En los dos primeros casos, estamos definiendo un puntero a un objeto de tipo `const NSString`. Por lo tanto, nos dará un error si intentamos utilizarlo en cualquier lugar en el que necesitemos tener un puntero a `NSString`, ya que el compilador los considera tipos distintos. Además, no es necesario hacer que `NSString` sea constante. Para ello en la API de Cocoa veremos que existen versiones mutables e inmutables de un gran número de objetos. Bastará con utilizar una cadena inmutable.

Las constantes se escribirán en *UpperCamelCase*, utilizando el prefijo de nuestra librería en caso de que sean globales.

3.2.4.2. Modificador static

Nos permite indicar que una variable se instancie sólo una vez. Por ejemplo en el siguiente código:

```
static NSString *cadena = @"Hola";
NSLog(cadena);
cadena = @"Adios";
```

La primera vez que se ejecute, la variable `cadena` se instanciará y se inicializará con el valor `@"Hola"`, que será lo que se escriba como *log*, y tras ello modificaremos su valor a

`@"Adios".` En las sucesivas ejecuciones, como la variable ya estaba instanciada, no se volverá a instanciar ni a inicializar, por lo que al ejecutar el código simplemente escribirá Adios.

Como veremos más adelante, este modificador será de gran utilidad para implementar el patrón *singleton*.

El comportamiento de `static` difiere según si la variable sobre la que se aplica tiene ámbito local o global. En el ámbito local, tal como hemos visto, nos permite tener una variable local que sólo se instancia una vez a lo largo de todas las llamadas que se hagan al método. En caso de aplicarlo a una variable global, indica que dicha variable sólo será accesible desde dentro del fichero en el que esté definida. Esto resultará de utilidad por ejemplo para definir constantes privadas que sólo vayan a utilizarse dentro de un determinado fichero .m. Al comienzo de dicho ficheros podríamos declararlas de la siguiente forma:

```
static NSString * const titulo = @"Menu";
```

Si no incluimos `static`, si en otro fichero se definiere otro símbolo global con el mismo nombre, obtendríamos un error en la fase de *linkado*, ya que existirían dos símbolos con el mismo nombre dentro del mismo ámbito.

3.2.4.3. Modificador extern

Si en el ejemplo anterior quisiéramos definir una constante global, no bastaría con declarar la constante sin el modificador `static` en el fichero .m:

```
NSString * const titulo = @"Menu";
```

Si sólo hacemos esto, aunque el símbolo sea accesible en el ámbito global, el compilador no va a ser capaz de encontrarlo ya que no hemos puesto ninguna declaración en los ficheros de cabecera incluidos. Por lo tanto, además de la definición anterior, tendremos que declarar dicha constante en algún fichero de cabecera con el modificador `extern`, para que el compilador sepa que dicho símbolo existe en el ámbito global.

Para concluir, listamos los tres posibles ámbitos en los que podemos definir cada símbolo:

- **Global:** Se declaran fuera de cualquier método para que el símbolo se guarde de forma global. Para que el compilador sepa que dicho símbolo existe, se deben declarar en los ficheros de cabecera con `extern`. Sólo se instancian una vez.
- **Fichero:** Se declaran fuera de cualquier método con modificador `static`. Sólo se podrá acceder a ella desde dentro del fichero en el que se ha definido, por lo que no deberemos declararlas en ningún fichero de cabecera que vaya a ser importado por otras unidades. Sólo se instancian una vez.
- **Local:** Se declaran dentro de un bloque de código, como puede ser una función, un método, o cualquier estructura que contengan los anteriores, y sólo será accesible dentro de dicho bloque. Por defecto se instanciarán cada vez que entremos en dicho

bloque, excepto si se declaran con el modificador `static`, caso en el que se instanciarán y se inicializarán sólo la primera vez que se entre.

3.3. Paso de mensajes

Como hemos comentado anteriormente, Objective-C es una extensión de C para hacerlo orientado a objetos, como es también el caso de C++. Una de las mayores diferencias entre ambos radica en la forma en la se ejecutan los métodos de los objetos. En Objective-C los métodos siempre se ejecutan de forma dinámica, es decir, el método a ejecutar no se determina en tiempo de compilación, sino en tiempo de ejecución. Por eso hablamos de *paso de mensajes*, en lugar de *invocar* un método. La forma en la que se pasan los mensajes también resulta bastante peculiar y probablemente es lo que primero nos llame la atención cuando veamos código Objective-C:

```
NSString* cadena = @"cadena-de-prueba";
NSUInteger tam = [cadena length];
```

Podemos observar que para pasar un mensaje a un objeto, ponemos entre corchetes `[...]` la referencia al objeto, y a continuación el nombre del método que queramos ejecutar.

Los métodos pueden tomar parámetros de entrada. En este caso cada parámetro tendrá un nombre, y tras poner el nombre del parámetro pondremos `:` seguido de el valor que queramos pasarle:

```
NSString* result = [cadena stringByReplacingOccurrencesOfString: @"-"
                                                       withString: @" "];
```

Podemos observar que estamos llamando al método `stringByReplacingOccurrencesOfString:withString:` de nuestro objeto de tipo `NSString`, para reemplazar los guiones por espacios.

Es importante remarcar que el nombre de un método comprende el de todos sus parámetros, por ejemplo, el método anterior se identificaría mediante `stringByReplacingOccurrencesOfString:withString:..`. Esto es lo que se conoce como un *selector*, y nos permite identificar los mensajes que se le van a pasar a un objeto.

No podemos sobrecargar los métodos, es decir, no puede haber dos métodos que correspondan a un mismo *selector* pero con distinto tipo de parámetros. Sin embargo, si que podemos crear varias versiones de un método con distinto número o nombres de parámetros. Por ejemplo, también existe el método `stringByReplacingOccurrencesOfString:withString:options:range:` que añade dos parámetros adicionales al anterior.

Es posible llamar a métodos inexistentes sin que el compilador nos lo impida, como mucho obtendremos un *warning*:

```
NSString* cadena = @"cadena-de-prueba";
[cadena metodoInexistente]; // Produce warning, pero compila
```

En el caso anterior, como `NSString` no tiene definido ningún método que se llame `metodoInexistente`, el compilador nos dará un *warning*, pero la aplicación compilará, y tendremos un error en tiempo de ejecución. Concretamente saltará una excepción que nos indicará que se ha enviado un mensaje a un selector inexistente.

En el caso en que nuestra variables fuese de tipo `id`, ni siquiera obtendríamos ningún *warning en la compilación*. En este tipo de variables cualquier mensaje se considera válido:

```
id cadena = @"cadena-de-prueba";
[cadena metodoInexistente]; // Solo da error de ejecucion
```

Por este motivo es por el que hablamos de paso de mensajes en lugar de llamadas a métodos. Realmente lo que hace nuestro código es enviar un mensaje al objeto, sin saber si el método existe o no.

3.4. Creación e inicialización

Para instanciar una clase deberemos pasarle el mensaje `alloc` a la clase de la cual queramos crear la instancia. Por ejemplo:

```
id instancia = [NSString alloc];
```

Con esto crearemos una instancia de la clase `NSString`, reservando la memoria necesaria para alojarla, pero antes de poder utilizarla deberemos inicializarla con alguno de sus métodos inicializadores. Los inicializadores comienzan todos por `init`, y normalmente encontraremos definidos varios con distintos parámetros.

```
NSString *cadVacia = [[NSString alloc] init];
NSString *cadFormato = [[NSString alloc] initWithFormat: @"Numero %d", 5];
```

Como podemos ver, podemos anidar el paso de mensajes siempre que el resultado obtenido de pasar un mensaje sea un objeto al que podemos pasarlo otro. Normalmente siempre encontraremos anidadas las llamadas a `alloc` e `init`, ya que son los dos pasos que siempre se deben dar para construir el objeto. Destacamos que `alloc` es un método de clase, que normalmente no será necesario redefinir, y que todas las clases heredan de `NSObject` (en Objective-C las clases también son objetos y los métodos de clase se heredan), mientras que `init` es un método de instancia (se pasa el mensaje a la instancia creada con `alloc`), que nosotros normalmente definiremos en nuestras propias clases (de `NSObject` sólo se hereda un método `init` sin parámetros).

Sin embargo, en las clases normalmente encontramos una serie de métodos alternativos para instanciarlas y construirlas. Son los llamados métodos factoría, y en este caso todos ellos son métodos de clase. Suele haber un método factoría equivalente a cada uno de los métodos `init` definidos, y nos van a permitir instanciar e inicializar la clase directamente pasando un único mensaje. En lugar de `init`, comienzan con el nombre del objeto que están construyendo:

```
NSString *cadVacia = [NSString string];
NSString *cadFormato = [NSString stringWithFormat: @"Número %d", 5];
```

Suelen crearse para facilitar la tarea al desarrollador. Estas dos formas de instanciar clases tienen una diferencia importante en cuanto a la gestión de la memoria, que veremos más adelante.

3.5. Algunas clases básicas de Cocoa Touch

Vamos a empezar viendo algunos ejemplos de clases básicas de la API Cocoa Touch que necesitaremos para implementar nuestras aplicaciones. En estas primeras sesiones comenzaremos con el *framework Foundation*, donde tenemos la librería de clases de propósito general, y que normalmente encontraremos con el prefijo NS.

3.5.1. Objetos

`NSObject` es la clase de la que normalmente heredarán todas las demás clases, por lo que sus métodos estarán disponibles en casi en todas las clases que utilicemos. Vamos a ver los principales métodos de esta clase.

3.5.1.1. Inicialización e instanciación

Ya conocemos algunos métodos de este grupo (`alloc` e `init`), pero podemos encontrar alguno más:

- `+ initialize`: Es un método de clase, que se llama cuando la clase se carga por primera vez, antes de que cualquier instancia haya sido cargada. Nos puede servir para inicializar variables estáticas.
- `+ new`: Este método lo único que hace es llamar a `alloc` y posteriormente a `init`, para realizar las dos operaciones con un único mensaje. No se recomienda su uso.
- `+ allocWithZone: (NSZone*)`: Reserva memoria para el objeto en la zona especificada. Si pasamos `nil` como parámetro lo aloja en la zona por defecto. El método `alloc` visto anteriormente realmente llama a `allocWithZone: nil`, por lo que si queremos cambiar la forma en la que se instancia el objeto, con sobrescribir `allocWithZone` sería suficiente (esto se puede utilizar por ejemplo al implementar el patrón *singleton*, para evitar que el objeto se instancie más de una vez). Utilizar zonas adicionales puede permitirnos optimizar los accesos memoria (para poner juntos en memoria objetos que vayan a utilizarse de forma conjunta), aunque normalmente lo más eficiente será utilizar la zona creada por defecto.
- `- copy / - mutableCopy`: Son métodos de instancia que se encargan de crear una nueva instancia del objeto copiando el estado de la instancia actual. No todos los objetos son copiables, a continuación veremos más detalles sobre la copia de objetos.

3.5.1.2. Objetos mutables e inmutables

En la API de Cocoa encontramos varios objetos que se encuentran disponibles en dos versiones: **mutable** e **inmutable**. Uno de estos objetos son por ejemplo las cadenas (`NSString` y `NSMutableString`).

Los objetos **inmutables** son objetos de los cuales no podemos cambiar su estado interno, y que una vez instanciados e inicializados, sus variables de instancia no cambiarán de valor. Por ejemplo, una vez instanciada una cadena de tipo `NSString`, no podremos modificar sus caracteres.

Por otro lado, los **mutables** son aquellos cuyo estado si puede ser modificado. Este es el caso de `NSMutableString`, que además de todos los métodos que ya tenía la clase `NSString`, incorpora también métodos como `appendString:` o `replaceCharactersInRange:withString:` que nos permiten modificar el contenido de la cadena.

3.5.1.3. Copia de objetos

La clase `NSObject` incorpora el método `copy` que se encarga de crear una copia de nuestro objeto. Sin embargo, no todos los objetos son copiables. Sólo se podrán copiar aquellos objetos que adopten el protocolo `NSCopying` (lo podremos comprobar en la documentación de la clase). Gran parte de los objetos de la API de Cocoa Touch implementan `NSCopying`, y por lo tanto son copiables.

Los objetos que existan en las modalidades mutable e inmutable pueden adoptar también el protocolo `NSMutableCopy`, que nos indica que podemos utilizar también el método `mutableCopy`, para poder crear una copia mutable del objeto. En estos objetos `copy` realizará la copia inmutable, mientras que `mutableCopy` creará una copia mutable.

Por ejemplo, en el caso de las cadenas tenemos las dos opciones. Tanto si nuestra cadena original es mutable como inmutable, podremos obtener una copia de ella en cualquiera de estas modalidades:

```
// La cadena original es inmutable
NSString *cadena = @"Mi cadena";

NSString *copiaInmutable = [cadena copy];
NSMutableString *copiaMutable = [cadena mutableCopy];

// La cadena original es mutable
NSMutableString *cadenaMutable = [NSMutableString stringWithCapacity: 32];

NSString *copiaInmutable = [cadenaMutable copy];
NSMutableString *copiaMutable = [cadenaMutable mutableCopy];
```

3.5.1.4. Información de la instancia

Al igual que ocurría en Java, `NSObject` implementa una serie de métodos que nos darán información sobre los objetos, y que nosotros podemos sobrescribir en nuestras clases para personalizar dicha información. Estos métodos son:

- `isEqual:` Comprueba si dos instancias de nuestra clase son iguales internamente, y nos devuelve un *booleano* (YES o NO).
- `description:` Nos da una descripción de nuestro objeto en forma de cadena de texto (`NSString`).
- `hash:` Genera un código *hash* a partir de nuestro objeto para indexarlo en tablas. Si hemos redefinido `isEqual`, deberíamos también redefinir `hash` para que dos objetos iguales generen siempre el mismo *hash*.

3.5.2. Cadenas

La clase `NSString` es la clase con la que representamos las cadenas en Objective-C, y hemos visto cómo utilizarla en varios de los ejemplos anteriores. Vamos ahora a ver algunos elementos básicos de esta clase.

3.5.2.1. Literales de tipo cadena

La forma más sencilla de inicializar una cadena es utilizar un literal de tipo `@ "cadena"`, que inicializa un objeto de tipo `NSString*`, y que no debemos confundir con las cadenas de C estándar que se definen como `"cadena"` (sin la @) y que corresponden al tipo `char*`.

Estos literales tienen la peculiaridad de que se crean de forma estática por el compilador, por lo que las operaciones de gestión de memoria no tienen efecto sobre ellos (nunca serán eliminados de la memoria). Las llamadas que realicemos sobre estos objetos a `retain`, `release` y `autorelease` serán ignoradas.

3.5.2.2. Cadenas C estándar y Objective-C

Como hemos comentado, las cadenas `@ "cadena"` y `"cadena"` son tipos totalmente distintos, por lo que no podemos utilizarlas en las mismas situaciones. Las clases de Cocoa Touch siempre trabajarán con `NSString`, por lo que si tenemos cadenas C estándar tendremos que convertirlas previamente. Para ello, en la clase `NSString` contamos con métodos inicializadores que crear la cadena Objective-C a partir de una cadena C:

```
- (id)initWithCString:(const char *)nullTerminatedCString
                  encoding:(NSStringEncoding)encoding
- (id)initWithUTF8String:(const char *)bytes
```

El primero de ellos inicializa la cadena Objective-C a partir de una cadena C y de la codificación que se esté utilizando en dicha cadena. Dado que la codificación más común es UTF-8, tenemos un segundo método que la inicializa considerando directamente esta codificación. También encontramos métodos de factoría equivalentes:

```
- (id)stringWithCString:(const char *)nullTerminatedCString
                  encoding:(NSStringEncoding)encoding
- (id)stringWithUTF8String:(const char *)bytes
```

De la misma forma, puede ocurrir que tengamos una cadena en Objective-C y que necesitemos utilizarla en alguna función C estándar. En la clase `NSString` tenemos

métodos para obtener la cadena como cadena C estándar:

```
NSString *cadena = @"Cadena";
const char *cadenaC = [cadena UTF8String];
const char *cadenaC = [cadena cStringUsingEncoding: NSASCIIStringEncoding];
```

3.5.2.3. Inicialización con formato

Podemos dar formato a las cadenas de forma muy parecida al `printf` de C estándar. Para ello contamos con el inicializador `initWithFormat:`:

```
NSString *cadena = [[NSString alloc]
                     initWithFormat: @"Duracion: %d horas", horas];
```

A los códigos de formato que ya conocemos de `printf` en C, hay que añadir `%@` que nos permite imprimir objetos Objective-C. Para imprimir un objeto utilizará su método `description` (o `descriptionWithLocale` si está implementado). Deberemos utilizar dicho código para imprimir cualquier objeto Objective-C, incluido `NSString`:

```
NSString *nombre = @"Pepe";
NSUInteger edad = 20;

NSString *cadena =
    [NSString stringWithFormat: @"Nombre: %@ (%d)", nombre, edad];
```

Atención

Nunca se debe usar el código `%s` con una cadena Objective-C (`NSString`). Dicho código espera recibir un *array* de caracteres acabado en `NULL`, por lo que si pasamos un puntero a objeto obtendremos resultados inesperados. Para imprimir una cadena Objective-C siempre utilizaremos `%@`.

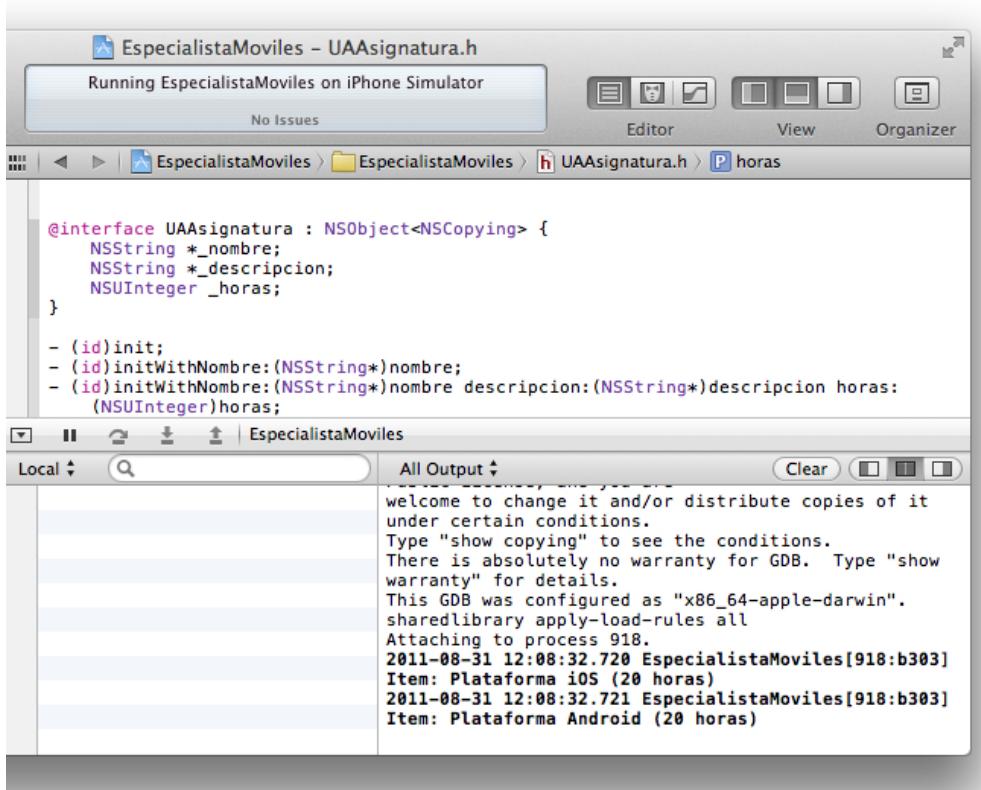
Los mismos atributos de formato se podrán utilizar en la función `NSLog` que nos permite escribir *logs* en la consola para depurar aplicaciones

```
NSLog(@"%@", i, obj);
```

Cuidado

Los *logs* pueden resultarnos muy útiles para depurar la aplicación, pero debemos llevar cuidado de eliminarlos en la *release*, ya que reducen drásticamente el rendimiento de la aplicación, y tampoco resulta adecuado que el usuario final pueda visualizarlos. Podemos ayudarnos de las macros (`#define`, `#ifdef`) para poder activarlos o desactivarlos de forma sencilla según la configuración utilizada.

Los *logs* aparecerán en el panel de depuración ubicado en la parte inferior de la pantalla principal del entorno. Podemos abrirlo y cerrarlo mediante en botón correspondiente en la esquina superior izquierda de la pantalla. Normalmente cuando ejecutemos la aplicación y ésta escriba *logs*, dicho panel se mostrará automáticamente.



Panel de depuración

3.5.2.4. Localización de cadenas

Anteriormente hemos comentado que las cadenas se puede externalizar en un fichero que por defecto se llamará `Localizable.strings`, del que podremos tener varias versiones, una para cada localización soportada. Vamos a ver ahora cómo leer estas cadenas localizadas. Para leerlas contamos con la función `NSLocalizedString(clave, comentario)` que nos devuelve la cadena como `NSString`:

```
NSString *cadenaLocalizada = NSLocalizedString(@"Titulo", @"Mobile UA");
```

Este método nos devolverá el valor asociado a la clave proporcionada según lo especificado en el fichero `Localizable.strings` para el *locale* actual. Si no se encuentra la clave, nos devolverá lo que hayamos especificado como comentario.

Existen versiones alternativas del método que no utilizan el fichero por defecto, sino que toman como parámetro el fichero del que sacar las cadenas.

Las cadenas de `Localizable.strings` pueden también contener códigos de formato. En estos casos puede ser interesante numerar los parámetros, ya que puede que en diferentes idiomas el orden sea distinto:

```
// es.lproj
```

```
"CadenaFecha" = "Fecha: %1$2d / %2$2d / %3$4d";
// en.lproj
"CadenaFecha" = "Date: %2$2d / %1$2d / %3$4d";
```

Podemos utilizar `NSLocalizedString` para obtener la cadena con la plantilla del formato:

```
NSString *cadena = [NSString stringWithFormat:
    NSLocalizedString(@"CadenaFecha", "Date: %2$2d / %1$2d / %3$4d"),
    dia, mes, año];
```

3.5.2.5. Conversión de números

La conversión entre cadenas y los diferentes tipos numéricos en Objective-C también se realiza con la clase `NSString`. La representación de un número en forma de cadena se puede realizar con el método `stringWithFormat` visto anteriormente, permitiendo dar al número el formato que nos interese.

La conversión inversa se puede realizar mediante una serie de métodos de la clase `NSString`:

```
NSInteger entero = [cadenaInt integerValue];
BOOL booleano = [cadenaBool boolValue];
float flotante = [cadenaFloat floatValue];
```

3.5.2.6. Comparación de cadenas

Las cadenas son punteros a objetos, por lo que si queremos comparar si dos cadenas son iguales nunca deberemos utilizar el operador `==`, ya que esto sólo comprobará si los dos punteros apuntan a la misma dirección de memoria. Para comparar si dos cadenas contienen los mismos caracteres podemos utilizar el método `isEqual`, al igual que para comparar cualquier otro tipo de objeto Objective-C, pero si sabemos que los dos objetos son cadenas es más sencillo utilizar el método `isEqualToString`.

```
if([cadena isEqualToString: otraCadena]) { ... }
```

También podemos comparar dos cadenas según el orden alfabético, con `compare`. Nos devolverá un valor de la enumeración `NSComparisonResult` (`NSOrderedAscending`, `NSOrderedSame`, o `NSOrderedDescending`).

```
NSComparisonResult resultado = [cadena compare: otraCadena];
switch(resultado) {
    case NSOrderedAscending:
        ...
        break;
    case NSOrderedSame:
        ...
        break;
    case NSOrderedDescending:
        ...
        break;
}
```

Tenemos también el método `caseInsensitiveCompare` para que realice la comparación ignorando mayúsculas y minúsculas.

Otros métodos nos permiten comprobar si una cadena tiene un determinado prefijo o sufijo (`hasPrefix`, `hasSuffix`), u obtener la longitud de una cadena (`length`).

3.5.3. Fechas

Otro tipo de datos que normalmente necesitaremos tratar en nuestras aplicaciones son las fechas. En Objective-C las fechas se encapsulan en `NSDate`. Muchos de los métodos de dicha clase toman como parámetro datos del tipo `NSTimeInterval`, que equivale a `double`, y corresponde al tiempo en segundos (con una precisión de submilisegundos).

La forma más rápida de crear un objeto fecha es utilizar su inicializador (o factoría) sin parámetros, con lo que se creará un objeto representando la fecha actual.

```
NSDate *fechaActual = [NSDate date];
```

También podemos crear la fecha especificando el número de segundos desde la fecha de referencia (1 de enero de 1970 a las 0:00).

```
NSDate *fecha = [NSDate dateWithTimeIntervalSince1970: segundos];
```

De la misma forma que en el caso de las cadenas, podemos comparar fechas con el método `compare`, que también nos devolverá un valor de la enumeración `NSComparisonResult`.

3.5.3.1. Componentes de la fecha

El objeto `NSDate` representa una fecha simplemente mediante el número de segundos transcurridos desde la fecha de referencia. Sin embargo, muchas veces será necesario obtener los distintos componentes de la fecha de forma independiente (día, mes, año, hora, minutos, segundos, etc). Estos componentes se encapsulan como propiedades de la clase `NSDateComponents`.

Para poder obtener los componentes de una fecha, o crear una fecha a partir de sus componentes, necesitamos un objeto calendario `NSCalendar`:

```
NSDate *fecha = [NSDate date];
NSCalendar *calendario = [NSCalendar currentCalendar];
NSDateComponents *componentes = [currentCalendar
    components:(NSDayCalendarUnit | NSMonthCalendarUnit |
    NSYearCalendarUnit)
    fromDate:fecha];

NSInteger dia = [componentes day];
NSInteger mes = [componentes month];
NSInteger anyo = [componentes year];
```

Con el método de clase `currentCalendar` obtenemos una instancia del calendario correspondiente al usuario actual. Con el método `components:fromDate:` de dicho

calendario podemos extraer los componentes indicados de la fecha (objeto `NSDate`) que proporcionemos. Los componentes se especifican mediante una máscara que se puede crear a partir de los elementos de la enumeración `NSCalendarUnit`, y son devueltos mediante un objeto de tipo `NSDateComponents` que incluirá los componentes solicitados como campos.

También se puede hacer al contrario, crear un objeto `NSDateComponents` con los campos que queramos para la fecha, y a partir de él obtener un objeto `NSDate`:

```
NSDateComponents *componentes = [[NSDateComponents alloc] init];
[componentes setDay: dia];
[componentes setMonth: mes];
[componentes setYear: anyo];

NSDate *fecha = [calendario dateFromComponents: componentes];
[componentes release];
```

Con el calendario también podremos hacer operaciones con fechas a partir de sus componentes. Por ejemplo, podemos sumar valores a cada componentes con `dateByAddingComponents:toDate:`, o obtener la diferencia entre dos fechas componente a componente con `components:fromDate:toDate:options:`.

3.5.3.2. Formato de fechas

Podemos dar formato a las fecha con `NSDateFormatter`. La forma más sencilla es utilizar alguno de los estilos predefinidos en la enumeración `NSDateFormatterStyle` (`NSDateFormatterNoStyle`, `NSDateFormatterShortStyle`, `NSDateFormatterMediumStyle`, `NSDateFormatterLongStyle`, `NSDateFormatterFullStyle`):

```
NSDateFormatter formato = [[NSDateFormatter alloc] init];
[formato setTimeStyle: NSDateFormatterNoStyle];
[formato setDateStyle: NSDateFormatterFullStyle];

NSString *cadena = [formato stringFromDate: fecha];
[formato release];
```

Podemos también especificar un formato propio mediante un patrón con `setDateFormat:`

```
[formato setDateFormat: @"dd/MM/yyyy HH:mm"];
```

También podemos utilizar nuestro objeto de formato para *parsear* fechas con `dateFromString:`

```
NSDate *fecha = [formato dateFromString: @"20/06/2012 14:00"];
```

3.5.4. Errores y excepciones

En Objective-C podemos tratar los errores mediante excepciones de forma similar a Java. Para capturar una excepción podemos utilizar la siguiente estructura:

```

@try
    // Código
@catch(NSErrorException *ex) {
    // Código tratamiento excepción
}
@catch(id obj) {
    // Código tratamiento excepción
}
@finally {
    // Código de finalización
}

```

Una primera diferencia que encontramos con Java es que se puede lanzar cualquier objeto (por ejemplo, en el segundo `catch` vamos que captura `id`), aunque se recomienda utilizar siempre `NSErrorException` (o una subclase de ésta). Otra diferencia es que en Objective-C no suele ser habitual heredar de `NSErrorException` para crear nuestros propios tipos de excepciones. Cuando se produzca una excepción en el código del bloque `try` saltará al primer `catch` cuyo tipo sea compatible con el del objeto lanzado. El bloque `finally` siempre se ejecutará, tanto si ha lanzado la excepción como si no, por lo que será el lugar idóneo para introducir el código de finalización (por ejemplo, liberar referencias a objetos).

Podremos lanzar cualquier objeto con `@throw`:

```

@throw [[[NSError alloc] initWithName: @"Error"
                                reason: @"Descripción del error"
                               userInfo: nil] autorelease];

```

Aunque tenemos disponible este mecanismo para tratar los errores, en Objective-C suele ser más común pasar un parámetro de tipo `NSError` a los métodos que puedan producir algún error. En caso de que se produzca, en dicho objeto tendremos la descripción del error producido:

```

NSError *error;
NSString *contenido = [NSString
    stringWithContentsOfFile: @"texto.txt"
    encoding: NSASCIIStringEncoding
    error: &error];

```

Este tipo de métodos reciben como parámetro la dirección del puntero, es decir, (`NSError **`), por lo que no es necesario que inicialicemos el objeto `NSError` nosotros. Si no nos interesa controlar los errores producidos, podemos pasar el valor `nil` en el parámetro `error`. Los errores llevan principalmente un código (`code`) y un dominio (`domain`). Los código se definen como constantes en Cocoa Touch (podemos consultar la documentación de `NSError` para consultarlos). También incorpora mensajes que podríamos utilizar para mostrar el motivo del error y sus posibles soluciones:

```

NSString *motivo = [error localizedFailureReason];

```

En Objective-C no hay ninguna forma de crear excepciones equivalentes a las excepciones de tipo *checked* en Java (es decir, que los métodos estén obligados a capturarlas o a declarar que pueden lanzarlas). Por este motivo, aquellos métodos en los que en Java utilizaríamos excepciones *checked* en Objective-C no será recomendable

utilizar excepciones, sino incorporar un parámetro `NSError` para así dejar claro en la interfaz que la operación podría fallar. Sin embargo, las excepciones si que serán útiles si queremos tratar posibles fallos inesperados del *runtime* (las que serían equivalentes a las excepciones *unchecked* en Java).

Acceso a la documentación

Como hemos comentado, mientras escribimos código podemos ver en el panel de utilidades ayuda rápida sobre el elemento sobre el que se encuentre el cursor en el editor. Tenemos también otros atajos para acceder a la ayuda. Si hacemos *option(alt)-click* sobre un elemento del código abriremos un cuadro con su documentación, a partir del cual podremos acceder a su documentación completa en *Organizer*. Por otro lado, si hacemos *cmd-click* sobre un elemento del código, nos llevará al lugar en el que ese elemento fue definido. Esto puede ser bastante útil para acceder de forma rápida a la declaración de clases y de tipos.

3.6. Colecciones de datos

En Objective-C encontramos distintos tipos de colecciones de datos, de forma similar al marco de colecciones de Java. Se trata de colecciones genéricas que pueden contener como datos cualquier objeto de Objective-C. Los tipos principales de colecciones que encontramos en Objective-C son `NSArray`, `NSSet`, y `NSDictionary`. De todos ellos podemos encontrar versiones tanto mutables como inmutables.

3.6.1. Wrappers de tipos básicos

Como hemos comentado, las colecciones pueden contener cualquier objeto de Objective-C, pero los tipos de datos básicos no son objetos (`int`, `float`, `char`, etc). ¿Qué ocurre si necesitamos crear una colección de elementos de estos tipos?

Para solucionar este problema encontramos objetos de Objective-C que se encargan de envolver datos de estos tipos en forma de objeto, para así poderlos incluir en colecciones. Estos objetos se conocen como *wrappers*.

El más sencillo es el que nos permite introducir un valor `nil` en la colección. Para ello utilizaremos un objeto de tipo `NSNull`. Dado que el valor `nil` es único, no necesitamos más que una instancia de dicha clase. Por este motivo dicho objeto se define como *singleton*, y podremos obtenerlo de la siguiente forma:

```
[NSNull null]
```

Otro tipo de datos común son los valores numéricos (`BOOL`, `int`, `float`, etc). Todos estos tipos pueden representarse mediante la clase `NSNumber`:

```
NSNumber *booleano = [NSNumber numberWithBool: YES];
NSNumber *entero = [NSNumber numberWithInt: 10];
NSNumber *flotante = [NSNumber numberWithFloat: 2.5];
...
```

```
BOOL valorBool = [booleano boolValue];
int valorEntero = [entero intValue];
float valorFlotante = [flotante floatValue];
```

Por último, tenemos el caso de las estructuras de datos. Este caso ya no es tan sencillo, ya que podemos tener cualquier estructura definida por nosotros. La única forma de tener una forma genérica para encapsular cualquier estructura de datos es almacenar sus *bytes*. Para ello utilizaremos la clase `NSValue`.

```
typedef struct {
    int x;
    int y;
} Punto;

Punto p;
p.x = 1;
p.y = 5;

NSValue *valorPunto = [NSValue valueWithBytes:&p
                                         objCType:@encode(Punto)];
```

Podemos observar que al construir el objeto `NSValue` debemos proporcionar la dirección de memoria donde se aloja el dato que queremos almacenar, y además debemos indicar su tipo. Para indicar su tipo utilizamos la directiva `@encode`. Esta directiva toma como parámetro un tipo de datos (podría ser cualquier tipo básico, compuesto, o incluso objetos), y nos devuelve la representación de dicho tipo de datos que utiliza internamente Objective-C.

Si queremos recuperar el valor guardado, deberemos proporcionar una dirección de memoria con el espacio necesario para alojar dicho valor:

```
Punto punto;
[valorPunto getValue:&punto];
```

3.6.2. Listas

Las listas son colecciones en las que los datos se guardan en un orden determinado. Cada elemento se almacena en un índice de la lista. Las listas se definen mediante los tipos `NSArray` (inmutable) y `NSMutableArray` (mutable).

En caso de la versión inmutable, deberemos inicializarlo a partir de sus elementos, ya que al ser inmutable no podremos añadirlos más adelante. En este caso normalmente utilizaremos un constructor que tome como parámetros los elementos de la lista. Este constructor recibe una lista de parámetros terminada en `nil`:

```
NSArray *lista = [NSArray arrayWithObjects: obj1, obj2, obj3, nil];
```

Es importante no olvidarnos de poner `nil` al final de la lista. Muchos métodos con número variable de parámetros se definen de esta forma. Podemos consultar la documentación para saber si debe llevar `nil` al final, o bien fijarnos en la firma del

método que aparece en Xcode al autocompletar (si aparece ... nil se trata de una lista acabada en nil). Si no incluimos nil al final obtendremos un *warning* del compilador y un error en tiempo de ejecución.

Su versión mutable, `NSMutableArray`, incorpora inicializadores adicionales en los que se indica la capacidad inicial de la lista, aunque ésta podría crecer conforme añadamos datos:

```
NSMutableArray *listaMutable = [NSMutableArray arrayWithCapacity: 100];
```

3.6.2.1. Acceso a los elementos de la lista

Podemos saber el número de elementos que tiene una lista con el método `count`:

```
NSUInteger numElementos = [lista count];
```

Esto se aplicará a cualquier tipo de colección (no exclusivamente a las listas). En el caso de las listas, los índices irán desde 0 hasta `count-1`. Podemos acceder al objeto que esté en un índice determinado con `objectAtIndex`:

```
id primerObjeto = [lista objectAtIndex: 0];
```

También podemos buscar el índice en el que está un objeto determinado con `indexForObject`:

```
NSUInteger indice = [lista indexForObject: obj];
if(indice==NSNotFound) {
    // Objeto no encontrado
}
```

El objeto proporcionado se comparará con cada objeto de la lista utilizando su método `isEqual`. Si lo que queremos es buscar la misma instancia, entonces deberemos utilizar `indexForObjectIdenticalTo`.

3.6.2.2. Modificación de los elementos de la lista

En el caso de que nuestra lista sea mutable, podremos modificar la lista de elementos que contiene para añadir, insertar, reemplazar o eliminar elementos:

```
// A partir del indice 5 se mueven a la siguiente posición
[listaMutable insertObject:obj atIndex:5];

// Lo añade al final de la lista
[listaMutable addObject:obj];

// A partir del indice 5 se mueven a la anterior posición
[listaMutable removeObjectAtIndex:5];

// Es más eficiente, con coste constante
[listaMutable removeLastObject];

[listaMutable replaceObjectAtIndex:5 withObject:obj];
```

También podremos encontrar gran cantidad de variantes de los métodos anteriores.

3.6.3. Conjuntos

Un conjunto es una colección no ordenada de elementos distintos. Dos objetos iguales (según `isEqual`) no pueden repetirse dentro del conjunto. De la misma forma que las listas, existen en versión inmutable `NSSet` y mutable `NSMutableSet`.

Se puede inicializar de forma similar a las listas, según sea mutable o inmutable:

```
NSSet *conjunto = [NSSet setWithObjects: obj1, obj2, obj3, nil];
NSMutableSet *conjuntoMutable = [NSMutableSet setWithCapacity: 100];
```

Al igual que en las listas, contamos con el método `count` para conocer el número de elementos del conjunto, y con el método `containsObject` que nos dirá si el conjunto contiene un determinado objeto.

```
BOOL pertenece = [conjunto containsObject: obj];
```

En el caso de los conjuntos mutables, tenemos métodos para añadir o eliminar elementos:

```
// Solo lo añade si no pertenece todavía al conjunto
[conjuntoMutable addObject:obj];
[conjuntoMutable removeObject:obj];
```

Además de estas operaciones, también encontramos métodos para realizar las operaciones habituales sobre conjuntos (unión, intersección, resta, etc).

Existe otra subclase de `NSSet` a parte de `NSMutableSet`: `NSCountedSet`. La diferencia con las anteriores consiste en que en este caso cada objeto lleva asociado un contador que indica cuántas veces se encuentra repetido en el conjunto. Si añadimos varias veces el mismo objeto, lo que estaremos haciendo es incrementar su contador. Podemos obtener el número de veces que se ha añadido un objeto dado con `countForObject`. Este tipo de conjuntos también se denomina bolsa de objetos.

3.6.4. Diccionarios

Los diccionarios son un tipo de colección en la que cada objeto se encuentra asociado a una clave. Realmente lo que almacenan es un conjunto de pares *clave-valor*. Igual que en los casos anteriores, tenemos un diccionario inmutable (`NSDictionary`) y uno mutable (`NSMutableDictionary`). Los diccionarios equivalen a la colección que en Java se denomina `Map`.

Al crear un diccionario ya no basta con dar una lista de objetos, sino que necesitaremos también una lista de claves:

```
NSDictionary *diccionario =
    [NSDictionary dictionaryWithObjectsAndKeys:
        obj1, @"clave1",
        obj2, @"clave2",
        obj3, @"clave3", nil];
```

Las claves pueden ser cualquier tipo de objeto, pero será conveniente que sean cadenas.

Podemos también crear un diccionario mutable vacío:

```
NSMutableDictionary *diccionarioMutable =  
    [NSMutableDictionary dictionaryWithCapacity: 100];
```

No puede haber más de una ocurrencia de la misma clave (esto se comprobará con `isEqual`). Del diccionario podemos sacar el objeto asociado a una clave con el método `objectForKey`:

```
id obj = [diccionario objectForKey:@"clave1"];
```

Si no hay ningún objeto asociado a dicha clave, este método nos devolverá `nil`.

También podemos sacar la lista de todas las claves (`allKeys`) o de todos los objetos almacenados en el diccionario (`allValues`). Estas listas se obtendrán como un objeto del tipo `NSArray`:

```
NSArray *claves = [diccionario allKeys];  
NSArray *valores = [diccionario allValues];
```

En el caso de los diccionarios mutables tenemos también métodos para establecer el objeto asociado a una clave dada, o para borrarlo:

```
[diccionario setObject:obj forKey:@"clave1"];  
[diccionario removeObjectForKey:@"clave1"];
```

3.6.5. Recorrer las colecciones

La forma más sencilla de recorrer una lista es utilizar la estructura `for-in`.

```
for(id obj in lista) {  
    NSLog(@"%@", obj);  
}
```

Si sabemos que todos los elementos de la lista son de un tipo concreto, podemos declarar los items con ese tipo. Por ejemplo, si tenemos una lista de cadenas podríamos recorrerlas con:

```
for(NSString *cadena in lista) {  
    NSLog(@"%@", cadena);  
}
```

Con esta estructura también podremos recorrer los elementos pertenecientes a un conjunto y las claves registradas en un diccionario. Si quisiésemos recorrer los objetos de un diccionario, sin tener que pasar por las claves, podríamos hacerlo de la siguiente forma:

```
for(id valor in [diccionario allValues]) {  
    NSLog(@"%@", valor);  
}
```

Lo habitual será recorrer las claves, ya que a partir de ellas es muy sencillo obtener sus valores asociados:

```
for(NSString *clave in diccionario) {  
    NSLog(@"%@", clave, [diccionario objectForKey: clave]);  
}
```

También podemos utilizar objetos enumeradores (`NSEnumerator`) para recorrer las listas. Estos objetos vienen de versiones anteriores de Objective-C en las que no existía la estructura `for-in`. Actualmente es más sencillo y limpio utilizar dicho tipo de `for` para recorrer las colecciones.

```
NSEnumerator *enumerador = [lista objectEnumerator];  
id obj;  
  
while (obj = [enumerador nextObject]) {  
    NSLog(@"Obtenido el objeto %@", valor);  
}
```

Cuidado

Si mientras estamos recorriendo una colección mutable (tanto con `for-in` como con un enumerador) tratamos de modificarla, obtendremos un error en tiempo de ejecución.

4. Ejercicios de Objective-C

4.1. Manejo de cadenas (1 punto)

Vamos a cambiar las cadenas de la aplicación que comenzamos a desarrollar en la sesión anterior. Se pide:

- a) En la clase `UAMasterViewController` localiza el método `initWithNibName:bundle:`. Hay una línea en la que se asigna el título mediante la instrucción `self.title =` Modifica el título para que ahora sea "Filmoteca".
- b) Abre la documentación de la clase `NSString`. Busca un método que permita convertir la cadena a mayúsculas, y aplícalo al título.
- c) Localiza el título de la aplicación utilizando la cadena con identificador "`AppName`" definida en la sesión anterior.
- d) Vamos a hacer que en la lista aparezcan varios ítems. Para ello localiza el método `tableView:numberOfRowsInSection:`, y haz que devuelva el valor 5. Comprueba que ahora aparecen cinco ítems en la lista.
- e) Ahora cambiaremos el texto de cada ítem, para que indique la posición de la lista en la que está. Por ejemplo, el primer ítem tendrá el texto Posición 0, el segundo Posición 1, etc. Para ello localiza el método `tableView:cellForRowAtIndexPath:`, y dentro de él la línea que comienza por `cell.textLabel.text =` Aquí es donde se asigna el texto de cada ítem. La posición del ítem que se está asignando se puede obtener llamando al método `row` del objeto `NSIndexPath`, y es un valor de tipo entero (consulta la documentación de la clase `NSIndexPath` en Organizer).

4.2. Manejo de fechas (1 punto)

Vamos a realizar operaciones con fechas dentro de la aplicación. Se pide:

- a) En el método `initWithNibName:bundle:` de `UAMasterViewController` crea una variable local `fechaActual` de tipo `NSDate` e inicialízala con la fecha actual. Escribe un *log* mediante la función `NSLog` para mostrar en la consola de depuración el valor de la variable anterior.
- b) Crea una segunda variable local `fechaPasada` también de tipo fecha, e inicialízala con la fecha *26 de octubre de 1985* ayudándote de la clase `NSDateComponents`.
- c) Calcula la diferencia entre las dos fechas en años. Escribe un *log* con la diferencia existente.

4.3. Gestión de errores (1 punto)

Vamos a ver como tratar errores al llamar a métodos que pueden fallar por factores externos. Utilizaremos el método `stringWithContentOfFile:encoding:error:` de la clase `NSString` para este ejercicio. Deberemos consultar la documentación de este método en Organizer. Se pide:

a) Crear dos ficheros de texto: `texto1.txt` y `texto2.txt` que contengan el texto `Primer fichero` y `Segundo fichero` respectivamente. El primero de ellos lo empaquetaremos en el raíz del *bundle*, y el segundo en una carpeta `/datos` (también se puede encontrar estos ficheros en las plantillas de la sesión).

Ayuda

Para obtener la ruta de un fichero en el sistema de ficheros del dispositivos, puedes utilizar el método `[[NSBundle mainBundle] pathForResource:@"nombreFichero" ofType:@"extension"]`.

b) En el método `initWithNibName:bundle:` de `UAMasterViewController` crearemos una cadena a partir de estos ficheros, y escribiremos un *log* con el contenido leído mediante la función `NSLog`.

c) Intenta leer el segundo fichero con la ruta `/texto2.txt`. Deberá dar error, ya que este fichero se empaqueta en `/datos/texto2.txt`. Utiliza el objeto `NSError` para detectar el error y escribe un *log* con el motivo del error.

5. Objetos y propiedades

5.1. Clases y objetos

En Objective-C cada clase normalmente se encuentra separada en dos ficheros: la declaración de la interfaz en un fichero .h, y la implementación en un .m. A diferencia de Java, el nombre del fichero no tiene que coincidir con el nombre de la clase, y podemos tener varias clases definidas en un mismo fichero. Podremos utilizar cualquiera de ellas siempre que importemos el fichero .h correspondiente.

El criterio que se seguirá es el de agrupar en el mismo fichero aquellas clases, estructuras, funciones y elementos adicionales que estén muy relacionados entre sí, y dar al fichero el nombre de la clase principal que contiene. Por ejemplo, las clases `NSString` y `NSMutableString` se definen en el mismo fichero, de nombre `NSString.h(m)`. La segunda es una subclase de la primera, que añade algunos métodos para poder modificar la cadena. Si estamos creando un *framework*, también deberemos crear un único fichero .h que se encargue de importar todos los elementos de nuestro *framework*. Por ejemplo, si queremos utilizar el *framework Foundation* sólo necesitamos importar `Foundation/Foundation.h`, a pesar de que las clases de esta librería se declaran en diferentes ficheros de cabecera.

La forma más rápida de crear una nueva clase con Xcode es seleccionar *File > New > New File... > Cocoa Touch > Objective-C class*. Nos permitirá especificar la superclase, poniendo por defecto `NSObject`, que es la superclase en última instancia de todas las clases, como ocurre en Java con `Object`. Tras esto, tendremos que dar un nombre y una ubicación a nuestra clase, y guardará los ficheros .h y .m correspondientes.

5.1.1. Declaración de una clase

En el fichero .h, en la declaración de la interfaz vemos que se indica el nombre de la clase seguido de la superclase:

```
@interface MiClase : NSObject  
@end
```

Podemos introducir entre llaves una serie de variables de instancia.

```
@interface UAAsignatura : NSObject {  
    NSString *_nombre;  
    NSString *_descripcion;  
    NSUInteger _horas;  
}  
@end
```

Una cosa importante que debemos tener en cuenta es que las variables de instancia en

Objective-C son **por defecto protegidas**. Podemos añadir los modificadores de acceso `@public`, `@protected`, y `@private`, pero no es recomendable declarar variables públicas. En su lugar, para poder acceder a ellas añadiremos métodos *accesores*. Lo más común será dejar el nivel de acceso por defecto.

```
@interface MiClase : NSObject {
    @public
        NSString *_varPublica;
        NSInteger _otraVarPublica;
    @protected
        NSString *_varProtegida;
        NSInteger _otraVarProtegida;
    @private
        NSString *_varPrivada;
        NSInteger _otraVarPrivada;
}
@end
```

Los nombres de las variables de instancia se escriben en *lowerCamelCase*, y su nombre debe resultar descriptivo, evitando abreviaturas, excepto las utilizadas de forma habitual (como `min` o `max`).

Los métodos se declaran dentro del bloque `@interface`, pero fuera de las llaves. La firma de cada método comienza por `+` o `-`, según sea un método de clase o de instancia respectivamente. Tras este indicador, se indica el tipo de datos que devuelve el método, y a continuación el nombre del métodos, sus parámetros y sus tipos:

```
@interface UAAsignatura : NSObject {
    NSString *_nombre;
    NSString *_descripcion;
    NSUInteger _horas;
}

+ (CGFloat)creditosParaHoras:(CGFloat)horas;
- (CGFloat)creditos;
- (CGFloat)tasaConPrecioPorCredito:(CGFloat)precioCredito
                                esBecario:(BOOL)becario;

@end
```

En el ejemplo hemos definido tres métodos que para calcular los créditos de una asignatura y su precio. El primero de ellos nos dice a cuántos créditos corresponde un número de horas dado como parámetro. Dado que no necesita acceder a las variables de ninguna instancia concreta lo hemos definido como método de clase, para así poderlo ejecutar simplemente a partir del nombre de la clase:

```
CGFloat cr = [UAAsignatura creditosParaHoras: 20];
```

Los otros métodos los tendremos que ejecutar a partir de una instancia concreta de la clase:

```
UAAsignatura *asignatura = // Instanciar clase
CGFloat creditos = [asignatura creditos];
CGFloat precio = [asignatura tasaConPrecioPorCredito: 60 esBecario: NO];
```

Espacios de nombres

En Objective-C no tenemos un espacio de nombres para nuestras clases como son los paquetes de Java, por lo que corremos el riesgo de que el nombre de una de nuestras clases se solape con el de las clases de las librerías que estemos utilizando. Para evitar esto es recomendable que utilicemos un mismo prefijo para todas nuestras clases. Las clases de Cocoa Touch tienen como prefijo dos letras mayúsculas que identifiquen cada librería y actúan como espacio de nombres (NS, CG, UI, etc), aunque no tenemos que utilizar por fuerza dos letras ni tampoco es obligatorio que sean mayúsculas. En los ejemplos estamos utilizando UA, ya que son las clases para una aplicación de la Universidad de Alicante. Esta nomenclatura también se aplica a otros símbolos globales, como estructuras, contantes, o funciones, pero nunca la utilizaremos para métodos ni variables de instancia. Esto es especialmente importante cuando estemos desarrollando librerías o frameworks que vayan a ser reutilizados.

Los métodos se escriben en *lowerCamelCase*. La composición del nombre resulta más compleja que en otros lenguajes, ya que se debe especificar tanto el nombre del método como el de sus parámetros. Si el método devuelve alguna propiedad, debería comenzar con el nombre de la propiedad. Si realiza alguna acción pondremos el verbo de la acción y su objeto directo si lo hubiese. A continuación deberemos poner los nombres de los parámetros. Para cada uno de ellos podemos anteponer una preposición (From, With, For, To, By, etc), y el nombre del parámetro, que podría ir precedido también de algún verbo. También podemos utilizar la conjunción and para separar los parámetros en el caso de que correspondan a una acción diferente.

Nota

En Objective-C es posible crear una clase que no herede de ninguna otra, ni siquiera de `NSObject`. Este tipo de clases se comportarán simplemente como estructuras de datos. También existe una segunda clase raíz diferente de `NSObject`, que es `NSProxy`, pero su uso es menos común (se utiliza para el acceso a objetos distribuidos). Normalmente siempre crearemos clases que en última instancia hereden de `NSObject`.

5.1.2. Implementación de la clase

La implementación se realiza en el fichero .m.

```
#import "UAAsignatura.h"
@implementation UAAsignatura
// Implementación de los métodos
@end
```

Vemos que siempre tenemos un `import` al fichero en el que se ha definido la interfaz. Deberemos añadir cualquier `import` adicional que necesitemos para nuestro código. Podemos implementar aquí los métodos definidos anteriormente:

```
#import "UAAsignatura.h"
```

```

const CGFloat UAHorasPorCredito = 10;
const CGFloat UADescuentoBecario = 0.5;

@implementation UAAsignatura

+ (CGFloat) creditosParaHoras:(CGFloat)horas
{
    return horas / UAHorasPorCredito;
}

- (CGFloat)creditos
{
    return [UAAsignatura creditosParaHoras: _horas];
}

- (CGFloat)tasaConPrecioPorCredito:(CGFloat)precioCredito
                                esBecario:(BOOL)becario
{
    CGFloat precio = [self creditos] * precioCredito;
    if(becario) {
        precio = precio * UADescuentoBecario;
    }
    return precio;
}

@end

```

Nota

Vemos que no es necesario añadir ningún import de las librerías básicas de Cocoa Touch porque ya están incluidos en el fichero Prefix.pch que contiene el prefijo común precompilado para todos nuestros fuentes.

Los métodos pueden recibir un número variable de parámetros. Para ello su firma deberá tener la siguiente forma:

```
+ (void)escribe:(NSInteger)n, ...;
```

Donde el valor n indica el número de parámetros recibidos, que pueden ser de cualquier tipo. En la implementación accederemos a la lista de los parámetros mediante el tipo va_list y una serie de macros asociadas (va_start, va_arg, y va_end):

```

+ (void)escribe:(NSInteger *)numargs, ... {
    va_list parametros;
    va_start(parametros, numargs);
    for(int i=0; i<numargs; i++) {
        NSString *cad = va_arg(parametros, NSString *);
        NSLog(cad);
    }
    va_end(parametros);
}

```

Hemos visto que para cada clase tenemos dos ficheros: el fichero de cabecera y el de implementación. Muchas veces necesitamos alternar entre uno y otro para añadir código o consultar lo que habíamos escrito en el fichero complementario. Para facilitar esta tarea, podemos aprovechar la vista del asistente en Xcode.

```

Build Succeeded | Today at 09:15
No Issues
Editor View Organizer
; | < > | A > Counterparts > h > C @int... | + x
// UAAsignatura.m
// EspecialistaMoviles
//
// Created by Miguel Angel Lozano on 24
// 08/11.
// Copyright 2011 __MyCompanyName__.
// All rights reserved.
//

#import "UAAsignatura.h"

const CGFloat UAHorasPorCredito = 10;
const CGFloat UADescuentoBecario = 0.5;

@implementation UAAsignatura

+ (CGFloat) creditosParaHoras:(CGFloat) horas
{
    return horas / UAHorasPorCredito;
}

- (CGFloat) creditos
{
    ...
}

// UAAsignatura.h
// EspecialistaMoviles
//
// Created by Miguel Angel Lozano on
// 24/08/11.
// Copyright 2011 __MyCompanyName__.
// All rights reserved.

@interface UAAsignatura : NSObject{
    NSString *nombre;
    NSString *descripcion;
    NSUInteger horas;
}
+
+ (CGFloat) creditosParaHoras:(CGFloat) horas;
- (CGFloat) creditos;
- (CGFloat) tasaConPrecioPorCredito:
    (CGFloat) precioCredito esBecario:
    (BOOL) becario;

```

Vista del asistente

Podemos seleccionar el modo asistente mediante los botones que hay en la esquina superior derecha de la interfaz, concretamente el botón central del grupo *Editor*. Por defecto la vista asistente nos abrirá el fichero complementario al fichero seleccionado en la principal, pero esto puede ser modificado pulsando sobre el primer elemento de la barra de navegación del asistente. Por defecto vemos que está seleccionado *Counterparts*, es decir, el fichero complementario al seleccionado. Cambiando este elemento se puede hacer que se abran las subclases, superclases o cualquier otro elemento de forma manual.

Atajo

Podemos abrir directamente un fichero en el asistente si hacemos *Option(alt)-Click* sobre él en el panel del navegador.

Podemos añadir nuevas vistas de asistente, o eliminarlas, mediante los botones (+) y (X) que encontramos en su esquina superior derecha. También podemos cambiar la disposición del asistente mediante el menú *View > Assistant Editor*.

5.1.3. Implementación de inicializadores

Vamos a ver ahora cómo implementar un método de inicialización. Estos serán métodos devolverán una referencia de tipo *id* por convención, en lugar de devolver un puntero del tipo concreto del que se trate. Por ejemplo podemos declarar:

```
- (id)initWithNombre:(NSString*)nombre
                descripcion:(NSString*)descripcion
                    horas:(NSUInteger)horas;
```

Nota

En algunas ocasiones veremos los inicializadores declarados sin especificar ningún tipo de retorno (`- init`). Esto es correcto, ya que en los métodos de Objective-C cuando no se declara tipo de retorno, por defecto se asume que es `id`.

Dentro de estos métodos lo primero que deberemos hacer es llamar al inicializador de la superclase, para que construya la parte del objeto relativa a ella. Si estamos heredando de `NSObject`, utilizaremos `[super init]` para inicializar esta parte, ya que es el único inicializador que define esa clase. Si heredamos de otra clase, podremos optar por otros inicializadores.

Una vez obtenido el objeto ya inicializado por la superclase, comprobaremos si la inicialización se ha podido hacer correctamente (mirando si el objeto devuelto es distinto de `nil`), y en ese caso realizaremos la inicialización pertinente. Finalmente devolveremos la referencia a nuestro objeto inicializado. Esta es la estructura que deberemos utilizar para los inicializadores de Objective-C:

```
- (id)initWithNombre:(NSString*)nombre
              descripcion:(NSString*)descripcion
                  horas:(NSUInteger)horas
{
    self = [super init];
    if(self != nil) {
        _nombre = nombre;
        _descripcion = descripcion;
        _horas = horas;
    }
    return self;
}
```

Aquí vemos por primera vez el uso de las referencias `self` y `super`. La primera de ellas es una referencia a la instancia en la que estamos (equivalente a `this` en Java), mientras que la segunda es una referencia a la superclase, que nos permite llamar sobre ella a métodos que hayan podido ser sobrescritos.

Es importante reasignar la referencia `self` al llamar al constructor de la superclase, ya que en ocasiones es posible que el inicializador devuelva una instancia distinta a la instancia sobre la que se ejecutó. Siempre deberemos utilizar la referencia que nos devuelva el inicializador, no la que devuelva `alloc`, ya que podrían ser instancias distintas, aunque habitualmente no será así.

Nota

Tras la llamada a `alloc`, todas las variables de instancia de la clase se habrán inicializado a 0. Si este es el valor que queremos que tengan, no hará falta modificarlo en el inicializador.

5.1.3.1. Inicializador designado

Normalmente en una clase tendremos varios inicializadores, con distintos número de

parámetros. Por ejemplo en el caso anterior podríamos tener:

```
- (id)init;
- (id)initWithNombre:(NSString*)nombre;
- (id)initWithNombre:(NSString*)nombre
    descripcion:(NSString*)descripcion
    horas:(NSUInteger)horas;
```

La forma de inicializar el objeto será parecida, y por lo tanto podremos encontrar código repetido en estos tres inicializadores. Para evitar que ocurra esto, lo que deberemos hacer es establecer uno de nuestros inicializadores como inicializador designado. El resto de inicializadores siempre deberán recurrir a él para inicializar el objeto en lugar de inicializarlo por si mismos. Para que esto pueda ser así, este inicializador designado debería ser aquel que resulte más genérico, y que reciba un mayor número de parámetros con los que podamos inicializar explícitamente cualquier campo del objeto. Por ejemplo, en el caso anterior nuestro inicializador designado debería ser el que toma los parámetros nombre, descripcion y horas.

Los otros constructores podrán implementarse de la siguiente forma:

```
- (id)init
{
    return [self initWithNombre: @"Sin nombre"];
}

- (id)initWithNombre:(NSString *)nombre
{
    return [self initWithNombre:nombre
        descripcion:@"Sin descripcion"
        horas:-1];
}
```

Como vemos, no es necesario que todos ellos llamen directamente al inicializador designado, sino que pueden llamar a uno más genérico que ellos que a su vez llamará a otro más general hasta llegar al designado. De esta forma conseguimos un código más mantenable, ya que si queremos cambiar la forma de inicializar el objeto (por ejemplo si cambiamos el nombre de los campos), bastará con modificar el inicializador designado.

Deberemos también sobrescribir siempre el inicializador designado de la superclase (en nuestro ejemplo `init`), ya que si no lo hacemos se podría inicializar el objeto con dicho inicializador y nos estaríamos saltando el inicializador designado de nuestra clase, por lo que nuestras variables de instancia no estarían siendo inicializadas.

La única forma de conocer cuál es el inicializador designado de las clases de Cocoa Touch es recurrir a la documentación. En ocasiones el inicializador designado de una clase vendrá heredado por la superclase (en caso de que la subclase no defina constructores).

5.1.4. Gestión de la memoria

En Objective-C existen un sistema de recolección de basura como en el caso de Java, pero

lamentablemente no está disponible para la plataforma iOS, así que deberemos hacer la gestión de memoria de forma manual. No obstante, si seguimos una serie de reglas veremos que esto no resulta demasiado problemático. Además, contamos con los analizadores estático y dinámico que nos permitirán localizar posibles fugas de memoria (objetos instanciados que no llegan a ser liberados).

La gestión manual de la memoria se hace mediante la cuenta del número de referencias. Cuando llamamos a `alloc`, la cuenta de referencias se pone automáticamente a 1. Podemos incrementar número de referencias llamando `retain` sobre el objeto. Podemos decrementar el número de referencias llamando a `release`. Cuando el número de referencias llegue a 0, el objeto será liberado automáticamente de memoria.

Cuando se haga una copia de un objeto mediante el método `copy` el nuevo objeto resultante de la copia tendrá su contador de referencias inicializado a 1 (pertenece al grupo de métodos que incrementan este contador, junto a `alloc` y `retain`).

Podemos decrementar las referencias también con el método `autorelease`. En este caso, en lugar de decrementarse inmediatamente, como ocurría con `release`, lo que se hace es programar una liberación pendiente, que se llevará a cabo cuando termine la pila de llamadas actual. Más adelante veremos la utilidad de este método.

Atención

Lo más importante es que el número total de llamadas a `alloc`, `retain` y `copy` sea el mismo que la suma de las llamadas a `release` y `autorelease`. Es decir, debemos balancear las llamadas a (`alloc-retain-copy`) y (`release-autorelease`). Si faltase alguna llamada a `release` tendríamos una fuga de memoria, ya que habría objetos que no se eliminan nunca de memoria. Si por el contrario se llamase más veces de las necesarias a `release`, podríamos obtener un error en tiempo de ejecución por intentar acceder a una zona de memoria que ya no está ocupada por nuestro objeto.

La cuestión es: ¿cuándo debemos retener y liberar las referencias a objetos? La regla de oro es que **quien retiene debe liberar**. Es decir, en una de nuestras clases nunca deberemos retener un objeto y confiar en que otra lo liberará, y tampoco debemos liberar un objeto que no hemos retenido nosotros.

Cuando un objeto vaya a ser liberado de memoria (porque su número de referencias ha llegado a 0), se llamará a su método `dealloc` (destructor). En dicho método deberemos liberar todas las referencias que tenga retenidas el objeto, para evitar fugas de memoria. Nunca deberemos llamar a este método manualmente, será llamado por el sistema cuando el número de referencias sea 0.

En la inicialización de nuestro objeto habitualmente deberemos retener los objetos referenciados por nuestras variables de instancia, para evitar que dichos objetos puedan ser liberados de la memoria durante la vida de nuestro objeto.

```
- (id)initWithNombre:(NSString*)nombre
               descripcion:(NSString*)descripcion
```

```
    horas:(NSUInteger)horas
{
    self = [super init];
    if(self != nil) {
        _nombre = [nombre retain];
        _descripcion = [descripcion retain];
        _horas = horas;
    }
    return self;
}
```

Todas aquellas referencias que hayan sido retenidas (conocidas como referencias fuertes), deberán ser liberadas antes de que nuestro objeto deje de existir. Para eso utilizaremos `dealloc`:

```
- (void)dealloc
{
    [_nombre release];
    [_descripcion release];
    [super dealloc];
}
```

Siempre deberemos llamar a `[super dealloc]` tras haber liberado las referencias retenidas de nuestras variables de instancia, para así liberar todo lo que haya podido retener la superclase.

5.1.4.1. Autorelease

Hemos visto que la regla que debemos seguir es que siempre deberá liberar las referencias quien las haya retenido. Pero a veces esto no es tan sencillo. Por ejemplo imaginemos un método que debe devolvernos una cadena de texto (que se genera dentro de ese mismo método):

```
- (NSString*) nombreAMostrar
{
    return [[NSString alloc] initWithFormat:@"%@", _nombre, _horas];
}
```

El método crea una nueva cadena, que tendrá el contador de referencias inicializado a 1 (tras llamar a `alloc`). Desde este método se devuelve la cadena y nunca más sabrá de ella, por lo que no podrá liberarla, lo cual hemos dicho que es responsabilidad suya. Tampoco podemos llamar a `release` antes de finalizar el método, ya que si hacemos eso quien reciba el resultado recibirá ya ese espacio liberado, y podría tener un error de acceso a memoria.

La solución es utilizar `autorelease`. Este método introduce un `release` pendiente para el objeto en un *pool* (conocido como *autorelease pool*). Estos `releases` pendientes no se ejecutarán hasta que haya terminado la pila de llamadas a métodos, por lo que podemos tener la seguridad de que al devolver el control nuestro método todavía no se habrá liberado el objeto. Eso sí, quien reciba el resultado, si quiere conservarlo, tendrá que retenerlo, y ya será responsabilidad suya liberarlo más adelante, ya que de no hacer esto,

el objeto podría liberarse en cualquier momento debido al `autorelease` pendiente. La forma correcta de implementar el método anterior sería:

```
- (NSString*) nombreAMostrar
{
    return [[[NSString alloc] initWithFormat:@"%@ (%d horas)",
                                         _nombre, _horas] autorelease];
}
```

Con esto cumplimos la norma de que quien lo retiene (`alloc`, `retain`, `copy`), debe liberarlo (`release`, `autorelease`).

Cambios en iOS 5

En iOS 5 se introduce una característica llamada *Automatic Reference Counting* (ARC), que nos libera de tener que realizar la gestión de la memoria. Si activamos esta característica, no deberemos hacer ninguna llamada a `retain`, `release`, o `autorelease`, sino que el compilador se encargará de detectar los lugares en los que es necesario realizar estas acciones y lo hará por nosotros. Por el momento hemos dejado esta opción deshabilitada, por lo que deberemos gestionar la memoria de forma manual. Más adelante veremos con más detalle el funcionamiento de ARC.

5.1.4.2. Autorelease pool

Las llamadas a `autorelease` funcionan gracias a la existencia de lo que se conoce como *autorelease pool*, que va acumulando estos `releases` pendientes y en un momento dado los ejecuta. Podemos ver el *autorelease pool* de nivel superior en el fichero `main.m`:

```
int main(int argc, char *argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

Podemos ver que la aplicación `UIApplication` se ejecuta dentro de un *autorelease pool*, de forma que cuando termine se ejecutarán todos los `autorelease` pendientes. Normalmente no deberemos preocuparnos por eso, ya que la API de Cocoa habrá creado *autorelease pools* en los lugares apropiados (en los eventos que llaman a nuestro código), pero en alguna ocasión puede ser necesario crear nuestros propios *autorelease pools*. Por ejemplo, en el caso en que tengamos un bucle que itera un gran número de veces y que en cada iteración crea objetos con `autoreleasepool`s pendientes. Para evitar quedarnos sin memoria, podemos introducir un *autorelease pool* propio dentro del bucle:

```
for(int i=0; i<n; i++) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSString *cadena = [NSString string];
    ...
    [pool release];
}
```

De esta forma estamos haciendo que en cada iteración del bucle se liberen todos los

autoreleases pendientes que hubiese. Cuando llamemos a `autorelease`, siempre se utilizará el *pool* de ámbito más cercano que exista.

5.1.5. Métodos factoría

Antes hemos visto que a parte de los inicializadores, encontramos normalmente una serie de métodos factoría equivalentes. Por ejemplo a continuación mostramos un par inicializador-factoría que se puede encontrar en la clase `NSString`:

```
- (id)initWithFormat:(NSString *)format ...;
- (id)stringWithFormat:(NSString *)format ...;
```

El método factoría nos permite instanciar e inicializar el objeto con una sola operación, es decir, él se encargará de llamar a `alloc` y al inicializador correspondiente, lo cual facilita la tarea del desarrollador. Pero si seguimos la regla anterior, nos damos cuenta de que si es ese método el que está llamando a `alloc` (y por lo tanto reteniendo el objeto), deberá ser él también el que lo libere. Esto se hará gracias a `autorelease`. La implementación típica de estos métodos factoría es:

```
+ (id)asignaturaWithNombre:(NSString*)nombre
                        descripcion:(NSString*)descripcion
                           horas:(NSUInteger)horas
{
    return [[[UAAsignatura alloc] initWithNombre:nombre
                                         descripcion:descripcion
                                         horas:horas] autorelease];
}
```

Podemos observar que estos métodos son métodos de clase (nos sirven para crear una instancia de la clase), y dentro de ellos llamamos a `alloc`, al método `init` correspondiente, y a `autorelease` para liberar lo que hemos retenido, pero dando tiempo a que quien nos llame pueda recoger el resultado y retenerlo si fuera necesario.

5.1.5.1. Patrón singleton

Para implementar el patrón *singleton* en Objective-C necesitaremos un método factoría que nos proporcione la instancia única del objeto, y una variable de tipo `static` que almacene dicha instancia. El método factoría tendrá el siguiente aspecto:

```
+ (UADatosCompartidos) sharedDatosCompartidos {
    static DatosCompartidos *datos = nil;
    if(nil == datos) {
        datos = [[DatosCompartidos alloc] init];
    }
    return datos;
```

Nota

Con esto no estamos impidiendo que alguien pueda crear nuevas instancias del objeto llamando directamente a `alloc`, en lugar de pasar por la factoría. Para evitar esto, podríamos sobrescribir el método `allocWithZone` para que sólo instancie el objeto una única vez, y también podríamos sobrescribir los métodos de gestión de memoria para inhabilitarlos (`retain`,

release y autorelease).

5.1.6. Copia de objetos

La clase `NSObject` incorpora el método `copy` que se encarga de crear una copia de nuestro objeto. Sin embargo, para que dicho método funcione necesita que esté implementado el método `copyWithZone` que no está en `NSObject`, sino en el protocolo `NSCopying`. Por lo tanto, sólo los objetos que adopten dicho protocolo serán copiables.

Gran parte de los objetos de la API de Cocoa Touch implementan `NSCopying`, lo cual quiere decir que son copiables. Si queremos implementar este protocolo en nuestros propios objetos, la forma de implementarlo dependerá de si nuestra superclase ya implementaba dicho protocolo o no:

- Si la superclase no implementa el protocolo, deberemos implementar el método `copyWithZone` de forma que dentro de él se llame a `allocWithZone` (o `alloc`) para crear una nueva instancia de nuestro objeto y al inicializador correspondiente para inicializarlo con los datos de nuestro objeto actual. Además deberemos copiar los valores de las variables de instancia que no hayan sido asignadas con el inicializador.

```
- (id)copyWithZone:(NSZone *)zone
{
    return [[UAAsignatura allocWithZone:zone]
            initWithNombre:_nombre
            descripcion:_descripcion
            horas:_horas];
}
```

- Si nuestra superclase ya era copiable, entonces deberemos llamar al método `copyWithZone` de la superclase y una vez hecho esto copiar los valores de las variables de instancia definidas en nuestra clase, que no hayan sido copiadas por la superclase.

```
- (id)copyWithZone:(NSZone *)zone
{
    id copia = [super copyWithZone: zone];

    // Copiar propiedades de nuestra clase
    [copia setNombre: _nombre];
    ...

    return copia;
}
```

Otra cosa que debemos tener en cuenta al implementar las copias, es si los objetos son **mutables** o **inmutables**. En caso de que nuestro objeto sea **inmutable**, podemos optimizar la forma de hacer las copias. Si el estado del objeto no va a cambiar, no es necesario crear una nueva instancia en memoria. Obtendremos el mismo resultado si en el método `copy` nos limitamos a retener el objeto con `retain` (esto es necesario ya que siempre se espera que `copy` retenga el objeto).

Los objetos que existan en ambas modalidades (mutable e inmutable) pueden adoptar

también el protocolo `NSMutableCopy`, que nos obligará a implementar el método `mutableCopyWithZone`, permitiéndonos utilizar `mutableCopy`, para así poder hacer la copia de las dos formas vistas anteriormente. En estos objetos `copy` realizará la copia inmutable simplemente reteniendo nuestro objeto, mientras que `mutableCopy` creará una copia mutable como una nueva instancia.

Vamos a ver como ejemplo de objeto que existe en ambas modalidades el caso de las cadenas: `NSString` y `NSMutableString`. Si de un objeto `NSString` hacemos un `copy`, simplemente se estará reteniendo el mismo objeto. Si hacemos `mutableCopy` estaremos creando una copia del objeto, que además será de tipo `NSMutableString`.

```
NSString *cadena = @"Mi cadena";
// copiaInmutable==cadena
NSString *copiaInmutable = [cadena copy];
// copiaMutable!=cadena
NSMutableString *copiaMutable = [cadena mutableCopy];
```

Si nuestro objeto original fuese una cadena mutable, cualquiera de los dos métodos de copia creará una nueva instancia. La copia inmutable creará un instancia de tipo `NSString` que no podrá ser alterada, mientras que la copia mutable creará una nueva instancia de tipo `NSMutableString`, que se podrá alterar sin causar efectos laterales con nuestro objeto original.

```
NSMutableString *cadenaMutable = [NSMutableString stringWithCapacity: 32];
// copiaInmutable!=cadena
NSString *copiaInmutable = [cadenaMutable copy];
// copiaMutable!=cadena
NSMutableString *copiaMutable = [cadenaMutable mutableCopy];
```

5.2. Propiedades de los objetos

Como hemos visto, las variables de instancia de los objetos normalmente son protegidas y accederemos a ellas a través de métodos accesores (*getters* y *setters*). La forma de escribir estos métodos será siempre la misma, por lo que suele resultar una tarea bastante repetitiva y anodina, que puede ser realizada perfectamente de forma automática. Por ejemplo, en Java normalmente los IDEs nos permiten generar automáticamente estos métodos.

En Objective-C contamos con las denominadas propiedades, que realmente son una forma de acceso que nos da el lenguaje a las variables de instancia sin que tengamos que implementar manualmente los métodos accesores.

Las propiedades se definen dentro de la declaración de la interfaz mediante la etiqueta `@property`. La etiqueta puede tomar una serie de atributos, como por ejemplo `nonatomic`, que indica que los métodos accesores no deben ser atómicos, es decir, que no se debe sincronizar el acceso a ellos desde diferentes hilos. Normalmente todas las

propiedades serán de este tipo, ya que no es frecuente que podamos tener problemas de concurrencia en las aplicaciones que desarrollaremos habitualmente, y sincronizar el acceso tiene un elevado coste en rendimiento. Tras `@property` y sus atributos declararemos el tipo y el nombre de la propiedad.

```
@interface UAAsignatura : NSObject{
    NSString *_nombre;
    NSString *_descripcion;
    NSUInteger _horas;
}

- (id)init;
- (id)initWithNombre:(NSString*)nombre;
- (id)initWithNombre:(NSString*)nombre
                  descripcion:(NSString*)descripcion
                  horas:(NSUInteger)horas;

+ (id)asignatura;
+ (id)asignaturaWithNombre:(NSString*)nombre;
+ (id)asignaturaWithNombre:(NSString*)nombre
                  descripcion:(NSString*)descripcion
                  horas:(NSUInteger)horas;

+ (CGFloat)creditosParaHoras:(CGFloat)horas;
- (CGFloat)creditos;
- (CGFloat)tasaConPrecioPorCredito:(CGFloat)precioCredito
                                esBecario:(BOOL)becario;

@property(nonatomic,retain) NSString *nombre;
@property(nonatomic,retain) NSString *descripcion;
@property(nonatomic,assign) NSUInteger horas;

@end
```

En el anterior ejemplo hemos declarado tres propiedades (`nombre`, `descripcion` y `horas`). Con esto realmente lo que estamos haciendo es declarar implícitamente que vamos a tener los métodos accesores siguientes:

```
- (NSString *)nombre;
- (void)setNombre: (NSString *)nombre;
- (NSString *)descripcion;
- (void)setDescripcion: (NSString *)descripcion;
- (NSUInteger)horas;
- (void)setHoras: (NSUInteger)horas;
```

Pero si sólo declaramos las propiedades en la interfaz obtendremos un *warning* del compilador, ya que no tenemos la implementación de los métodos implícitos que hemos declarado. Además, simplemente estamos declarando los métodos accesores, por el momento no los estamos vinculando a ninguna variable de instancia. Podemos hacer que la implementación se genere de forma automática incluyendo la etiqueta `@synthesize` seguida del nombre de la propiedad cuyos métodos accesores queremos implementar. Si el nombre de la propiedad no coincide con el nombre de la variable de instancia a la que vamos a acceder (como es el caso de nuestro ejemplo), podemos vincularla añadiendo `= nombreVariable` tras el nombre de la propiedad:

```
@implementation UAAsignatura
```

```
@synthesize nombre = _nombre;
@synthesize descripcion = _descripcion;
@synthesize horas = _horas;

// Implementacion del resto de metodos
...
@end
```

Con esto se crearán las implementaciones de los *getters* y *setters* sin tener que escribirlos de forma explícita en el fichero de código (el compilador ya sabe cómo crearlos).

Nota

Es recomendable hacer que la propiedad tenga un nombre diferente al de la variable de instancia a la que se accede para evitar confusiones (un error común es utilizar la variable de instancia cuando se debería estar utilizando la propiedad). Habitualmente encontraremos que a las variables de instancia se le añade el prefijo `_` para evitar confusiones, como hemos hecho en nuestro ejemplo. Si las nombramos de forma distinta e intentamos utilizar la propiedad donde debería ir la variable de instancia (o viceversa) nos aparecerá un error de compilación, y pinchando sobre él en el entorno nos permitirá corregirlo automáticamente (*fix-it*).

Como alternativa a `@synthesize` también podríamos implementar manualmente los accesores para cada propiedad, de forma que podríamos personalizar la forma de acceder a los campos, o incluso introducir propiedades que no estén vinculadas a ninguna variable de instancia.

Nota

Si utilizamos `@synthesize` no es necesario declarar manualmente las variables de instancia asociadas a las propiedades. En caso de no existir las variables de instancia especificadas, el compilador se encargará de crearlas por nosotros.

5.2.1. Acceso a las propiedades

Las propiedades definen una serie de métodos accesores, por lo que podremos acceder a ellas a través de estos métodos, pero además encontramos una forma alternativa de acceder a ellas mediante el operador `..`. Por ejemplo podríamos utilizar el siguiente código:

```
asignatura.nombre = @"Plataforma iOS";
 NSLog(@"Nombre: %@", asignatura.nombre);
```

Este código es equivalente a llamar a los métodos accesores:

```
[asignatura setNombre: @"Plataforma iOS"];
 NSLog(@"Nombre: %@", [asignatura nombre]);
```

Advertencia

Hay que destacar que el operador `.` no sirve para acceder a las variables de instancia directamente, sino para acceder a las propiedades mediante llamadas a los métodos accesores.

Este es uno de los motivos por los que es conveniente que las propiedades no tengan el mismo nombre que las variables de instancia, para evitar confundir ambas cosas.

5.2.2. Gestión de la memoria

Lo más habitual es que los objetos que referenciamos desde las variables de instancia de nuestros objetos deban ser retenidos para evitar que se libere su memoria durante la vida de nuestro objeto. Esto se tendrá que tener en cuenta en los *setters* generados para que se libere la referencia anterior (si la hubiese) y retenga la nueva. Esta es la política que se conoce como `retain`. En otros casos no nos interesa que se retenga, sino simplemente asignar la referencia (para así evitar posibles referencias cíclicas que puedan causar fugas de memoria). Esta política se conoce como `assign`. Por último, puede que nos interese no sólo retener el objeto, sino retener una copia del objeto que se ha recibido como parámetro para evitar efectos laterales al modificar un objeto mutable desde lugares diferentes. Esta política se conoce como `copy`, y sólo la podremos aplicar si el tipo de datos de la propiedad implementa el protocolo `NSCopying`.

La política de gestión de memoria se indica como atributo de la etiqueta `@property` (`assign`, `retain`, `copy`). Vamos a ver cada una de ellas con más detalle.

El caso más sencillo es el de la política `assign`. Esta será la política que se aplique por defecto, o la que se debe aplicar en cualquier tipo de datos que no sea un objeto, aunque también podemos aplicarla a objetos. Los accesores generados con ella tendrán la siguiente estructura:

```
- (NSString *)nombre {
    return _nombre;
}

- (void)setNombre:(NSString *)nombre {
    _nombre = nombre;
}
```

La política más común para las propiedades de tipo objeto será `retain`. Con ella el *getter* será igual que en el caso anterior, mientras que el *setter* tendrá la siguiente estructura:

```
- (void)setNombre:(NSString *)nombre {
    if(_nombre != nombre) {
        [nombre retain];
        [_nombre release];
        _nombre = nombre;
    }
}
```

En este caso en el *setter* eliminamos la referencia anterior, si la hubiese (si no la hay, `_nombre` será `nil` y enviar el mensaje `release` no tendrá ningún efecto). El nuevo valor se retiene y se asigna a la variable de instancia.

Podemos ver también que se comprueba si el nuevo valor es el mismo que el anterior. En tal caso no se hace nada, ya que no merece la pena realizar tres operaciones innecesarias.

Por último, tenemos el modificador `copy`, que realiza una copia del objeto que pasamos como parámetro del *setter*. El *getter* será como en el caso anterior.

```
- (void)setNombre:(NSString *)nombre {
    NSString *nuevoObjeto = [nombre copy];
    [_nombre release];
    _nombre = nuevoObjeto;
}
```

En este caso el objeto que pasemos debe implementar el protocolo `NSCopying` para poder ser copiado. Siempre se realizará la copia inmutable (`copy`). Si queremos realizar una copia mutable deberemos definir nuestro propio *setter*. En este *setter* vemos también que no se comprueba si los objetos son el mismo, ya que en el caso de las copias es improbable (aunque podría pasar, ya que los objetos inmutables se copian simplemente reteniendo una referencia).

Tanto si se retiene (`retain`) como si se copia (`copy`) nuestro objeto tendrá una referencia pendiente de liberar. Se liberará cuando se asigne otro objeto a la propiedad, o bien cuando nuestro objeto sea destruido. En este último caso nosotros seremos los responsables de escribir el código para la liberación de la referencia en el método `dealloc`, ya que `@synthesize` sólo se ocupa de los *getters* y *setters*. En nuestro caso deberíamos implementar `dealloc` de la siguiente forma:

```
- (void)dealloc
{
    [_nombre release];
    [_descripcion release];
    [super dealloc];
}
```

Estamos liberando todas las variables de instancia asociadas a propiedades con política `retain` o `copy`. Nunca deberemos liberar propiedades con política `assign`.

De la misma forma, en el inicializador también deberíamos retener las variables de instancia si se les da un valor, ya que al utilizar estos modos se espera que las variables de instancia tengan una referencia pendiente de liberar cuando sean distintas de `nil`. Las variables que correspondan a propiedades de tipo `assign` nunca deberán ser retenidas, ya que eso provocaría una fuga de memoria cuando asignásemos un nuevo valor:

```
- (id)initWithNombre:(NSString*)nombre descripcion:(NSString*)descripcion
horas:(NSUInteger)horas
{
    self = [super init];
    if(self != nil) {
        _nombre = [nombre retain];
        _descripcion = [descripcion retain];
        _horas = horas;
    }
    return self;
}
```

Referencias cíclicas

Las referencias en las que se retiene el objeto (`retain`, `copy`) son conocidas como referencias fuertes, mientras que las de tipo `assign` son referencias débiles. Un problema que podemos

tener con las referencias fuertes es el de las referencias cíclicas. Imaginemos una clase A con una referencia fuerte a B, y una clase B con una referencia fuerte a A. Puede que ninguna otra clase de nuestra aplicación tenga una referencia hacia ellas, por lo que estarán inaccesibles, pero si entre ellas tienen referencias su cuenta de referencias será mayor que 0 y por lo tanto nunca serán borradas (tenemos una fuga de memoria).

Para evitar las referencias cíclicas fundamentalmente debemos seguir la regla de que, dada una jerarquía de clases, sólo debe haber referencias fuertes desde las clases padre a las clases hijas. Las referencias que pueda haber desde una clase a sus ancestros deberán ser siempre débiles, para evitar así los ciclos. Si una clase A que ha sido referenciada débilmente desde otra clase B va a ser eliminada, siempre deberá avisarse a B de este hecho poniendo la referencia a `nil`, para evitar posibles accesos erróneos a memoria.

5.2.3. Propiedades atómicas

En los ejemplos anteriores hemos considerado siempre propiedades no atómicas (con el modificador `nonatomic`), que serán las que utilizaremos en casi todos los casos. Si no incluyésemos este modificador, la propiedades se considerarían atómicas, y el código de todos los *getters* y *setters* quedaría envuelto en un bloque sincronizado:

```
- (void)setNombre:(NSString *)nombre {
    @synchronized(self) {
        if(_nombre != nombre) {
            [nombre retain];
            [_nombre release];
            _nombre = nombre;
        }
    }
}
```

Con la directiva `@synchronized` estamos creando un bloque de código sincronizado que utilizará como cerrojo el objeto actual (`self`). Esto es equivalente a declarar un método como `synchronized` en Java.

Pero donde encontramos una mayor diferencia es en los *getters* para los modos `retain` y `copy`. En las propiedades no atómicas estos métodos se limitaban a devolver el valor de la variable de instancia. Sin embargo, cuando la propiedad sea atómica tendrán el siguiente aspecto:

```
- (NSString *)nombre {
    @synchronized(self) {
        return [[_nombre retain] autorelease];
    }
}
```

¿Por qué se llama a `retain` y a `autorelease`? Esto nos va a garantizar que el objeto devuelto va a seguir estando disponible hasta terminar la pila de llamadas. Evitará que podamos tener problemas en casos como el siguiente:

```
NSString *oldNombre = asignatura.nombre;
asignatura.nombre = @"Nuevo nombre"; // Hace un release del nombre antiguo
NSLog(@"Antiguo nombre: %@", oldNombre);
```

Si no se hubiese retenido el nombre antiguo antes de devolverlo, el *setter* lo habría borrado de la memoria y podríamos tener un error de acceso en el `NSLog`. Con propiedades atómicas no tendremos problemas en este caso, pero con las no atómicas, que son las que utilizaremos más comúnmente este código daría error.

Como vemos, las propiedades atómicas, a parte de ser sincronizadas, incorporan dos operaciones adicionales de gestión de memoria en los *getters*, por lo que si se accede a ellas muy frecuentemente vamos a tener un impacto en el rendimiento. Por este motivo utilizaremos normalmente las propiedades no atómicas, aunque deberemos llevar cuidado para evitar problemas como el anterior.

5.2.4. Automatic Reference Counting (ARC)

A partir de Xcode 4.2 se incluye la característica *Automatic Reference Counting* (ARC), que nos permite dejar que sea el compilador quien se encargue de la gestión de la memoria, liberando al desarrollador de esta tarea. Es decir, ya no será necesario hacer ninguna llamada a los métodos `retain`, `release`, y `autorelease`, sino que el compilador se encargará de detectar dónde es necesario introducir las llamadas a dichos métodos y lo hará por nosotros.

Podemos activar ARC al crear un nuevo proyecto, o si ya tenemos un proyecto creado podemos migrarlo mediante la opción *Edit > Refactor > Convert to Objective-C ARC ...*.

Esta es una característica aportada por el compilador LLVM 3.0 de Apple. Por lo tanto, para comprobar si está activa deberemos ir a *Build Settings > Apple LLVM Compiler 3.0 - Language > Objective-C Automatic Reference Counting*.

Si decidimos trabajar con ARC, debemos seguir una serie de reglas:

- No sólo no es necesario utilizar `retain`, `release`, `autorelease`, sino que intentar llamar a cualquiera de estos métodos resultará en un error de compilación. Tampoco se permitirá consultar el contador de referencias (propiedad `retainCount`), ni deberemos llamar al método `dealloc`. Ya no será necesario por lo tanto implementar el método `dealloc` en nuestras clases, a no ser que queramos liberar memoria que no corresponda a objetos de Objective-C, o que queramos realizar cualquier otra tarea de finalización de recursos. En caso de implementar dicho método, no deberemos llamar a `[super dealloc]`, ya que esto nos daría un error de compilación. La cadena de llamadas al `dealloc` de la superclase está automatizada por el compilador.
- No debemos crear referencias a objetos Objective-C dentro de estructuras C, ya que quedaría fuera de la gestión que es capaz de hacer ARC. En lugar de estructuras C, podemos crear objetos Objective-C que hagan el papel de estructura de datos.
- No se debe crear un *autorelease pool* de forma programática como hemos visto anteriormente, sino que debemos utilizar un bloque de tipo `@autoreleasepool`:

```
@autoreleasepool {
    ...
}
```

- No se permite hacer un *cast* directo entre objetos de Objective-C (`id`) y punteros genéricos (`void *`).

ARC nos evita tener que realizar la gestión de la memoria, pero no nos protege frente a posibles retenciones cíclicas, las cuales causarían una fuga de memoria. Para evitar que esto ocurra deberemos marcar de forma adecuada las propiedades como fuertes o débiles, evitando siempre que se crucen dos referencias fuertes. Para esto se introducen dos nuevos tipos de propiedades:

- `strong`: Es equivalente a `retain`, aunque se recomienda pasar a usar `strong`, ya que en futuras versiones `retain` podría desaparecer (en las plantillas creadas veremos que siempre se utiliza `strong`, incluso cuando no está activado ARC). Las propiedades marcadas con `strong` retienen los objetos a los que referencian en memoria.
- `weak`: Referencia de forma débil a un objeto, es decir, sin retenerlo en memoria. Para que el objeto siga existiendo debe existir en algún otro lugar del código una referencia fuerte hacia él. Se diferencia de `assign` en que si el objeto es liberado, la propiedad `weak` pasará automáticamente a valer `nil`, lo cual nos protegerá de accesos inválidos a memoria.
- `unsafe_unretained`: Es equivalente a `assign`. Lo normal para objetos de Objective-C será utilizar siempre `strong` o `weak`, pero para tipos básicos deberemos utilizar `assign`. El tipo `unsafe_unretained` es equivalente y se podría utilizar también en estos casos, pero por semántica, resultará más adecuado para referencias débiles a objetos que no se vayan a poner a `nil` de forma automática cuando se libere el objeto.

Atención

Sólo podremos utilizar propiedades `weak` en dispositivos iOS 5 (o superior) y en aplicaciones con ARC activo. Si queremos que nuestra aplicación sea compatible con versiones anteriores de iOS, podremos utilizar ARC (ya que las operaciones de gestión de memoria se añaden en tiempo de compilación), pero no podremos utilizar referencias de tipo `weak`. En su lugar deberemos utilizar el tipo `unsafe_unretained`, y asegurarnos de poner a `nil` el puntero de forma manual cuando el objeto vaya a ser liberado en otro lugar del código.

En el caso de las variables (no propiedades), por defecto siempre establecerán una referencia fuerte durante su tiempo de vida. Sin embargo, encontramos una serie de modificadores para cambiar este comportamiento:

- `__strong`: Es el tipo por defecto. No hace falta declararla explícitamente de esta forma, ya que cualquier variable por defecto establecerá una referencia fuerte con el objeto al que apunta.
- `__weak`: Funciona de la misma forma que las propiedades de tipo `weak`. Debemos llevar cuidado al utilizar este tipo, ya que si el objeto al que referenciamos no está referenciado de forma fuerte desde ningún otro sitio, será liberado al instante. Por ejemplo, en el siguiente código:

```
NSString __weak *cadena = [[NSString alloc]
                           initWithFormat:@"Edad: %d", edad];
```

```
NSLog(cadena);
```

Estaremos haciendo un *log* de `nil`, ya que la cadena recién creada no ha sido referenciada de forma fuerte por nadie, y por lo tanto ha sido liberada al instante.

- `__unsafe_unretained`: Funciona de la misma forma que las propiedades de tipo `unsafe_unretained`.
- `__autoreleasing`: Este tipo es útil cuando vayamos a pasar una variable por referencia, para evitar que el compilador se confunda con su ciclo de vida. Por ejemplo, se utilizará cuando pasemos un parámetro de error:

```
NSError __autoreleasing *error = nil;  
[objeto realizarOperacionConError:&error];
```

El objeto `NSError` se instancia dentro del método al que llamamos, pero dentro de ese método se ve como una variable local, por lo que para el compilador su vida termina con la finalización del método, y por lo tanto nosotros no podríamos ver su contenido. Para evitar que esto ocurra, se utiliza el tipo `__autoreleasing` que añadirá el objeto al *autorelease pool* más cercano, y por lo tanto lo tendremos disponible en toda la pila de llamadas hasta llegar a la liberación del *pool*.

5.2.5. Métodos generados

Otros modificadores que podemos incorporar a las propiedades son `readonly` y `readwrite`. Con ellas realmente lo que se está indicando es si sólo queremos declarar los *getters* (`readonly`), o si además también se quieren declarar los *setters* (`readwrite`). Por defecto se considerará que son de lectura/escritura.

Además, también podemos modificar el nombre de los *getters* y *setters* de una propiedad dada mediante los modificadores `getter=nombreGetter` y `setter=nombreSetter`.

5.2.6. Acceso directo a las variables de instancia

Si hemos declarado las variables de instancia como públicas, podremos acceder a ellas directamente con el operador `->`:

```
asignatura->_nombre = @"Plataforma iOS";
```

Hemos de tener en cuenta que si accedemos de esta forma no se estarán ejecutando los *getters* y *setters*, sino que estaremos manipulando directamente la variable, por lo que no se estará realizando ninguna gestión de las referencias al objeto.

Por este motivo se desaconseja totalmente esta forma de acceso. Siempre deberemos acceder a los campos de nuestros objetos con el operador `..`, o bien con los *getters* y *setters*.

Nota

El operador `.` no sólo se limita a las propiedades declaradas, sino que puede aplicarse a cualquier método definido en el objeto. Sin embargo, no debemos abusar de este posible uso, y deberíamos limitar la utilización de este operador al acceso a propiedades del objeto.

5.3. Key-Value-Coding (KVC)

KVC es una característica del lenguaje Objective-C que nos permite acceder a las propiedades de los objetos proporcionando una cadena con el nombre de la propiedad. Todos los objetos incorporan un método `valueForKey:` que nos permite acceder a las propiedades de esta forma:

```
NSString *nombre = [asignatura valueForKey: @"nombre"];
NSNumber *horas = [asignatura valueForKey: @"horas"];
```

Esto será útil cuando no conocemos el nombre de la propiedad a la que queremos acceder hasta llegar a tiempo de ejecución. Podemos observar también que cuando tenemos propiedades que no son objetos, las convierte al *wrapper* adecuado (en el ejemplo anterior `NSUInteger` a pasado a `NSNumber`).

También podemos modificar el valor de las propiedades mediante KVC con el método `setValue:forKey:`:

```
[asignatura setValue:@"Proyecto iOS" forKey:@"nombre"];
[asignatura setValue:[NSNumber numberWithInteger:30] forKey:@"horas"];
```

Debemos destacar que KVC siempre intentará acceder a las variables de instancia a través de los métodos accesores (para que los encuentre deberán llamarse de la forma estándar: `nombre` y `setNombre`), pero si estos métodos no estuviesen disponibles, accedería a ella directamente de la misma forma, incluso tratándose de una variable privada. El modificador `@private` sólo afecta al compilador, pero no tiene ningún efecto en tiempo de ejecución.

También podemos acceder a propiedades mediante una ruta de claves (*key path*). Esto será útil cuando tengamos objetos anidados y desde un nivel alto queramos consultar propiedades de objetos más profundos en la jerarquía. Para ello tenemos `valueForKeyPath:` y `setValue:forKeyPath:`:

```
NSString *nombreCoordinador =
[asignatura valueForKeyPath:@"coordinador.nombre"];
```

Cada elemento de la ruta se separará mediando el símbolo punto (.).

5.3.1. KVC y colecciones

Cuando aplicamos KVC sobre colecciones de tipo `NSArray` o `NSSet`, lo que estaremos haciendo es acceder simultáneamente a todos los elementos de la colección. Esto nos permite modificar una determinada propiedad de todos los elementos con una única operaciones, u obtener la lista de valores de una determinada propiedad de los objetos de la colección.

```
[lista setValue:[NSNumber numberWithInteger:50] forKey:@"horas"];
NSArray *nombres = [lista valueForKey:@"nombre"];
```

```
for(NSString *nombre in nombres) {
    NSLog(@"Nombre asignatura: %@", nombre);
}
```

El caso de `NSDictionary` es distinto. De hecho, la forma de acceder a las propiedades de los objetos mediante KVC (`valueForKey:`) nos recuerda mucho a la forma en la que se accede a los objetos en los diccionarios a partir de su clave (`objectForKey:`). La diferencia entre ambos métodos es que en KVC las claves siempre deben ser cadenas, mientras que en un diccionario podemos utilizar como clave cualquier tipo de objeto. Sin embargo, si tenemos un diccionario en el que las claves sean cadenas, también podremos acceder a los objetos mediante el método `valueForKey:` de KVC. Por este motivo se recomienda utilizar siempre cadenas como claves.

5.3.2. Acceso mediante listas

Puede que algunas variables de instancia de nuestros objetos sean colecciones de datos, y podremos acceder a ellas y a sus elementos como hemos visto anteriormente. Pero puede que tengamos alguna propiedad que nos interese que se comporte como un *array*, pero que realmente no esté asociada a una variable de tipo `NSArray`.

Cuando teníamos propiedades con un único valor (por ejemplo `nombre`), nos bastaba con tener definidos dos métodos accesores: el *getter* con el mismo nombre que la propiedad (`nombre`) y el *setter* con el prefijo `set` (`setNombre`), aunque no existiese realmente ninguna variable de instancia llamada `nombre`.

En el caso de las listas necesitaremos definir métodos adicionales. Por ejemplo, vamos a considerar que tenemos un propiedad `profesores`, pero no tenemos a los profesores almacenados en ningún `NSArray`. Podemos hacer que KVC acceda a ellos como si se tratase de una lista definiendo los siguientes métodos:

```
- (NSUInteger) countOfProfesores {
    return numeroProfesores;
}

- (id) objectInProfesoresAtIndex: (NSUInteger) index {
    [self obtenerProfesorEnPosicion: index];
}
```

Si definimos estos métodos, aunque los profesores no estén almacenados en ningún *array*, podremos obtenerlos en forma de `NSArray` mediante KVC:

```
NSArray *profesores = [asignatura valueForKey: @"profesores"];
```

Podríamos también obtener el listado de una determinada propiedad de los profesores, utilizando un *key path*, o modificar la misma propiedad para todos los profesores.

Esta característica resultará de gran utilidad para el acceso a bases de datos, ya que nos permitirá de forma sencilla definir objetos que mapeen los datos almacenados en la BD a objetos.

Podemos incluso acceder al objeto mediante un *array* de tipo mutable. Para ello necesitamos implementar dos métodos adicionales:

```
- (void) insertObject:(id)obj inProfesoresAtIndex:(NSUInteger)index {
    [self insertarProfesor: obj enPosicion: index];
}

- (void) removeObjectFromProfesoresAtIndex: (NSUInteger) index {
    [self eliminarProfesorEnPosicion: index];
}
```

Si añadimos estos métodos podremos obtener un `NSMutableArray` para manipular los profesores, aunque internamente estén almacenados de otra forma:

```
NSMutableArray *profesores =
    [asignatura mutableArrayValueForKey: @"profesores"];
```

5.4. Protocolos

Los protocolos son el equivalente a las interfaces en Java. Definen una serie de métodos, que los objetos que los adopten tendrán que implementar. Los protocolos se definen de la siguiente forma:

```
@protocol UACalificable
- (void)setNota: (CGFloat) nota;
- (CGFloat)nota;
@end
```

Para hacer que una clase adopte lo especificaremos entre `< . . . >` tras el nombre de la superclase en la declaración de nuestra clase:

```
@interface UAAsignatura : NSObject<UACalificable>
...
@end
```

Podemos utilizar el tipo referencia genérica `id` junto a un nombre de protocolo para referenciar cualquier clase que adopte el protocolo, independientemente de su tipo concreto, y que el compilador verifique que dicha clase al menos define los métodos declarados en el protocolo (en caso contrario obtendríamos un *warning*):

```
id<UACalificable> objetoCalificable;
```

Los protocolos en Objective-C nos permiten definir tanto métodos de implementación obligatoria (por defecto) como opcional:

```
@protocol MiProtocol
- (void)metodoObligatorio;

@optional
- (void)metodoOpcional;
- (void)otroMetodoOpcional;

@required
- (void)otroMetodoObligatorio;
```

```
@end
```

Los nombres de los protocolos, al igual que las clases, se deberán escribir en notación `UpperCamelCase`, pero en este caso se recomienda utilizar gerundios (p.ej. `NSCopying`) o adjetivos.

5.5. Categorías y extensiones

Las categorías nos permiten añadir métodos adicionales a clases ya existentes, sin necesidad de heredar de ellas ni de modificar su código. Para declarar una categoría especificaremos la clase que extiende y entre paréntesis el nombre de la categoría. En la categoría podremos declarar una serie de métodos que se añadirán a los de la clase original, pero no podemos añadir variables de instancia:

```
@interface NSString ( CuentaCaracteres )
- (NSUInteger)cuentaCaracter:(char)caracter;
@end
```

La implementación se definirá de forma similar. El fichero en el que se almacenan las categorías suele tomar como nombre `NombreClase+NombreCategoria.h` (`.m`). Por ejemplo, en nuestro caso se llamaría `NSString+CuentaCaracteres`:

```
#import "NSString+CuentaCaracteres.h"

@implementation NSString ( CuentaCaracteres )
- (NSUInteger)cuentaCaracter:(char)caracter { ... }
@end
```

Las categorías resultarán de utilidad para incorporar métodos de utilidad adicionales que nuestra aplicación necesite a las clases que correspondan, en lugar de tener que crear clases de utilidad independientes (como el ejemplo en el que hemos extendido `NSString`). También serán útiles para poder repartir la implementación de clases complejas en diferentes ficheros.

Por otro lado tenemos lo que se conoce como extensiones. Una extensión se define como una categoría sin nombre, y nos fuerza a que los métodos que se definen en ella estén implementados en su clase correspondiente. Esto se usa frecuentemente para declarar métodos privados en nuestras clases. Anteriormente hemos comentado que en Objective-C todos los métodos son públicos, pero podemos tener algo similar a los métodos privados si implementamos métodos que no estén declarados en el fichero de cabecera público de nuestra clase. Estos métodos no serán visibles por otras clases, y si intentamos acceder a ellos el compilador nos dará un *warning*. Sin embargo, en tiempo de ejecución el acceso si que funcionará ya que el método realmente está implementado y como hemos comentado, realmente el acceso privado a métodos no existe en este lenguaje.

El problema que encontramos haciendo esto es que si nuestro método no está declarado en ningún sitio, será visible por los métodos implementados a continuación del nuestro, pero no los que aparecen anteriormente en el fichero de implementación. Para evitar esto

deberíamos declarar estos métodos al comienzo de nuestro fichero de implementación, para que sea visible por todos los métodos. Para ello podemos declarar una extensión en el fichero de implementación:

```
@interface MiObjeto ()  
- (void)metodoPrivado;  
@end  
  
@implementation MiObjeto  
  
// Otros métodos  
...  
- (void)metodoPrivado {  
    ...  
}  
@end
```

Atención

No deberemos utilizar el prefijo `_` para los métodos privados. Esta nomenclatura se la reserva Apple para los métodos privados de las clases de Cocoa Touch. Si la utilizásemos, podríamos estar sobrescribiendo por error métodos de Cocoa que no deberíamos tocar.

6. Ejercicios de objetos, propiedades y colecciones

6.1. Creación de objetos (1 punto)

Vamos a crear una clase para representar las películas. La clase tendrá como nombre `UAPelicula`, y contendrá los siguientes datos:

- Título
- Director
- Calificación de edad (puede ser TP, NR7, NR13 o NR18)
- Puntuación (número real de 0 a 10)
- Fecha de estreno

- a) Crea la clase, e introduce las variables de instancia necesarias para representar los datos anteriores, utilizando el tipo de datos que consideres más adecuado.
- b) Implementa un inicializador sin parámetros, otro que lleve sólo el título, y otro que incluya todos los datos. ¿Cuál será el inicializador designado? Impleméntalo teniendo esto en cuenta.
- c) Añade en el inicializador designado las retenciones de todos los campos que consideres oportunas para evitar que sean borrados de memoria mientras nuestra clase se esté utilizando.
- d) Añade el método `dealloc` a la clase y libera todos los campos retenidos. Recuerda llamar también al `dealloc` de la superclase para que también libere su memoria.
- e) Crea un método factoría para cada inicializador. Controla la memoria de forma adecuada en estos métodos factoría.
- f) Sobrescribe el método `description` para que imprima el título de la película, seguido del nombre del director entre paréntesis. Por ejemplo: `El Resplandor (Stanley Kubrick)`.
- g) Vamos ahora a utilizar la clase que acabamos de crear dentro de la aplicación. Para ello vamos a introducir en la clase `UAMasterViewController` una variable de instancia `_pelicula` para almacenar un objeto de la clase `UAPelicula`.

Ayuda

Cuando declaramos en el `.h` una variable cuyo tipo es una clase creada por nosotros, no conviene hacer un `#import` de dicha clase, para evitar inclusiones cíclicas, pero es necesario que el compilador sepa que es una clase válida. Para hacer esto haremos una declaración adelantada: `@class UAPelicula;`. La importación del fichero donde está definida la clase se hará en el `.m`, que es donde si se necesita saber todo lo que contiene dicha clase.

h) En el inicializador de `UAMasterViewController` instanciaremos un objeto de la clase `UAPelicula` y lo asignaremos a la variable de instancia anterior. Aplica la gestión de memoria que consideres adecuada a dicho objeto (tanto la retención al crearlo, como la liberación en `dealloc`). La película tendrá los siguientes datos:

- **Título:** El Resplandor
- **Director:** Stanley Kubrick
- **Calificación de edad:** NR18
- **Puntuación:** 9.0
- **Fecha de estreno:** 19 de diciembre de 1980

i) Como texto de las celdas, mostraremos la descripción del objeto `UAPelicula` creado. Para ello deberemos localizar el método `tableView:cellForRowAtIndexPath:`, y dentro de él la línea que comienza por `cell.textLabel.text =`, como se hizo en la sesión anterior. En esta línea haremos que como texto de la celda muestre `[_pelicula description]`.

j) Añade a la clase `UAPelicula` un método `antiguedad` que nos devuelva la antigüedad de la película en años. Por ejemplo, si la película se estrenó en diciembre de 1980 y estamos en octubre de 2012, nos devolverá 31 años. Puedes aprovechar para esto el código realizado en uno de los ejercicios de la sesión anterior. Escribe un `log` con la antigüedad de la película instanciada en el inicializador de `UAMasterViewController`.

6.2. Propiedades (1 punto)

Vamos a añadir propiedades a la clase `UAPelicula` creada en la sesión anterior.

a) Crea propiedades para cada variable de instancia definida en la clase. Las variables de instancia pueden eliminarse, ya que no es necesario que estén declaradas explícitamente.

Importante

Las variables de instancia implícitas asociadas a las propiedades creadas deben llevar siempre el prefijo `"_"` para no confundirlas con el nombre de la propiedad. Recuerda especificar esto siempre que se haga `@synthesize`.

b) En `UAMasterViewController`, haz que la película sea una propiedad de la clase, en lugar de una variable de instancia. La propiedad se llamará `pelicula`, y su variable de instancia asociada `_pelicula`.

c) Haz que en cada celda, en lugar de mostrar la descripción de la película muestre sólo el título, accediendo a él a través de la propiedad definida anteriormente mediante el operador `". ."`.

6.3. Listas (1 punto)

Vamos ahora a utilizar colecciones de datos en nuestra aplicación. Se pide:

- a) Añade a las películas una lista de actores. Esta lista debe poder ser accedida como propiedad.
- b) Ahora en lugar de guardar una única película vamos a tener una lista de ellas. Para ello en la clase `UAMasterViewController` sustituiremos la propiedad de tipo `UAPelicula` por una de tipo `NSArray`, y en el constructor crearemos una serie de películas de prueba y las añadiremos a la lista.
- c) Modifica el método `tableView:numberOfRowsInSection:`, y haz que devuelva el número de componentes de la lista, en lugar de un valor fijo.
- d) Modifica el método `tableView:cellForRowAtIndexPath:`, para que muestre como texto de la celda el título de la película correspondiente a la fila actual. Esta vez, accede a la propiedad `row` de `NSIndexPath` mediante el operador `'.'`. Comprueba que la lista de películas se muestra de forma correcta en la lista.

6.4. Gestión de memoria con ARC (0 puntos)

Vamos a convertir el proyecto anterior a ARC para que la gestión de la memoria la haga automáticamente el compilador.

- a) Realizar un *snapshot* del proyecto antes de realizar la conversión.
- b) Convertir el proyecto de forma automática con la opción *Edit > Refactor > Convert to Objective-C ARC* ¿Qué cambios se producen?

7. Programación de eventos

Hemos visto que en Objective-C no existe sobrecarga de métodos. Los métodos se identifican por lo que se conoce como el *selector*, que consiste en el nombre del método seguido de los nombres de sus parámetros. Por ejemplo `setValue:forKey:`. Cada parámetro se indica mediante : en el selector, y no importa de qué tipo sea (por eso no existe sobrecarga).

Los selectores se puede representar en el código mediante el tipo `SEL`, y podemos crear valores de este tipo mediante la directiva `@selector()`:

```
SEL accesoKVC = @selector(setValue:forKey:);
```

Podremos utilizar esta variable de tipo `SEL` para ejecutar el *selector* indicado sobre cualquier objeto, utilizando el método `performSelector:` (o cualquiera de sus variantes) del objeto sobre el que lo queramos ejecutar.

```
[asignatura performSelector:accesoKVC
    withObject:@"Plataforma iOS"
    withObject:@"nombre"];
```

Esta forma de ejecutar métodos va a ser de gran utilidad para definir *callbacks*. Cuando queramos que se nos notifique de un determinado evento en el momento que se produzca, podemos proporcionar una referencia a nuestro objeto (al que nos referiremos como *target*) y el *selector* al que queremos que se nos avise.

7.1. Patrón target-selector

Como hemos comentado, la forma anterior de ejecutar selectores va a permitirnos definir *callbacks* de forma muy sencilla. Por ejemplo, la clase `NSTimer` utiliza este esquema. En la mayoría de sus inicializadores toma como parámetros:

```
NSTimer *temporizador = [NSTimer
    scheduledTimerWithTimeInterval:5.0
        target: self
        selector: @selector(tick:)
        userInfo: nil
        repeats: YES];
```

De esta forma el temporizador llamará cada 5 segundos al método `tick` de nuestro objeto (`self`). Cuando utilizamos este esquema, la firma del método al que se llama está determinada por quien genera los eventos. Normalmente, este tipo de métodos recibirán un parámetro con una referencia al objeto que produjo el evento. En el caso del temporizador anterior el método deberá tomar un parámetro de tipo `NSTimer*`, en el que recibiremos el objeto temporizador que está generando los eventos.

```
- (void) tick: (NSTimer*)temporizador;
```

Normalmente el objeto especificado como *target* nunca se retendrá. Esto es así porque es

muy frecuente que la clase que se registra como *target* sea ancestra de la que genera los eventos. Por ejemplo, en el caso del temporizador, lo más común es que la clase que crea el temporizador y lo retiene en una de sus variables de instancia sea también la clase que se registra como destino de los eventos del temporizador (especificando `target: self` al crear el temporizador). Si el temporizador retuviese el *target* tendríamos una retención cíclica y por lo tanto una posible fuga de memoria.

Por lo tanto, cuando utilicemos este patrón para definir un *callback*, deberemos llevar cuidado cuando el objeto desaparezca. Antes siempre deberemos eliminarlo de todos los lugares en los que lo hubiésemos establecido como *target* (a no ser que el *target* se haya definido como una propiedad de tipo `weak`, con las que esto se haría de forma automática).

7.2. Notificaciones

Otra forma de informar de que un evento ha ocurrido es mediante el uso de notificaciones. Las notificaciones son una especie de mensajes de tipo *broadcast* que podemos enviar, o bien escuchar los que otros envían. Una diferencia entre las notificaciones y otros mecanismos de comunicación es que las notificaciones se envían sin saber quién va a ser el receptor, podría incluso no haber ningún receptor o haber varios de ellos. Esto nos va a permitir comunicar objetos lejanos en el diagrama de clases, que muchas veces resultan difícilmente accesibles el uno desde el otro.

La gestión de las notificaciones se hace mediante el *notification center*, que se define como *singleton*:

```
[NSNotificationCenter defaultCenter]
```

A través de este centro de notificaciones podemos difundir objetos de tipo `NSNotification`. Estos objetos se componen de tres elementos básicos:

- `name`: Nombre de la notificación que sirve para identificarla.
- `object`: Objeto que se adjunta a la notificación. Suele ser el objeto que la envía.
- `userInfo`: Información adicional que queramos añadir, en forma de diccionario (`NSDictionary`). La información aquí incluida dependerá del tipo de notificación, deberemos consultar la documentación de cada una para obtener esta información.

Podemos crear una notificación de la siguiente forma:

```
NSNotification *notificacion = [NSNotification
    notificationWithName:@"SincronizacionCompletada"
        object:self
        userInfo:nil];
```

Existen otros métodos factoría que toman menos parámetros, por ejemplo, en muchas ocasiones no necesitamos proporcionar ninguna información en `userInfo`. En esos casos podemos utilizar un método factoría más sencillo sin este parámetro.

Una vez tenemos la notificación, podemos enviarla a través del centro de notificaciones.

```
[ [NSNotificationCenter defaultCenter] postNotification: notificacion];
```

En lugar de crear el objeto `NSNotification` previamente, existe un atajo para que se cree y se envíe con una única operación:

```
[ [NSNotificationCenter defaultCenter]
    postNotificationName:@"SincronizacionCompletada" object:self];
```

Por otro lado, si lo que queremos es escuchar las posibles notificaciones de un determinado tipo (nombre) que se produzcan en la aplicación, tendremos que registrar un observador en el centro de notificaciones.

```
[ [NSNotificationCenter defaultCenter]
    addObserver:self
    selector:@selector(sincronizado:)
    name:@"SincronizacionCompletada"
    object:nil];
```

Podemos ver que para registrarnos como observador estamos utilizando el patrón *target-selector* visto anteriormente. Cada vez que se produzca una notificación de tipo `@"SincronizacionCompleta"`, se enviará un mensaje al método `sincronizado:` de nuestro objeto (`self`). Este método deberá recibir como parámetro un objeto de tipo `NSNotification`, con el que recibirá la notificación cuando se produzca.

```
- (void)sincronizado:(NSNotification *) notificacion {
    NSLog(@"Sincronizacion completada");
}
```

El último parámetro nos permite especificar si queremos que sólo nos llegan notificaciones enviadas por un objeto concreto. Si queremos cualquier notificación del tipo indicado, independientemente de su fuente, especificaremos `object:nil`.

Podemos dejar de ser observador con el siguiente método:

```
[ [NSNotificationCenter defaultCenter] removeObserver: self];
```

Con esto nos elimina como observador de cualquier notificación. Si queremos eliminar sólo una notificación concreta debemos usar otra versión de este método (`removeObserver:name:object:`).

Importante

Siempre deberemos llamar a `removeObserver:` antes de que nuestro objeto sea desalojado de la memoria, ya que el centro de notificaciones no retiene a los observadores, siguiendo las normas que hemos comentado anteriormente para el patrón *target-selector*.

Los nombres de notificaciones se suelen definir como constantes. En el ejemplo hemos creado un tipo propio de notificación, pero si utilizamos notificaciones incorporadas en la API de Cocoa Touch siempre deberemos utilizar las constantes que se definen para ellas, y nunca la cadena de texto que corresponde a dichas constantes. Si en un momento dado cambiase la cadena de texto, o bien nos equivocásemos al escribir algún carácter, dejaríamos de recibir las notificaciones correctamente.

7.3. Key Value Observing (KVO)

KVO nos permitirá observar los posibles cambios que se produzcan en las propiedades de los objetos. Recordemos que KVC nos permitía acceder a dichas propiedades de forma dinámica mediante cadenas de texto. KVO se podrá utilizar en cualquier propiedad que implemente KVC. Para registrarnos como observador de cualquier objeto, tenemos el siguiente método:

```
[_asignatura addObserver:self  
    forKeyPath:@"nombre"  
    options:NSMutableArrayObservingOptionNew  
    context:NULL];
```

Cuando la propiedad `nombre` del objeto `_asignatura` cambie de valor, recibiremos el siguiente mensaje en nuestro objeto (`self`):

```
observeValueForKeyPath:ofObject:change:context:
```

Por lo tanto, deberemos implementar el correspondiente método en nuestro observador:

```
- (void)observeValueForKeyPath:(NSString *)keyPath  
    ofObject:(id)object  
    change:(NSDictionary *)change  
    context:(void *)context
```

En `keyPath` recibiremos la ruta de la propiedad que ha cambiado, en `object` la referencia al objeto observado en el que se produjo el cambio, en `change` el valor que ha cambiado, y por último en `context` nos llegará la misma información que indicamos en dicho parámetro cuando nos registramos como observador.

Destacamos que el valor de la propiedad que ha cambiado lo recibimos como diccionario en el parámetro `change`. Esto es así porque podremos recibir varios valores (por ejemplo, el valor antiguo anterior al cambio, y el nuevo). Esto dependerá de lo que indicásemos en el parámetro `options` cuando nos registramos como observador. En el ejemplo anterior sólo hemos solicitado obtener el valor nuevo. Podremos obtener dicho valor a partir del diccionario de la siguiente forma:

```
id valor = [change objectForKey: NSKeyValueChangeNewKey];
```

En este caso hemos utilizado también el patrón observador, como en los casos anteriores en los que hemos utilizado *target-selector*. La única diferencia de este último caso es que no nos ha permitido especificar el selector, como ocurría en los casos anteriores, sino que éste ya está establecido. Igual que en los casos anteriores, el objeto observado no retendrá al observador, por lo que es importante que eliminemos el observador antes de que se libere de la memoria. Para ello utilizaremos `removeObserver:forKeyPath:`.

7.4. Objetos delegados y protocolos

Otra forma habitual de dar respuesta a eventos es sobrescribir una serie de métodos a los

que esperamos que el sistema llame cada vez que se produzca un evento. Sin embargo, muchas veces no es conveniente heredar de determinadas clases de la API (especialmente no contando con herencia múltiple) cuando lo único que nos interesa es dar respuesta a una serie de eventos que se pueden producir.

Podemos resolver esto mediante el patrón de **objetos delegados**, que nos permite que una determinada clase delegue en una clase secundaria para determinadas tareas. Este patrón se lleva a cabo mediante la inclusión de una propiedad `delegate`. Si dicha propiedad es distinta de `nil`, cuando ocurran determinados eventos pasará mensajes a `delegate` para que dicho objeto delegado realice las tareas oportunas.

Un ejemplo de uso de objetos delegados lo tenemos en la clase `UIApplication`. Para controlar el ciclo de vida de las aplicaciones no estamos creando una subclase de `UIApplication`, sino que pasamos dicha tarea a un objeto delegado.

Los objetos delegados suelen implementar un determinado protocolo, que les forzará a definir una serie de métodos que corresponderán a los mensajes que se espera que envíe el objeto principal. Este es el uso principal de los protocolos. Por ejemplo, en el caso del delegado de `UIApplication`, vemos que implementa el protocolo `UIApplicationDelegate` que incorpora una serie de métodos para controlar el ciclo de vida de la aplicación.

Podemos verlo como algo similar a los *listeners* en Java, pero con algunas diferencias. En el caso de los delegados, muchos métodos definidos en los protocolos son de implementación opcional, e incluso en algunas ocasiones puede que el delegado no se defina con un protocolo, sino que simplemente se documente qué métodos se espera que nos envíen. Esto es posible gracias a que Objective-C permite pasar cualquier mensaje a los objetos, aunque en su tipo no esté declarado el método.

¿Cómo podemos entonces saber en el objeto principal si el método al que le vamos a pasar el mensaje existe en el delegado? Si enviamos un mensaje a `nil` no pasa nada (obtenemos `nil`), pero si enviamos un mensaje a un objeto distinto de `nil` que no implementa ningún método para dicho mensaje, obtendremos una excepción, con lo que deberemos llevar cuidado con esto. Para poder comprobar si el método está implementado o no, podemos utilizar los mecanismos de introspección que veremos a continuación.

7.5. Bloques

Los bloques son una nueva característica del lenguaje incorporada a partir de iOS 4.0. Nos permiten definir un bloque de código que se podrá pasar como parámetro para que se ejecute cuando un determinado evento suceda, o bien devolver un bloque como resultado de la ejecución de un método. Los bloques se declaran mediante el símbolo ^:

```
int (^Suma)(int, int) = ^(int num1, int num2) {
    return num1 + num2;
};
```

Dicho bloque tendrá como nombre `Suma` y recibirá dos parámetros numéricos. Podremos ejecutarlo como si se tratase de una función, pero realmente se está definiendo como una variable que puede ser pasada como parámetro o devuelta como resultado:

```
int resultado = Suma(2, 3);
```

Los bloques son una forma simple de definir *callbacks*. La ventaja de los bloques es que el código del *callback* se puede definir en el mismo lugar en el que se registra, sin necesitar crear un método independiente. Por ejemplo podemos registrarnos como observadores para una notificación usando bloques de la siguiente forma:

```
- (id)addObserverForName:(NSString *)name
    object:(id)obj
    queue:(NSOperationQueue *)queue
    usingBlock:(void (^)(NSNotification *))block
```

En este caso vemos que como último parámetro define un bloque que tomará un parámetro de tipo `NSNotification *` y devuelve `void`. Lo podríamos utilizar de la siguiente forma:

```
[[NSNotificationCenter defaultCenter]
    addObserverForName:@"SincronizacionCompletada"
    object:nil
    queue:nil
    usingBlock:^(NSNotification *notificacion) {
        NSLog(@"Se ha completado la sincronizacion");
    }];
}
```

7.6. Introspección

En muchas ocasiones en nuestro código podemos recibir objetos con referencias de tipo `id`, en los que puede que no estemos seguros del tipo de objeto del que se trate, ni de los *selectores* que incorpora, o incluso los objetos pueden implementar protocolos que definan métodos opcionales, que puedan no estar presentes.

Por este motivo son necesarios mecanismos de introspección, que nos permitan en tiempo de ejecución determinar de qué tipo es un objeto dado y si responde ante un determinado *selector*.

7.6.1. Tipo de la clase

Para conocer si un objeto es de una determinada clase podemos utilizar los siguientes métodos:

```
[asignatura isMemberOfClass:[UAAsignatura class]]; // YES
[asignatura isMemberOfClass:[NSObject class]]; // NO
[asignatura isKindOfClass:[UAAsignatura class]]; // YES
[asignatura isKindOfClass:[NSObject class]]; // YES
```

Podemos ver que `isMemberOfClass` comprueba si una clase es estrictamente del tipo especificado, mientras que `isKindOfClass` indica si son tipos compatibles (es decir, si es

la misma clase, o una subclase de la indicada). También podemos ver que podemos obtener el objeto que representa una clase dada mediante el método de clase `+class` que podremos encontrar en todas las clases. Nos devolverá un objeto de tipo `Class` con información de la clase.

También podemos saber si un objeto implementa un protocolo determinado con `conformsToProtocol:`. En este caso necesitaremos una variable de tipo `Protocol`, que podemos obtener mediante la directiva `@protocol`:

```
[asignatura conformsToProtocol:@protocol(NSCopying)];
```

7.6.2. Comprobación de selectores

Pero incluso hay protocolos que definen métodos opcionales, por lo que la forma más segura de saber si un objeto implementa un determinado método es comprobarlo mediante `respondsToSelector:`.

```
[asignatura respondsToSelector:@selector(creditos)];
```

Esta comprobación será muy común cuando utilicemos el patrón de objetos delegados, para comprobar si el delegado implementa un determinado método.

En las clases podemos encontrar un método similar llamado `instancesRespondToSelector:`, que indica si las instancias de dicha clase responden al *selector* indicado.

También podemos consultar la firma (*signature*) de un método a partir de su *selector*:

```
NSMethodSignature *firma = [asignatura methodSignatureForSelector:  
    @selector(tasaConPrecioPorCredito:esBecario:)];
```

Al igual que existe una variable implícita `self` que nos da un puntero al objeto en el que estamos actualmente, cuando estamos ejecutando un método disponemos también de otra variable implícita de nombre `_cmd` y de tipo `SEL` que indica el selector en el que estamos.

7.6.3. Llamar a métodos mediante implementaciones

A partir del *selector* de un método podemos obtener su implementación, de tipo `IMP`. La implementación es la dirección de memoria donde realmente se encuentra el método a llamar. Podemos utilizar la implementación para llamar al método como si se tratase de una función C. Esto nos permitirá optimizar las llamadas, ya que no será necesario resolver la dirección a la que llamar cada vez que se ejecuta de esta forma, se resolverá una única vez al obtener la implementación, y a partir de ese momento todas las llamadas que hagamos ya conocerán la dirección a la que llamar.

```
SEL selDescripcion = @selector(description);  
IMP descripcion = [asignatura methodForSelector:selDescripcion];  
  
NSString *descripcion = descripcion(asignatura, selDescripcion);
```

Vemos que en la llamada a la implementación debemos pasarle al menos dos parámetros, que corresponden a las dos variables implícitas con las que siempre contaremos en los métodos de nuestros objetos: `self` y `_cmd`. De hecho, el tipo `IMP` se define de la siguiente forma:

```
typedef id (*IMP)(id, SEL, ...);
```

Esto nos servirá siempre que el método al que queramos llamar devuelva un objeto (`id`). Si no fuese así, tendríamos que crear nuestro propio tipo de función. Por ejemplo, para `(CGFloat) tasaConPrecioPorCredito:(CGFloat)precioCredito esBecario:(BOOL)becario;` podríamos definir el siguiente tipo de función:

```
typedef CGFloat (*UACalculaTasa)(id, SEL, CGFloat, BOOL);
```

A la hora de obtener la implementación habrá que hacer una conversión *cast* al tipo de función al que se ajusta nuestro método:

```
SEL selTasa = @selector(tasaConPrecioPorCredito:esBecario:);
UACalculaTasa calcularTasa =
    (UACalculaTasa)[asignatura methodForSelector:selTasa];

CGFloat tasa = calcularTasa(asignatura, selTasa, 50, NO);
```

Podemos también obtener la implementación de un método de instancia a partir de un objeto de tipo `Class` mediante el método `instanceMethodForSelector:`.

7.6.4. Reenvío de mensajes

Cuando un objeto reciba un mensaje para el cual no tenga ningún método definido, realmente lo que ocurrirá es que se llamará a su método `forward::`, que recibirá el `selector` y la lista de parámetros que se han recibido. Por defecto este método lo que hará será lanzar una excepción indicando que el objeto no reconoce el `selector` solicitado y se interrumpirá la aplicación.

El método `forward::` de `NSObject` es privado y no debe ser sobrescrito, pero si que podemos sobrescribir `forwardInvocation:` para modificar este comportamiento. Podríamos por ejemplo reenviar los mensajes a un objeto delegado, o bien simplemente ignorarlos.

Podemos aprovechar esta característica para crear accesores a propiedades en tiempo de ejecución. En ese caso, para evitar que el compilador nos dé un *warning* por no haber utilizado `@synthesize` para generar dichos accesores, utilizaremos la directiva `@dynamic` para indicar al compilador que no debe preocuparse por la propiedad indicada, y que puede confiar en que hemos implementado algún mecanismo para acceder a ella.

7.6.5. Objetos y estructuras

Objective-C nos permite convertir de forma sencilla un objeto a una estructura de datos. Para ello tenemos la directiva `@defs()`, que genera en tiempo de compilación la lista de

variables de instancia de una clase. De esta forma, podemos generar una estructura de datos equivalente:

```
typedef struct {
    @defs(UAAsignatura);
} UAAsignaturaStruct;
```

Si tenemos un objeto de tipo `UAAsignatura`, podemos asignarlo directamente a una estructura equivalente mediante un *cast*:

```
UAAsignatura *asignatura = [[UAAsignatura alloc] init];
UAAsignaturaStruct *asigStruct = (UAAsignaturaStruct *) asignatura;
asigStruct->nombre = @"Plataforma iOS";
```

Advertencia

No debemos hacer esto si utilizamos ARC, ya que en este caso las conversiones directas entre `id` y `void *` están prohibidas.

7.7. Ciclo de vida de las aplicaciones

Ya hemos visto el patrón delegado, que se utiliza frecuentemente en Cocoa Touch. El primer uso que le daremos será para definir el delegado de la aplicación. La aplicación se implementa en la clase `UIApplication`, pero para poder tratar los eventos de su ciclo de vida necesitaremos crear un delegado en el que programaremos las tareas a realizar para cada uno de ellos. Este delegado implementará el protocolo `UIApplicationDelegate`, que define los métodos que podemos implementar para tratar cada uno de los eventos del ciclo de vida de la aplicación.

El evento fundamental es `application:didFinishLaunchingWithOptions:`, que se ejecutará cuando la aplicación ha terminado de cargarse y se va a poner en marcha. Aquí tendremos que programar las tareas a realizar para poner en marcha nuestra aplicación. Una implementación típica consiste en configurar el contenido a mostrar en la ventana de la aplicación y mostrarla en pantalla:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window.rootViewController = self.navigationController;
    [self.window makeKeyAndVisible];

    return YES;
}
```

En la próxima sesión veremos con más detalle como trabajar con los objetos de la interfaz.

El método opuesto al anterior es `applicationWillTerminate:`. Con él se nos notifica que la aplicación va a ser cerrada, por lo que deberemos liberar los objetos de memoria, parar las tareas en curso, y guardar los datos pendientes.

Durante la ejecución de la aplicación es posible que suceda algún evento externo, como por ejemplo una llamada entrante, que haga que nuestra aplicación se detenga temporalmente, para luego continuar por donde se había quedado (en el ejemplo anterior se reanudaría al terminar la llamada). Los eventos que nos notificarán la pausa y reanudación de la aplicación son `applicationWillResignActive:` y `applicationDidBecomeActive:` respectivamente.

A partir de iOS 4.0 aparece la multitarea, permitiéndonos dejar una aplicación en segundo plano sin finalizar su ejecución. Esto es diferente al caso anterior, ya que siempre se ha permitido que la aplicación entre en pausa temporalmente cuando suceden determinados eventos del dispositivo (como las llamadas entrantes), pero con la multitarea podemos dejar la aplicación en segundo plano para ejecutar cualquier otra aplicación, y posteriormente poder volver a recuperar el estado anterior de nuestra aplicación. Para implementar la multitarea deberemos definir los métodos `applicationDidEnterBackground:` y `applicationWillEnterForeground:`.

Deshabilitar la multitarea

Por defecto, cuando salimos de una aplicación realmente la estaremos dejando en segundo plano. La aplicación sólo se finalizará cuando la cerramos explícitamente desde la lista de aplicaciones recientes, o cuando el dispositivo la cierre por falta de recursos. Sin embargo, en algunos casos nos puede interesar deshabilitar la multitarea para que al salir de la aplicación, ésta se finalice. Para que esto sea así, añadiremos la propiedad "UIApplicationExitsOnSuspend" = YES en Info.plist.

Otro evento que resulta recomendable tratar es `applicationDidReceiveMemoryWarning:`. Recibiremos este mensaje cuando el dispositivo se esté quedando sin memoria. Cuando esto ocurra, deberemos liberar tanta memoria como podamos (objetos que puedan ser fácilmente recreados más tarde).

8. Ejercicios de gestión de eventos

8.1. Temporizadores (1 punto)

Crearemos ahora un temporizador con `NSTimer`, que transcurridos unos segundos escribirá un log.

- a) Cuando se lance la aplicación, en el método `application:didFinishLaunchingWithOptions:` de la clase `UAMAppDelegate`, programa un temporizador que se dispare en 10 segundos. Tras esos 10 segundos deberá llamar a un método `temporizadorDisparado:`, que deberemos crear dentro de la clase anterior.
- b) Escribe un *log* en el momento en el que se programa el temporizador (en `application:didFinishLaunchingWithOptions:`), y otro en el momento en el que se dispara (dentro del método `temporizadorDisparado:`).
- c) Haz ahora que el temporizador se dispare de forma periódica. Si queremos que este temporizador se empiece a disparar a una hora concreta, ¿qué propiedad de `NSTimer` podríamos utilizar? Programa el temporizador para que se ejecute cada 10 segundos empezando a la hora actual.

8.2. Notificaciones (1 punto)

Vamos a hacer ahora que el temporizador anterior produzca una notificación que pueda ser observada desde otro lugar de la aplicación:

- a) Al dispararse el temporizador publica a través del centro de notificaciones una notificación con identificador "TemporizadorCompletado".
- b) En el inicializador de la clase `UAMasterViewController` registra un método `notificacionRecibida:` (que deberemos previamente crear) como observador de la notificación "TemporizadorCompletado". En dicho método haz que se añada una nueva película (sin título) a la lista, y actualiza la interfaz llamando a `[self.tableView reloadData]`.

Ayuda

Para poder añadir una película a la lista, la lista deberá ser mutable. Por lo tanto, tendrás que utilizar el tipo `NSMutableArray` tanto en la declaración como en la instanciación de la lista de películas.

- c) Modifica el código anterior para registrar un bloque como observador de la notificación, en lugar del método `notificacionRecibida:`. El bloque contendrá el

mismo código que dicho método.

d) Haz que el observador anterior añada a la lista de películas un objeto de tipo `NSString`, en lugar de `UAPelicula`. Al mostrar la lista de películas, cuando se añada la cadena dará un error, ya que estamos intentando mostrar la propiedad `item.titulo`, que no existe en `NSString`. Modificar el código de `tableView:cellForRowAtIndexPath:` para que compruebe si el item de la lista responde al selector `titulo`. De no ser así, mostraremos como texto la descripción del propio objeto.

8.3. Delegados (1 punto)

Vamos a hacer ahora que al dispararse el temporizador aparezca en el móvil una pantalla de alerta, en lugar de añadir una película.

a) Al producirse la notificación, ejecutaremos el siguiente código para mostrar la alerta en pantalla:

```
UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Alerta"
                                              message:@"Se ha disparado el temporizador"
                                             delegate: nil
                                         cancelButtonTitle:@"Cerrar"
                                         otherButtonTitles: nil];
```

b) Modifica la alerta para que tenga un segundo botón con el texto "Ver".

c) Al crear la alerta vemos que uno de los campos es un delegado. Consulta la documentación de `UIAlertView` para ver de qué tipo debe ser dicho delegado. Haz que la propia clase `UAMasterViewController` se comporte como delegada, implementando el protocolo correspondiente, e implementa el método del delegado que nos permita saber cuando se hace *click* sobre uno de los botones de la alerta. Escribe un *log* indicando el índice del botón que se haya pulsado cuando se ejecute dicho método del delegado.

9. Depuración y pruebas

La depuración y las pruebas son dos procesos indispensables a la hora de comprobar y solucionar comportamientos erróneos en la ejecución de nuestras aplicaciones. En esta sesión comenzaremos tratando los dos tipos de depuración existentes en iOS: uso de la consola para mostrar información determinada de los procesos y el manejo de la herramienta de *debugging* de XCode. Después comentaremos el uso de la aplicación *Instruments*, con la cual podremos averiguar qué objetos pueden provocar fallos de gestión de memoria así como depurar más en profundidad nuestra aplicación. Para terminar la sesión explicaremos, mediante ejemplos, la implementación de pruebas de unidad en XCode.

9.1. Depuración clásica: Uso de directivas NSLog y Asserts

La depuración clásica consiste en alterar el código fuente de forma temporal añadiendo una serie de directivas con el fin de mostrar mensajes informativos en la consola. La función habitual para enviar mensajes a la consola de XCode es `NSLog()`. NSLog recibe como parámetro un `NSString` con uno o varios *especificadores de formato*. Un especificador de formato es una cadena de texto que será sustituida en tiempo de ejecución por el valor que se especifique. Todos los especificadores de formato empiezan con el símbolo del porcentaje % seguido de un carácter que indica el formato del elemento a mostrar.

Los distintos tipos de especificadores de formato que podemos encontrar en Objective C son los siguientes:

- **%@:** Cadena de texto, descripción y valor de objetos.
- **%i:** Entero (integer).
- **%f:** Decimal (float).
- **%.02f:** Decimal (float) con dos decimales.
- **%ld:** Entero Long
- **%p:** Puntero, referencia de un objeto.

El más usado es %@ ya que, además de mostrar una cadena de texto también puede mostrar la descripción de un método si este está preparado para ello. Un ejemplo podría ser el siguiente:

```
NSLog(@"%@", "Descripción del objeto window: %@", self.window);
```

En este caso se mostrará por la consola la información más relevante del elemento *window* como el tamaño del frame, opacidad, autoresize, etc. Existen más especificadores de formato no tan comunes que se pueden consultar en la documentación de Apple, dentro del apartado "String format Specifiers".

Otro uso muy útil del especificador de formato de cadena es para mostrar los valores de un diccionario `NSDictionary`:

```
NSLog(@"%@", Valores y claves del diccionario: %@", miDiccionario);
```

Un error muy común a la hora de especificar una cadena de texto en una directiva `NSLog` es indicar distinto número de especificadores de formato que parámetros tiene la cadena o equivocarse a la hora de especificar un tipo de parámetro determinado, por ejemplo, mostrar un entero cuando en realidad es una cadena. Estos errores pueden derivar en una excepción en tiempo de ejecución o simplemente en mostrar un valor que no corresponda en absoluto a lo esperado.

El uso de este tipo de depuración mediante trazas en el código es bastante usado ya que es fácil de implementar (sólo una función) y, además, a veces es el único modo de depurar el código. Las directivas `NSLog` funcionan con cualquier tipo de configuración (Debug o Release) y en cualquier tipo de ejecución (en el simulador o en el dispositivo), incluso funcionan en otros dispositivos que ejecuten nuestra aplicación a modo de testeo. Para poder visualizar la salida de consola de una aplicación en fase beta se debe conectar el dispositivo al XCode y ejecutarla, el log de la consola se guardará en el dispositivo y podremos acceder a él desde la ventana *Organizer* de XCode.

Atención

Antes de generar el binario de distribución para la *App Store* deberemos de eliminar todas las llamadas a `NSLog` que tengamos en el código para así dejar " limpia" la aplicación y evitar una posible relentización de la ejecución debido a la escritura en la consola.

Otro tipo de función con la cual podemos depurar nuestro código por consola es el llamado **Assert**. Los *Asserts* (o aserciones) son condiciones que se deben de cumplir para que continúe la ejecución en un determinado momento. Estas deben de devolver "true". En el caso de que algún *assert* no se cumpla se producirá una excepción en fase de ejecución que detendrá la aplicación en ese mismo instante almacenando todo el "log" en XCode para su posterior depuración.

El uso de *asserts* es una manera muy buena de asegurarse que una determinada situación dentro de nuestro código cumpla con las expectativas que deseemos. Muchos desarrolladores dejan incluso estas directivas *assert* a la hora de distribuir la aplicación, lo cual no es del todo recomendable en la mayoría de casos.

La directiva *assert* en Objective-C viene en forma de macro habitualmente `assert()` a la cual se le pasa una condición que será que se tenga que evaluar a verdadero (true) o falso (false). Si se evalua falso la aplicación se detendrá mostrando en la consola un log y si se evalua a verdadero la aplicación continuará con su ejecución. Los *asserts* son usados frecuentemente en APIs así como en códigos en fases de testeo.

Los *asserts* también se pueden programar mediante la clase de Objective-C `NSAssert2` de

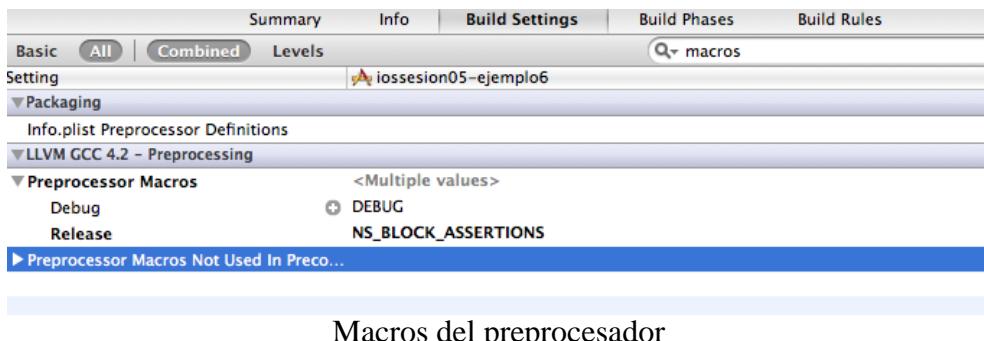
más alto nivel que la macro comentada anteriormente. Mediante esta clase podremos especificar de una manera más clara y descriptiva el tipo de condición que se debe de cumplir, para que en el caso de que esta no se cumpla aparezca un mensaje más claro en consola. A continuación podemos ver como usar los dos tipos de assert que disponemos:

```
// Uso de la macro assert()
assert(valor < maximoValor && @"El valor es demasiado grande!");

// Uso de NSAssert2()
NSAssert2(valor < maximoValor, @"El valor %i es demasiado grande
(max.:%i)!", valor,
maximoValor);
```

Como podemos ver, el segundo tipo es más descriptivo que el primero y, por tanto, más adecuado de cara a la depuración. Internamente NSAssert2 usa la macro assert().

Por último, para terminar, queda ver como desactivar todos los asserts que hayamos escrito en el código mediante NSSAssert2. Para desactivar todos los *asserts*, por ejemplo para compilar el proyecto para distribución, deberemos declarar la directiva `NS_BLOCK_ASSERTIONS` dentro de las macros del preprocesador en la configuración de "release" como se muestra en la imagen siguiente.



9.2. Usando el depurador de XCode

Después de comentar el método básico de depuración usando la consola de XCode pasamos a explicar el funcionamiento del potente depurador que viene integrado en el propio XCode y que nos será de mucha utilidad en determinadas ocasiones. Cuando se está ejecutando la aplicación en modo "debug" podremos pararla y observar el estado en ese momento, como cualquier depurador de código que conozcamos.

Para activar el seguimiento de los *breakpoints* y que el depurador se detenga en ellos deberemos arrancar nuestra aplicación con la opción de depuración activada, esto se hace pulsando sobre el botón en forma de flecha que se encuentra en la parte superior de la interfaz de XCode.



Botón Breakpoints en XCode

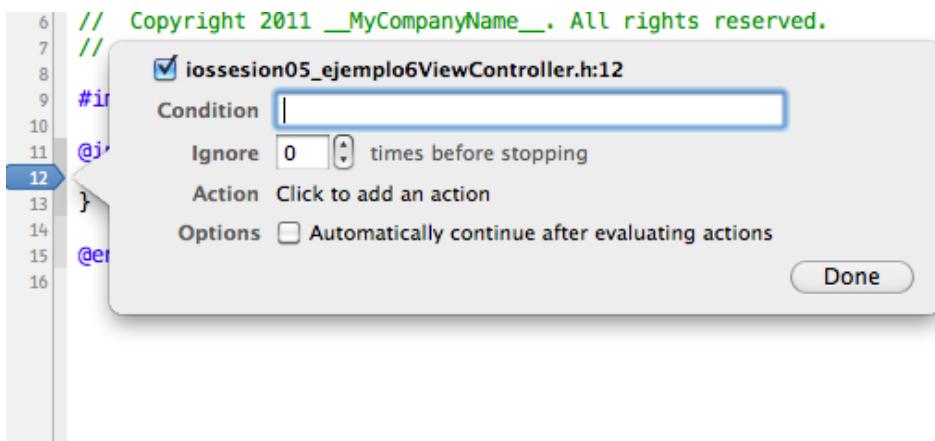
Para crear un punto de parada o *breakpoint* seleccionaremos en el editor la línea de código en donde queramos pausar la ejecución y pulsamos sobre *Product > Debug > Add Breakpoint at Current Line*, también podremos crearlo de una forma más rápida simplemente pulsando en la parte izquierda de la línea de código. El *breakpoint* se indica con una flecha de color azul. Si este está creado pero desactivado se mostrará semitransparente.

A screenshot of the XCode code editor. On the left, a vertical line number column shows lines 8 through 15. Line 12 is highlighted with a blue arrow pointing to the left, indicating it has a breakpoint. The code itself is as follows:

```
8
9
10
11
12 @interface iossesion05_6
13 {
14
15 @end
```

Breakpoint activado

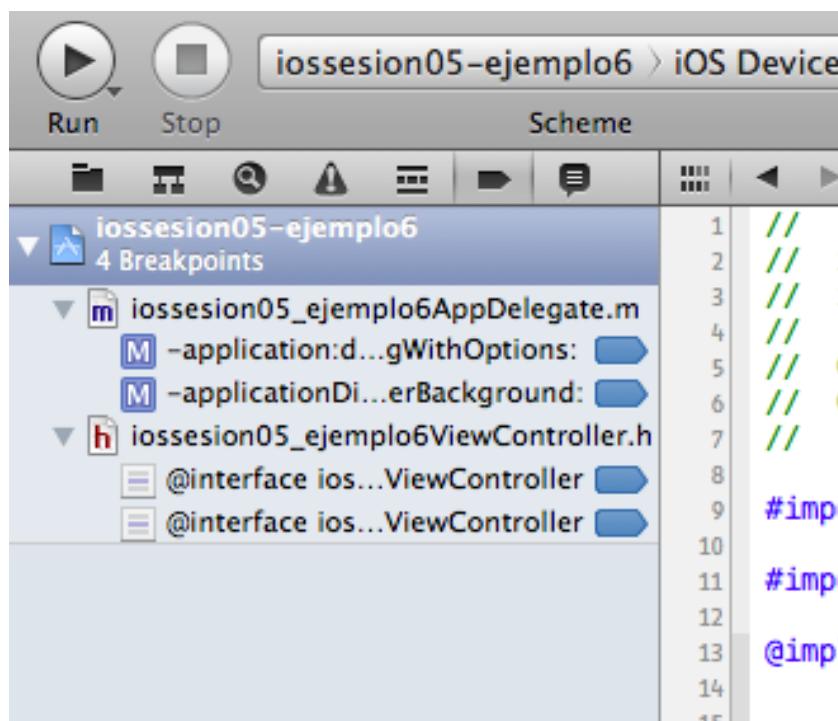
Los *breakpoints* se pueden desactivar o borrar, en estos casos el depurador no se detendrá en esa línea, de esta forma en el caso de que no queramos utilizar un *breakpoint* en un momento dado pero que más adelante puede que si, lo marcaremos como desactivado pero sin borrarlo. También podremos editar los *breakpoints* para, por ejemplo, indicar que sólo se detenga la ejecución cuando se cumple una determinada condición o si ha pasado por la línea un determinado número de veces. Para acceder a estas opciones deberemos hacer *ctrl + click* sobre el punto de parada y seleccionar la opción *Edit breakpoint*. Los *breakpoints* se pueden activar y desactivar de forma global desde *Product > Debug > Activate/Deactivate Breakpoints*.



Condición en los breakpoints

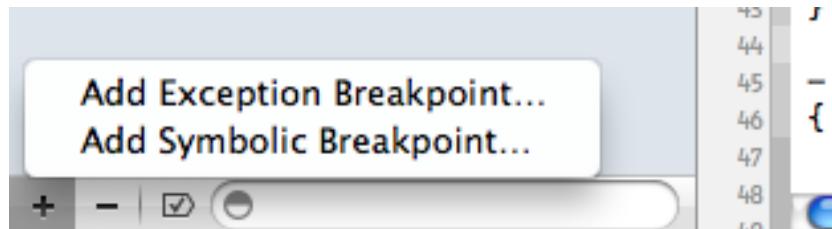
Otra de las características que tienen los puntos de parada en XCode es que podemos configurarlos para que no se detenga la ejecución cuando se pasa por ellos. Esto es una alternativa a la depuración clásica comentada en el primer apartado, evitaremos llenar el código con `NSLogs` que después deberemos controlar.

El entorno de XCode 4 dispone además de un navegador de *breakpoints*, al cual se puede acceder desde la pestaña en forma de flecha con color negro que se encuentra en la parte superior de la columna de la izquierda. En este navegador veremos en un golpe de vista todos los breakpoints que tiene nuestro proyecto separados por ficheros. En la parte inferior hay dos botones, uno para añadir nuevos breakpoints y otro para eliminarlos.



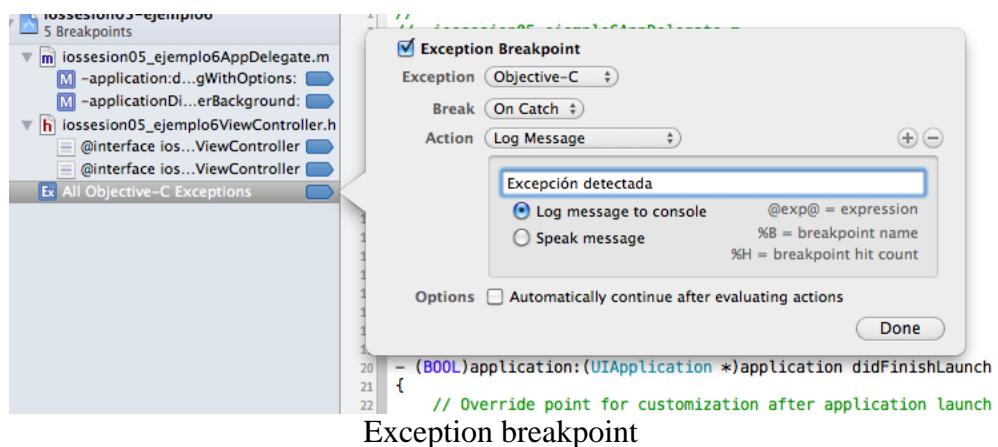
Navegador de breakpoints

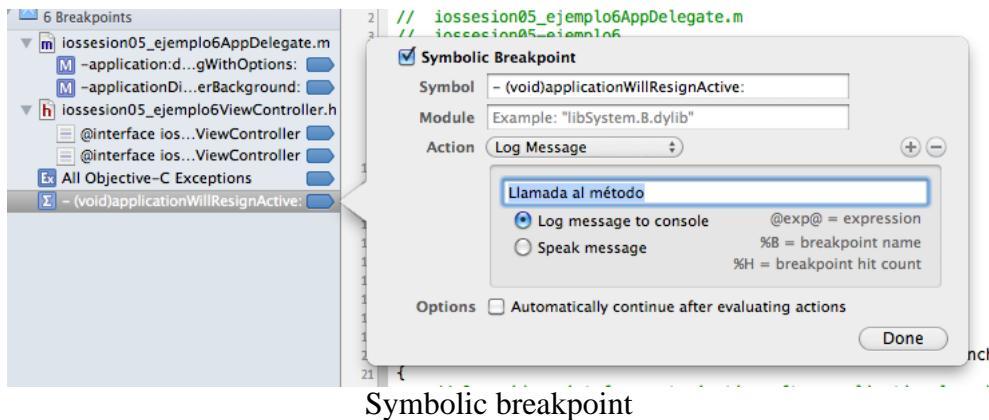
Al pulsar sobre el botón "+" nos aparecerá una ventana emergente en la que nos da a elegir si queremos crear un *breakpoint* de tipo excepción o de tipo simbólico. Estos son los dos tipos que existen en XCode junto con el más común que hemos explicado antes.



Tipos de breakpoint

- **Exception breakpoint:** Este tipo de breakpoint se ejecuta cuando salta una excepción en la aplicación. Es recomendable crear este tipo de breakpoints dentro del bloque "catch" de la sentencia *try-catch* para que en el momento en el que se ejecute se pueda ver toda la traza de manera completa y así detectar el problema que causa la excepción.
- **Symbolic breakpoint:** Este tipo de breakpoint se usa cuando queremos interrumpir la aplicación al ejecutar un determinado método o función. Deberemos indicar el método concreto por ejemplo `pathsMatchingExtensions:`, un método de una clase: `[SKTLine drawHandlesInView]`, `people::Person::name()`, o un nombre de función: `_objc_msgForward`.





A la hora de depurar la aplicación, la ejecutaremos en modo "Debug" y esta se detendrá en cuanto detecte un breakpoint activo. En la ventana principal de XCode se mostrará el fichero que contiene el breakpoint, el cual se mostrará mediante una flecha verde. Esta linea será la que se vaya a ejecutar en cuanto continuemos con la ejecución de la aplicación. Una vez que tenemos parada la ejecución deberemos analizar el estado en que está la aplicación, para ello podremos realizar cualquiera de las siguientes acciones:

- **Situarse en el programa:** Lo primero que debemos hacer cuando la aplicación se detiene en un breakpoint es ver la traza por dónde ha pasado la ejecución hasta llegar a este punto. Los métodos que se listan en la traza que están en color negro y con el icono de usuario son nuestros y los que son de color gris son de la propia API, la cual no tenemos acceso a su código fuente. Para analizar un determinado método nuestro deberemos pulsar sobre el.
- **Analizar los valores de las variables:** Esta es una de las razones más importantes por la que depuramos nuestro código. Cada una de las variables iniciadas en el código se puede analizar y ver los valores de todos sus componentes. El depurador de XCode nos marca las variables han cambiado su valor recientemente ya que estas son las que más nos pueden interesar. También disponemos de un pequeño buscador para filtrar por nombre de variable o por su valor.
- **Crear un watchpoint:** Un *watchpoint* es similar a un *breakpoint* pero que depende de una variable en concreto y se puede crear de forma dinámica mientras estamos ejecutando el depurador. Este se detendrá siempre que el valor de la variable cambie. Para crear un *watchpoint* haremos *ctrl+click* en la variable, dentro del listado y seleccionaremos *Watch Address of Variable*. Una vez creado se mostrará dentro del listado de breakpoints junto con el resto.
- **Usar la linea de comandos:** El uso de la línea de comandos del depurador de XCode es un método alternativo al uso de la interfaz. Mediante este podremos realizar las mismas funciones que con la interfaz e incluso más. El depurador que utiliza XCode es *GDB*.
- **Editar el listado de breakpoints dinamicamente:** Se pueden crear, editar y borrar los puntos de parada siempre que deseemos, incluso estando la aplicación en ejecución.

- **Continuar hasta la siguiente línea o continuar hasta el siguiente breakpoint:** A estas dos acciones se le llaman *step* y *continue* respectivamente. Cuando queremos avanzar en la depuración hasta el siguiente breakpoint ejecutaremos la opción de "continue" seleccionando *Product > Debug > Continue*. Por otro lado si queremos ir avanzando línea a línea en el código deberemos de seleccionar una de las opciones de *step* (*step over*, *step into* o *step out*) que se encuentran en *Product > Debug*. *Step over* detendrá la ejecución en la siguiente línea, *Step Into* la detendrá dentro del método que se "llame" desde la línea actual y *Step out* detendrá la ejecución cuando "salgamos" del método que se esté ejecutando. A todas estas opciones se puede acceder directamente desde los botones que están en la parte superior del panel de depuración.
- **Empezar de nuevo o abortar la ejecución:** Para detener la ejecución de la aplicación pulsaremos sobre el botón de "Stop" que se encuentra en la barra superior de XCode o desde *Product > Stop*.

Botón de Home

Si pulsamos el botón de *Home* en el simulador la aplicación no se detendrá ya que seguirá ejecutándose en segundo plano. Esto ocurre desde iOS 4 en donde la multitarea se introdujo. Si queremos detener la ejecución o depuración de una aplicación deberemos pulsar sobre el botón "Stop" que se encuentra en la barra superior de XCode.

Información depurador

El depurador de XCode nos ofrece una interfaz que nos facilita enormemente la depuración de nuestras aplicaciones. XCode hace uso de **GDB**, el famoso depurador de GNU. Recuerda que siempre que quieras puedes usar la línea de comandos si te sientes más cómodo. Puedes encontrar toda la documentación en su sitio oficial (<http://www.gnu.org/s/gdb/documentation/>).

9.3. Usando Instruments para detectar problemas de memoria

XCode, dentro de su conjunto de utilidades entre las que se encuentra el simulador de iPhone/iPad y el depurador, encontramos una serie de aplicaciones para depurar en un nivel algo más avanzado nuestras aplicaciones. Entre estas utilidades se encuentra **Instruments**, la cual es muy recomendable usar al menos una vez al final de nuestro desarrollo para capturar posibles fallos de memoria. Debemos de tener en cuenta que la memoria es un "bien" muy apreciado en todos los dispositivos móviles incluyendo los de Apple.

Los "famosos" *Memory leaks* o "fugas de memoria" son porciones de memoria que fueron reservadas o creadas en nuestra aplicación y que el programa pierde en un momento determinado. Estas partes de memoria nunca serán liberados por la aplicación y por tanto puede provocar múltiples fallos a nivel de ejecución que difícilmente podremos detectar si no es con la ayuda de las utilidades de XCode. Esto suele ocurrir cuando usamos dentro de nuestro código los métodos *new*, *malloc* o *alloc* y no hacemos posteriormente

`delete`, `free` o `release` respectivamente.

Cuando reservamos memoria haciendo uso de uno de los tres métodos comentados anteriormente (`new`, `malloc` o `alloc`) el sistema operativo espera que la liberemos cuando no la necesitemos más (usando `free`, `delete` o `release`). Los *Memory leaks* aparecen cuando no liberamos estos fragmentos de memoria. Esta mala gestión de la memoria puede provocar en el mejor de los casos que simplemente esa memoria se libere cuando paremos la aplicación y no pase nada, en el peor de los casos puede provocar la salida de la aplicación de forma inesperada por falta de memoria en el caso que el problema ocurra con bastante frecuencia durante la ejecución.

Memory Leaks

Puedes encontrar más información sobre los *Memory leaks* en esta dirección de la Wikipedia: http://en.wikipedia.org/wiki/Memory_leak

Algunos *Memory leaks* son fáciles de detectar en nuestro código, pero otros no. Para detectar los más complicados utilizaremos la herramienta que incluye el "kit" de desarrollador que se llama **Instruments**. A continuación explicaremos el uso de *Instruments* para la detección de estos problemas relacionados con la memoria, para ello realizaremos un ejemplo sencillo en el que veremos su funcionamiento. ¡A por ello!

Novedades en iOS 5

Con la llegada del SDK de iOS 5, los problemas con las fugas de memoria se han reducido considerablemente ya que no tendremos que preocuparnos nunca más de liberar la memoria reservada (usar "release") o de retenerla ("retain"). Esto se puede gracias a lo que en Apple han llamado *Automatic Reference Counting (ARC)*, activando esta opción en nuestros proyectos indicamos que es el nuevo compilador de Apple LLVM el encargado de hacer estas funciones de gestión de memoria reduciendo en gran medida los problemas de "fugas de memoria" comentados anteriormente.

9.3.1. Probando la aplicación de ejemplo

Para realizar las pruebas con *Instruments* y el depurador de XCode vamos a crear una aplicación desde cero. Arrancamos XCode (versión 4.2 en adelante) y seleccionamos *File > New > New Project > Single View Application*. Escribimos el nombre "sesion06-ejemplo1" y desmarcamos todas las opciones (incluyendo la comentada anteriormente "Use automatic reference counting").

Al haber desmarcado la opción de usar *ARC* la gestión de la memoria corre a nuestra cuenta, por lo que deberemos de tener mucho cuidado al respecto. Ahora abrimos el fichero `ViewController.m` y escribimos lo siguiente dentro del método `viewDidLoad`:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
```

```

    // Do any additional setup after loading the view, typically from
a nib.

NSString *str = [NSString stringWithFormat:@"Hola"];

NSMutableArray *array = [[NSMutableArray alloc] init];
[array addObject:@"opción 1"];

 NSLog(@"%@", Probando, probando..., str);

 [str release];
}

```

Ahora ejecutamos la aplicación mediante el botón de "Run" de la barra superior de la interfaz. Veremos que después de compilar y al poco de ejecutarse esta se para y aparece de nuevo la ventana de XCode mostrándonos algo similar a esto:

```

37 0x0131fde1 <+0145> movl $0x15,(%esp)
38 0x0131fde8 <+0152> call 0x138dc10 <__CFRecordAllocatic
39 0x0131fded <+0157> mov -0x10(%ebp),%eax
40 0x0131fdf0 <+0160> mov %eax,(%esp)
41 0x0131fdf3 <+0163> call 0x13fbff6 <dyld_stub_sel_getNa
42 0x0131fdf8 <+0168> mov %edi,0x10(%esp)
43 0x0131fdfc <+0172> mov %eax,0xc(%esp)
44 0x0131fe00 <+0176> add $0xa,%esi
45 0x0131fe03 <+0179> mov %esi,0x8(%esp)
46 0x0131fe07 <+0183> lea 0x12e6aa(%ebx),%eax
47 0x0131fe0d <+0189> mov %eax,0x4(%esp)
48 0x0131fe11 <+0193> movl $0x3,(%esp)
49 0x0131fe18 <+0200> call 0x136aad0 <CFLog>
50 0x0131fe1d <+0205> int3
51 0x0131fe1e <+0206> call 0x13fb8b4 <dyld_stub_getpid>
52 0x0131fe23 <+0211> mov %eax,(%esp)
53 0x0131fe26 <+0214> movl $0x9,0x4(%esp)
54 0x0131fe2e <+0222> call 0x13fb908 <dyld_stub_kill>
55 0x0131fe33 <+0227> xor %edi,%edi
56 0x0131fe35 <+0229> jmp 0x132012e <__forwarding__+99
57 0x0131fe3a <+0234> lea 0xed29f(%ebx),%eax
58 0x0131fe40 <+0240> mov %eax,0x4(%esp)
59 0x0131fe44 <+0244> mov %esi,(%esp)
60 0x0131fe47 <+0247> call 0x13fbcb6 <dyld_stub_strcmp>
61 0x0131fe4c <+0252> test %eax,%eax
62 0x0131fe4e <+0254> mov %edi,%eax
63 0x0131fe50 <+0256> mov %eax,%edi
64 0x0131fe52 <+0258> mov %eax,-0x14(%ebp)
65 0x0131fe55 <+0261> jne 0x131fe63 <__forwarding__+27

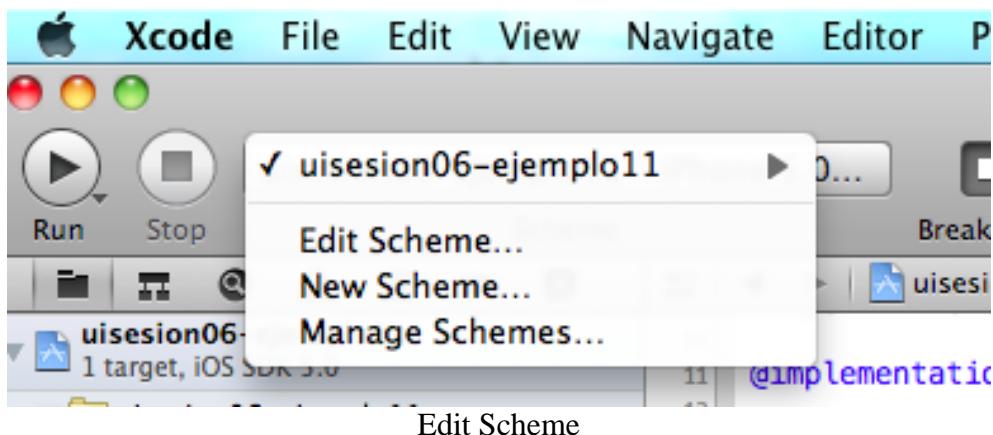
```

Pila de llamadas

Como podemos ver, a menos que sepamos programar en ensamblador, es imposible detectar el error de mediante esta traza. Para obtener algo más de información del error de memoria vamos a activar la opción de *Zombie*.

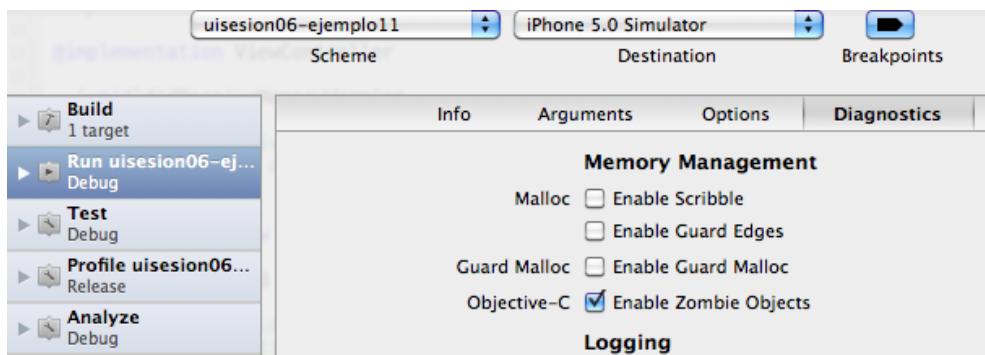
9.3.2. Usando NSZombieEnabled

NSZombieEnabled no es más que un parámetro de compilación cuya finalidad es proporcionarnos información adicional cuando intentemos acceder a un objeto que ya haya sido liberado de memoria. En este caso el compilador, en la mayoría de los casos, nos indicará dónde se ha producido el error (en qué línea de nuestro código) y qué objeto lo ha provocado. Para activar este parámetro en versiones de XCode superiores a la 4.0 deberemos pulsar sobre la barra superior de selección de esquema y seleccionar Edit Scheme.



Edit Scheme

Ahora en la ventana emergente que aparece seleccionamos en la columna de la izquierda el objetivo "Run" y en la derecha la pestaña de Diagnostics. Ahí deberemos marcar la opción de Enable Zombie Objects dentro del bloque de Memory Management. Pulsamos OK para guardar los cambios y cerrar la ventana.



Activando los objetos Zombie

Ahora volvemos a ejecutar la aplicación y veremos que cuando esta se detiene, en la ventana de la consola nos aparecerá algo más de información, como por ejemplo el tipo de objeto que ha provocado el fallo de memoria.

```
This GDB was configured as "x86_64-apple-darwin".
sharedlibrary apply-load-rules all
Attaching to process 11137.
2011-10-16 13:01:13.706 uisession06-ejemplo11[11137:207] Hola. Probando, probando...
2011-10-16 13:01:13.713 uisession06-ejemplo11[11137:207] *** -[CFString release]: message sent to
deallocated instance 0x6b1f890
(gdb)
```

Mensaje de consola del objeto zombie

Como podemos ver en la salida de la consola el objeto que se está gestionando incorrectamente es `NSString` al hacer `release` de él. Esto ocurre porque al crearlo usando el método `stringWithFormat` hacemos que `NSString` sea "autoliberable", o *autorelease*, por lo que es el propio compilador el que gestiona la memoria de este. Al forzar nosotros la liberación de la memoria usando `[str release]` estamos provocando el error, ya que el objeto puede haber desaparecido de la memoria anteriormente.

Para corregir este error, simplemente deberemos de suprimir la línea `[str release]`. De esta forma ya debe de funcionar correctamente la aplicación. Lo comprobamos volviéndola a ejecutar.

9.3.3. Encontrando fugas de memoria (memory leaks)

En este apartado vamos a detectar posibles fugas de memoria que se produzcan en nuestras aplicaciones. Estas son causadas, como hemos comentado en puntos anteriores por no liberar objetos de memoria en determinadas ocasiones dentro de nuestro código. A partir de iOS 5, estas liberaciones de memoria se realizarán de forma automática por el compilador y no deberemos de preocuparnos por ello, pero en versiones anteriores sí que lo tendremos que tener en cuenta. Para entender de una forma más clara lo que es una fuga de memoria y lo que implica vamos a continuar con el ejemplo que hemos realizado en el punto anterior.

Sobre los memory leaks y ARC

Aunque usando ARC no debemos de preocuparnos en exceso de los memory leaks, siempre es importante saber qué son y, en el caso de que nuestra aplicación funcione de manera incorrecta debido a fallos de memoria, poder solucionarlos satisfactoriamente. Aunque el compilador esté preparado para usar ARC, este no es efectivo al 100% y puede fallar en casos determinados.

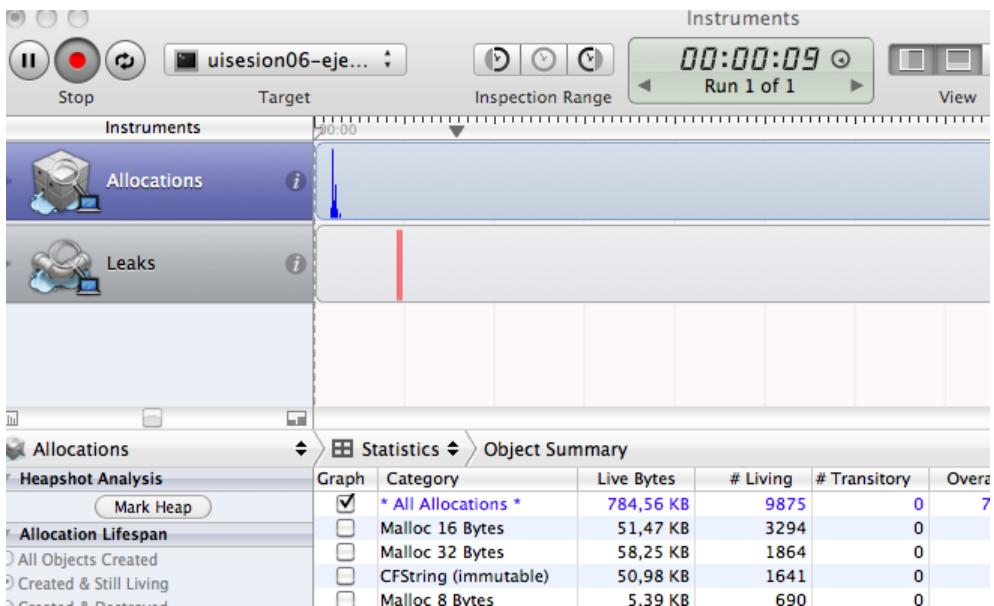
Para la detección y corrección de las posibles fugas de memoria que se produzcan en nuestras aplicaciones usaremos las herramientas de **Instruments** que incorpora XCode y que con la salida de *iOS 5* han mejorado notablemente. Siguiendo con el ejemplo anterior y comentando la línea `[str release]`, volvemos a ejecutar la aplicación. En este caso no notaremos nada extraño, pero internamente sí está pasando algo que a la larga puede perjudicar al rendimiento de la aplicación seriamente. Vamos a adivinar qué es...

Para analizar con detalle los *memory leaks* pulsamos sobre la opción del menú principal *Product > Profile*. En ese momento XCode compilará la aplicación y seguidamente abrirá una ventana emergente con distintas opciones, cada una de ellas corresponde a una utilizada del paquete de *Instruments*. En este caso deberemos seleccionar *Leaks*.



Pantalla principal de Instruments

Al pulsar sobre *Leaks* se abrirá una nueva interfaz en una ventana separada de XCode que estará divida en varias partes. Por un lado en la parte superior de la pantalla se encuentra el menu principal que tendrá los botones de control de ejecución: "pausa", "grabación/ejecución" y "reset", el "target" que estamos analizando, los distintos rangos de inspección, por defecto analizaremos toda la traza de ejecución y los distintos tipos de vista que están activadas.



Ejecución de Instruments



Barra superior de Instruments

En la parte central de la pantalla se encuentra toda la información que *Instruments* va recopilando a medida que la traza de ejecución va pasando. Al seleccionar *Leaks* utilizaremos dos instrumentos distintos: *Allocations* y *Leaks*. El primero nos indica en azul los objetos que la aplicación en ejecución va reservando en memoria. El segundo por otro lado nos indica en rojo los objetos que hemos creado en memoria y no hemos liberado cuando teníamos que haberlo hecho, las *fugas de memoria*.

A medida que la ejecución del programa va pasando, las gráficas de *Allocations* y de *Leaks* se van actualizando cada cierto tiempo, este intervalo de tiempo entre actualizaciones del estado de la memoria de la aplicación se puede cambiar desde el menu de la columna de la izquierda, dentro del bloque de *Snapshots* de *Leaks*. Por defecto el intervalo de actualización es de 10 segundos.

Cuando *Leaks* detecta una fuga de memoria la indica mediante una barra de color rojo dentro de la línea de tiempo. En el caso de que se detecten varios leaks al mismo tiempo, esta barra será mayor y en este caso la vista gráfica se irá alejando automáticamente para representar de una forma más clara los datos. En la parte justo inferior a las barras gráficas de *Allocations* y *Leaks* se encuentra la misma información pero en forma de listado. Ahí se mostrarán todos los objetos a los que se hace referencia en las barras gráficas. Si pulsamos sobre la propia barra de *Leaks* se seleccionará el objeto al que hace referencia dentro del listado.

Statistics		Object Summary						
Graph	Category	Live Bytes	# Living	# Transitory	Overall Bytes	# Overall	# Allocations (Net / Overall)	
<input checked="" type="checkbox"/>	* All Allocations *	783,54 KB	9867	0	783,54 KB	9867	+++	
<input type="checkbox"/>	Malloc 16 Bytes	51,45 KB	3293	0	51,45 KB	3293		
<input type="checkbox"/>	Malloc 32 Bytes	58,22 KB	1863	0	58,22 KB	1863		
<input type="checkbox"/>	CFString (immutable)	50,98 KB	1641	0	50,98 KB	1641		
<input type="checkbox"/>	Malloc 8 Bytes	5,38 KB	689	0	5,38 KB	689		
<input type="checkbox"/>	CFString (store)	51,33 KB	361	0	51,33 KB	361		
<input type="checkbox"/>	Malloc 48 Bytes	14,77 KB	315	0	14,77 KB	315		
<input type="checkbox"/>	Malloc 64 Bytes	11,44 KB	183	0	11,44 KB	183		
<input type="checkbox"/>	CFBasicHash (value-store)	19,05 KB	145	0	19,05 KB	145		
<input type="checkbox"/>	CFDictionary (mutable)	6,28 KB	134	0	6,28 KB	134		
<input type="checkbox"/>	CFBasicHash (key-store)	18,14 KB	116	0	18,14 KB	116		
<input type="checkbox"/>	Malloc 80 Bytes	8,20 KB	105	0	8,20 KB	105		
<input type="checkbox"/>	Malloc 96 Bytes	5,91 KB	63	0	5,91 KB	63		
<input type="checkbox"/>	CFArray (mutable-vari...	2,05 KB	62	0	2,05 KB	62		
<input type="checkbox"/>	Malloc 1,00 KB	54,00 KB	54	0	54,00 KB	54		

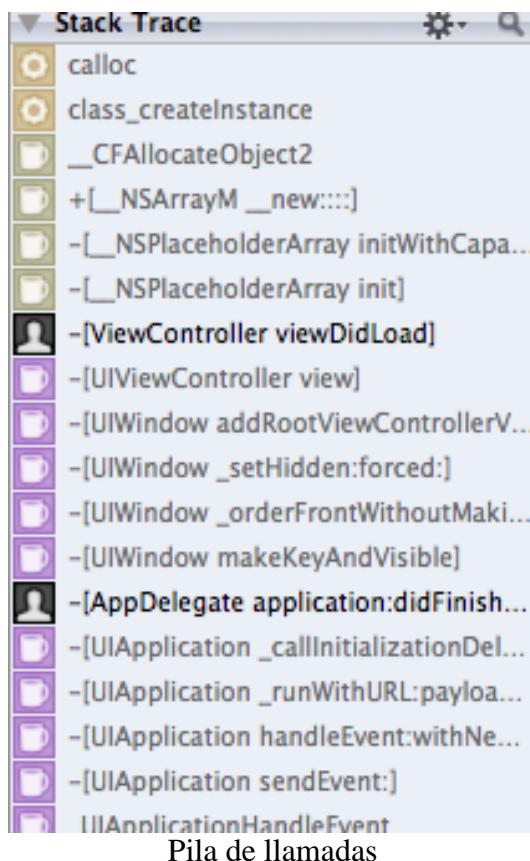
Listado de objetos usados

Si seleccionamos uno de los objetos en los que *Instruments* ha detectado un *leak* y hacemos doble click sobre el accederemos a la línea exacta de nuestro código en donde se referencia a ese objeto. De esta forma podremos situarnos en el objeto en el que no liberamos su memoria y podremos realizar los cambios oportunos en nuestro código para remediarlo. Esto se puede hacer directamente desde *Instruments*.

Una vez que hemos visto las principales características del funcionamiento de *Instruments* y su detección de *Leaks* vamos a proceder a realizar lo propio con nuestra aplicación de ejemplo. Para ello la arrancamos mediante *Product > Profile*. Seleccionamos la opción de *Leaks* y esperamos a que arranque *Instruments*. En ese momento veremos como la línea de tiempo empieza a avanzar detallando en la gráfica superior las reservas de memoria y en la barra inferior los *leaks* encontrados. En este caso *Instruments* detecta dos *leaks* en el segundo seis aproximadamente, aunque el momento puede variar por muchos factores externos.

En cualquier momento de la ejecución podemos pausar la depuración por *Instruments* pulsando sobre el botón de pausa de la barra superior del menu. También podemos detener el análisis y guardar los resultados para más adelante analizarlos. Si después de pausar la ejecución la volvemos a poner en marcha veremos como la linea de tiempo se posicionará en el momento en donde le corresponde y dejará un hueco vacío en medio sin datos.

Nosotros vamos a detener la ejecución pulsando sobre el botón rojo de *stop*. Una vez detenido el análisis y la aplicación nos situamos sobre los *leaks* detectados (la única barra roja que hay) y vemos como en el listado de objetos aparecen dos: `malloc 16 bytes` y `NSMutableArray`. Si desplegamos la columna de la derecha pulsando sobre el botón de la derecha del todo del apartado "view" del menu superior veremos toda la traza de la aplicación y, en negrita, los métodos en los que se ha detectado el *leak* seleccionado. Como podemos ver el método al que hace referencia es `[ViewController viewDidLoad]`. Haciendo doble click sobre el nombre del método podremos ver el código y subrayado en color violeta los métodos responsables del *leak*.



Pila de llamadas

```

NSString *str = [NSString stringWithFormat:@"Hola"];
NSMutableArray *array = [[NSMutableArray alloc] init];
[array addObject:@"opción 1"];
 NSLog(@"%@", Probando, probando..., str);

```

Detección de Memory Leak

Volviendo al código y analizándolo con un poco más de detalle vemos que la variable array de tipo NSMutableArray se crea en memoria usando alloc y en ningún momento la liberamos, por lo que incumplimos la norma básica de la gestión de memoria en iOS: todo lo que se crea en memoria se debe liberar una vez que no lo necesitemos. Por tanto, para solucionar esta *fuga de memoria* deberemos llamar a release al final del método quedando de la siguiente manera:

```

- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from
    // a nib.

    NSString *str = [NSString stringWithFormat:@"Hola"];
}

```

```

NSMutableArray *array = [[NSMutableArray alloc] init]; //Reserva memoria
[array addObject:@"opción 1"];

NSLog(@"%@", Probando, probando..., str);
[array release]; //Libera la memoria: evita el leak
}

```

Ahora volvemos a ejecutar la aplicación con *Instruments* y comprobaremos que ya no nos aparece ningún *Leak*. ¡Problema resuelto!

9.4. Pruebas de unidad

Los test de unidad o unitarios son una forma de probar el correcto funcionamiento de un módulo o método concreto de la aplicación que desarrollemos. La finalidad de este tipo de pruebas es comprobar que cada uno de los módulos de la aplicación funcione según lo esperado por separado. Por otro lado, los tests de integración comprueban que la aplicación funciona correctamente en su totalidad. Dentro de este módulo veremos como implementar las pruebas de unidad en nuestras aplicaciones iOS.

XCode facilita en gran manera la generación de las pruebas de unidad, proporciona un entorno adaptado y simple. Las pruebas de unidad están compuestas por los *tests*, los cuales comprueban que un bloque de código devuelve un determinado resultado, en caso contrario el test falla. Los *grupos de tests* son un conjunto de tests que prueban una determinada característica o funcionalidad de la aplicación. El *framework* en el que se basa XCode para su entorno de testing es de código abierto y se llama *SenTestingKit*.

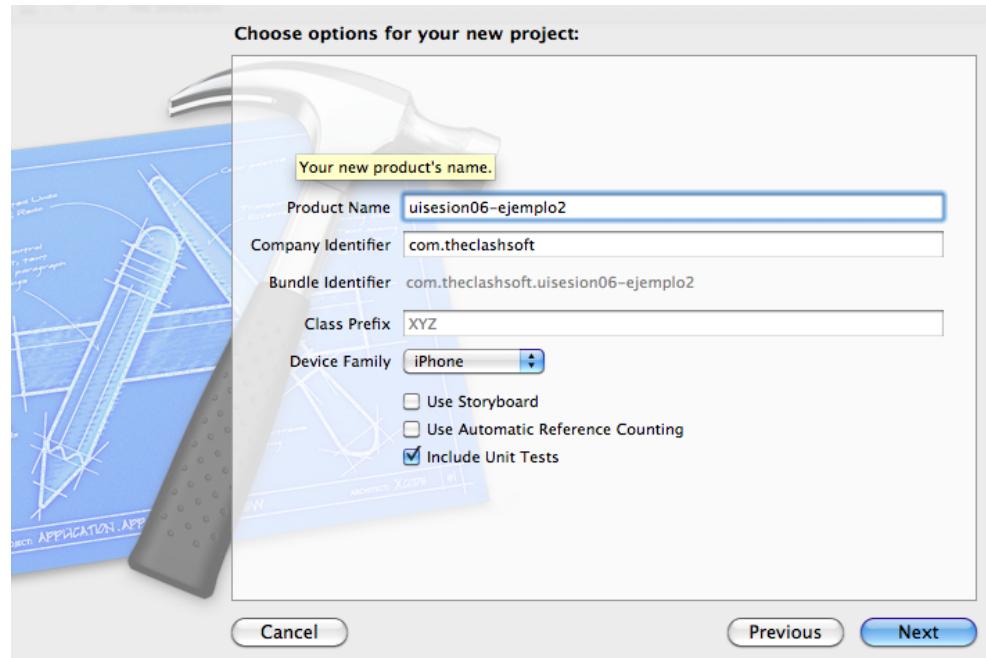
En XCode encontramos dos tipos de tests de unidad:

- **Tests de lógica:** Comprueban el correcto funcionamiento de cada uno de los métodos por separado, no en su conjunto en toda la aplicación. Los tests de lógica se pueden implementar para hacer pruebas de estres, poniendo a prueba la aplicación para situaciones extremas no habituales en una ejecución nominal. Estos tipos de pruebas sólo se pueden probar con el simulador de iOS.
- **Test de aplicación:** Comprueban el correcto funcionamiento de la aplicación en su conjunto. Incluyen tests de conectividad con las interfaces de usuario, tests de sensores, etc. Este tipo de pruebas sí se pueden realizar sobre dispositivos iOS, a parte de los simuladores.

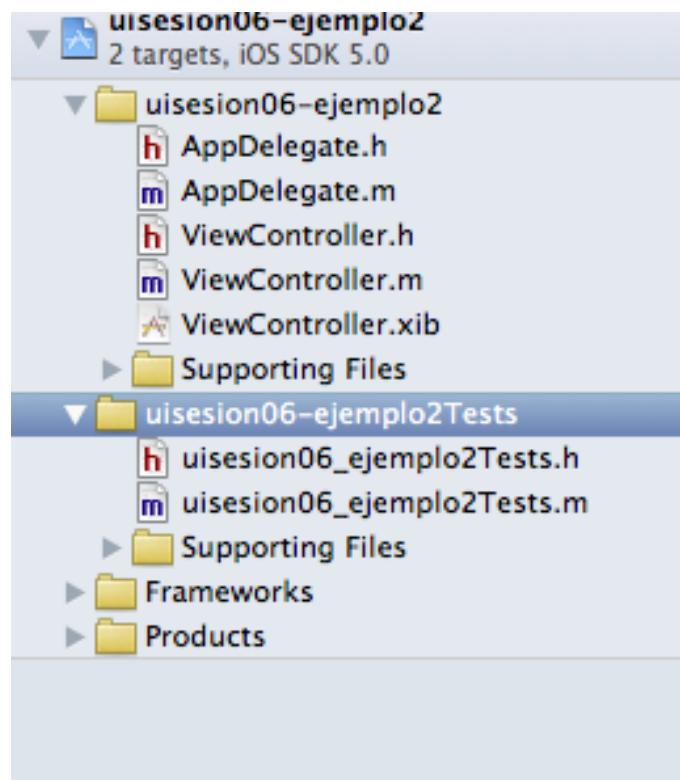
9.4.1. Configurando XCode para ejecutar tests

En el ejemplo que vamos a realizar a continuación vamos a realizar los dos tipos de pruebas comentadas anteriormente: de lógica y de aplicación. Para ello empezamos creando un nuevo proyecto en XCode de tipo *Single View Application* que llamaremos *uisesion06-ejemplo2*. Debemos de acordarnos de marcar la opción de *Include Unit*

Tests, de esta forma el propio XCode nos creará un subdirectorio dentro del proyecto en donde se ubicarán los distintos tests de unidad que realicemos.

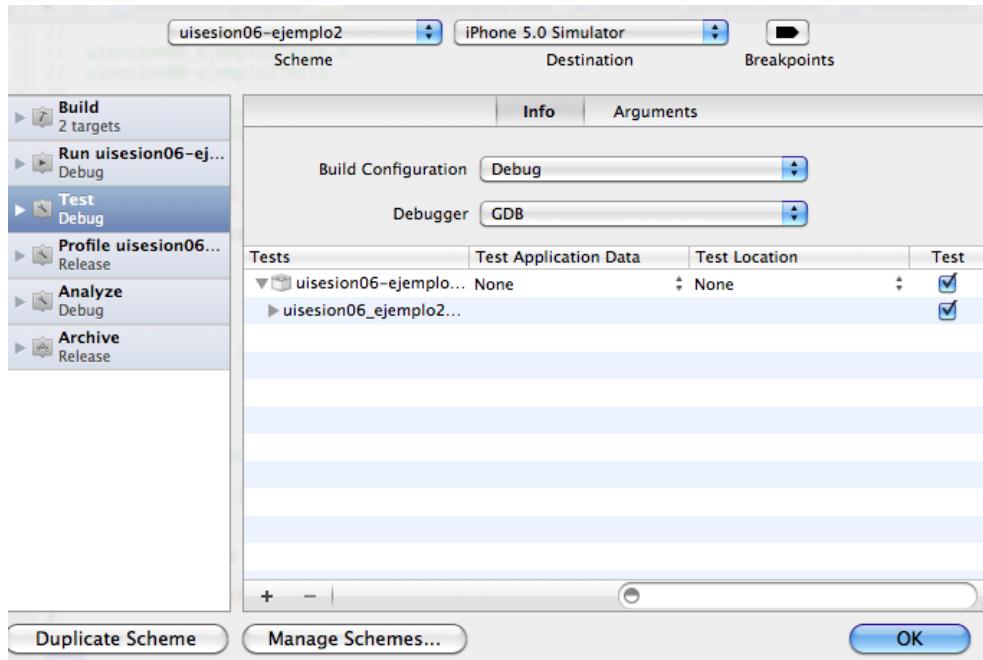


Propiedades del proyecto



Carpeta de Tests del proyecto

XCode también ha creado de forma automática un *target* nuevo para tests que podremos modificar a nuestro antojo y un *esquema*. Esto lo podemos ver pulsando sobre el raíz del proyecto y pulsando sobre `Edit Scheme` dentro del listado de esquemas del proyecto respectivamente.



Target de Tests en el proyecto

Para ejecutar los tests que se crean por defecto al generar el proyecto deberemos de pulsar sobre *Product > Test* y esperar a que se compile el proyecto y arranquen los tests. Veremos que este caso falla uno, el correspondiente al método `(void)testExample`. Como podemos ver los tests que no han pasado se muestran en color rojo tanto en el código como en el navegador de la izquierda:

```

1 // 
2 // uisession06_ejemplo2Tests.m
3 // uisession06_ejemplo2Tests
4 //
5 // Created by Javier Aznar de los Ríos on 17/10/11.
6 // Copyright (c) 2011 __MyCompanyName__. All rights reserved.
7 //

8 #import "uisession06_ejemplo2Tests.h"

9 @implementation uisession06_ejemplo2Tests

10 - (void)setUp
11 {
12     [super setUp];
13     // Set-up code here.
14 }
15 
16 - (void)tearDown
17 {
18     // Tear-down code here.
19     [super tearDown];
20 }
21 
22 - (void)testExample
23 {
24     STFailf(@"Unit tests are not implemented yet in uisession06_ejemplo2Tests");
25 }
26 
27 @end
28 
```

Fallo de test

Si desplegamos la sección de abajo y el log de salida veremos un resumen de los tests que se han pasado y sus resultados:

```
Test Suite 'All tests' started at 2011-10-21 10:48:33 +0000
Test Suite
'/Users/javikr/Documents/Projects/builds/uisesion06-ejemplo2-eqfkbbmboigtv
gumflvqvufkhlb/Build/Products/Debug-iphonesimulator/uisesion06-ejemplo2Tests.octest
(Tests)' started at 2011-10-21 10:48:33 +0000
Test Suite 'uisesion06_ejemplo2Tests' started at 2011-10-21 10:48:33 +0000
Test Case '-[uisesion06_ejemplo2Tests testExample]' started.
/Users/javikr/Documents/Projects/CURSO/uisesion06-ejemplo2/uisesion06-ejemplo2Tests/
uisesion06_ejemplo2Tests.m:29: error: -[uisesion06_ejemplo2Tests
testExample] :
Unit tests are not implemented yet in uisesion06-ejemplo2Tests
Test Case '-[uisesion06_ejemplo2Tests testExample]' failed (0.000
seconds).
Test Suite 'uisesion06_ejemplo2Tests' finished at 2011-10-21 10:48:33
+0000.
Executed 1 test, with 1 failure (0 unexpected) in 0.000 (0.000) seconds
Test Suite
'/Users/javikr/Documents/Projects/builds/uisesion06-ejemplo2-eqfkbbmboig
tvggumflvqvufkhlb/Build/Products/Debug-iphonesimulator/uisesion06-ejemplo2Tests.octest
(Tests)' finished at 2011-10-21 10:48:33 +0000.
Executed 1 test, with 1 failure (0 unexpected) in 0.000 (0.001) seconds
Test Suite 'All tests' finished at 2011-10-21 10:48:33 +0000.
Executed 1 test, with 1 failure (0 unexpected) in 0.000 (0.025) seconds
```

Una vez que hemos configurado nuestro proyecto de XCode para ejecutar tests de unidad vamos a ver cómo escribir estos tests de una forma eficiente.

9.4.2. Escribiendo tests de unidad

Un test no deja de ser una instancia de un conjunto o *suite* de tests, el cual no recibe parámetros ni devuelve ningún tipo de objeto, sólo `void`. El objetivo del test es probar la API de la aplicación y detectar si produce el resultado esperado. Este resultado puede ser una excepción o un valor cualquiera. Los tests hacen uso de una serie de *macros*, las cuales debemos de conocer saber las distintas posibilidades que disponemos a la hora de diseñar los casos de prueba. Para crear una *suite de tests* deberemos crear una clase que herede de `SenTestCase`. Para crear nuevos tests dentro de una *suite de tests* ya creada simplemente deberemos de implementarlo siguiendo la siguiente estructura:

Macros disponibles

El framework de `SenTestingKit` ofrece un amplio listado de macros que podemos utilizar en nuestras pruebas de unidad. El listado completo lo podemos ver dentro de la documentación oficial de iOS en [esta](#) dirección. Existen macros para producir fallos de forma incondicional, para comprobar dos valores concretos, para comprobar referencias a valores nulos, comprobaciones de *booleanos*, otros que comprueban si se lanza o no una excepción y de qué tipo...

```
- (void)testMiTestDePrueba {
    ...
    // Configuración inicial (setup)
```

```
ST...    // Asserts
...
}           // Liberación de memoria y variables auxiliares
```

Normalmente, para configurar las variables auxiliares que son necesarias para ejecutar los casos de pruebas implementaremos dos métodos: `(void)setup` y `(void)tearDown`. El primero creará las variables y las configurará de forma adecuada, el segundo las liberará de memoria. Esta sería la estructura básica de ambos métodos, los cuales están por defecto incluidos al crear los tests de unidad con XCode:

```
- (void)setUp {
    objeto_test = [[[MiClase alloc] init] retain];
    STAssertNotNil(objeto_test, @"No se puede crear un objeto de la clase
MiClase");
}

- (void)tearDown {
    [objeto_test release];
}
```

Fallo en setup o tearDown

Cuando XCode detecta algún fallo en los métodos `Setup` o `tearDown`, estos se reportarán dentro del caso de pruebas que se esté probando y desde el que se hizo la llamada.

10. Depuración y pruebas - Ejercicios

10.1. Usando NSLog con distintos tipos de datos (0.5 puntos)

En este primer ejercicio vamos a practicar el uso de las sentencias `NSLog` con diferentes tipos de variables, para ello nos descargamos la [plantilla](#) del ejercicio y completamos el método `viewDidLoad` del fichero `UAVViewController.m` con el código necesario.

10.2. Breakpoints y análisis de variables (1 punto)

En este ejercicio vamos a practicar el uso del depurador de XCode creando *breakpoints* para analizar el código fuente y detectar distintos tipos de errores. El objetivo de este ejercicio será obtener el valor de algunas variables en la aplicación de la plantilla que podemos descargarlos [aquí](#).

- Pon un *breakpoint* antes del bucle `for` para ver el valor de los objetos del array
- Pon un *breakpoint* dentro del bucle `for` para ver como va cambiando el valor del string `str`

10.3. NSZombie: Detectando las excepciones de memoria (0.5 puntos)

Usando la misma plantilla que el ejercicio anterior, desactivamos o borramos los *breakpoints* y volvemos a ejecutar la aplicación. Si ejecutamos la aplicación... ¿Por qué falla? Vamos a analizar el código para encontrar el error.

Nota

Esta aplicación **no** está usando ARC (Automatic Reference Counting).

Usando la opción de compilación **NSZombieEnabled** ¿Qué error nos aparece por la consola? ¿Cómo podemos arreglarlo?

Una vez solucionado el problema, volvemos a arrancar la aplicación y esta deberá de funcionar correctamente.

10.4. Instruments: detectando memory leaks (1 puntos)

Siguiendo con el ejercicio anterior, ya con la excepción de memoria solucionada, vamos a practicar el uso de la herramienta *Instruments* de XCode. Esta herramienta la usaremos para detectar posibles fugas de memoria (o *memory leaks*) en nuestras aplicaciones iOS.

Ejecutamos Instruments en XCode pulsando sobre *Product > Profile* y escogemos la opción de *Leaks*. Ejecuta la aplicación y responde a las siguientes cuestiones:

- ¿Se produce algún memory leak durante la ejecución?
- Si se produce un memory leak, ¿Qué objeto del código lo genera?
- ¿Cómo podemos evitar el memory leak? Si se puede solucionar, solúcnalo.

