Introducción a Objective-C

Índice

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.

1. Tipos de datos

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;
```

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;
```

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);
```

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`.

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.

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.

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.

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`).

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
```

2.4. Modificadores

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

2.4.1. Modificador const

Indica que el valor de una variable no va a poder se 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.

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 inicilizar, 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 instancie 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 definiese 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.

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. 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.

4. 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.

4.1. 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`.

4.2. 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:

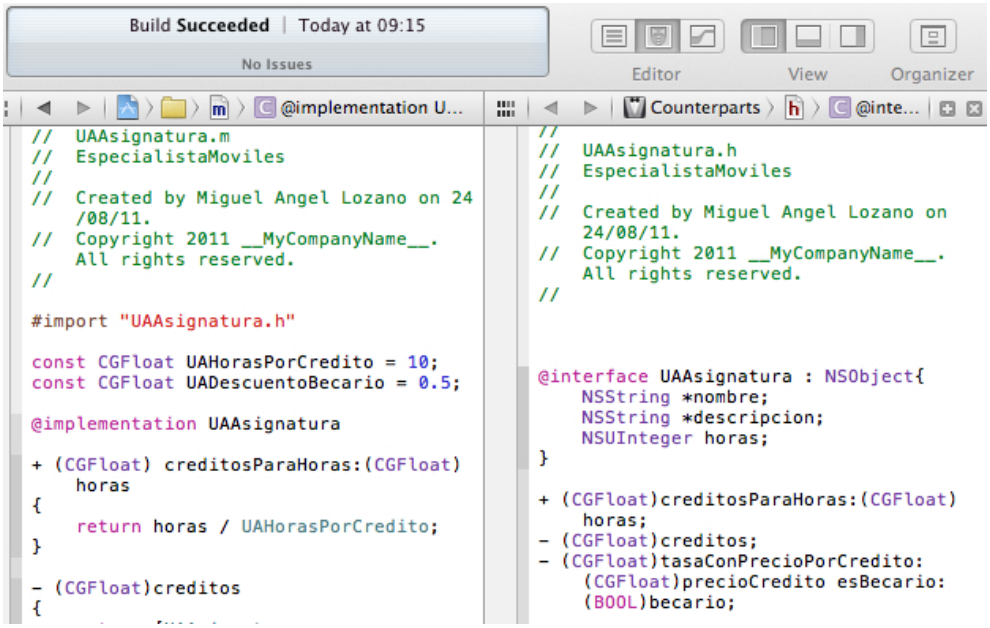
```
+ (void)escribe:(NSInteger)n, ...;
```

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 complementarío. Para facilitar esta tarea, podemos aprovechar la vista del asistente en Xcode.



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*.

4.3. 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];
NNString *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 pasarle 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 heredarán 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];
NNString *cadFormato = [NSString stringWithFormat: @"Numero %d", 5];
```

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

4.3.1. 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.

4.3.2. 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 mantenible, 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 nuestra 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).

4.4. 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.

Atención

Lo más importante es que el número de llamadas a `alloc` y `retain` sea el mismo que a `release`. 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 la referencias retenidas de nuestras variables de instancia, para así liberar todo lo que haya podido retener la superclase.

4.4.1. 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:@"%@ (%d horas)",
                                _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 si, 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`), 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.

4.4.2. 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.

4.4.3. 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 `autoreleases` 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.

4.4.4. 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 objecto 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. 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.

6. 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 definan 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 MiObjecto ()
- (void)metodoPrivado;
@end


@implementation MiObjeto

// Otros metodos
...

- (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.

7. 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`.

7.1. 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.

7.1.1. 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. Estos métodos pondrán a uno el contador de referencias de la nueva instancia (pertenecen al grupo de métodos que incrementan este contador, junto a `alloc` y `retain`). No todos los objetos son copiables, a continuación veremos más detalles sobre la copia de objetos.

7.1.2. 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 la API de Cocoa encontramos varios objetos que se encuentran disponibles en ambas versiones, como 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 otro lado, los mutables son aquellos cuyo estado si puede ser modificado.

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.

Un objeto que existe en ambas modalidades es por ejemplo 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 = [cadena copy];

// copiaMutable!=cadena
NSMutableString *copiaMutable = [cadena mutableCopy];
```

7.1.3. 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 nuestra 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*.

7.2. 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.

7.2.1. 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.

7.2.2. 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 cStringWithEncoding: NSASCIIStringEncoding];
```

7.2.3. 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: %@ (edad %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 `%@`.

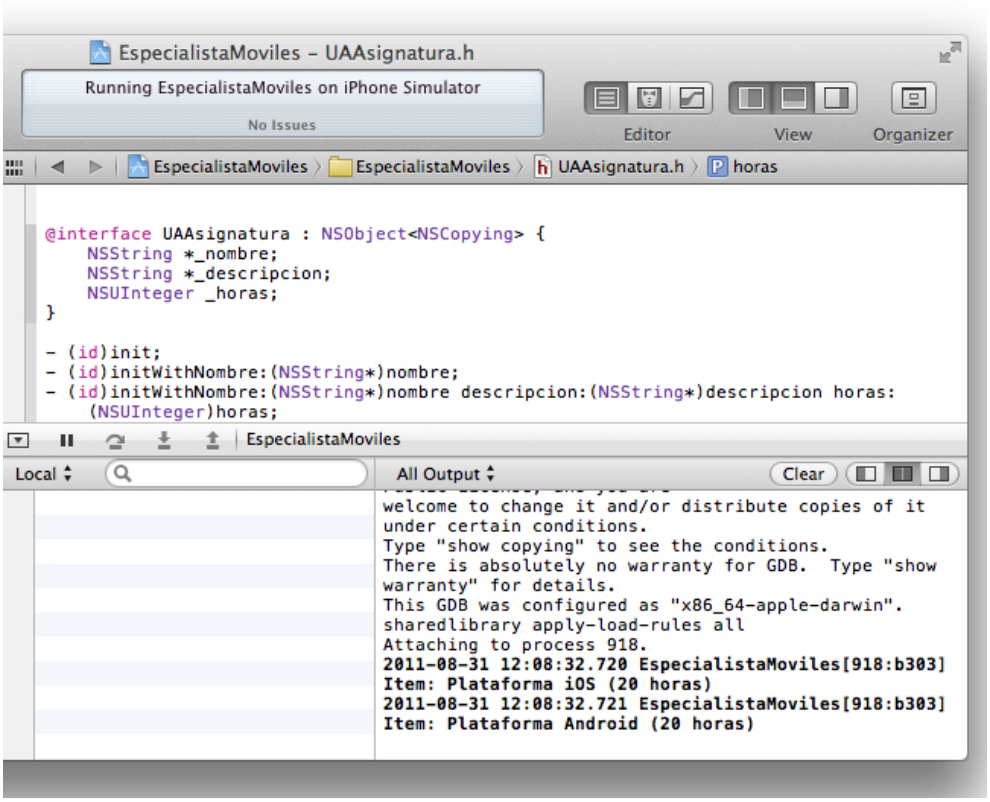
Los mismos atributos de formato se podrán utilizar en la función `NSLog` que nos permite escribir *logs* en la consola para depurar aplicaciones

```
NSLog(@"i = %d, obj = %@", 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

7.2.4. 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, anyo ];
```

7.2.5. 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];
```

7.2.6. 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`).

7.3. 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`.

7.3.1. 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:`, u obtener la diferencia entre dos fechas componente a componente con `components:fromDate:toDate:options:`.

7.3.2. 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 podremos utilizar nuestro objeto de formato para *parsear* fechas con `dateFromString:`

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

7.4. 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
    // Codigo
@catch(NSException *ex) {
    // Codigo tratamiento excepcion
}
@catch(id obj) {
    // Codigo tratamiento excepcion
}
@finally {
    // Codigo 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 `NSException` (o una subclase de ésta). Otra diferencia es que en Objective-C no suele ser habitual heredar de `NSException` 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 [[[NSException alloc] initWithName: @"Error"
                                   reason: @"Descripcion 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.