Propiedades, colecciones y gestión de la memoria

Índice

1. 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.

La 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 `@synthetize` 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 crealos).

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.

1.1. 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.

1.2. 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 ancestras 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.

1.3. Propiedades atómicas

En los ejemplos anteriores hemos considerado siempre propiedades no atómicas (con el modificador `nonatomic`), que serán las que utilicemos 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.

1.4. 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, liberándo 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 > Refector > 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*.

1.5. 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`*.

1.6. 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.

2. 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.

2.1. 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];
```

2.2. 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];
```

2.2.1. 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 `indexOfObject`:

```
NSUInteger indice = [lista indexOfObject: obj];
if(indice==NSNotFount) {
    // 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 `indexOfObjectIdenticalTo`.

2.2.2. 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.

2.3. 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.

2.4. 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"];
```

2.5. Recorrer las colecciones

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

```
for(id obj in lista) {
    NSLog(@"Obtenido el objeto %@", 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(@"Obtenida la cadena %@", 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(@"Obtenido el objeto %@", 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.

2.6. Almacenamiento persistente

Las colecciones se pueden grabar en ficheros de forma muy sencilla. Para hacer esto todas ellas nos proporcionan un método `writeToFile:atomically:` que se encarga de volcar el contenido de la colección al fichero cuya ruta especificamos. A la hora de trabajar con ficheros, no podremos guardarlos en cualquier ruta, sino que deberemos utilizar las rutas que el dispositivo reserva a nuestra aplicación. Un directorio habitual donde podemos almacenar los datos es el directorio de documentos (representado por `NSDocumentDirectory`). Podemos obtener la ruta de dicho directorio de la siguiente forma:

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                                     NSUserDomainMask,
                                                     YES);

if ([paths count] > 0)
{
    NSString *directory = [paths objectAtIndex:0];
    NSString *filename = [directory
             stringByAppendingPathComponent:@"coleccion.plist"];

    BOOL guardado = [coleccion writeToFile:filename
```

```
                                    atomically:YES];
}
```

Nota

El contenido el directorio de documentos será guardado automáticamente en las copias de seguridad que se realicen mediante iTunes o iCloud. Según los datos que almacenemos, deberemos elegir el directorio apropiado de nuestra aplicación. Para más información sobre el sistema de archivos se puede consultar el tutorial *File System Basics* de la documentación proporcionada por Apple.

La colección se almacena en un fichero con formato `plist` (hemos visto que gran parte de los ficheros de configuración del proyecto iOS utilizan este mismo formato). Debido a las limitaciones del formato, debemos tener en cuenta que sólo podremos guardar colecciones que contengan únicamente los siguientes tipos de datos:

- `NSString`
- `NSData`
- `NSDate`
- `NSNumber`
- `NSArray`
- `NSDictionary`

Si alguno de los componentes de la colección fuese de cualquier otro tipo la colección no se guardaría (la llamada a `writeToFile:atomically:` devolvería `NO`).

También podemos leer los datos de una colección a partir de un fichero `plist`. Para ello contamos con un inicializador, y su correspondiente método factoría:

```
- (id)initWithContentsOfFile:
+ (id)arrayWithContentsOfFile:
```

En este caso, podemos utilizarlo para leer los ficheros que hayamos almacenado anteriormente en el directorio de documentos:

```
BOOL fileExists = [[NSFileManager defaultManager]
                                    fileExistsAtPath:filename];
if (fileExists) {
    coleccion = [NSArray arrayWithContentsOfFile:filename];
}
```

Pero también podríamos leer ficheros `plist` que hayamos creado con Xcode y hayamos empaquetado junto a la aplicación (estos ficheros serán de sólo lectura):

```
NSString *filename = [[NSBundle mainBundle] pathForResource:@"datos"
                                                     ofType:@"plist"];

coleccion = [NSArray alloc arrayWithContentsOfFile:filename];
```

Si queremos almacenar objetos propios que no pertenezcan a la lista de tipos de datos mencionada anteriormente, podemos hacerlo adoptando el protocolo `NSCoding`:

```
@interface UAAsignatura : NSObject<NSCoding>
```

```
    ...
@end
```

Este protocolo nos obliga a definir dos métodos que se encargarán de serializar y deserializar el objeto:

```
- (id)initWithCoder:(NSCoder *)aDecoder {
    self = [super init];
    if(self!=nil) {
        self.nombre = [aDecoder decodeObjectForKey:@"nombre"];
        self.descripcion = [aDecoder decodeObjectForKey:@"descripcion"];
        self.horas = [aDecoder decodeIntegerForKey:@"horas"];
    }
    return self;
}

- (void)encodeWithCoder:(NSCoder *)aCoder {
    [aCoder encodeObject:self.nombre forKey:@"nombre"];
    [aCoder encodeObject:self.descripcion forKey:@"descripcion"];
    [aCoder encodeInteger:self.horas forKey:@"horas"];
}
```

Una vez definidos estos métodos, podremos guardar o leer el objeto de forma sencilla con `NSKeyedArchiver` y `NSKeyedUnarchiver`:

```
+ (UAAsignatura *) load {

    UAAsignatura *asig = nil;

    NSArray *dirs = NSSearchPathForDirectoriesInDomains(
                        NSDocumentDirectory, NSUserDomainMask, YES);
    if([dirs count] > 0) {
        NSString *filename = [[dirs objectAtIndex:0]
                              stringByAppendingPathComponent: @"datos"];
        asig = [NSKeyedUnarchiver unarchiveObjectWithFile: filename];
    }

    return asig;
}

- (BOOL) save {
    NSArray *dirs = NSSearchPathForDirectoriesInDomains(
                        NSDocumentDirectory, NSUserDomainMask, YES);
    if([dirs count] > 0) {
        NSString *filename = [[dirs objectAtIndex:0]
                              stringByAppendingPathComponent: @"datos"];

        return [NSKeyedArchiver archiveRootObject: self
                                           toFile: filename];
    } else {
        return NO;
    }
}
```

Nota

Debemos destacar que todas las colecciones implementan `NSCoding`, por lo que podemos almacenarlas también de la forma anterior.

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 conozcamos 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 (`.`).

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.

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 instancia 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"];
```

4. 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.

4.1. 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 hubiesemos 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).

4.2. 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 caracter, dejaríamos de recibir las notificaciones correctamente.

4.3. 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:NSKeyValueObservingOptionNew
                 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 liberé de la memoria. Para ello utilizaremos `removeObserver:forKeyPath:`.

4.4. 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.

4.5. 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");
                }];
```

5. 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.

5.1. 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)];
```

5.2. 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 `instancesRespondsToSelector:`, 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.

5.3. 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 u 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:`.

5.4. 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.

5.5. 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.

6. 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 cerremos 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, éste 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).