aula MENTOR

GOBIERNO DE ESPAÑA

MINISTERIO DE EDUCACIÓN, CULTURA Y DEPORTE

Programación de aplicaciones para Iphone y Ipad

AULA MENTOR

aulamentor.es

Catálogo de publicaciones del Ministerio: www.educacion.gob.es
Catálogo general de publicaciones oficiales: www.publicacionesoficiales.boe.es

Autor
Francisco Javier Honrubia López

Coordinación pedagógica
Hugo Alvarez Garrote
Susana Pérez Marín

Edición y maquetación de contenidos
Almudena Bretón

Diseño gráfico y portada
Almudena Bretón

MINISTERIO
DE EDUCACIÓN, CULTURA
Y DEPORTE

Edita:
© SECRETARÍA GENERAL TÉCNICA
Subdirección General
de Documentación y Publicaciones

NIPO: 030-13-092-3
ISBN: 978-84-369-5460-9

ÍNDICE

Pág.

UNIDAD 1. Introducción a Objetive-C ... 5
1. Intruducción ... 5
2. Clases y objetos en Objective-C ... 6
2.1 Creación de clases en Objective-C ... 7
2.2 Comunicación entre objetos: paso de mensajes ... 7
2.2.1 Creación o instanciación de un objeto ... 7
2.2.2 Paso de argumentos en los mensajes ... 9
2.2.3 Destrucción de objetos ... 9
2.3 Herencia en Objective-C ... 10
3. Variables de instancia ... 12
3.1 Tipos de datos en Objective-C ... 12
4. Métodos: de instancia y de clase ... 13
4.1 Métodos de instancia y de clase ... 14
4.2 Métodos de acceso ... 15
5. Inicializadores ... 16

UNIDAD 2. PATRONES DE DISEÑO ... 18
1. Introducción ... 18
2. Definición de patrones de diseño ... 19
3. Patrón Modelo-Vista-Controlador ... 20
4. Patrón Delegate ... 21
4.1 Protocolos ... 24
5. Patrón paso de mensajes ... 25
5.1 Centro de notificaciones ... 25
5.2 Observación Clave-Valor ... 27
5.3 Target-Action ... 27

UNIDAD 3. INTRODUCCIÓN A XCODE 4 ... 29
1. Introducción ... 29
1.1 Estructura de la unidad didáctica ... 29
2. Descarga e instalación de Xcode 4.5 ... 29
2.1 Utilizando la App Store ... 30
2.2 Ejecutando Xcode ... 32
3. Herramientas integradas en Xcode 4.5 ... 34
4. Primeros pasos en Xcode 4.5: el entorno ... 35
5. La primera aplicación ... 45
5.1 Consulta de la documentación ... 47

6. El simulador de dispositivos 48
6.1 Carga de versiones de iOS en el simulador 51
6.2 Instalar la aplicación en un dispositivo físico 53

UNIDAD 4. GESTIÓN DE MEMORIA 55
1. Introducción 55
1.1 Estructura de la unidad didáctica 55
2. Gestión de memoria en C 56
2.1 Heap 56
2.2 Stack 57
3. Gestión de memoria en iOS 5: Reference Counting 58
4. Gestión de memoria en iOS 5: Automatic Reference Counting 61
4.1 Declaración de atributos 62
5. El recolector de basura 65
6. Evitando problemas 66
6.1 Ciclo de retención 66

UNIDAD 5. CLASE UIView 68
1. Introducción 68
1.1 Estructura de la unidad didáctica 68
2. Elementos de UIView. Jerarquía de clases 69
3. Vistas personalizadas. 75
3.1 El método drawRect 80
4. Controladores de vistas 82
4.1 Controlador de vistas de tabla (TableView Controller) 83
4.2 Controlador de navegación (Navigation Controller) 84
4.3 Controlador de pestañas (TabBar Controller) 91
4.4 SplitView Controller 94
4.5 Popover 95
5. Ficheros XIB y NIB 96
6. Storyboards 103

UNIDAD 6. LAS CLASES UITABLEVIEW Y UITABLEVIEWCONTROLLER 112
1. Introducción 112
1.1 Estructura de la unidad didáctica 113
2. Vistas de tablas: UITableView. 113
2.1 UITableViewCells 114
3. Controlador de vista de tabla: UITableViewController 114
4. Mostrando datos en tablas 118
5. Editando datos de tablas 121
5.1 Borrado de elementos 124
5.2 Modificación en el orden de los elementos 125
5.3 Añadir nuevos elementos 126

UNIDAD 7. ARRAYS 129
1. Introducción 129
1.1 Estructura de la unidad didáctica 130
2. Arrays mutables e inmutables: Jerarquía de clases 130
2.1 Arrays mutables 130
2.2 Arrays inmutables 131
3. Controladores de arrays 133

UNIDAD 8. DICCIONARIOS ... 134
1. Introducción ... 134
1.1 Estructura de la unidad didáctica ... 135
2. Diccionarios mutables e inmutables. Jerarquía de clases ... 135
2.1 Preferencias de usuario (NSUserDefaults) ... 136
3. Clases Singleton ... 140
4. Accediendo a la cámara del iPhone-iPad ... 147
5. Conexión con las vistas ... 152

UNIDAD 9. PERSISTENCIA DE DATOS ... 157
1. Introducción ... 157
1.1 Estructura de la unidad didáctica ... 157
2. Mecanismos de persistencia de datos ... 157
2.1 Archiving ... 159
3. SQLite ... 164
4. Creación y gestión de una base de datos ... 165
5. Creación y gestión del árbol ... 166

UNIDAD 10. CORE DATA ... 168
1. Introducción ... 168
1.1 Estructura de la unidad didáctica ... 168
2. Creación y gestión del modelo ... 169
2.1 Agregar un Framework al proyecto ... 169
2.2 Creación del fichero del modelo ... 171
2.3 Gestión del fichero del modelo ... 173
3. Controladores ... 176
4. Enlace de datos ... 180
4.1 Consulta y obtención de datos ... 183
4.2 Añadir datos ... 185
4.3 Eliminar datos ... 185
5. Comparación de SQLite ... 186

IMDb
RUNNER'S
WORLD

UNIDAD 1. Introducción a Objetive-C

1. Introducción

Actualmente existen una gran cantidad de lenguajes de programación en el mercado que abarcan una gran cantidad de paradigmas de programación estructurada, orientada a objetos, etc). Entre ellos, uno de los que más se utiliza es el lenguaje de programación C, debido, entre cosas a la gran potencia que proporciona a la hora del acceso a bajo nivel del computador con relativa facilidad. Sin embargo presenta un problema y es que, desde su creación en 1972 por Dennis M. Ritchie en los laboratorios Bell, y, a pesar de su gran potencia, no se adapta a los paradigmas de programación que posteriormente han ido apareciendo.

Por esa razón, con el tiempo han ido surgiendo extensiones a dicho lenguaje que pretenden adaptarlo a las nuevas necesidades aparecidas en el mundo de la programación. Quizás la extensión más conocida y aceptada sea C++, diseñado por Bjarne Stroustrup con el objetivo de proporcionar orientación a objetos al lenguaje C.

No obstante, aunque C++ sea la extensión más aceptada y conocida, no es la única y mucho menos la primera. Brad Cox realizó una extensión a C, incorporando los conceptos de otro lenguaje de programación orientado a objetos (Smalltalk) y cuyo resultado fue Objective-C. Esta extensión, aunque menos conocida que C++, presenta ciertas ventajas sobre ella:

- Se trata de una pequeña extensión del lenguaje C, por lo que, poseyendo el conocimiento del primero, resulta bastante sencillo (una vez que se comprenden los conceptos sobre la orientación a objetos), el aprendizaje del mismo.
- No se trata de un lenguaje de programación propietario. Esto quiere decir que se trata de un estándar abierto que se ha incluido en el compilador gcc (versión estándar del compilador de C), de forma que resulta fácilmente comprensible por este último.

No obstante, aunque Objective-C ha sido una extensión anterior a C++, no ha sido hasta hace relativamente pocos años, donde se ha popularizado su uso. La responsabilidad de este hecho recae en Apple, ya que, con la rápida adopción de sus dispositivos en el mercado, ha “obligado” a los desarrolladores ha rescatar Objetive-C, ya que, el entorno oficial de programación para estos dispositivos (Xcode) únicamente acepta dicho lenguaje, y los frameworks de desarrollo, también se encuentran programados en él.

Por tanto, si se desea programar para los dispositivos de Apple, como iPhone o iPad resulta muy conveniente la utilización del entorno oficial de programación (Xcode) y por lo tanto el aprendizaje de Objective-C. No obstante, estas no son las únicas alternativas disponibles para el desarrollo para estos dispositivos, existen otros lenguajes de programación y entornos, pero presentan el problema de que, aunque soportados por Apple, no son oficiales, y pueden presentar problemas de compatibilidad.

A lo largo de este tema, se va a asumir que el alumno ya tiene conocimiento de C y de programación orientada a objetos, y va a centrarse en la presentación de Objective-C y de las extensiones que aporta sobre el lenguaje en el que si apoya: C.

2. Clases y objetos en Objective-C

Como se ha dicho anteriormente, Objective-C fue creado con la intención de adaptar C a un paradigma de programación orientada a objetos, el cual se apoya básicamente en dos conceptos clave: clases y objetos.

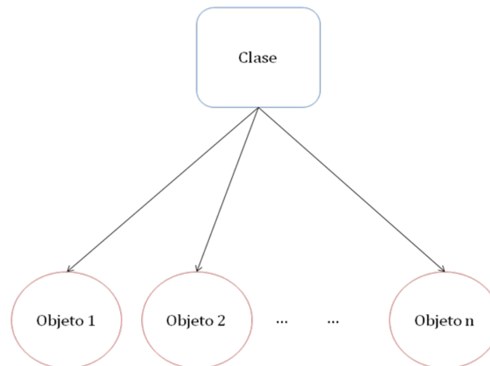

Figura 1. Representación de una clase y sus objetos

Las clases son las "plantillas" que van a permitir la creación (en adelante se denominará instanciación) de objetos. En Objetive-C van a disponer de dos tipos de ficheros cuando se declara[1] una clase:

- **Fichero de cabecera (header file)**: este tipo de fichero se identifica porque posee la extensión .h y lo que va a contener es una descripción (interfaz), de lo que va a contener la clase.
- **Fichero de implementación (implementation file)**: este tipo de fichero se identifica porque posee la extensión .m y contiene el detalle de la clase, es decir, que desarrolla el contenido de lo que se describe en el fichero de cabecera.

Hay que tener en cuenta además que estos ficheros no pueden estar separados, es decir, una clase siempre va a tener asociada estos dos ficheros, no puede existir uno sin el otro.

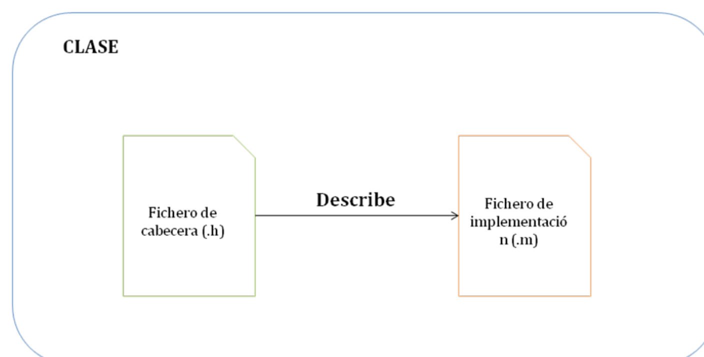

Figura 2. Relación entre los ficheros de cabecera e implementación de una clase en Objective-C

---

[1] Esta forma de organizar los ficheros no es algo que aporte Objective-C, ya que el propio lenguaje C trabaja utilizando esta dualidad de ficheros: el de interfaz y el de implementación. Sin embargo, lo que en este caso se aporta es la introducción deL concepto de clase y su organización.

2.1 Creación de clases en Objective-C

Como se ha dicho anteriormente, las clases en Objective-C, van a tener asociadas dos ficheros: el de cabecera y el de implementación. Por tanto, a la hora de crear una clase, se deben llevar a cabo las siguientes acciones:

- 1. Declaración de la clase en el fichero de cabecera. Esto se hace mediante las palabras reservadas[2] @interface y @end de la siguiente forma:

    @interface <nombre_de_la_clase>
    …
    @end

- 2. Implementación de la clase en el fichero de implementación. Esto se consigue mediante las palabras reservadas @implementation y @end de la siguiente forma:

    @implementation <nombre_de_la_clase>
    …
    @end

2.2 Comunicación entre objetos: paso de mensajes

En orientación a objetos, tanto las clases como los objetos, no son entidades aisladas, sino que se comunican unas con otras. Esta comunicación se realiza a través del intercambio de mensajes. Estos mensajes, se definen en las clases y se denominan métodos.

En Objective-C el envío de mensajes posee tres elementos encerrados entre un par de corchetes:

- **Receptor**: se trata del objeto o de la clase que va a recibir el mensaje. Este elemento es obligatorio para el envío del mensaje.
- **Selector**: se trata del nombre de método que ha de ejecutarse. Este elemento es obligatorio para el envío del mensaje.
- **Argumentos**: se tratan de los valores suministrados como parámetros al método. Estos elementos no son de uso obligatorio para el envío de un mensaje ya que puede existir mensajes que no tengan ningún tipo de parámetros.

En definitiva, la estructura que posee el envío de mensajes en Objective-C es la siguiente:

[Receptor Selector Argumentos]

2.2.1 Creación o instanciación de un objeto

Un objeto es una instancia de una clase. Para poder usar dicha instancia, Objective-C utiliza una variable que apunta al mismo. Por tanto, una variable que referencia a un objeto, no es más que un puntero a la zona de memoria donde se encuentra.

Por consiguiente, lo primero que hay que hacer para poder instanciar un objeto es declarar una

[2] Las palabras reservadas de un lenguaje de programación son las palabras propias del mismo. Objective-C añade un conjunto de palabras reservadas a las ya existentes de C, diferenciándose entre sí porque las pertenecientes a Objective-C se les antepone el carácter @.

variable que sea del tipo de la clase y que sea un puntero que apunte a la zona de memoria en la que se encuentra el objeto. Esto, en Objective-C se hace de la siguiente manera:

*Clase *[3]nombre_de_la_variable*

Por ejemplo, si se desea crear un objeto de tipo NSMutableArray, lo primero que hay que hacer es declarar un puntero de ese tipo de la siguiente forma:

*NSMutableArray *arr;*

Aunque ya se tiene el puntero a la zona de memoria que va a contener el objeto, la operación anterior realmente no apunta a nada. Hay que decir a Objective-C que reserve la zona de memoria para el objeto.

Como se ha visto antes, toda comunicación que se desee realizar con los objetos o las clases debe realizarse a través del paso de mensajes, y por tanto la instanciación de un objeto no es una excepción. Ahora lo que se desea hacer es decirle a Objective-C: "Reserva memoria para el objeto de esta clase". En Objective-C, esto se haría de la siguiente forma:

[NSMutableArray alloc]

Como puede verse, el receptor del mensaje en este caso sería la clase NSMutableArray y el selector (nombre del mensaje), sería alloc. En este caso no existen argumentos, por lo que no aparecen en el mensaje.

Con el mensaje anterior, lo que se ha conseguido es reservar la cantidad de memoria necesaria para almacenar un objeto de tipo NSMutableArray. Sin embargo, el objeto todavía no puede usarse ya que, aunque se dispone de la memoria necesaria para el mismo, las variables del objeto no se encuentran inicializadas. Para ello es necesario volver a enviar un nuevo mensaje al objeto de forma que se inicien todas las variables necesarias y que se quede listo para poder ser usado. El mensaje que ha de enviarse ahora es init quedando el proceso de instanciación de la siguiente forma:

*NSMutableArray *arr;*
arr = [NSMutableArray alloc];
[arr init];

Las tres líneas anteriores pueden resumirse en una sola de la siguiente forma:

*NSMutableArray *arr = [[NSMutableArray alloc] init];*

Como se puede observar, en una misma línea se ha declarado un puntero al objeto de tipo NSMutableArray (denominado arr), se ha reservado memoria para el mismo (enviando a la clase el mensaje alloc) y se ha inicializado con los valores por defecto (enviando el mensaje init).

Resumiendo, en la creación de un objeto en Objective-C, se llevan a cabo tres pasos:

1. Creación de una variable de tipo puntero cuyo tipo es el nombre de la clase y que va a apuntar a la zona de memoria donde se almacena el objeto.

[3] El * antes del nombre de la variable indica que se trata de un puntero a una zona de memoria.

2. Reserva de la zona de memoria para el objeto.
3. Inicialización del objeto.

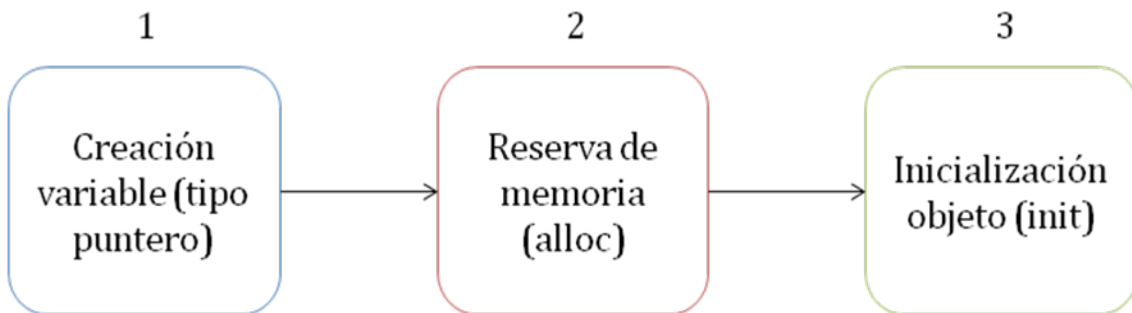

Figura 3. Pasos en la creación de un objeto

2.2.2 Paso de argumentos en los mensajes

Hasta ahora, en los ejemplos mostrados de intercambio de mensajes, estos únicamente estaban compuesto del Receptor y del Selector, pero no enviaban al receptor ningún tipo de argumento. Este paso de argumentos es un poco diferente de otros lenguajes de programación como C++ o Java. Por ejemplo, en C++ un mensaje con parámetros podría escribirse de la siguiente forma:

x.intersectsArcs(25.0, 9.0, 33.0, 80.0, 110.0)

Aquí x, sería el receptor; intersectsArcs sería el selector, y los cuatro datos numéricos que se encuentran dentro de los paréntesis serían los argumentos que se envían en el mensaje. Sin embargo, esta forma de enviar los mensajes (por otro lado muy difundida en los lenguajes de programación orientados a objetos) posee un gran problema: a simple vista no es posible distinguir a que se refieren cada uno de los valores que se envían en el mensaje.

Por ello, la forma de pasar de enviar los parámetros en Objective-C es un poco diferente. Así el mensaje anterior tendría la siguiente sintaxis:

[x intersectsArcWithRadius:25.0
centeredAtX: 9.0
Y: 33.0
fromAngle:80.0
toAngle:110]

Ahora, x sigue siendo el receptor, intersectsArcWithRadius:centerdAtX:Y:fromAngle:toAngle sería en este caso el selector y los valores numéricos serían los parámentros o argumentos del mensaje. De esta forma, queda perfectamente claro a que se refieren cada uno de los argumentos que constituyen el mensaje.

2.2.3 Destrucción de objetos

Los objetos son entidades que ocupan memoria , y por tanto, aquellos que no se utilicen deben ser destruidos ya que los dispositivos para los que se va a programar poseen una cantidad de memoria limitada que deben compartir una gran cantidad de aplicaciones.

Como se ha visto anteriormente, los objetos van a encontrarse almacenados en una dirección de memoria a la que se va a apuntar mediante una variable de tipo puntero.

Por tanto, para destruir un objeto bastaría con hacer que el puntero que apunta a la dirección de memoria que lo contiene apuntara hacia otro sitio. El "sitio" al que debe apuntar un objeto que se desea destruir es *nil*. Por tanto si se quiere destruir el objeto al que apunta *arr de los ejemplos anteriores, bastaría con hacer lo siguiente:

**arr = nil;*

Lo que realmente indica nil al puntero es que no debe apuntar hacia ningún sitio, destruyendo así la referencia que tenía al objeto previo.

No obstante, en la unidad 4, se va a revisar la gestión de memoria de iOS y se volverá sobre el tema de la escasez de memoria y la creación y destrucción de objetos. Por el momento, lo único que debe tenerse claro es el uso de nil para indicar a un puntero a un objeto que rompa la referencia al mismo.

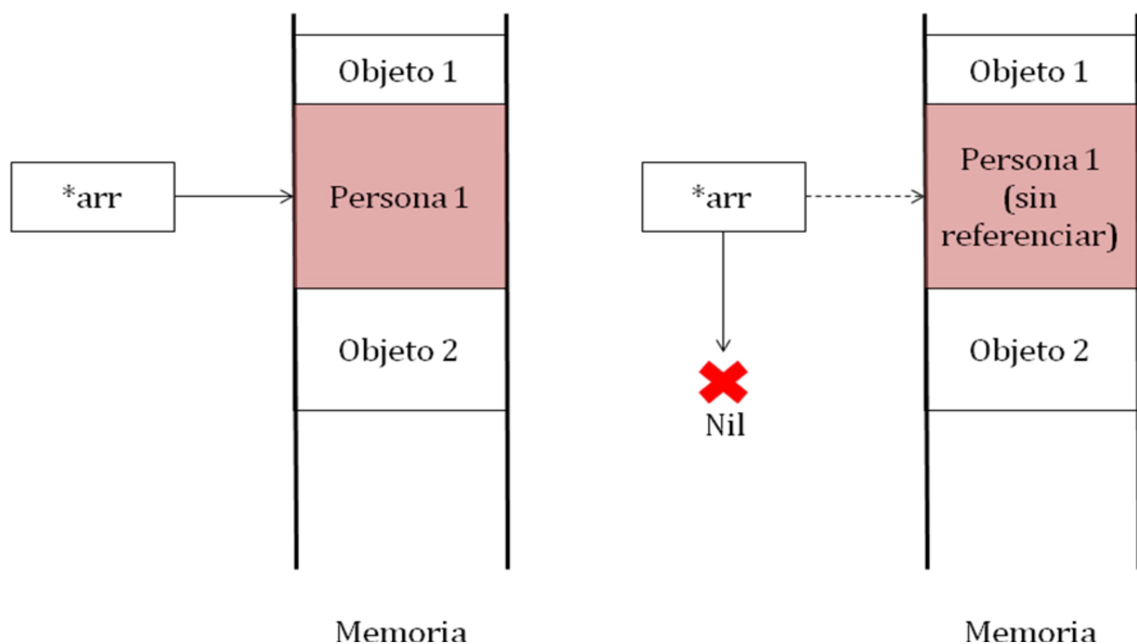

Figura 4. Destrucción de un objeto

2.3 Herencia en Objective-C

En orientación a objetos, el concepto de herencia hace referencia a la relación que existe entre dos o más clases, de forma que una o varias de ellas (a las que se denominan subclases) heredan las características de una o varias clases (a las que se denominan superclases).

Se trata de un concepto muy útil a la hora de trabajar con clases, y se encuentra muy extendido en los lenguajes de programación que trabajan con el paradigma de orientación a objetos como C++ o Java.

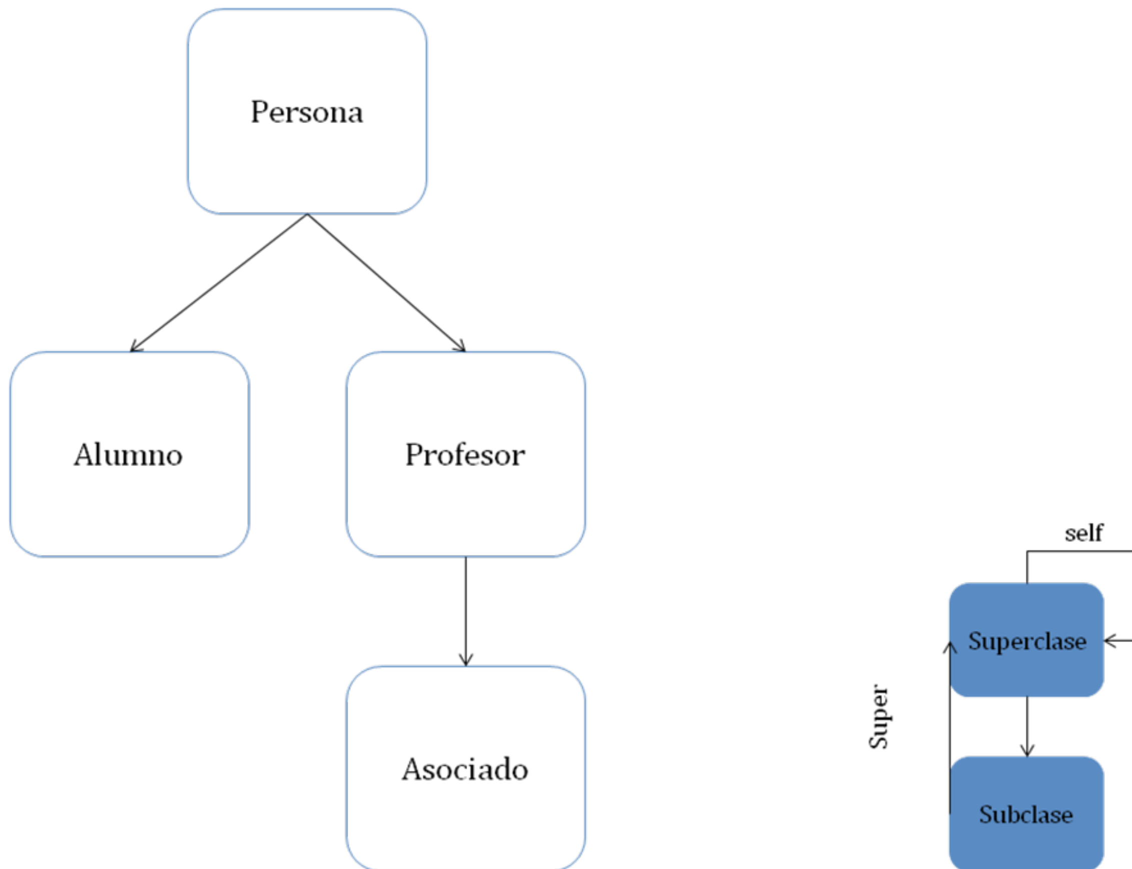

Figura 5. Ejemplo de herencia y palabras reservadas en Objective-C

En Objective-C, una clase puede heredar de una única clase, posponiendo a la definición de la misma dos puntos seguidos del nombre de la clase de la que hereda de la siguiente forma:

@interface <nombre_de_la_clase> : <nombre_de_la_superclase>

@end

Decir que obviamente la declaración debe realizarse en el fichero de cabecera que se encuentra asociado a la clase.

Por ejemplo, en Objective-C, existe una clase de la que heredan todas las demás, llamada NSObject. Si se quisiera definir una clase que heredase directamente de NSObject, se haría de la siguiente forma:

@interface miClase: NSObject

@end

Debido a la existencia de la herencia, es posible que una clase quiera referirse a un determinado método de la superclase. Cuando se de esta situación, la forma de que una subclase se refiera a la superclase inmediatamente superior es mediante la palabra reservada *super*. Por otro lado, si dentro de una clase hay que referirse a ella misma, se utilizará la palabra reservada *self*. El uso

de esta última forma quedará ejemplificado cuando más adelante se vean los inicializadores.

3. Variables de instancia

A la hora de trabajar con clases, debe existir alguna forma de almacenar las características de la misma. Por ejemplo, si se tiene una clase *Persona*, sería conveniente que la misma mantuviese información sobre el color de ojos, el color de pelo, si es hombre o mujer o el número de hermanos que tenga. Toda esta información se almacena en una serie de variables que se definen dentro de la clase y que se denominan variables de instancia.

Por tanto, es en el fichero de cabecera asociado a la clase, donde se han de definir las variables que van a contener información característica de la misma. Siguiendo con el anterior ejemplo sobre la clase persona, el fichero de cabecera quedaría como sigue:

```
@interface Persona
{
	NSString *colorOjos;
	NSString *colorPelo;
	NSString *sexo;
	int numeroHermanos;
}
@end
```

Como se puede observar las variables que aportan información sobre la característica de la clase, se encuentran definidas entre las palabras @interface y @end de la declaración de la clase entre paréntesis. Su definición se hace anteponiendo el tipo de la variable al nombre de la misma.

3.1 Tipos de datos en Objective-C

Al ser Objective-C una extensión del lenguaje C, posee los mismos tipos de datos que este, es decir, siguen siendo válidos tipos de datos como int o float para declarar datos numéricos o char para declarar caracteres. No obstante, se han añadido además los siguientes tipos:

- **id**: se trata de un puntero a cualquier tipo de objetos.
- **BOOL**: es lo mismo que el tipo de datos char pero con la diferencia de que se usa como un valor booleano.
- **YES**: es lo mismo que 1.
- **NO**: es lo mismo que 0.
- **nil**: es lo mismo que NULL, pero solo se usa con punteros a objetos.
- **IBOutlet**: se trata de una macro que se evalúa como nada. Se usa únicamente en la declaración de variables de instancia dentro del fichero de cabecera.
- **IBAction**: se trata de una macro que se evalúa como nada. Tendría el mismo significado que void y solo se usa en la declaración de métodos.

Además de estos tipos de datos, también puede consideran como tipos de datos para la creación de objetos, los nombres de las clases creadas y existentes dentro del lenguaje. Así por tanto, en el ejemplo que se ha presentado anteriormente, se utiliza el tipo NSString para referirse a la clase estándar para el tratamiento de cadenas de texto.

Por tanto, en el ejemplo anterior, existen dos tipos de variables de instancia: aquellas

que contienen tipos de datos estándar del lenguaje, por ejemplo la variable numeroHermanos es de tipo int; aquellas que son instancias de otras clases, por ejemplo las variables colorPelo, colorOjos y sexo son punteros a zonas de memoria que contienen objetos de la clase NSString.

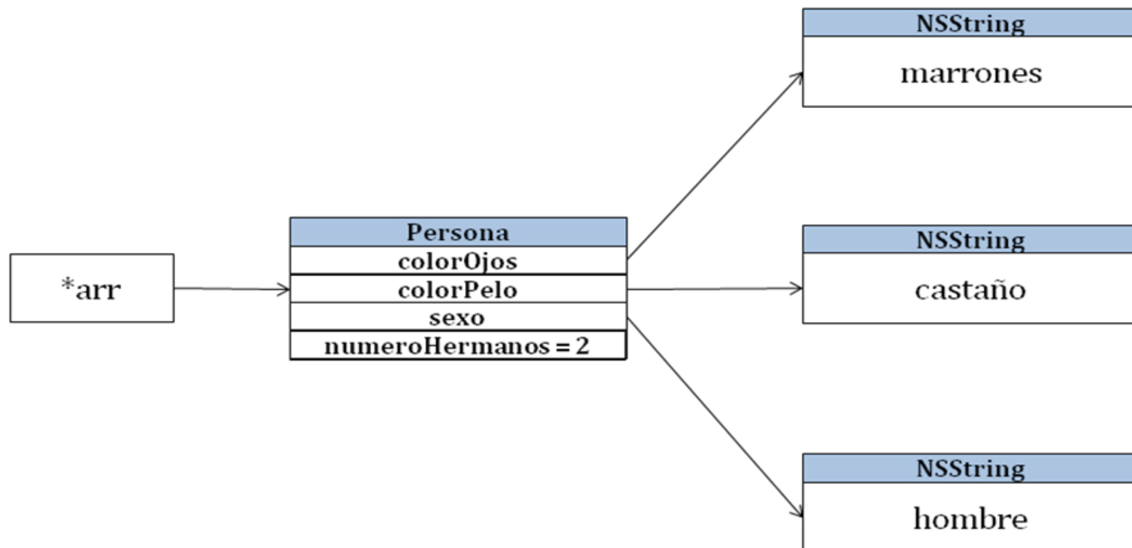

Figura 6. Referencias entre objetos

4. Métodos: de instancia y de clase

Anteriormente, se ha dicho que tanto los objetos como las clases se comunican mediante el paso de mensajes entre sí. Estos mensajes, implementados dentro de las clases reciben en nombre de métodos. Cuando en Objective-C se desee incluir un nuevo método en una clase, se han de llevar a cabo dos acciones:

1. Se ha de definir el método dentro del fichero de cabecera. Esta definición debe incluir los siguientes elementos del método: el nombre del método, los parámetros o argumentos que se le pasan, y el tipo de datos que retorna. Por ejemplo, retomando la siguiente clase Persona que se definió anteriormente, se le desea añadir un método llamado *description* que no tengo ningún parámetro y que retorne como un objeto de tipo *NSString* (en unidades posteriores se describirá con mayor detalle dicha clase, ahora mismo basta que saber que es una clase que trata con cadenas de caracteres). La definición del fichero de cabecera para la clase quedaría de la siguiente forma:

```
@interface Persona
{
        NSString *colorOjos;
        NSString *colorPelo;
        NSString *sexo;
        int numeroHermanos;
}
- (NSString *)description;
@end
```

Si también se quiere añadir un método llamado *cambiarColorOjosPelo* en el que no se retorne ningún valor y cuyos argumentos sean dos objetos de tipo *NSString* que representen el nuevo color de ojos y de pelo, el fichero de cabecera para la definición de la clase que daría ahora de la siguiente forma (en negrita se resalta el nuevo método que se ha incluido):

```
@interface Persona
{
        NSString *colorOjos;
        NSString *colorPelo;
        NSString *sexo;
        int numeroHermanos;
}
        - (NSString *)description;
        - (void)cambiarColorOjosPelo:(NSString *)cOjos pelo(NSString)cPelo;
@end
```

Como regla general, la definición de un método posee los siguientes elementos:

(<tipo_datos_retorno>)<nombre_del_método>:(<tipo_datos_parámetro1>)<nombre_parámetro1> …

2. Una vez que realizada la definición del método, este debe implementarse dentro del fichero de implementación entre las directivas *@implementation* y *@end*, que marcan el inicio y el final de la implementación de la clase. Así el fichero de implementación correspondiente a la clase Persona junto con los dos métodos que se han incluido en el punto anterior quedarían de la siguiente forma:


```
@implementation Persona
{
        - (NSString *)description
        {
…
}
        - (void)cambiarColorOjosPelo:(NSString *)cOjos pelo:(NSString *)cPelo
        {
…
}
}
@end
```

4.1 Métodos de instancia y de clase

A la hora de implementar un método, hay que decidir si cuando este se ejecute va a ir dirigido a una instancia (*métodos de instancia*) de la o bien a la propia clase (*métodos de clase).* De esta forma, los métodos de instancia actúan sobre una instancia en particular de la clase, de todas las posibles que pueda tener, mientras que los métodos de clase no operan sobre una instancia en particular ni tampoco tienen acceso a las variables de instancia que se hayan definido. Estos métodos de clase normalmente se utilizan para crear nuevas instancias de la clase o recibir ciertas propiedades de la misma.

Para poder diferenciar estos dos tipos de métodos hay que ver que carácter se les antepone en su definición. Si se les antepone el carácter –, el método es de instancia, mientras que si el carácter que se le antepone es +, el método es de clase. Por ejemplo, los dos métodos que se han añadido a la clase persona en el apartado anterior, son método de instancia, ya que se les ha antepuesto el carácter -.

4.2 Métodos de acceso

Según el paradigma de orientación a objetos, as variables de instancia que se han definido en una clase, deberían ser accedidas mediante métodos, es decir, aunque el lenguaje de programación permita el acceso directo a dichas variables, si se desean evitar problemas, se deberían incluir como mínimo dos métodos: uno para consultar el valor de la variable (método get) y otro para poder modificar su valor (método set). Estos métodos reciben el nombre de *métodos de acceso*.

Obsérvese el ejemplo de la clase Persona. Esta contiene cuatro variables de instancia, que deben poder leerse y modificarse. Por tanto, se deberían definir 8 métodos de acceso (4 métodos get y 4 métodos set correspondientes a cada una de las 4 variables de instancia). De esta forma, el fichero de cabecera quedaría de la siguiente forma:

```
@interface Persona
{
        NSString *colorOjos;
        NSString *colorPelo;
        NSString *sexo;
        int numeroHermanos;
}
        - (NSString *)description;
        - (void)cambiarColorOjosPelo:(NSString *)cOjos pelo:(NSString *)cPelo;

        - (NSString)colorOjos;
        - (void)setColorOjos:(NSString *)cOjos;
        - (NSString)colorPelo;
        - (void)setColorPelo:(NSString *)cPelo;
        - (NSString)sexo;
        - (void)setSexo:(NSString *)s;
        - (NSString)numeroHermanos;
        - (void)setNumeroHermanos:(int)n;

@end
```


Como puede apreciarse, los métodos get coinciden con el nombre de la variable de instancia y los métodos set se consiguen anteponiendo la palabra set al nombre de la variable de instancia correspondiente. A lo largo de este curso, se van a ver una serie de convenciones a la hora de nombrar los elementos. Este hecho no solo es para dar uniformidad al código, sino que el entorno que se utilizará tiene en cuenta dichas convenciones a la hora de trabajar. Por tanto, se recomienda seguir dichas reglas ya que de no ser así, pueden producirse errores a la hora de ejecutar los programas.

5. Inicializadores

Al inicio de esta unidad, cuando se presentaron las clases y los objetos, se explicó el proceso de creación de una instancia, por el cual se enviaba el mensaje alloc a la clase para reservar espacio en memoria y posteriormente, a la instancia se le enviaba en mensaje init con el fin de inicializar las variables de la misma. Este segundo mensaje que se le envía a la instancia, se denomina *inicializador* o *método de iniciación*.

Una clase puede tener más de un inicializador, de forma que con todos ellos se cubran todos los posibles escenarios de inicialización para la instancia de la clase. Por norma, todos los inicializadotes deben comenzar con la palabra init. Por tanto, siguiendo con el ejemplo de la clase Persona que se ha ido desarrollando en esta unidad, podrían presentarse por ejemplo dos inicializadores: uno en el que se le pase como argumento todos los valores correspondientes a las variables de instancia de la clase y otro en el que no se le pase ningún valor para dichas variables y estas tengan que iniciarse con valores por defecto.

Sin embargo, aunque una clase posea varios inicializadotes distintos, se debe elegir uno de ellos como principal, al que todos los demás deben llamar internamente. A este inicializador se le denomina *inicializador designado*. Siguiendo con el ejemplo de la clase Persona, se va a escoger el inicializador que acepta parámetros como designado. De esta forma, la implementación de dichos inicializadotes, quedaría como sigue en su fichero .m:

```
- (id)initConParametros:(NSString *)cOjos colorPelo:(NSString *)cPelo sexo:(NSString *)s
hermanos:(int)n
{
	//Se llama al inicializador designado de la superclase
	self = [super init];

	//Si no ocurre ningún problema en la inicialización, se asignan los valores //de los parámetros a las variables de instancia.
	if (self) {
		[self setColorOjos:cOjos];
		[self setColorPelo:cPelo];
		[self setSexo:s];
		[self setNumeroHermanos:n];
	}

	//Se retorna el propio objeto
	return self;
}

- (id)init
{
	//Se llama al inicializador designado de la clase
	return [self initConParametros:@””
		colorPelo:@””
		sexo:@””
		hermanos:0];
```


Como puede observarse, el inicializador designado (*initConParametros*) es el que, inicialmente llama al inicializador de su superclase y comprueba que no ha dado ningún error (esta operación es así por convención), para, posteriormente asignar los valores que se le han pasado como parámetro a las variables de instancia de la clase.

Por otro lado, el inicializador restante (*init*) simplemente llama al inicializador designado

con los valores por defecto para cada una de las variables de instancia de la clase.