



**El lenguaje
Objective-C
para
programadores
C++ y Java**

Acerca de este documento

Este tutorial está diseñado para que programadores procedentes de otros entornos descubran las ventajas que tiene utilizar un lenguaje orientado a objetos tan simple, potente y versátil como Objective-C, el lenguaje que eligió NeXT y luego Apple para desarrollar aplicaciones en su sistema operativo.

A lo largo de este tutorial comentaremos las diferencias que tiene Objective-C con C++ o Java, es decir, este tutorial no está pensado para personas que quieren empezar a programar usando Objective-C. Para bien o para mal, la experiencia ha demostrado que Objective-C no es un lenguaje que se suela escoger como primer lenguaje de programación, sino que son C++ y Java los lenguajes más elegidos por los recién llegados al mundo de la programación. Sin embargo, lenguajes como Objective-C son muchas veces seleccionados por programadores que ya conocen C++ o Java, con lo que esperamos que las precondiciones que fija este tutorial, más que desanimar, animen a muchos más programadores a usar este documento para introducirse en este apasionante mundo.

Al acabar este tutorial el lector debería de haber aprendido todos los detalles del lenguaje Objective-C necesarios para empezar a estudiar el Foundation Framework, que son el conjunto de clases comunes a Mac OS X y iPhone OS, Cocoa, la librería de programación orientada a objetos de Mac OS X, y Cocoa Touch, la librería de programación orientada a objetos de iPhone OS.

Nota legal

Este tutorial ha sido escrito por Fernando López Hernández para MacProgramadores, y de acuerdo a los derechos que le concede la legislación española e internacional el autor prohíbe la publicación de este documento en cualquier otro servidor web, así como su venta, o difusión en cualquier otro medio sin autorización previa.

Sin embargo el autor anima a todos los servidores web a colocar enlaces a este documento. El autor también anima a bajarse o imprimirse este tutorial a cualquier persona interesada en aprender el lenguaje Objective-C.

Madrid, Diciembre del 2008

Para cualquier aclaración contacte con:

fernando@DELITmacprogramadores.org

Tabla de contenido

TEMA 1: Empezando a programar con Objective-C

1. Introducción	10
2. Compilando con las GCC.....	11
2.1. Crear un ejecutable	11
2.2. Framework y runtime de Objective-C	12
2.3. Programar con el framework de clases de GNU	13
2.4. Programar con el framework de clases de NeXTSTEP	15
2.5. Crear una librería estática o dinámica.....	16
3. Compilando con Xcode	17
3.1. Crear un programa	17
3.2. Crear una librería de enlace estático	20
3.3. Crear una librería de enlace dinámico.....	22
3.4. Crear un framework.....	26

TEMA 2: Características del lenguaje

1. Qué es Objective-C	32
2. Lenguaje marcadamente dinámico	33
3. Asociación, agregación y conexiones	36
4. Componentes vs. frameworks	37

TEMA 3: Objetos y clases

1. Clases	39
1.1. La interfaz.....	39
1.2. Implementación	41
2. Objetos	43
2.1. Instanciar objetos.....	43
2.2. Tipos estáticos y dinámicos	44
3. Variables de instancia.....	46
4. Métodos	47
4.1. Declaración de un método.....	47
4.2. Implementación de un método	49
4.3. Name mangling	50
4.4. Ejecutar un método	51
4.5. Número variable de parámetros.....	52
5. Encapsulación.....	54
6. Clases como estructuras de datos	55
6.1. Clases sin clase base.....	55
6.2. Paso de objetos por valor.....	56
7. Objetos cadena.....	57
7.1. Crear y manipular objetos cadena.....	57
7.2. Formatear cadenas	60

7.3. Leer y escribir cadenas de ficheros y URLs	62
8. Tipos de datos de 32 y 64 bits	64
9. Declaraciones adelantadas.....	66

TEMA 4: Profundizando en el lenguaje

1. Herencia	70
1.1. La clase raíz	70
1.2. Redefinir métodos y variables de instancia	70
1.3. Los receptores especiales <code>self</code> y <code>super</code>	71
2. Objetos clase.....	73
2.1. Los objetos clase	73
2.2. La variable de instancia <code>isa</code>	75
2.3. Crear instancias de una clase	76
2.4. Personalización con objetos clase	76
2.5. Introspección	77
2.6. Variables de clase.....	77
2.7. Inicializar un objeto clase	78
3. Otros receptores especiales	79
4. Ruta de un mensaje durante su ejecución.....	80
5. Objetos metaclase	82
5.1. Objetos de instancia, de clase, y de metaclase.....	82
5.2. Obtener la metaclase de una clase.....	83
5.3. La variable de instancia <code>super_class</code> en los objetos clase y metaclase	84
5.4. Métodos de la clase raíz en los objetos clase	84
6. Ciclo de vida de un objeto	86
6.1. Creación e inicialización	86
6.2. Implementar la inicialización.....	92
6.3. Desinicialización y liberación.....	96
7. Categorías.....	97
7.1. Qué son las categorías	97
7.2. Declarar la interfaz de una categoría	98
7.3. Implementación de una categoría	99
7.4. Sobrescribir métodos con categorías	101
7.5. Categorías en la clase raíz	101
8. Protocolos	102
8.1. Declarar un protocolo	103
8.2. Adoptar un protocolo	104
8.3. Tipificación estática de prototipo.....	106
8.4. Jerarquía de protocolos.....	108
8.5. El protocolo <code>NSObject</code>	109
8.6. Objetos protocolo	109
8.7. Declaración adelantada de protocolos	110
8.8. Protocolos informales	111
8.9. Proxies y delegados	112
8.10. Protocolos formales con métodos opcionales	113

9. Extensiones	114
10. Clases abstractas	116
11. Cluster de clases	117

TEMA 5: El runtime de Objective-C

1. Interactuar con el runtime de Objective-C.....	120
2. El sistema de paso de mensajes.....	121
2.1. Los selectores	121
2.2. Ejecutar métodos a través de selectores.....	122
2.3. El patrón de diseño target-action	123
2.4. Evitar errores en el envío de mensajes	124
2.5. Parámetros implícitos.....	125
2.6. Cómo se envían los mensajes	126
3. Gestión de memoria	128
3.1. Técnicas de gestión de memoria.....	128
3.2. Mantener la cuenta de referencias de un objeto.....	129
3.3. Política de gestión de la cuenta de referencias	131
3.4. Retornar un objeto	132
3.5. Recibir un objeto de un ámbito superior	132
3.6. Métodos factory.....	133
3.7. Métodos setter	133
3.8. Retenciones cíclicas	134
3.9. Referencias débiles	135
3.10. Validez de los objetos compartidos	136
3.11. Autorelease pools	137
4. Las clases raíz	141
4.1. Creación, copia y liberación de objetos.....	141
4.2. Identificar objetos y clases	143
4.3. Introspección de jerarquía y protocolos	145
4.4. Introspección de métodos	146
5. Copia de objetos	147
5.1. Copia de objetos <code>Object</code>	148
5.2. Copia de objetos <code>NSObject</code>	149
5.3. Método getter/setter y liberación de objetos agregados.....	150
6. Gestión de excepciones	151
6.1. El bloque <code>@try-@catch</code>	151
6.2. Lanzar excepciones con <code>@throw</code>	152
6.3. Usar varios bloques <code>@catch</code>	153
6.4. El bloque <code>@finally</code>	154
6.5. Excepciones y errores	155
6.6. El handler de excepciones no capturadas	157
6.7. El handle de excepciones por defecto.....	158
7. Bloques sincronizados	160

TEMA 6: Objective-C 2.0

1. Las propiedades.....	163
1.1. Declarar propiedades	163
1.2. Implementar propiedades	164
1.3. Acceso a propiedades	165
1.4. Modificadores de la propiedad	165
1.5. Personalizar la implementación	168
1.6. El operador punto.....	171
1.7. Redefinir modificadores de propiedad	175
1.8. Diferencias en el runtime	176
2. El recolector de basura	178
2.1. Activar el recolector de basura.....	178
2.2. Técnicas de recolección de basura	179
2.3. La librería auto	183
2.4. Recolector de memoria con tipos fundamentales.....	184
2.5. Gestión de memoria de objetos puente	184
2.6. Clases para recolección de basura.....	186
2.7. Finalizar un objeto	188
2.8. Ventajas e inconvenientes.....	189
2.9. Trazar la recolección de basura	190

TEMA 7: Las objetos de colección

1. Objetos arrays	193
1.1. Objetos arrays inmutables.....	193
1.2. Objetos array mutables	196
1.3. Rangos y concatenaciones	197
1.4. Copia de objetos array	197
1.5. Objetos array de punteros.....	198
2. Objetos conjunto	199
2.1. Objetos conjunto inmutables	200
2.2. Objetos conjunto mutables.....	201
2.3. Operaciones con conjuntos.....	202
2.4. Objetos conjunto con repeticiones	202
3. Recorrer los elementos de una colección	203
3.1. Enumeraciones.....	203
3.2. Enumeraciones rápidas	204
3.3. Ejecutar un selector	205
3.4. Ordenar elementos	205
3.5. Filtrar elementos	207
4. Objetos diccionario	208
4.1. Objetos diccionario inmutables	208
4.2. Objetos diccionario mutables.....	211
4.3. Objetos diccionario de punteros.....	211
5. Tipos fundamentales en colecciones.....	212

TEMA 8: Key-Value Coding

1. Qué es KVC	214
1.1. Tecnologías relacionadas.....	214
1.2. Terminología	215
2. Métodos de acceso.....	216
2.1. Lecturas simples.....	216
2.2. Escrituras simples	220
2.3. Soporte para escalares y estructuras.....	221
2.4. Lectura de propiedades uno a muchos	221
2.5. Escritura de propiedades uno a muchos	223
3. Métodos de validación	225
3.1. Validar una propiedad	225
3.2. Métodos de patrón para validación.....	226
4. Operadores en caminos de claves	228
5. Describir las propiedades.....	229

TEMA 9: Key-Value Observing

1. Qué es KVO.....	231
2. Registrar observadores.....	231
2.1. Registrar el objeto observador	232
2.2. Recibir notificaciones de cambio	233
2.3. Eliminar un objeto observador	234
3. Notificaciones automáticas y manuales	234
3.1. Cóctel de punteros isa.....	234
3.2. Notificaciones automáticas	235
3.3. Notificaciones manuales	235
3.4. Desactivar la notificación automática.....	236
3.5. Registrar propiedades dependientes.....	237

TEMA 10: Aprovechando toda la potencia del lenguaje

1. Directivas del preprocesador y compilador	241
1.1. Directivas del preprocesador	241
1.2. Directivas del compilador	241
2. Zonas de memoria	247
2.1. Creación y gestión de zonas de memoria.....	247
2.2. Reservar y liberar memoria en una zona	248
3. Forwarding	249
3.1. Forwarding con Object	249
3.2. Forwarding con NSObject	250
3.3. Delegados y herencia.....	254
3.4. Posing	254
4. Mensajes remotos	256
4.1. Programación distribuida.....	256
4.2. Modificadores de tipo.....	258

4.3. Mensajes síncronos y asíncronos.....	259
4.4. Paso de punteros.....	259
4.5. Paso de objetos.....	263
5. Tipos de datos y constantes predefinidas.....	264
5.1. Tipos de datos predefinidos.....	264
5.2. Constantes predefinidas	265
6. Optimización del acceso a métodos	266
7. Estilos de codificación.....	267
7.1. Clases, categorías y protocolos formales	268
7.2. Prefijos	268
7.3. Métodos y variables de instancia.....	269
7.4. Funciones, variables globales y constantes	270
7.5. Variables locales	270
8. Objective-C++.....	270

Tema 1

Empezando a programar con Objective-C

Sinopsis:

Este primer tema de naturaleza introductoria pretende fijar conceptos fundamentales, y describir cómo se usan las principales herramientas que sirven para programar en Objective-C.

Antes de que el Tema 2 se adentre en los conceptos del lenguaje, creemos que puede ser interesante empezar describiendo cómo manejar las herramientas de programación para hacer un programa sencillo con Objective-C. Para ello en este primer tema vamos a empezar mostrando cómo se compilan y enlazan aplicaciones.

Primero veremos cómo compilar aplicaciones Objective-C desde la consola usando las herramientas de programación de GNU, para en la segunda parte de este tema ver cómo, usando la herramienta gráfica Xcode, podemos mejorar la productividad del programador.

1. Introducción

Cocoa, la API de programación orientada a objetos de Mac OS X tradicionalmente se ha dividido en dos grandes grupos de clases:

- **Foundation Framework:** Son un conjunto de clases de apoyo que permiten representar estructuras de datos complejas (arrays, listas, diccionarios, ...) y dentro de las que no se incluyen las clases relacionadas con la interfaz gráfica. Esta librería es común a Mac OS X y iPhone OS.
- **Application Kit Framework (AppKit Framework):** Aquí se incluyen casi todas las demás clases. Como puedan ser clases relacionadas con la interfaz gráfica, con la impresión, el acceso a audio y vídeo, el acceso a red, y todos los demás aspectos que podemos gestionar desde Cocoa. Dentro del AppKit encontramos otros Kit de desarrollo especializados en determinadas tareas como puedan ser: Image Kit, QuickTime Kit, Apple Scripting Kit.

Ambos frameworks son accesibles desde Java, Python, Ruby y Objective-C, aunque es Objective-C el lenguaje con el que vamos a poder sacar el máximo rendimiento a estos frameworks.

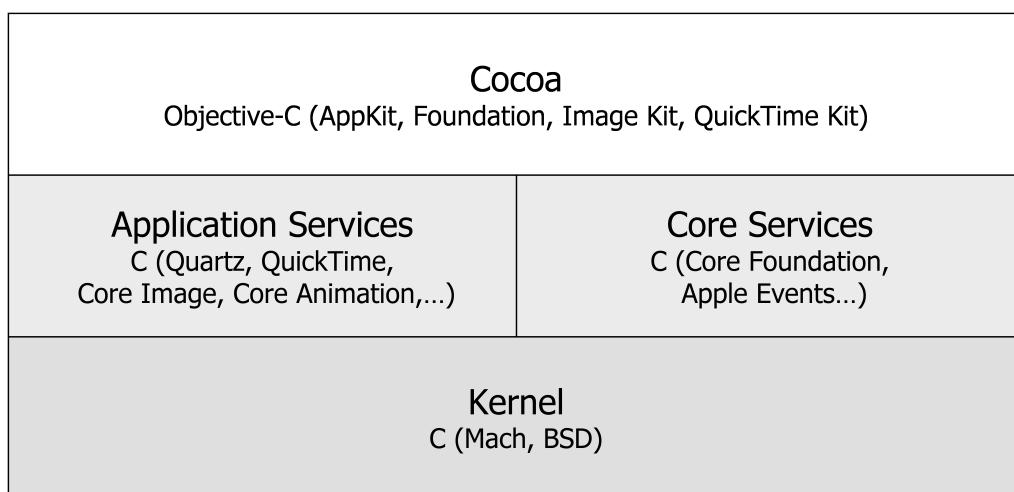


Figura 1.1: Capas software de programación en Mac OS X

La Figura 1.1 muestra un resumen de las capas software de programación del sistema operativo Mac OS X. Vemos que la principal característica de las librerías que se agrupan bajo el nombre de Cocoa es que son accesibles desde Objective-C, aunque luego existen librerías C que proporcionan servicios de más bajo nivel. Por ejemplo, Core Foundation envuelve en funciones C las operaciones del Foundation Framework.

En este tutorial vamos a centrarnos en estudiar el lenguaje Objective-C. También estudiaremos algunas clases del Foundation Framework. Las clases del

Application Kit Framework se dejan para otros tutoriales que conviene leer una vez que usted domine el lenguaje.

2. Compilando con las GCC

Las GCC (GNU Compiler Collection)¹ son un conjunto de herramientas que proporciona GNU para programar en varios lenguajes y plataformas. Mac OS X y iPhone OS se apoyan en estas herramientas para realizar las tareas de compilación y enlazado de sus aplicaciones.

Para mejorar la productividad y facilidad de uso de estas herramientas Apple creó Xcode, un IDE (Integrated Development Environment) que permite realizar de forma gráfica las tareas comunes de programación, y que hace llamadas a las GCC para realizar las tareas de compilación y enlazado de forma transparente al programador.

En este apartado veremos cómo utilizar las herramientas de programación de GNU para compilar y enlazar aplicaciones Objective-C desde la consola. En el siguiente apartado veremos cómo se puede utilizar Xcode para mejorar la productividad del programador.

En este tema vamos a pasar por encima los detalles relativos al lenguaje, con lo que si no conoce Objective-C puede que muchos aspectos de lenguaje le resulten confusos. En el Tema 2 introduciremos el lenguaje a nivel conceptual, y en el Tema 3 empezaremos a detallar todos los aspectos del lenguaje con el nivel de detalle que se requiere para su correcta comprensión.

2.1. Crear un ejecutable

Debido a que Objective-C es una extensión al lenguaje C, un programa C compila en Objective-C sin necesidad de cambios. El Listado 1.1 muestra un programa básico Objective-C que sólo usa la sintaxis de C excepto, por dos aspectos:

El primer aspecto es que en vez de usar la directiva del preprocesador `#include` se ha usado la directiva del preprocesador `#import`. Ambas directivas realizan la operación de incluir un fichero dentro de otro fichero, pero a diferencia de `#include`, la directiva `#import` asegura que el fichero incluido no se incluya nunca más de una vez, con lo que en los ficheros de cabecera incluidos con esta directiva no es necesario hacer un control de inclusiones

¹ Si cree que todavía no conoce lo suficiente de estas herramientas, puede que le resulte interesante leer el tutorial "Compilar y depurar aplicaciones con las herramientas de programación de GNU" que también encontrará publicado en MacProgramadores.

con la directiva del preprocesador `#ifdef`, tal como se acostumbra a hacer en C y C++.

La otra diferencia está en que la extensión de los ficheros Objective-C es `.m`, extensión usada por el comando `gcc` para saber que se trata de un programa Objective-C.

```
/* holamundo.m */
#import <stdio.h>
#import <stdlib.h>

int main() {
    printf("Hola desde Objective-C\n");
    return EXIT_SUCCESS;
}
```

Listado 1.1: Programa Objective-C básico

Para compilar y ejecutar este programa bastaría con ejecutar los comandos:

```
$ gcc holamundo.m -o holamundo
$ ./holamundo
Hola desde Objective-C
```

2.2. Framework y runtime de Objective-C

Actualmente, para programar en Objective-C disponemos de dos frameworks distintos: El primer framework es el **framework de clases de GNU**, que son un conjunto de clases inicialmente desarrollado por NeXTSTEP para Objective-C que fueron abiertas bajo licencia GNU, y cuya clase base es la clase `Object`. El segundo es el **framework de clases de NeXTSTEP**, que es el conjunto de clases que desarrolló NeXTSTEP en 1994, cuya clase base es `NSObject`, y que actualmente es el usado por Mac OS X para implementar Cocoa. El código fuente del framework de clases de NeXTSTEP no está abierto.

Actualmente el framework de clases de GNU ha dejado de actualizarse y GNU también está haciendo una implementación de código fuente abierto del nuevo framework de clases de NeXTSTEP llamado llama GNUStrp. Esta implementación también utiliza la clase base `NSObject` así como el resto de clases del framework de NeXTSTEP, pero actualmente no está terminada.

Para usar el framework de clases de GNU debemos de enlazar con el fichero `libobjc.a` usando la opción del enlazador `-lobjc`. Por el contrario, para enlazar con el framework de clases de NeXTSTEP debemos de enlazar con el framework `Foundation.framework` usando la opción del enlazador `-`

framework Foundation. Lógicamente podemos enlazar con ambos frameworks de clases usando ambas opciones durante el enlazado.

Es importante no confundir los *frameworks de Objective-C*, que son librerías de clases, con el *runtime de Objective-C*, que es un conjunto de funciones de librería, escritas en C, en las que se basan las clases de Objective-C para alcanzar el potencial dinámico característico de este lenguaje: Por defecto en Mac OS X se usa el **runtime de NeXTSTEP**, el cual actualmente da soporte tanto al framework de clases de GNU, como al framework de clases de NeXTSTEP. Podemos pedir usar el **runtime de GNU** usando la opción del enlazador `-fobjc-runtime`, pero en este caso sólo tendremos acceso al framework de clases de GNU, y además deberemos enlazar con la librería `libobjc-gnu.a` usando la opción `-lobjc-gnu`.

2.3. Programar con el framework de clases de GNU

En este apartado vamos a ver cómo programar usando el framework de clases de GNU. En el siguiente apartado veremos las diferencias cuando usamos el framework de clases de NeXTSTEP.

Normalmente cada clase Objective-C consta de dos ficheros: Uno con la extensión `.h` que contiene la interfaz, y otro con la extensión `.m` que contiene la implementación. El Listado 1.2 y Listado 1.3 muestran respectivamente un ejemplo de interfaz e implementación de una clase Objective-C llamada Saludador. Observe que estamos usando como clase base la clase `Object` situada en el fichero de cabecera `<objc/Object.h>`.

```
/* Saludador.h */
#import <objc/Object.h>

@interface Saludador : Object {
    char* saludo;
}
- init;
- (void)setSaludo:(char*)unSaludo;
- (void)setSaludo:(char*)unSaludo y:(char*)unaColetilla;
- (void)saluda;
@end
```

Listado 1.2: Interfaz de una clase Objective-C con el framework de clases de GNU

```
/* Saludador.m */
#import "Saludador.h"
#import <stdio.h>
#import <stdlib.h>
#import <string.h>

@implementation Saludador
```

```

- init {
    if (self = [super init]) {
        saludo = "Hola mundo";
    }
    return self;
}
- (void)setSaludo:(char*)unSaludo {
    saludo = unSaludo;
}
- (void)setSaludo:(char*)unSaludo y:(char*)unaColetilla {
    saludo = malloc(strlen(unSaludo)+strlen(unaColetilla)+1);
    strcpy(saludo,unSaludo);
    strcat(saludo,unaColetilla);
}
- (void)saluda {
    printf("%s\n",saludo);
}
@end

```

Listado 1.3: Implementación de una clase Objective-C con el framework de clases de GNU

Una vez tengamos definida la clase, para instanciar y usar un objeto de esta clase, necesitamos un programa principal como el del Listado 1.4.

```

/* pidesaludo.m */
#import "Saludador.h"

int main() {
    Saludador* s = [[Saludador alloc] init];
    [s saluda];
    [s setSaludo: "Hola de nuevo"];
    [s saluda];
    [s setSaludo: "Hola buenos dias,"
                 y: "encantado de verle"];
    [s saluda];
    [s free];
    return EXIT_SUCCESS;
}

```

Listado 1.4: Programa que usa un objeto Objective-C del framework de clases de GNU

Al estar usando el framework de clases de GNU, el programa puede ser compilado y enlazado con los comandos:

```

$ gcc -c Saludador.m
$ gcc -c pidesaludo.m
$ gcc Saludador.o pidesaludo.o -lobjc -o pidesaludo

```

O bien realizar los tres pasos a la vez con el comando:

```
$ gcc pidesaludo.m Saludador.m -lobjc -o pidesaludo
```

2.4. Programar con el framework de clases de NeXTSTEP

El Listado 1.5, Listado 1.6 y Listado 1.7 muestran cómo implementar y usar la clase `Saludador` del apartado anterior enlazando con el framework de clases de NeXTSTEP. La principal diferencia está en que ahora derivamos de `NSObject` en vez de derivar de `Object`. La segunda diferencia está en que importamos el fichero `<Foundation/NSObject.h>` en vez de importar el fichero `<objc/Object.h>`.

Para compilar y ejecutar ahora el programa podemos usar los comandos:

```
$ gcc Saludador.m pidesaludo.m -framework Foundation -o
pidesaludo
$ ./pidesaludo
```

```
/* Saludador.h */
#import <Foundation/NSObject.h>

@interface Saludador : NSObject {
    char* saludo;
}
- init;
- (void)setSaludo:(char*)unSaludo;
- (void)setSaludo:(char*)unSaludo y:(char*)unaColetilla;
- (void)saluda;
@end
```

Listado 1.5: Interfaz de una clase Objective-C con el framework de clases de NeXTSTEP

```
/* Saludador.m */
#import "Saludador.h"
#import <stdio.h>
#import <stdlib.h>
#import <string.h>

@implementation Saludador
- init {
    if (self = [super init]) {
        saludo = "Hola mundo";
    }
    return self;
}
- (void)setSaludo:(char*)unSaludo {
    saludo = unSaludo;
}
- (void)setSaludo:(char*)unSaludo y:(char*)unaColetilla {
    saludo = malloc(strlen(unSaludo)+strlen(unaColetilla)+1);
    strcpy(saludo,unSaludo);
    strcat(saludo,unaColetilla);
}
- (void)saluda {
```

```

    printf("%s\n", saludo);
}
@end

```

Listado 1.6: Implementación de una clase Objective-C con el framework de NeXTSTEP

```

/* pidesaludo.m */
#import "Saludador.h"

int main() {
    Saludador* s = [[Saludador alloc] init];
    [s saluda];
    [s setSaludo: "Hola de nuevo"];
    [s saluda];
    [s setSaludo: "Hola buenos dias,"
                y: "encantado de verle"];
    [s saluda];
    [s release];
    return EXIT_SUCCESS;
}

```

Listado 1.7: Programa que usa un objeto Objective-C del framework de clases de NeXTSTEP

2.5. Crear una librería estática o dinámica

Al igual que con C o C++, con Objective-C también podemos crear una librería de enlace estático o dinámico, que luego usemos desde otros programas. Por ejemplo, para crear una librería de enlace estático con la clase `Saludador` del Listado 1.6 podemos generar el fichero de librería con el comando:

```
$ ar -r libsaludos.a Saludador.o
ar: creating archive libsaludos.a
```

En caso de estar programando con el framework de NeXTSTEP, podemos enlazar con él desde el programa del Listado 1.7 con el comando:

```
$ gcc pidesaludo.m libsaludos.a -framework Foundation -o pide
saludo
```

Si lo que queremos es crear una librería de enlace dinámico, podemos crear la librería de enlace dinámico, y enlazar el programa principal del Listado 1.7 con la librería, usando los comandos:

```
$ gcc -dynamiclib Saludador.o -framework Foundation -o libSalu
dos.dylib
$ gcc pidesaludo.m libSaludos.dylib -framework Foundation -o p
idesaludo
```

Análogamente podríamos haber creado una librería de enlace estático o dinámico que usase el framework de clases de GNU cambiando `NSObject` por `Object` y `-framework Foundation` por `-lobjc`.

3. Compilando con Xcode

Cuando creamos un nuevo proyecto con Xcode, usando la opción `File|New Project`, Xcode nos muestra un diálogo donde nos pregunta qué tipo de proyecto queremos crear. Aunque hay muchos tipos de proyectos que usan Cocoa, en este apartado vamos a ver sólo cuatro tipos de proyectos: Un proyecto que crea una aplicación de consola Objective-C, un proyecto que crea una librería de enlace estático, otro proyecto que crea una librería de enlace dinámico, y otro proyecto más que crea un framework. También veremos cómo desde otro proyecto Xcode podremos enlazar con estos tres últimos tipos de proyectos².

3.1. Crear un programa

En este apartado vamos a ver cómo se crea un proyecto donde podemos hacer un programa de consola Objective-C. Aunque el uso de Xcode puede resultar muy cómodo, tenga en cuenta que Xcode está llamando por usted a las GCC, con lo que procure tener siempre presentes qué opciones son las que Xcode pasa por usted al compilador y enlazador de las GCC.

Para crear un proyecto de consola, ejecute Xcode y elija la opción de menú `File|New Project`. Aparecerá un diálogo donde se le preguntará qué tipo de proyecto desea crear, dentro del grupo `Command Line Utility` seleccionaremos el tipo `Foundation Tool`³. Esto nos permite crear un proyecto Objective-C que enlaza con el Foundation Framework. A continuación se le pedirá un directorio y nombre para el proyecto. En nuestro caso hemos elegido el nombre de proyecto `pidesaludo`, lo cual creará un comando de consola `pidesaludo` similar al comando que hicimos en el apartado 2, sólo que ahora lo vamos a desarrollar desde Xcode.

El proyecto Xcode se guarda en un fichero llamado `pidesaludo.xcodeproj` en el subdirectorio que usted haya especificado⁴. Además, como puede apre-

² Podrá encontrar información más detallada sobre qué son, y cómo se configuran estos tipos de proyectos, en el tutorial "Compilar y depurar aplicaciones con las herramientas de programación de GNU" que también encontrará publicado en MacProgramadores.

³ Esta es la localización de los proyectos de consola que usan el Foundation Framework en Xcode 3.0. Apple ha cambiado ya varias veces la distribución de los tipos de proyectos en este diálogo, y no sería extraño que en futuras versiones la localización de este tipo de proyecto volviese a cambiar.

⁴ Realmente `pidesaludo.xcodeproj` es un subdirectorio con el atributo de bundle que permite que usted lo vea desde el Finder como si fuese un único fichero.

ciar en la Figura 1.2, dentro del grupo External Frameworks and Libraries se ha enlazado con el Foundation Framework, con lo que tenemos ya acceso a todas las clases Objective-C definidas en este framework.

Por defecto se nos habrá creado un fichero llamado `pidesaludo.m` con el contenido que muestra la Figura 1.2. En este esqueleto de programa se nos sugiere crear un objeto `NSAutoreleasePool`, el cual se usa para implementar el sistema de recogida automática de memoria de Cocoa tal como veremos en el apartado 3.11 del Tema 5. Nosotros vamos a cambiar el código generado automáticamente por el del Listado 1.7.

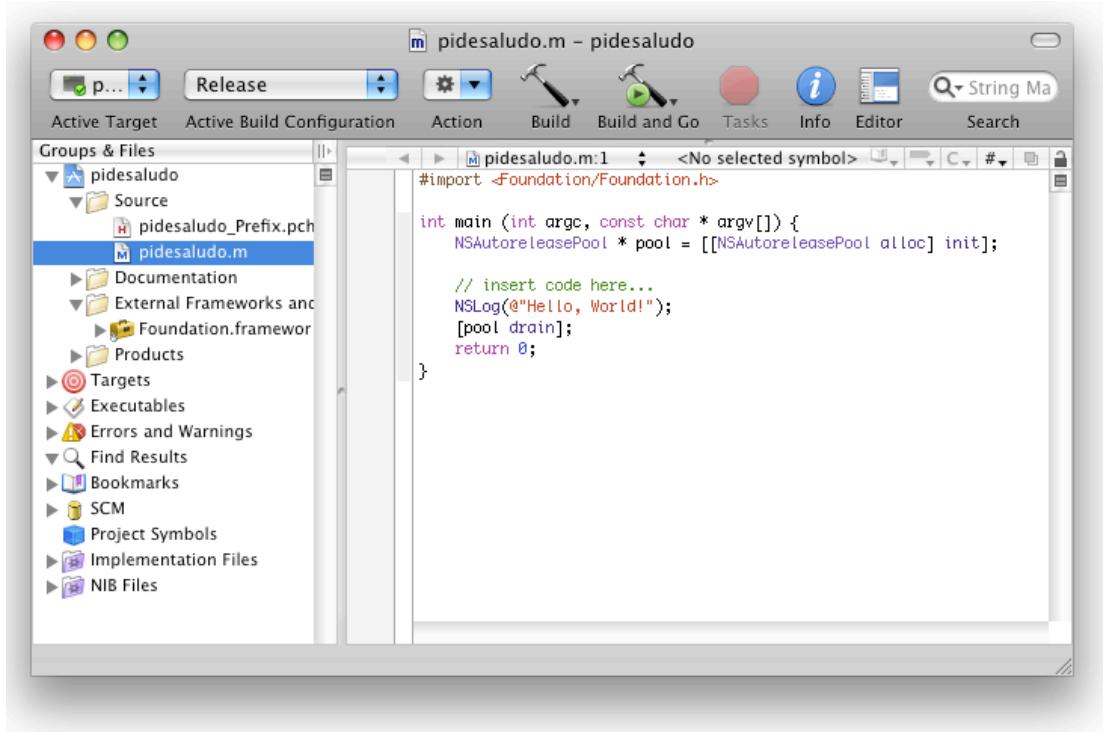


Figura 1.2: Programa de consola básico

También vamos a añadir al proyecto la clase `Saludador` que hicimos en el apartado anterior. Para ello podemos usar la opción `File|New File` y seleccionamos el tipo `Objective-C class`. Como nombre de fichero elegiremos `Saludador.m`, y dejando seleccionada la opción `Also create "Saludador.h"` se nos creará también el fichero de cabecera correspondiente. Después sólo tenemos que cambiar el código que genera automáticamente Xcode por el que se muestra en el Listado 1.5 y Listado 1.6.

Una vez introducido el código de la clase `Saludador` el proyecto generado debería tener la forma de la Figura 1.3. Ahora podemos usar la opción de menú `Build|Build` para compilar el programa y la opción `Debug|Run Executable` para ejecutar el programa.

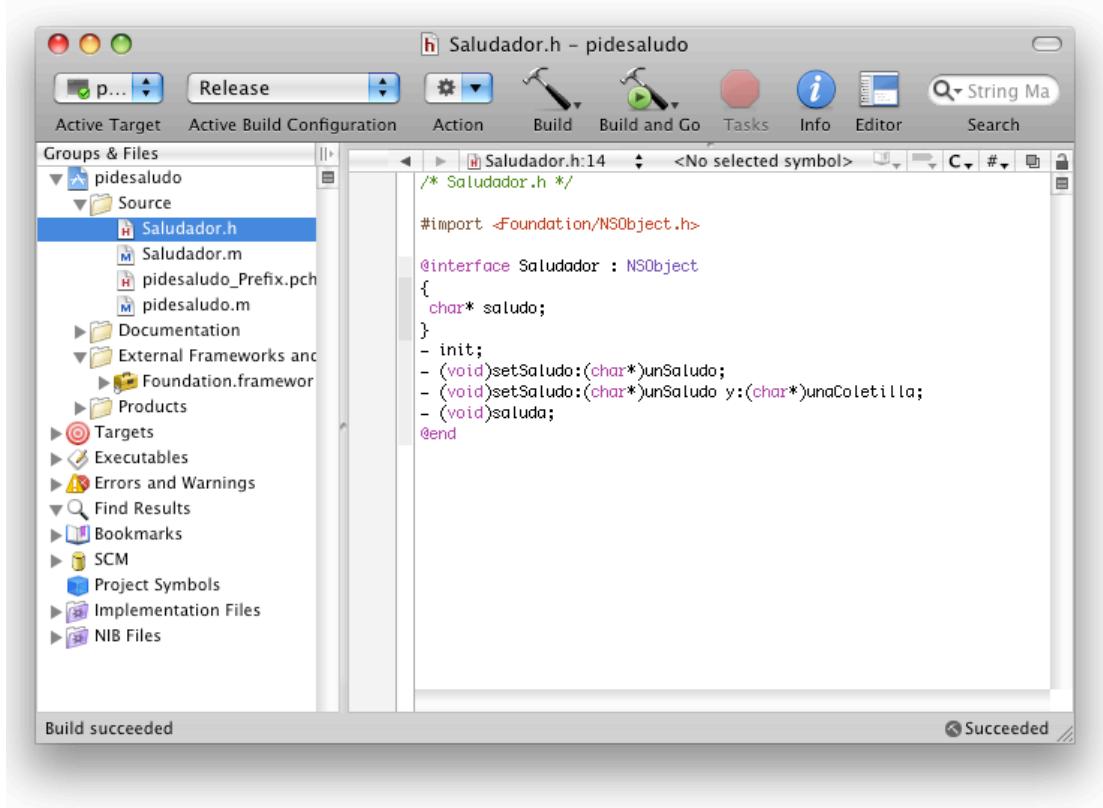


Figura 1.3: Programa Objective-C terminado

Al iniciar el proceso de compilación se genera el subdirectorio `build` con el resultado de la compilación. También es posible iniciar el proceso de compilación desde el terminal con el comando `xcodebuild`. Para obtener la lista de targets y de configuraciones de compilación podemos ejecutar desde el directorio `pidesaludo`:

```
$ xcodebuild -list
Information about project "pidesaludo":
Targets:
    pidesaludo (Active)

Build Configurations:
    Debug
    Release (Active)

If no build configuration is specified "Release" is used.
```

Para indicar la configuración y target que queremos lanzar usaríamos:

```
$ xcodebuild -configuration Debug -target pidesaludo
```

Una vez termina el proceso de compilación, se creará dentro del directorio `build` un subdirectorio para la configuración elegida (Debug en este caso).

3.2. Crear una librería de enlace estático

En el apartado 2.5 creamos una librería de enlace estático llamada `libsaludos.a` usando las GCC. Ahora vamos a crear otra vez esta librería, pero desde Xcode.

Para empezar vamos a crear un nuevo proyecto con Xcode usando `File | New Project` y dentro del grupo `Static Library` elegimos ahora el tipo `Cocoa Static Library`. Si queremos que nos genere un fichero de librería llamado `libsaludos.a`, como nombre de fichero deberemos elegir `saludos`, lo cual generará el nombre de fichero `saludos.xcodeproj`.

La Figura 1.4 muestra qué forma tendrá ahora este proyecto. Observe que en el grupo `External Frameworks and Libraries` se nos han incluido varias librerías. Realmente (véase apartado 2.5) para crear una librería de enlace estático no es necesario enlazar con ningún framework, ya que es la aplicación que usa la librería de enlace estático la que debe enlazar con los frameworks. En consecuencia podemos eliminar los frameworks tal como muestra la Figura 1.5.

En el grupo `Products` de la Figura 1.4 vemos que se nos indica que el producto generado por este proyecto será el fichero de librería estática `libsaludos.a`. El color rojo indica que el fichero actualmente no existe.

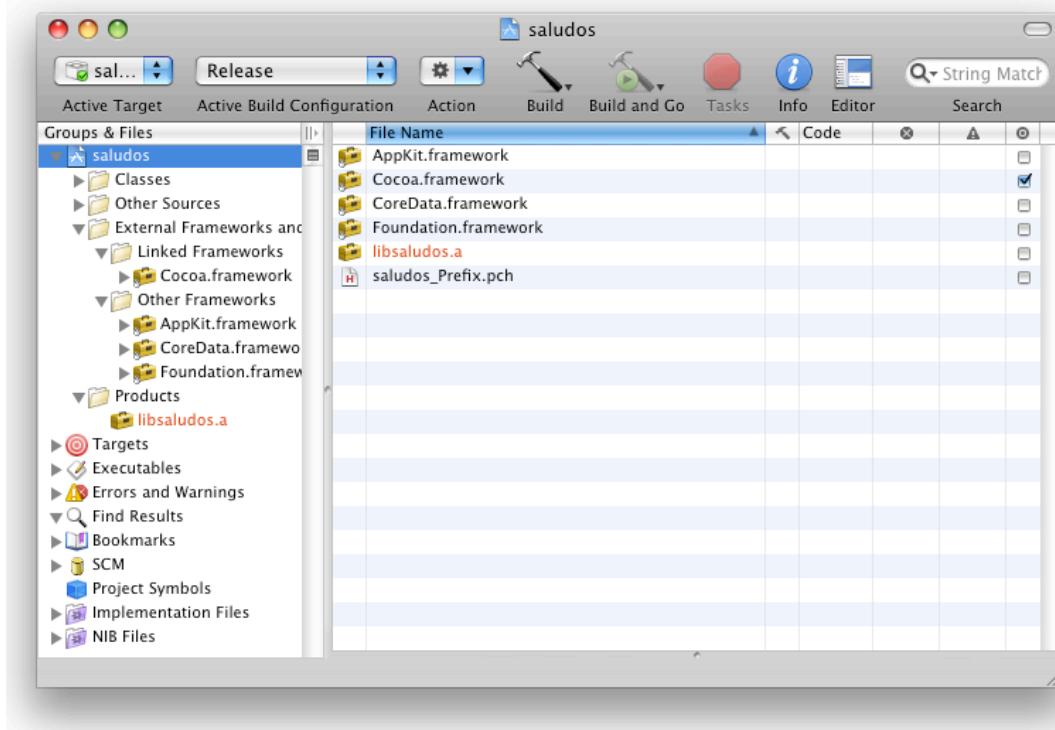
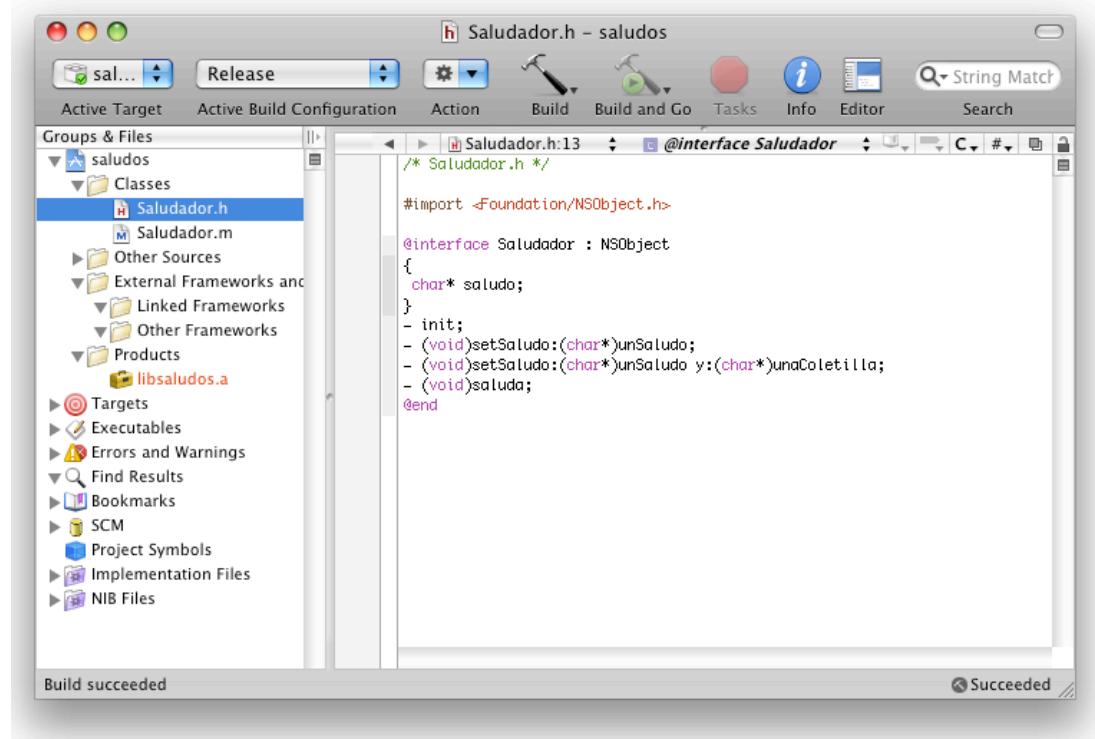
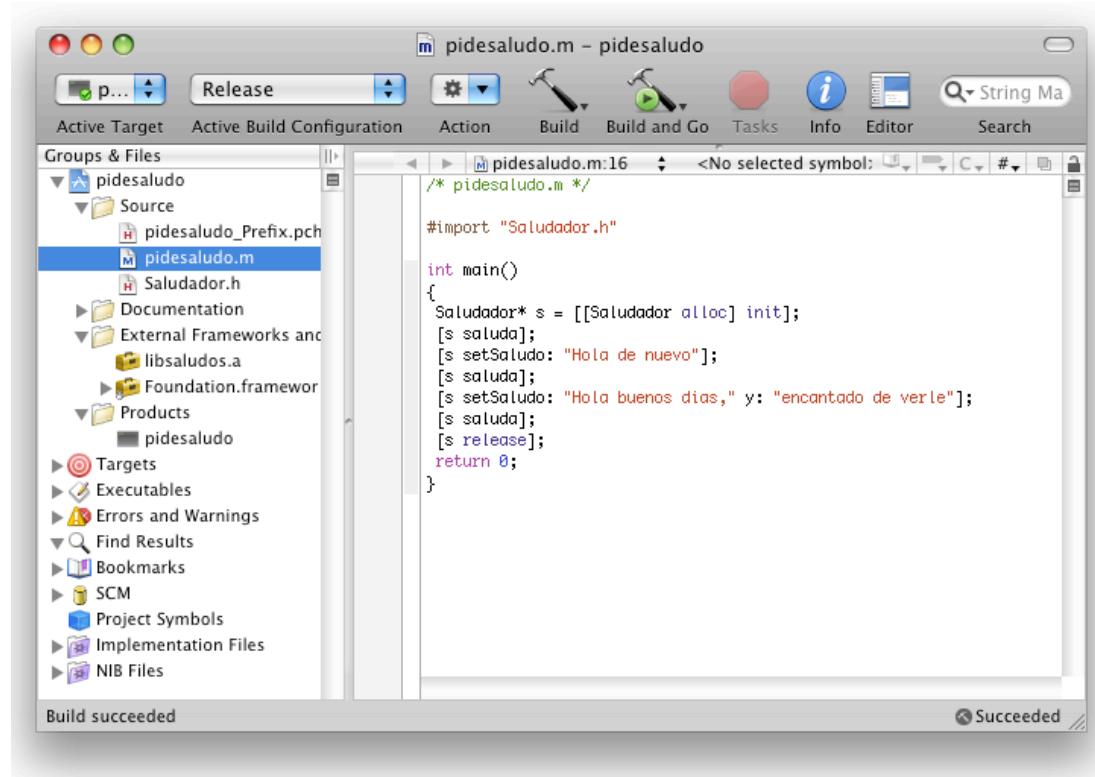


Figura 1.4: Proyecto de librería estática de partida

**Figura 1.5:** Resultado de crear una librería estática**Figura 1.6:** Programa que usa una librería estática

También vemos que, a diferencia de cuando creamos una aplicación de consola en el apartado 3.1, ahora Xcode no ha creado ningún fichero de código fuente, con lo que nosotros vamos a volver a crear los ficheros `Saludador.h` y `Saludador.m` usando la opción de menú `File|New File|Objective-C class`. La Figura 1.5 muestra el resultado de crear la librería estática.

Por otro lado vamos a crear un proyecto con una aplicación de consola que llame a la librería. Para ello, igual que en el apartado 3.1, cree un proyecto de tipo `Foundation Tool` llamado `pidesadulo`. El código fuente del fichero `pidesaludo.m` lo podemos cambiar para que sea de nuevo idéntico al del Listado 1.7. Aunque lógicamente no necesitamos incluir la implementación de la clase, ya que la vamos a coger de la librería estática, sí que necesitamos incluir el fichero `Saludador.h` con la interfaz de clase. Para ello podemos usar la opción de menú `Project|Add To Project`

Para usar la librería de enlace estático `libsaludos.a` en nuestro proyecto también podemos usar la opción `Project|Add To Project`. Tras añadir esta librería el proyecto tendrá la forma de la Figura 1.6, y podremos compilar y ejecutar el programa.

3.3. Crear una librería de enlace dinámico

En Mac OS X las librerías de enlace dinámico suelen tener la extensión `.dylib`. Para crearlas desde Xcode volvemos a usar la opción de menú `File|New Project`, y elegimos dentro del grupo `Dynamic Library` el tipo `Cocoa Dynamic Library`. Como nombre de proyecto podemos elegir `libsaludos`. Esto generará una librería de enlace dinámico llamada `libsaludos.dylib`⁵. La Figura 1.7 muestra la configuración inicial de este proyecto.

Observe que, al igual que cuando creamos un proyecto de librería estática en el apartado 3.2, en el grupo `External Frameworks and Libraries` se nos han incluido varias librerías. A diferencia de las librerías de enlace estático, las librerías de enlace dinámico necesitan indicar los frameworks de los que dependen. Esto hace que antes de cargarse la librería de enlace dinámico se carguen los frameworks de los que dependa. Tal como Xcode ha creado el proyecto, Cocoa es el único framework que enlazará con nuestra librería de enlace dinámico. Esto no se debe a que el framework esté colocado en la subcarpeta `Linked Frameworks` (aunque el nombre de la subcarpeta sirve para indicárnoslo), sino a que es el único framework activo en el target del proyecto.

⁵ Observe que en el apartado anterior Xcode asignaba al fichero de librería estática el nombre del proyecto precedido por `lib`, cosa que no hace por defecto con las librerías de enlace dinámico. Como también es muy común preceder a los nombres de librería de enlace dinámico con `lib`, nosotros hemos puesto este prefijo al nombre del proyecto.

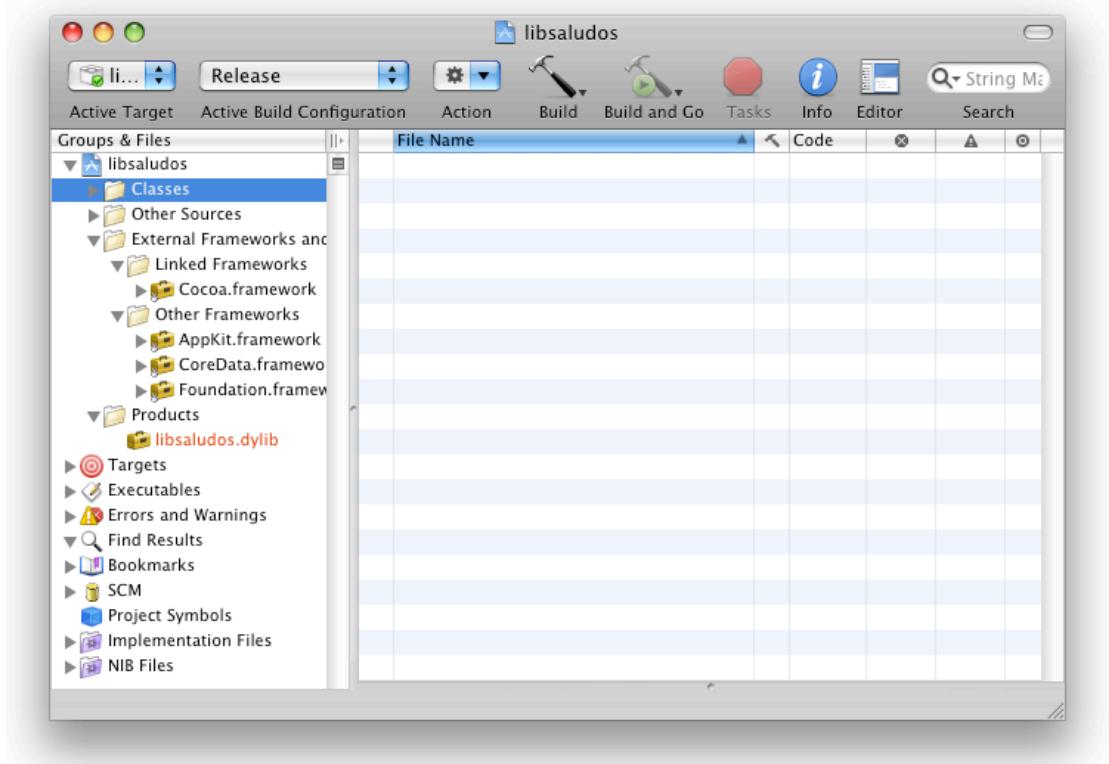


Figura 1.7: Proyecto inicial de librería de enlace dinámico

Los frameworks que aparecen en la subcarpeta `Other Frameworks` son frameworks inicialmente inactivos para el target (aunque en breve veremos cómo se activarían). La razón por la que Xcode nos los pone en el proyecto es porque, al ser frameworks muy usados, podríamos querer activarlos.

Para activar o desactivar un framework (véase la Figura 1.8), debe seleccionar el framework, y desde el inspector de propiedades (Command+I) en la pestaña `Targets` podrá indicar si desea o no activar el target.

Como nuestra librería de enlace dinámico no necesita enlazar con Cocoa⁶, sólo con el Foundation Framework, podemos desactivar los demás frameworks del proyecto y activar el Foundation Framework en la pestaña `Targets`. Tras hacer esto el proyecto debería tener la forma de la Figura 1.8. Ahora puede comprobar que el Foundation Framework es el único framework de la etapa del target `Link binary With Libraries`.

El resto del proceso de creación de la librería de enlace dinámico es similar al de la librería de enlace estático. Cree los ficheros de ejemplo `Saludador.h` y `Saludador.m` y compile el proyecto para producir el fichero `libsaludos.dylib`.

⁶ Cocoa es un umbrella framework que engloba al Foundation Framework.

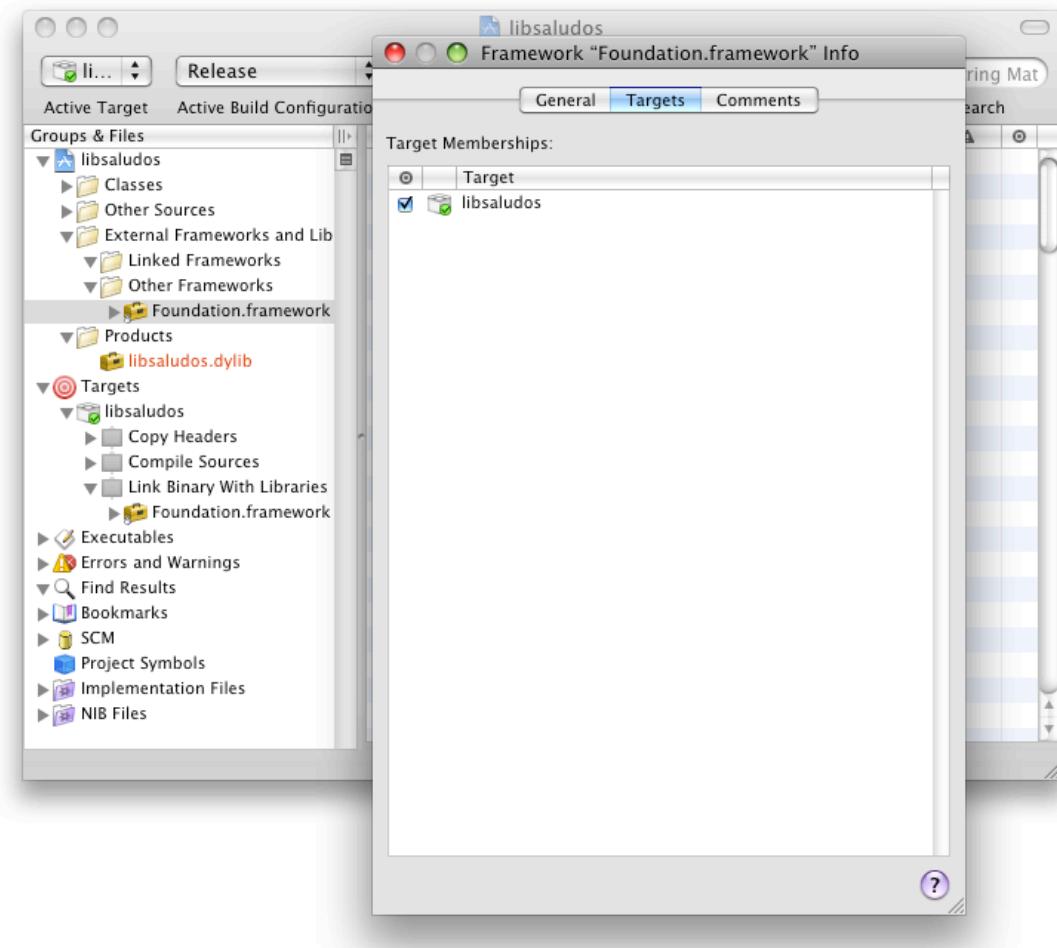


Figura 1.8: Proyecto con el Foundation Framework activado en el target

De nuevo puede crear una aplicación de consola como la del apartado anterior desde la que acceder a la librería de enlace dinámico. Para añadir la librería de enlace dinámico al proyecto de la aplicación puede volver a usar la opción de menú Project | Add To Project

Seguramente cuando vaya a ejecutar ahora el programa `pidesaludo` obtendrá un mensaje de error de la forma:

```
dyld: Library not loaded: /usr/local/lib/libsaludos.dylib
  Referenced from: pidesaludo
  Reason: image not found
pidesaludo has exited due to signal 5 (SIGTRAP).
```

La razón está en que las librerías de enlace dinámico tienen un **nombre de instalación**⁷, que es el directorio donde se supone que debería estar instalada la librería. Cuando Xcode creó el proyecto de la librería de enlace dinámico, asignó a `libsaludos.dylib` el nombre de instalación `/usr/`

⁷ Puede consultar el nombre de instalación de la librería de enlace dinámico usando el comando `otool -D libSaludos.dylib`

local/lib/libsaludos.dylib, y al no encontrarse ahí, falla la carga del programa pidesaludo.

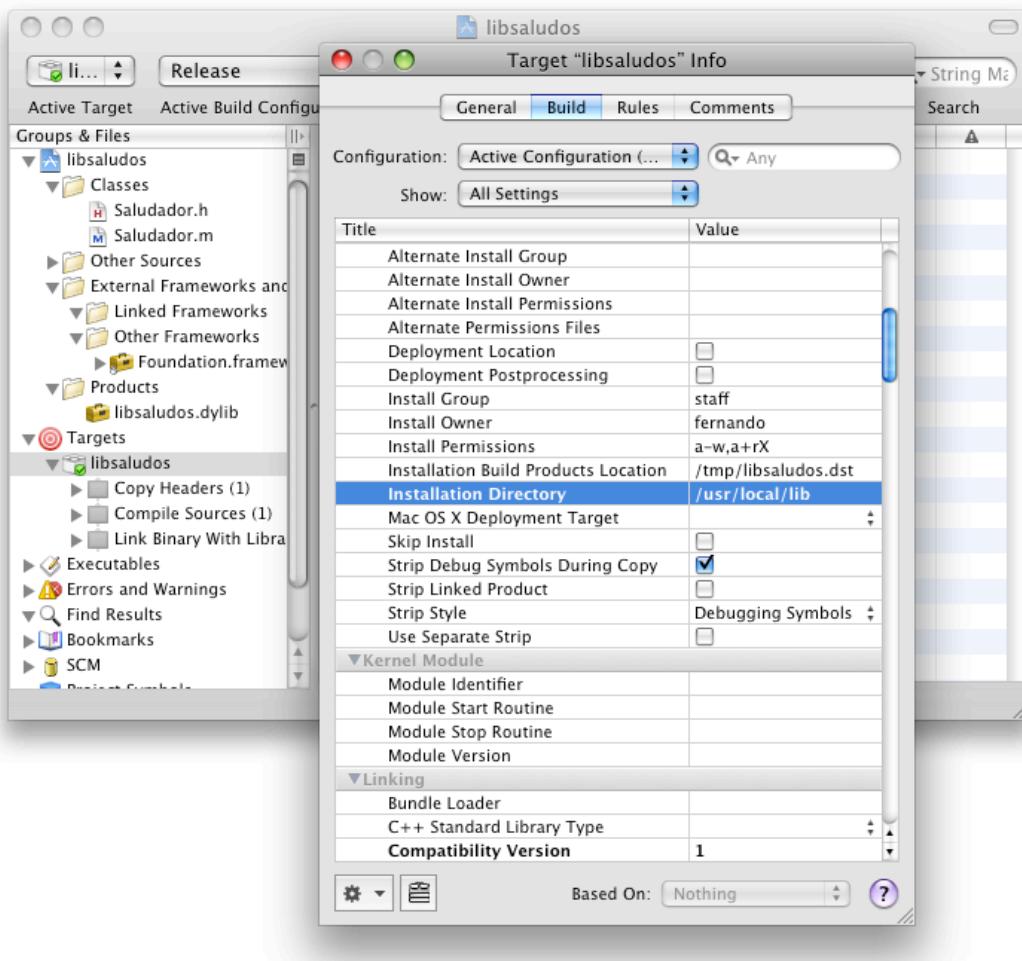


Figura 1.9: Cambiar el nombre de instalación

Para solucionarlo siempre puede copiar la librería de enlace dinámico en la ruta sugerida, pero si esa ruta no es de su agrado puede cambiarla en el target del proyecto de la librería de enlace dinámico. Para ello, como muestra la Figura 1.9, seleccione en el target la librería que estamos generando, y usando el inspector de propiedades (Command+I), en la pestaña Build encontrará la opción Installation Directory. En esta opción puede indicar el nombre de instalación de la librería a generar.

El nombre de instalación de una librería también se puede pasar al comando `gcc` cuando generamos la librería con la opción `-install_name`. Por ejemplo, en la librería de enlace dinámico que generamos en el apartado 2.5 podríamos haber ejecutado el comando `gcc` así:

```
$ gcc -dynamiclib Saludador.o -framework Foundation -install_name $HOME/libSaludos.dylib -o libSaludos.dylib
```

Y el directorio `$HOME/libSaludos.dylib` sería el nombre de instalación:

```
$ otool -D libSaludos.dylib
libSaludos.dylib: /Users/fernando/libSaludos.dylib
```

3.4. Crear un framework

Para acabar esta introducción al manejo de Xcode con Objective-C, vamos a ver cómo crear un framework. Hasta ahora hemos usado frameworks como Cocoa o Foundation, ahora vamos a ver cómo podemos crear uno propio. La ventaja de los frameworks de Mac OS X está en que, junto con una librería de enlace dinámico podemos guardar sus ficheros de cabecera, su documentación, y otros recursos que use el framework (p.e. iconos).

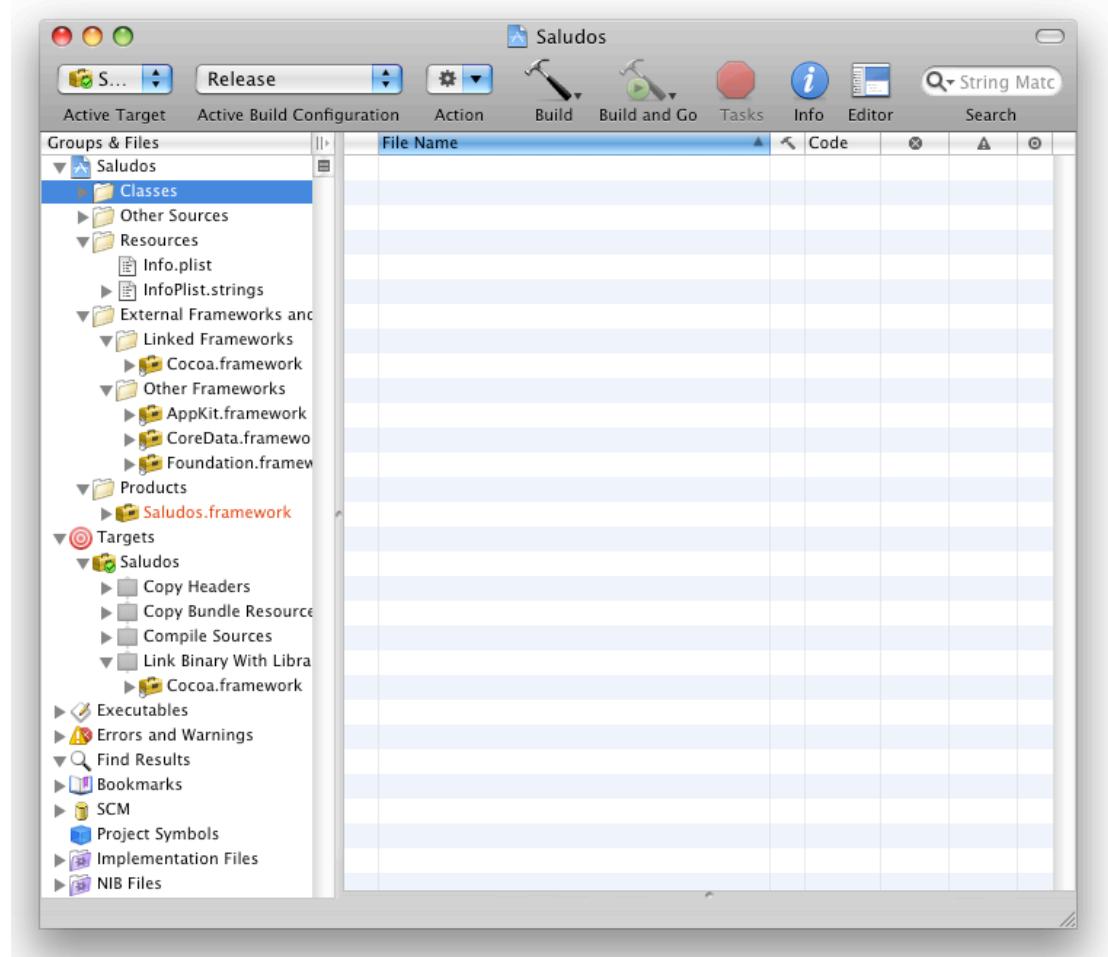


Figura 1.10: Proyecto inicial de un framework

Los framework que crea el programador se suelen guardar en subcarpetas dentro de `/Library/Frameworks` (los frameworks que crea Apple son una excepción respecto a que son los únicos que se guardan en `/System`

/Library/Frameworks). Cada framework es una subcarpeta con una estructura de subdirectorios estandarizada⁸ y con la extensión .framework.

Para crear un framework, de nuevo usamos la opción de menú File|New Project, y dentro del grupo Framework elegimos el tipo Cocoa Framework. Para nuestro ejemplo vamos a crear un framework llamado Saludos.framework (el nombre del framework suele empezar con una mayúscula), con lo que vamos a crear un proyecto llamado Saludos. El proyecto que inicialmente obtendremos tiene la forma de la Figura 1.10.

De nuevo podemos eliminar todos los frameworks de los que depende nuestro proyecto excepto el Foundation Framework. También debemos de acordarnos de activarlo en el target, para lo cual una forma alternativa de hacerlo es arrastrar al framework a la etapa Link Binary With Libraries del target.

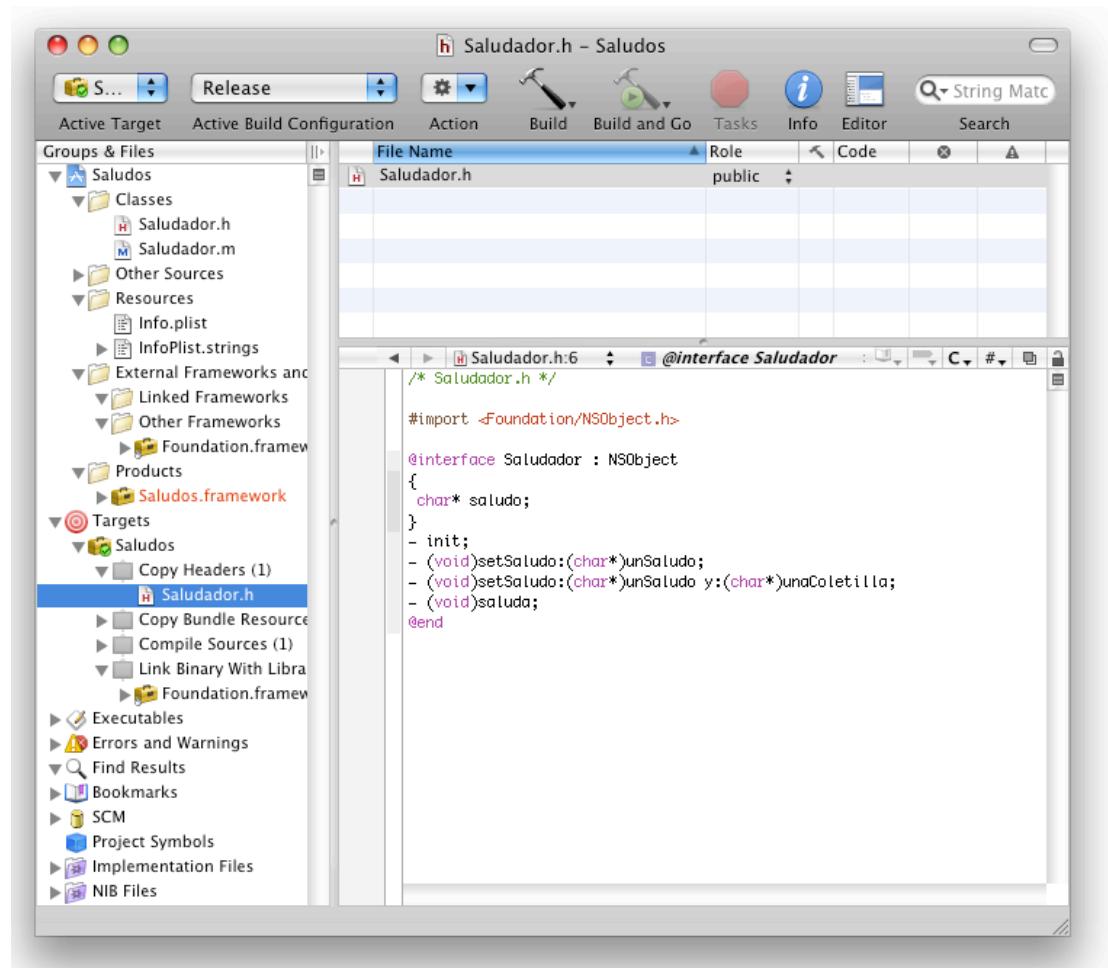


Figura 1.11: Proyecto del framework resultante

⁸ Puede consultar el tutorial "Compilar y depurar aplicaciones con las herramientas de programación de GNU" que también encontrará publicado en MacProgramadores para obtener una descripción más detallada de la estructura y funcionalidad de un framework.

Lo otro que tenemos que hacer, igual que en los anteriores proyectos, es añadir los ficheros `Saludador.h` y `Saludador.m` al proyecto.

Como hemos explicado antes, los framework incluyen, junto a una librería de enlace dinámico, los ficheros de cabecera y documentación de un proyecto. En nuestro caso vamos a hacer que el fichero `Saludador.h` se incluya dentro de las cabeceras del framework de forma que otras aplicaciones que usen nuestro framework puedan usarlo. Para ello observe que el fichero `Saludador.h` ha sido automáticamente incluido en la etapa del target (véase Figura 1.11), pero aun así no será incluido en el framework, para que lo sea debe seleccionar el fichero `Saludador.h` en el target y cambiar el atributo `Role` del fichero del valor `project` al valor `public`.

Por otro lado vamos a usar el framework desde una aplicación de consola. Para ello volvemos a crear un proyecto de consola, como el de los apartados anteriores, e insertamos el framework `Saludos.framework` en el proyecto usando la opción de menú `Project|Add To Project` La Figura 1.12 muestra la forma de este proyecto.

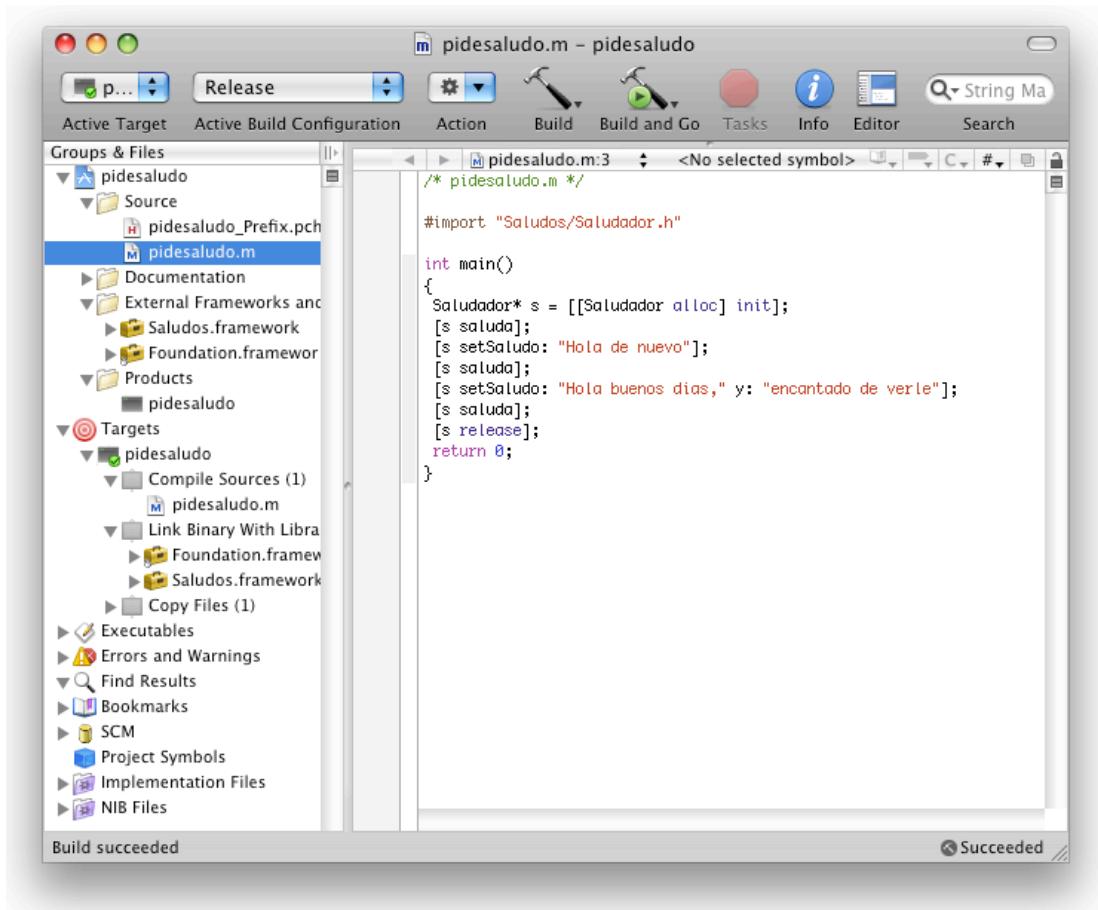


Figura 1.12: Programa que usa el framework

El otro aspecto interesante es que no es necesario incluir en el proyecto el fichero de cabecera del framework (con `Project|Add To Project`), ya que basta con que nuestro código se refiera al fichero de cabecera del framework usando el formato especial:

```
#include <Framework/Cabecera.h>
```

Donde `Framework` es el nombre del framework y `Cabecera.h` un fichero de cabecera del framework. Luego, para referirnos al fichero de cabecera `Saludador.h` usamos la forma (véase Figura 1.12):

```
#include <Saludos/Saludador.h>
```

Ahora podríamos compilar y ejecutar la aplicación de consola.

3.5. Los ficheros de configuración Xcode

El inspector de propiedades (ver **Figura 1.9**) permite configurar las variables de entorno que usa GCC para un proyecto. Si disponemos de un conjunto de variables de entorno comunes a muchos proyectos resulta engorroso copiarlas de un proyecto a otro cada vez que abrimos un nuevo proyecto. Una forma elegante de solucionar este problema es creando un fichero de configuración Xcode (con la extensión `.xcconfig`). Este fichero puede ser compartido por muchos proyectos, o copiado a otros proyectos. Para crear este fichero tenemos la opción `File|New|Other|Configuration Settings File`.

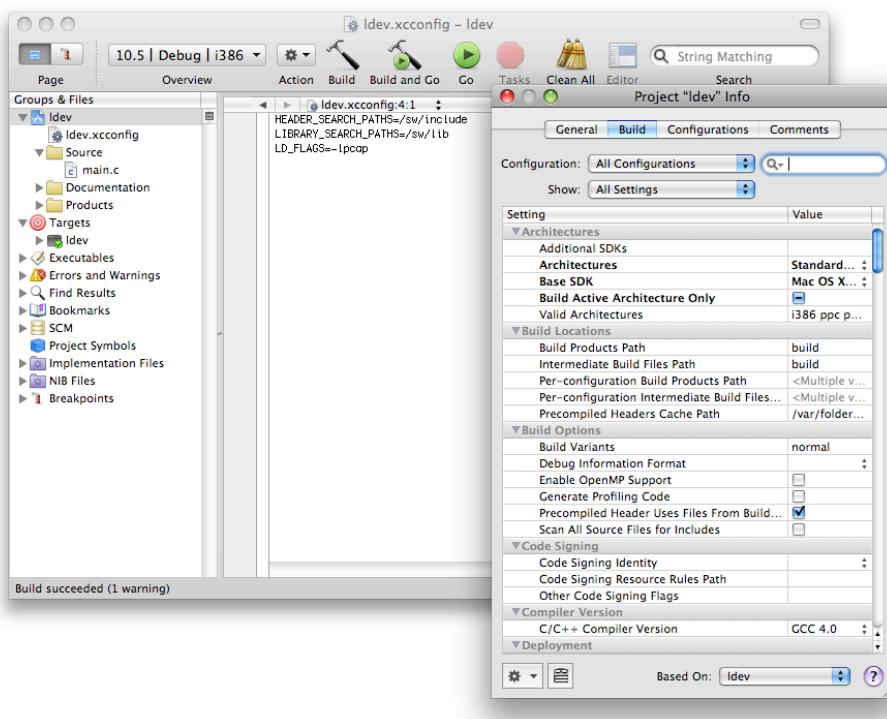


Figura 1.13: Ejemplo de fichero de configuración Xcode

En la **Figura 1.13** hemos creado un fichero de configuración Xcode llamado `ldev.xcconfig` con opciones de compilación que nos permiten acceder a la librería `libpcap` del paquete Fink. Una vez creado este fichero debemos abrir el inspector de propiedades (Command+I) y en la opción `Based on` seleccionar el fichero de configuración Xcode a utilizar.

Si este documento le está resultando útil puede plantearse el ayudarnos a mejorarlo:

- Anotando los errores editoriales y problemas que encuentre y enviándlos al sistema de Bug Report de Mac Programadores.
- Realizando una donación a través de la web de Mac Programadores.

Tema 2

Características del lenguaje

Sinopsis:

En este segundo tema vamos a describir, a nivel conceptual, las principales características del lenguaje Objective-C con el fin de asentar una serie de conceptos fundamentales. En el Tema 3 describiremos cómo el lenguaje implementa estos conceptos.

Empezaremos viendo por qué Objective-C es un lenguaje tan marcadamente dinámico, y qué ventajas introduce este dinamismo. Después describiremos los tipos de relaciones que se dan entre los objetos Objective-C, y en qué se diferencia Cocoa de otras librerías de clases orientadas a objetos.

1. Qué es Objective-C

Al igual que C++, Objective-C es una extensión de C para hacerlo orientado a objetos, pero a diferencia de C++, Objective-C está basado en ideas del mundo de Smalltalk, lo cual hace que Objective-C sea un lenguaje más limpio, pequeño y rápido de aprender que C++.

Sin embargo Objective-C es un lenguaje mucho menos usado que C++. El mundo de Mac OS X es quizá una excepción a esta regla debido a que Objective-C es el lenguaje utilizado para programar en Cocoa, la nueva API orientada a objetos de Mac OS X que pretende mejorar la antigua API de programación Carbon.

A diferencia de otros lenguajes de las GCC, Objective-C no ha sido estandarizado por ningún organismo internacional, sino que fue NeXTSTEP, y ahora Mac OS X quienes han contribuido a crear este lenguaje. NeXT, la empresa que creó NeXTSTEP cedió la implementación de Objective-C a las GCC, y ambos están ahora trabajando en mantener y mejorar el lenguaje.

Debido a que Objective-C es una extensión de C compatible hacia atrás, muchas características de la sintaxis de C han sido heredadas por Objective-C, entre ellas:

- Sentencias de control de flujo.
- Los tipos de datos fundamentales, estructuras y punteros.
- Conversiones implícitas y explícitas entre tipos.
- Los ámbitos de las variables: Globales, estáticas y locales.
- Las funciones y su sintaxis.
- Las directivas del preprocesador, aunque veremos que Objective-C añade más directivas del preprocesador, y también añade las llamadas directivas del compilador.

Como ya vimos en el tema anterior, los ficheros de código fuente de Objective-C tienen la extensión `.m`, y en ellos podemos usar tanto la sintaxis de C como la de Objective-C. Además, a partir de las GCC 2.95 podemos compilar en lo que se ha llamado Objective-C++, que no es más que la sintaxis de Objective-C más la de C++. Los ficheros Objective-C++ se caracterizan por tener la extensión `.mm` o `.M`. En el apartado 8 del Tema 10 comentaremos cómo se hace esto, y veremos algunas limitaciones que se producen al intentar combinar ambos lenguajes. La gran ventaja de Objective-C++ está en que vamos a poder acceder a librerías escritas en C++ desde Objective-C, y viceversa.

2. Lenguaje marcadamente dinámico

Si hubiera que elegir una característica que diferencie a Objective-C de otros lenguajes, ésta sería que es un lenguaje muy dinámico, en el sentido de que muchas decisiones que muchos lenguajes toman en tiempo de compilación, Objective-C las deja para el tiempo de ejecución.

La gran ventaja de este dinamismo se aprecia en las herramientas de desarrollo, donde las herramientas tienen acceso a todo el runtime del programa, con lo que las herramientas de desarrollo pueden instanciar los objetos del programa, representarlos visualmente, personalizarlos, monitorizarlos, y depurarlos de forma muy cómoda para el programador.

Los cinco tipos de dinamismo que diferencian a Objective-C de otros lenguajes, y que vamos a describir con más detalle en esta sección son:

1. Memoria dinámica
2. Tipos dinámicos
3. Introspección
4. Enlace dinámico
5. Carga dinámica

1. Memoria dinámica

En los primeros lenguajes la cantidad de memoria que usaba un programa quedaba fijada durante su compilación. Rápidamente los lenguajes empezaron a ser conscientes de la importancia de que un programa pudiera decidir en tiempo de ejecución la cantidad de memoria que quería reservar. Por ejemplo C introdujo la función `malloc()` para reservar memoria dinámicamente.

Lenguajes como C++ permitieron que, no sólo la memoria usada para almacenar datos, sino la memoria ocupada por los objetos se pudiera decidir en tiempo de ejecución. De hecho en C++ podemos reservar memoria para los objetos tanto en la pila (memoria estática) como en el heap (memoria dinámica).

Posteriormente, los lenguajes de programación se dieron cuenta de que reservar memoria para los objetos en la pila hacía que el tamaño de ésta fuera mucho más difícil de predecir, debido a que siempre había que dejar reservada una gran cantidad de memoria para la pila "por si acaso", lo cual daba lugar a un bajo aprovechamiento de la memoria. Lenguajes más modernos como Java o Objective-C solucionaron el problema obligando a que los objetos se creen siempre en memoria dinámica, evitando así los desbordamientos de pila. En Objective-C, a diferencia de C++, los objetos siempre se crean en memoria dinámica.

2. Tipos dinámicos

Asociar un tipo a una variable es una buena idea ya que ayuda al compilador a identificar errores de codificación. Por ejemplo muchos lenguajes no permiten asignar directamente cadenas a números (debido a que la cadena podría no contener un número), o números en coma flotante a enteros (donde se produciría un redondeo). Además la tipificación de los objetos permite al compilador informar al programador de que está intentando acceder a un método o variable de instancia que no existe en el objeto.

La tipificación de variables puede ser de dos tipos: **Tipificación estática**, que se produce cuando es el compilador quien lleva la cuenta de los tipos de las variables para identificar errores, y **tipificación dinámica**, que se da cuando es el runtime del lenguaje el que en tiempo de ejecución detecta y usa el tipo de las variables.

Lenguajes como C++ tienen una tipificación dinámica muy limitada, en concreto C++ permite realizar tipificación dinámica durante el enlace dinámico de métodos marcados como `virtual`. Más allá del enlace dinámico C++ dispone de una extensión al lenguaje llamada RTTI (RunTime Type Information) que permite obtener algo de información dinámica sobre los objetos, pero de forma bastante limitada.

Java y Objective-C son dos lenguajes que proporcionan una tipificación dinámica mucho más rica, como vamos a ver a continuación.

3. Introspección

La **introspección** es la característica que tienen algunos lenguajes – como Objective-C o Java – de observar y manipular como datos el estado de su ejecución.

Con la introspección podemos preguntar en tiempo de ejecución a un objeto cosas como: A qué clase pertenece, de qué clase deriva, qué protocolos implementa, qué métodos tiene, qué parámetros reciben sus métodos, etc.

4. Enlace dinámico

Los lenguajes orientados a objeto reemplazan el concepto de llamada a función por el de envío de mensajes. La diferencia está en que el mismo mensaje puede ejecutar diferentes funciones dependiendo del objeto que reciba el mensaje. A esta capacidad que tienen los objetos de responder al mismo mensaje de distinta forma es a lo que se ha venido a llamar **polimorfismo**.

En lenguajes como C++ el polimorfismo lo implementa el compilador construyendo una tabla, llamada v-table o virtual-table, de las cuales se crea una por

cada clase que tenga métodos virtuales, y donde en tiempo de ejecución se decide qué función ejecutar para cada mensaje que reciba el objeto.

Por el contrario, en Objective-C es el runtime el que, una vez recibido un mensaje, busca en la clase del objeto, y en las clases base, la función a ejecutar. En el apartado 4 del Tema 4 veremos con más detalle cómo se realiza esta búsqueda. El tener que hacer una búsqueda lineal en vez de indirigir una entrada de una tabla tiene el inconveniente de un coste de ejecución mayor, pero gracias a que el runtime de Objective-C cachea las búsquedas, el aumento de coste es despreciable. En el apartado 3.4 del Tema 10 veremos que una de las ventajas de la forma en que Objective-C implementa el enlace dinámico es el **polymorphism**, mediante el cual un objeto puede cambiar la clase de la que deriva en tiempo de ejecución.

5. Carga dinámica

La carga dinámica es una característica que tienen algunos lenguajes como Java u Objective-C, consistente en poder cargar sólo un conjunto básico de clases al empezar el programa, y luego, en función de la evolución del flujo de programa ir cargando las clases de nuevos objetos que se necesiten instanciar.

Quizá la ventaja más importante de la carga dinámica de clases es que hace a los programas extensibles. Los plug-ins son la forma en la que Cocoa implementa este concepto. Ejemplos de plug-ins son los componentes de las preferencias del sistema, los componentes de Interface Builder, o los inspectores de formatos del Finder.

Una vez que se carga una clase, los objetos de esta clase se tratan como cualquier otro objeto Objective-C, lo único que tiene que hacer un programa que quiera ser extensible mediante plug-ins es definir un protocolo que deban implementar los plug-ins.

3. Asociación, agregación y conexiones

Parte del trabajo de hacer un programa orientado a objetos consiste en definir los diagramas de objetos que muestran las relaciones entre los objetos, y cómo estas relaciones evolucionan a lo largo del ciclo de vida del programa. Algunas relaciones pueden ser totalmente transitorias, mientras que otras se mantienen durante toda la ejecución del programa.

En programación orientada a objetos se suele llamar **asociación** a cualquier tipo de relación entre objetos. Estas asociaciones pueden ser peer-to-peer, es decir donde ningún objeto domina sobre el otro, en cuyo caso se llaman asociaciones **extrínsecas**, o por el contrario ser asociaciones **intrínsecas**, es decir donde un objeto domina sobre otro.

Dos tipo de asociaciones intrínsecas muy conocidas son la composición y la agregación: En la **composición** la vida de un objeto está limitada por la de su contenedor, y en C++ se implementa haciendo que las variables de instancia de un objeto (objeto contenedor) sean instancias de objetos de otra clase (objeto contenido). En la **agregación** la vida del objeto contenido no está estrictamente limitada por la de su contenedor, y en C++ se representa con variables de instancia del contenedor que son punteros a los objetos contenidos. En el caso de Java y Objective-C el único tipo de asociación intrínseca es la agregación, ya que tanto en Java como en Objective-C las variables de instancia de un objeto siempre deben ser tipos fundamentales, o bien punteros a objetos.

Interface Builder permite al programador crear un **grafo de objetos** gráficamente y después almacenar este grafo de objetos en un fichero .nib (NeXTSTEP Interface Builder). Cuando el usuario ejecute el programa, el grafo de objetos se recuperará del fichero .nib a memoria. El fichero .nib no sólo almacena los objetos, sino también las conexiones entre los objetos. Se llama **conexión** a cualquier asociación del grafo de objetos que se almacena de forma persistente. Las conexiones se dividen en:

1. **Conexiones outlet**, que son punteros de un objeto a otro objeto.
2. **Conexiones target-action**, las cuales crean relaciones que permiten enviar en mensaje llamado **action** a un objeto llamado **target**.
3. **Bindings**, que permiten mantener sincronizadas las propiedades de dos objetos distintos. Se usan principalmente en Cocoa Bindings con el fin de mantener sincronizadas las propiedades del objeto modelo y objeto vista del patrón de diseño Modelo-Vista-Controlador⁹.

⁹ En este paradigma normalmente Apple implementa los objetos de la vista. El controlador es un mecanismo de desacoplamiento entre vista y modelo, normalmente implementado por el programador de interfaces gráficas. El modelo es a veces implementado con objetos predefinidos por Apple (Core Data), y a veces, es implementado por el programador de aplicaciones.

4. Componentes vs. frameworks

Actualmente los entornos de programación son tan grandes y completos que el tiempo dedicado a aprender un lenguaje es despreciable respecto al tiempo que lleva aprender la funcionalidad de la librería de clases del entorno de programación. El que las librerías de clases sean tan grandes ayuda a reducir mucho el coste de desarrollo de una aplicación, pero a cambio, el tiempo que dedica el programador en aprender a manejar el entorno es cada vez mayor.

A la hora de construir una librería de clases de apoyo al programador los entornos de programación han seguido una de estas dos aproximaciones:

- **Librerías de componentes.** Son un conjunto de clases que representan las distintas partes de un programa (botones, cajas de texto, ventanas, ficheros, ...), y donde es tarea del programador el juntar objetos de estas clases para formar su aplicación.
- **Frameworks.** Los frameworks, además de disponer de las clases de los principales componentes del programa, proporcionan una serie de patrones de diseño que el programador debe seguir para construir su aplicación. Normalmente el framework sugiere un esqueleto de aplicación, y después el programador va extendiendo este esqueleto añadiendo al programa tanto nuevos componentes, como nuevas relaciones entre ellos. De esta forma el programador no sólo adapta su programa al framework, sino que el framework genérico se especializa para el propósito de la aplicación.

Ejemplos de librerías de componente son las clases de REALbasic o las de Java. Ejemplo de frameworks son las MFC (Microsoft Foundation Class) o Cocoa.

La ventaja que ofrecen las librerías de componentes es que dejan libertad al programador para definir las relaciones entre los objetos de la librería. La ventaja que proporcionan los frameworks es que el código generado es más fácil de mantener, ya que cuando llega un nuevo programador al proyecto, éste conoce los patrones de diseño estándar del framework. Normalmente la relación entre objetos en aplicaciones hechas mediante librerías de componentes son tan particulares y peculiares, que sólo los programadores que iniciaron la construcción de la aplicación son capaces de explicarlas. Si a esto añadimos el hecho de que es muy común que el diseño de las aplicaciones no se documente, es muy fácil encontrarse con aplicaciones hechas con librerías de componentes, que luego son extremadamente difíciles de mantener.

Tema 3

Objetos y clases

Sinopsis:

En el Tema 2 hemos introducido los conceptos más fundamentales del lenguaje Objective-C. En este tema vamos a empezar a describir la sintaxis de Objective-C.

Empezaremos viendo cómo se crean clases y objetos, así como concretando aspectos relativos a la forma de implementar sus métodos y variables de instancia. En la segunda parte del tema veremos cómo implementa Objective-C la encapsulación, cómo manejar cadenas, y cómo interpretar una clase Objective-C como si fuera una estructura de datos C.

1. Clases

Las clases en Objective-C constan de una interfaz y una implementación. La **interfaz** indica la estructura del objeto, y la **implementación** contiene la implementación de sus métodos.

La interfaz y la implementación se suelen poner en dos ficheros distintos (con las extensiones `.h` y `.m` respectivamente). El código fuente del fichero de interfaz debe estar disponible para ser usados por otros programadores. Sin embargo el fichero de implementación puede entregarse compilado.

Al igual que ocurre en el lenguaje C++, un fichero de implementación puede contener tanto la interfaz como la implementación de la clase, o bien un mismo fichero de implementación puede contener varias implementaciones de clases. En general lo recomendable es poner cada interfaz y cada implementación en un fichero distinto, pero a veces puede resultar útil poner la interfaz y la implementación en un mismo fichero de implementación. Esto ocurre cuando creamos una pequeña clase auxiliar que no vamos a usar más que desde de la implementación de otra clase. En este caso se puede poner la interfaz e implementación de la clase auxiliar dentro del fichero de implementación de la otra clase.

1.1. La interfaz

El Listado 3.1 muestra un ejemplo de interfaz. Normalmente una clase comienza importando las clases a las que hace referencia (en este caso `NSObject` y `NSString`)¹⁰. La declaración de la interfaz va desde la directiva del compilador `@interface` hasta la directiva del compilador `@end`.

Lo primero que se indica es el nombre de la clase y la clase de la que deriva. En Objective-C si una clase no deriva de `NSObject` (o de `Object`), no puede utilizar las ventajas del runtime de Objective-C. Como veremos en el apartado 6, aunque es posible que una clase Objective-C no derive de ninguna clase, esto sólo se hace cuando la clase se va a utilizar como un almacén de datos (al estilo de las estructuras C), pero no se van a ejecutar métodos sobre ella. También veremos que otra razón por la que Objective-C nos permite que una clase no tenga clase base es por eficiencia.

En general, es buena idea el que una clase derive siempre de otra clase base con una funcionalidad común básica. Algunos lenguajes como Java nos obligan a ello. En otros lenguajes como C++, el estándar no obliga a un objeto a derivar de una clase base, pero muchas librerías han creado una clase base

¹⁰ Al importar `Foundation.h` importamos todas las clases del Foundation Framework.

donde ponen una funcionalidad de runtime básica a compartir por todos los objetos.

Lo primero que se indica en la declaración de la interfaz de una clase son las variables de instancia de ésta. Estas se ponen siempre entre llaves, y su sintaxis es similar a la de C++ o Java. En el apartado 5 veremos cómo se puede indicar la visibilidad de cada variable de instancia.

Como comentamos en el apartado 3 del Tema 2, una diferencia que presenta Objective-C con C++ es que las variables de instancia no pueden ser objetos, sólo punteros a objetos. Otra diferencia que presenta Objective-C con C++ o Java es que las variables de instancia no pueden ser compartidas entre clases, es decir, no existen variables de clase (las marcadas con el modificador `static` en C++ o Java), aunque la misma funcionalidad se consigue declarando una variable estática o global en el módulo. La variable `nPuntos` del Listado 3.2 sería una variable de módulo (declaración `static`) que equivaldría a una variable estática privada de C++. Para conseguir el equivalente a una variable estática pública de C++, tendríamos que definir una variable global en el módulo y hacer una declaración `extern` en el fichero de interfaz.

```
/* Punto.h */
#import <Foundation/Foundation.h>

@interface Punto : NSObject {
    NSInteger x;
    NSInteger y;
}
- init;
+ (NSString*)sistema;
- (NSInteger)x;
- (void)setX: (NSInteger)paramX;
- (void)setX: (NSInteger)paramX
    incrementando: (NSInteger)paramSumar;
- (NSInteger)y;
- (void)setY: (NSInteger)paramY;
- (void)setY: (NSInteger)paramY
    incrementando: (NSInteger)paramSumar;
- (void)setX: (NSInteger)paramX Y: (NSInteger)paramY;
+ (Punto*)suma: (Punto*)p1 : (Punto*)p2;
+ (Punto*)suma: (NSInteger) n , ...;
- (void)dealloc;
@end
```

Listado 3.1: Ejemplo de interfaz

Después de cerrar las llaves aparece la declaración de los métodos de la clase. Los métodos que empiezan por `-` son métodos de instancia, y los que empiezan por `+` son métodos de clase. El lenguaje nos obliga a indicar si un método es de instancia o de clase. La sintaxis del prototipo de los métodos y la forma de llamarlos la veremos con más detalle en el apartado 4. Por ahora sólo comentaremos que las variables de instancia, métodos de instancia, y

métodos de clase pueden tener el mismo nombre sin que se produzcan conflictos entre ellos. Por ejemplo, en el Listado 3.1 las variables de instancia `x`, `y` tienen el mismo nombre que los métodos getter `x`, `y`.

1.2. Implementación

El Listado 3.2 muestra un ejemplo de implementación de la interfaz anterior. Lo primero que se suele hacer en la implementación de una clase es importar su interfaz, y después se usan las directivas del compilador `@implementation` y `@end` para encerrar la implementación de la clase.

```
/* Punto.m */
#include <stdarg.h>
#import "Punto.h"

static NSInteger nPuntos=0;

@implementation Punto
- init {
    if (self = [super init]) {
        nPuntos++;
    }
    return self;
}
+ (NSString*)sistema {
    return @"Cartesiano";
}
- (NSInteger)x {
    return x;
}
- (void)setX:(NSInteger)paramX {
    x = paramX;
}
- (void)setX:(NSInteger)paramX
incrementando:(NSInteger)paramSumar {
    x = paramX - paramSumar;
}
- (NSInteger)y {
    return y;
}
- (void)setY:(NSInteger)paramY {
    y = paramY;
}
- (void)setY:(NSInteger)paramY
incrementando:(NSInteger)paramSumar {
    y = paramY - paramSumar;
}
- (void)setX:(NSInteger)paramX Y:(NSInteger)paramY {
    x = paramX;
    y = paramY;
}
```

```

+ (Punto*) suma:(Punto*)p1 :(Punto*)p2 {
    Punto* sol = [Punto new];
    sol->x = [p1 x] + [p2 x];
    sol->y = [p1 y] + [p2 y];
    return sol;
}
+ (Punto*) suma:(NSInteger)n ,... {
    Punto* sol = [Punto new];
    va_list parametros;
    va_start(parametros,n);
    Punto* p;
    while (n-->0) {
        p = va_arg(parametros,Punto*);
        sol->x += p->x;
        sol->y += p->y;
    }
    va_end(parametros);
    return sol;
}
- (void) dealloc {
    nPuntos--;
    [super dealloc];
}
@end

```

Listado 3.2: Ejemplo de implementación

Sólo si una clase no tiene métodos, podemos omitir su implementación. Observe que, a diferencia de la interfaz, en la implementación se indica el nombre de la clase, pero no se vuelve a indicar de qué clase deriva.

Dentro de la implementación de una clase no se pueden declarar nuevas variables de instancia, pero sí que es posible declarar métodos en la implementación de una clase que no aparezcan en la interfaz. En este caso los métodos son tratados como privados, y sólo podrán ser llamados desde la implementación de la clase.

Como veremos en el apartado 6.1 del Tema 4, el método `init` es el equivalente al constructor en C++ o Java. También a diferencia de C++ o Java, este método suele acabar retornando `self`, que es el equivalente a `this` en C++ o Java. En el apartado 6.3 del Tema 4 también veremos que `dealloc` es el equivalente al destructor en el caso de C++ o Java.

2. Objetos

En C++ los objetos se pueden crear tanto en la pila como en memoria dinámica. Por el contrario en Java o en Objective-C, los objetos sólo se pueden crear en memoria dinámica. Esto da lugar a que en Objective-C sólo nos podamos referir a los objetos mediante punteros, es decir:

```
Punto p; // Error de compilación  
Punto* p; // correcto
```

También, a diferencia de C++, en Objective-C no se pueden pasar a las funciones objetos por valor, sólo se pueden pasar (y sólo pueden devolver) por referencia, es decir, pasar (y devolver) punteros a objetos¹¹.

En Objective-C cuando decimos que `p` es un objeto de la clase `Punto`, lo que debemos entender es que `p` es un puntero a un objeto de la clase `Punto`.

2.1. Instanciar objetos

Para instanciar un objeto debemos llamar a los métodos `alloc` e `init`:

```
Punto* p = [Punto alloc];  
p = [p init];
```

El método de clase `alloc` (que sería el equivalente al operador `new` en C++ o Java) reserva la memoria dinámica del objeto, y pone a cero todas las variables de instancia del objeto. El método de instancia `init` (que sería el equivalente al constructor en C++ o Java), se encarga de inicializar las variables de instancia del objeto. En el apartado 6.1 del Tema 4 veremos que `init` también puede recibir parámetros.

Ambos pasos se pueden realizar en una sola línea de la forma:

```
Punto* p = [[Punto alloc] init];
```

Debido a que ambas operaciones son muy comunes de realizar juntas, podemos usar el método `new` que realiza ambas operaciones a la vez:

```
Punto* p = [Punto new];
```

Como los objetos Objective-C siempre se crean en memoria dinámica, es importante acordarse de liberarlos al acabar de trabajar con ellos. En el apartado 3 del Tema 5 veremos que Cocoa implementa para tal fin un sistema de

¹¹ En el apartado 6.2 veremos una excepción a esta regla

gestión de memoria por cuenta de referencias, y en el apartado 2 del Tema 6 veremos que Objective-C también implementa un sistema de recogida automática de basura, pero mientras tanto basta con decir que la forma de liberar un objeto es con `release`¹²:

```
[p release];
```

2.2. Tipos estáticos y dinámicos

Ya sabemos que cuando trabajamos con objetos en Objective-C, necesitamos referirnos a ellos mediante punteros. En C++ y en Java, la única forma de referirse a un objeto es mediante **tipos estáticos** que, como explicamos en el apartado 2 del Tema 2, son variables donde es el compilador el que conoce el tipo de las variables. En el caso de los objetos la tipificación estática se consigue mediante *variables de tipo puntero a la clase del objeto*, con lo que a los tipos estáticos usados para apuntar objetos también se les llama **punteros a objetos estáticos**, por ejemplo nuestro:

```
Punto* ps = [Punto new];
```

Por el contrario, los **tipos dinámicos** serían variables cuyo tipo no es conocido por el compilador, sólo por el runtime. Objective-C permite usar tipos dinámicos, pero sólo con puntero a objetos, es decir, el tipo del objeto no es conocido por el compilador, pero el runtime en tiempo de ejecución sí que puede conocer el tipo del objeto. Para indicar en Objective-C que queremos crear un **puntero a objeto dinámico**, declaramos la variable puntero del tipo `id`. Por ejemplo, para referirnos a un objeto `Punto` con un puntero a objeto dinámico hacemos:

```
id pd = [Punto new];
```

Observe que `id` no lleva asterisco, ya que por sí mismo es un puntero. Al igual que con un puntero a objeto estático, con un puntero a objeto dinámico también vamos a poder llamar a los métodos del objeto, pero a diferencia de los punteros estáticos, el compilador no comprueba que el método exista cuando se trate de un puntero a objeto dinámico, sino que será responsabilidad del runtime comprobar que el método existe, o producir una excepción si éste no existiese. Por ejemplo:

```
Punto* ps = [Punto new];
id pd = [Punto new];
[ps setX:4]; // Correcto
[ps canta]; // Error de compilacion (metodo no existe)
[pd setX:4]; // Correcto
```

¹² `release` es el método que proporciona `NSObject` para decrementar la cuenta de referencias y liberar la memoria dinámica de un objeto. La clase `Object` de GNU proporciona otro método equivalente llamado `free`.

```
[pd canta]; // Compila correctamente pero falla en ejecucion
```

Aunque en muchas ocasiones resulta útil que sea el compilador el que compruebe la existencia de los métodos, dejar que esta comprobación se haga en tiempo de ejecución va a permitirnos trabajar con objetos altamente polimórficos, ya que la variable lo único que dice sobre el objeto es que es un objeto.

Además del identificador del preprocesador heredado de C `NULL`, Objective-C proporciona las palabras reservadas `nil` (puntero a objeto sin inicializar) y `Nil` (puntero a clase o metaclass sin inicializar). Realmente estos tres símbolos valen `0`, con lo que son totalmente intercambiables, pero tienen un significado semántico para el programador que conviene recordar.

En el caso de los punteros a objetos estáticos los casting y mensajes que permite el compilador son los mismos que en C++:

- Un puntero a clase derivada será implícitamente convertido a puntero a clase base.
- Un puntero a clase base necesita una conversión explícita (casting) para ser convertido en puntero a clase derivada.
- Las conversiones entre punteros a objetos no relacionados jerárquicamente necesitan conversión explícita (casting), aunque lógicamente en tiempo de ejecución pueden fallar.
- Sobre un puntero a objeto estático sólo podemos ejecutar los métodos que tenga la clase del puntero.

Por el contrario, los punteros a objetos dinámicos son más flexibles:

- Podemos enviar cualquier mensaje a un puntero a objeto dinámico. En tiempo de ejecución se producirá una excepción si este método no existe en el objeto.
- Podemos asignar cualquier puntero a objeto estático o dinámico a una variable de tipo `id`.
- Podemos asignar una variable de tipo `id` a cualquier puntero a objeto (estático o dinámico). En este caso asumimos el riesgo de asignar la variable a un puntero a objeto de tipo incompatible.

Objective-C no es tan tolerante como acabamos de decir, de hecho Objective-C puede producir warnings que nos ayuden a identificar errores. En concreto Objective-C produce un warning cuando ejecutamos sobre un puntero a objeto dinámico un método cuyo nombre no esté definido en ninguna clase. Esto, en muchas ocasiones, ayuda a detectar errores a la hora de escribir el nombre del método.

3. Variables de instancia

Como muestra el Listado 3.1, las variables de instancia se declaran entre llaves dentro de la interfaz. Además la interfaz es el único lugar donde se pueden declarar variables de instancia, la implementación no puede indicar nuevas variables de instancia, ni aunque sean privadas. La razón por la que es necesario que las variables de instancia estén sólo en la interfaz es para poder conocer en tiempo de compilación el tamaño de los objetos.

En caso de que queramos acceder a la variable de instancia desde fuera de la clase se usa el **operador flecha**, por ejemplo:

```
Punto* ps = [Punto new];
ps->x = 3;
```

Como veremos en el apartado 5, por defecto las variables de instancia son protegidas, aun así actualmente Objective-C nos permite acceder a ellas produciendo un warning, y avisando de que en el futuro esta forma de acceso se considerará un error.

Dentro de un método se pueden acceder directamente a las variables de instancia indicando su nombre, o bien usar `self` (que es el equivalente a `this` en C++ o Java) para acceder a la variable de instancia. Por ejemplo:

```
x = 3; // Acceso a una variable de instancia desde un método
self->x = 3; // Forma alternativa
```

El operador flecha sólo se puede usar con punteros a objetos estáticos, los punteros a objetos dinámicos no pueden acceder a las variables de instancia del objeto directamente, sino que siempre deben acceder usando métodos `getter/setter`. Es decir:

```
id pd = [Punto new];
pd->x = 3; // Error de compilación
[pd setX:3]; // Correcto
```

En C++ podemos poner el modificador `const` a una variable de instancia para que sea de sólo lectura. Sin embargo, en Java y Objective-C no existe la noción de variable de instancia de sólo lectura. Toda variable de instancia que se puede leer, se puede también modificar.

4. Métodos

Los métodos son operaciones asociadas con un objeto, y se usan, o bien como interfaces para leer y cambiar el estado de un objeto, o bien como un mecanismo para pedir al objeto que realice una acción.

Como comentamos en el apartado 2 del Tema 2, en el caso de C++ los métodos son funciones que cuando se ejecutan reciben como parámetro el puntero `this` para poder acceder a las variables de instancia del objeto. En C++ el método a ejecutar se decide en tiempo de compilación (estáticamente), a no ser que declaremos al método con el modificador `virtual`, en cuyo caso el método a ejecutar se decide en tiempo de ejecución (dinámicamente). Por el contrario, por defecto en Java los métodos se resuelven dinámicamente, y si queremos que se resuelvan estáticamente debemos de marcar al método con el modificador `final`. La posibilidad de decidir tanto estáticamente como dinámicamente el método a ejecutar, hace que en C++ y Java se utilicen indistintamente los términos *llamar* a un método de un objeto (decisión estática) y *enviar un mensaje* a un objeto (decisión dinámica).

En el caso de Objective-C el método a ejecutar *siempre* se decide dinámicamente, usando el nombre del mensaje para buscar el método a ejecutar en la clase del objeto. En consecuencia, en Objective-C es más correcto hablar de *enviar un mensaje* a un objeto, que hablar de *llamar* a un método de un objeto. Aun así en la práctica ambos términos se utilizan indistintamente en la literatura.

4.1. Declaración de un método

El Listado 3.1 muestra una clase en la que primero se declaran las variables de instancia del objeto entre llaves, y luego sus métodos. Las principales partes de la declaración de un método son:

- El **nivel** del método, que es obligatorio, e indica quién recibe el mensaje, si la clase (en cuyo caso se pone un `+`), o las instancias de la clase (en cuyo caso se pone un `-`).
- El **tipo de retorno**, que indica el tipo de la variable que retorna el método. En caso de que no retorne nada se usa `void`. El tipo de retorno es opcional, si no se indica el tipo de retorno (como en el caso de `init` en el Listado 3.1) por defecto se retorna un `id`¹³. En los métodos Objective-C es obligatorio poner entre paréntesis los tipos de retorno y de los parámetros.

¹³ En el caso de Objective-C, al igual que en el caso de C, las funciones, si no indican tipo de retorno, por defecto retornan un `int`.

- El **nombre del método** que es obligatorio, y junto con el nombre de los parámetros permiten identificar de forma única a los métodos de un objeto.
- Los **parámetros** del objeto, los cuales usan una notación infija heredada de Smalltalk.

Un método tiene tantos parámetros como veces pongamos el símbolo :. En el caso del Listado 3.1 el método `init` es un ejemplo de un método que no recibe parámetros.

Cuando un método recibe parámetros, cada parámetro no sólo tiene un tipo y una variable local donde recibe el parámetro, sino que también está precedido por una **etiqueta** y el símbolo :. Llamamos **nombre completo** de un método a la unión del nombre del método y sus etiquetas. Esta forma de nombrar los parámetros está heredada de Smalltalk y permite crear un código mejor autodocumentado. Si es usted un programador C++ o Java, puede que al principio le resulte excesiva, pero cuando se familiarice con esta forma de nombrar los parámetros, seguramente la acabe echando de menos en C++. Por ejemplo el método C++:

```
pedido->anadeCliente(1, "Juan", 23, 260);
```

Queda mejor autodocumentado cuando lo ejecutamos en Objective-C:

```
[pedido anadeCliente:1 nombre:@"Juan" edad:23 saldo:260];
```

Cuando un método tiene un sólo parámetro el nombre del método sirve para documentar el nombre del parámetro, como ocurre por ejemplo en:

```
- (void)setX:(NSInteger)paramX;
```

Por el contrario, cuando un método tiene más de un parámetro se asignan nuevas etiquetas a los parámetros, que luego además sirven para diferenciar métodos con distintos name mangling en función de las etiquetas de los parámetros. Por ejemplo:

```
- (void)setX:(NSInteger)paramX Y:(NSInteger)paramY;
```

También es cierto que hay ocasiones en las que poner etiquetas a cada parámetro resulta innecesario, especialmente cuando todos los parámetros tienen la misma naturaleza. En este caso podemos omitir la etiqueta, pero no los :, como en el caso de:

```
+ (Punto*)suma:(Punto*)p1 :(Punto*)p2;
```

4.2. Implementación de un método

El cuerpo de un método aparece en el fichero de implementación de la clase. El método empieza con un prototipo idéntico al de la interfaz, excepto que no acaba en punto y coma, sino que entre llaves se indica su implementación. Por ejemplo:

```
- (void)setX:(NSInteger)paramX Y:(NSInteger)paramY {  
    x = paramX;  
    y = paramY;  
}
```

A diferencia de lo que ocurría con las variables de instancia, la implementación de una clase puede tener **métodos privados** que son métodos que no aparecen en la interfaz de la clase. En este caso los métodos privados deberán aparecer en la implementación antes que los métodos públicos. En caso de llamar a un método privado desde la implementación de un método público y donde el método privado aparece declarado después del método público, el compilador emite un warning indicando que el método privado podría no existir. Los métodos privados permiten una mayor encapsulación.

En C++ y en Java es común que los nombres de los métodos getter/setter empiecen por `get` y `set` respectivamente. Por el contrario, en Objective-C lo recomendado es que los métodos getter tengan el mismo nombre que la variable de instancia a la que accedemos (p.e. `x` del Listado 3.1), esto es posible porque las variables de instancia y los métodos de una clase están en namespaces distintos¹⁴. Por el contrario, el nombre de los métodos setter sí que suele empezar por `set`.

También en C++ y en Java los métodos setter suelen tener parámetros cuyo nombre coincide con el nombre de la variable de instancia a la que asignarle, por el contrario en Objective-C los parámetros suelen tener un nombre distinto al de la variable de instancia a la que le asignamos. Por ejemplo:

```
- (void)setX:(NSInteger)paramX Y:(NSInteger)paramY {  
    x = paramX;  
    y = paramY;  
}
```

¹⁴ No use el prefijo `get` para los métodos getter. Por convenio, como se explicará en el apartado 7.3 del Tema 10, en Objective-C cuando el nombre de un método empieza por `get` indica que el método recibe una dirección de memoria en la que depositar el dato.

4.3. Name mangling

Al igual que C++ tiene su propia forma de hacer name mangling, Objective-C también tiene su propio convenio para el name mangling. Podemos usar el comando `nm` sobre un fichero de código objeto, generado a partir del Listado 3.2, para ver cómo realiza Objective-C el name mangling:

```
$ nm Punto.o
00000200 t +[Punto suma::]
00000310 t +[Punto suma:]
00000054 t +[Punto sistema]
00000000 t -[Punto init]
00000404 t -[Punto dealloc]
000001c0 t -[Punto setX:Y:]
000000bc t -[Punto setX:]
000000ec t -[Punto setX:incrementando:]
00000154 t -[Punto setY:]
00000184 t -[Punto setY:incrementando:]
00000090 t -[Punto x]
00000128 t -[Punto y]
    U .objc_class_name_NSConstantString
    U .objc_class_name_NSObject
00000000 A .objc_class_name_Punto
    U __NSConstantStringClassReference
000006cc b __nPuntos
    U __objc_msgSend
    U dyld_stub_binding_helper
```

En Objective-C el name mangling de los métodos de instancia se hace de la forma `-[clase metodo]`, donde `clase` es el nombre de la clase y `metodo` el nombre del método junto con las etiquetas de sus parámetros. Los métodos de clase se nombran de la misma forma, pero precedidos por el símbolo `+`.

Observe que, durante el name mangling, en el nombre de los métodos se guardan los nombres de las etiquetas de los parámetros, pero a diferencia de C++ o de Java, en el nombre del método no se guarda su tipo. Estas reglas de name mangling hacen que en C++ o Java métodos con el mismo número y tipo de parámetros no se puedan sobrecargar (overload). Por el contrario en Objective-C se pueden sobrecargar métodos incluso aunque tengan el mismo número y tipo de parámetros, siempre que tengan etiquetas distintas. Por ejemplo, en el Listado 3.1 encontramos sobrecargados los métodos:

- (void)setX:(NSInteger)paramX
incrementando:(NSInteger)paramSumar;
- (void)setX:(NSInteger)paramX Y:(NSInteger)paramY;

En C++ o Java no hubiera sido posible sobrecargar dos funciones que reciben dos enteros¹⁵.

4.4. Ejecutar un método

Siempre que ejecutamos un método tiene que existir un receptor (que puede ser la clase o una instancia de la clase), y un nombre de método usado para indicar el mensaje a enviar. Para ejecutar el método encerramos el nombre del objeto receptor y el del método a ejecutar entre corchetes de la forma:

```
[p setX:4 Y:3];
```

Si el método tiene parámetros éstos también se meten dentro de los corchetes separados por espacios.

Una llamada a método es una expresión, con lo que si el método retorna un valor, podemos asignárselo a una variable:

```
NSInteger ancho = [p x];
```

Lógicamente, las llamadas a métodos también se pueden anidar:

```
[p2 setX:[p x]];
```

Para ejecutar un método sobre una clase, en vez de indicar como receptor a un objeto, indicamos como receptor a una clase (en el apartado 2 del Tema 4 veremos que en Objective-C las clases también son objetos). Por ejemplo podemos ejecutar el método `sistema` del Listado 3.1 así:

```
NSString* s = [Punto sistema];
```

Al igual que vimos en el apartado anterior que las variables de instancia y los métodos estaban en namespaces distintos, y en consecuencia podían tener el mismo nombre dentro de una clase, también los métodos de clase y de instancia están en namespaces distintos, con lo que pueden compartir el mismo nombre. A la hora de ejecutarlos unos se ejecutan sobre la clase y otros sobre instancias de la clase.

Cuando el compilador genera código objeto para llamar a un método, necesita depositar los parámetros en la pila, así como recoger el retorno de la llamada. Esta operación es una de las pocas operaciones no dinámicas de Objective-C, y la razón de no serlo es que si lo fuera el programa sufriría una

¹⁵ Si nos fijamos el nombre completo de la función, podemos afirmar que en Objective-C no existe sobrecarga ya que cada método tiene un nombre completo distinto. Algunos autores afirman que en Objective-C no existe la sobrecarga, argumentando que el nombre completo del método siempre es distinto.

importante pérdida de rendimiento en tiempo de ejecución. Una consecuencia de que el paso de parámetros y su recogida no sea una operación dinámica es que el compilador debe conocer en tiempo de compilación el tipo de los parámetros y del retorno de los métodos.

En el apartado 2.2 vimos que Objective-C era tan tolerante que si usábamos punteros a objetos dinámicos (de tipo `id`) era posible llamar a métodos no conocidos por el compilador. En tiempo de ejecución se buscará el método, y si éste no existe se producirá una excepción.

Un caso interesante de analizar más detenidamente es qué ocurre cuando el compilador genera una llamada a un método desconocido en tiempo de compilación. En este caso, al no conocer el compilador el método, no puede conocer el tipo de sus parámetros, con lo que no puede hacer una comprobación estática de que estos parámetros estén bien pasados. Cuando esto ocurre el compilador genera un warning porque puede ser que simplemente nos hayamos equivocado al escribir el nombre del método.

Si el compilador no ha visto ningún método con ese nombre completo, asume que tanto el tipo de los parámetros como el tipo de retorno es `id`. Si más tarde (en tiempo de ejecución) el método que recibe la llamada no tiene estos tipos de parámetros pueden producirse inconsistencias si los parámetros no son del tamaño de la palabra (4 bytes en aplicaciones compiladas para el runtime de 32 bits y 8 bytes en aplicaciones compiladas para el runtime de 64 bits).

Por el contrario, basta con que el compilador haya visto un método con ese nombre completo en cualquier otra clase (aunque no tenga ningún tipo de relación con la nuestra) para que el compilador compruebe que los parámetros tienen el mismo tipo que el método que conoce.

4.5. Número variable de parámetros

Al igual que las funciones C, los métodos Objective-C pueden ser llamados con un número variable de parámetros. El Listado 3.1 y Listado 3.2 muestran un ejemplo de cómo hacerlo con uno de sus métodos `suma::`. El prototipo del método con un número variable de parámetros tiene la sintaxis:

```
+ (Punto*) suma: (NSInteger) n , ...;
```

Donde la elipsis `...` precedida por una coma indica que ahí va un número variable de parámetros.

La forma de llamar a estos métodos también es peculiar porque los parámetros variables se separan por comas, y no por dos puntos como los paráme-

tos normales. Por ejemplo para ejecutar el método anterior con dos puntos como parámetros podemos hacer:

```
Punto* p3 = [Punto suma: 2, p1, p2];
```

La forma de acceder a los parámetros variables de un método Objective-C es equivalente a cómo se hace en C, usando la librería `<stdarg.h>` de la forma:

```
+ (Punto*) suma:(NSInteger)n ,... {
    Punto* sol = [Punto new];
    va_list parametros;
    va_start(parametros,n);
    Punto* p;
    while (n-->0) {
        p = va_arg(parametros,Punto*);
        sol->x += p->x;
        sol->y += p->y;
    }
    va_end(parametros);
    return sol;
}
```

Esta librería nos pide que empecemos inicializando una variable de tipo `va_list` que representa la lista de parámetros de la función. En nuestro caso esta variable la hemos llamado `parametros`.

A continuación tenemos que inicializar la variable `parametros` con el macro `va_start()` el cual, además de la variable, recibe el parámetro anterior a la lista de parámetros variable de la función (y no el número de parámetros como pudiera parecer).

En cada llamada al macro `va_arg()` obtendremos un parámetro de la lista de parámetros variables. A este macro además de pasarle la variable `parametros`, debemos pasarle el tipo de la variable que queremos leer de la lista de parámetros. En nuestro ejemplo leemos siempre un puntero a `Punto`, pero funciones como `printf()` pueden leer en cada llamada una variable de un tipo distinto.

Por último, al acabar de leer los parámetros se debe llamar al macro `va_end()` pasándole la variable `parametros` con la que hemos terminado de trabajar.

5. Encapsulación

Para facilitar la **encapsulación**, es decir, para poder ocultar las partes del objeto que otros programadores no necesitan conocer para manejar nuestro objeto, Objective-C permite limitar el ámbito desde el que podemos acceder a las variables de instancia de un objeto.

Para declarar los niveles de encapsulación de las variables de instancia se usan los **modificadores de acceso** `@public`, `@protected` y `@private`. Estas directivas del compilador pueden aparecer tantas veces como sea necesario, y afectan a todas las variables de instancia desde su aparición hasta el nuevo modificador de acceso. Si no existe modificador de acceso, por defecto las variables de instancia son `@protected`. Los efectos de los modificadores de acceso son los siguientes:

1. Cuando una variable de instancia tiene el modificador de acceso `@public`, la variable de instancia es accesible desde cualquier parte del programa.
2. Cuando una variable de instancia tiene el modificador de acceso `@private`, entonces si se accede a la variable de instancia desde dentro del objeto la variable de instancia es visible, en cualquier otro caso la variable de instancia no es visible.
3. Cuando una variable de instancia tiene el modificador de acceso `@protected`, las reglas de acceso son similares a las de `@private` excepto que también se permite acceder a la variable de instancia desde una clase derivada.

Observe que Objective-C sigue las mismas reglas que C++ en lo que respecta a los modificadores de acceso.

A diferencia de C++ o Java, los programadores Objective-C no tienen la costumbre de indicar el ámbito de accesibilidad de las variables de instancia. Sólo en las ocasiones en las que realmente sea importante marcar a una variable como pública o privada se usan sus correspondientes directivas del compilador. Este documento cumple con este hábito.

Como adelantamos en el apartado 3, en Objective-C podemos acceder a variables de instancia protegidas y privadas de un objeto desde fuera del objeto, pero el compilador generará un warning avisando de que en el futuro esta forma de acceso se considerará un error de compilación.

Una peculiaridad de Objective-C, que no encontramos ni en C++ ni en Java, es que los modificadores de acceso afectan sólo a las variables de instancia, no a los métodos de la clase. Los métodos en Objective-C pueden ser sólo de dos tipos: Públicos, si están declarados en la interfaz del objeto, o privados, si están declarados en la implementación del objeto.

Otra peculiaridad de Objective-C es que nos permite llamar a métodos privados (ya sean de clase o de objeto). Durante la compilación se producirá un warning avisando de que el método podría no existir, pero en tiempo de ejecución el método se encuentra y ejecuta correctamente. Aun así, esta característica puede cambiar en el futuro, con lo que si vamos a ejecutar un método desde fuera del objeto, debemos declararlo de acceso público.

6. Clases como estructuras de datos

6.1. Clases sin clase base

En Objective-C, al igual que en C++, podemos declarar clases que no deriven de ninguna clase, pero esto invalida el poder usar todo el soporte que da el runtime de Objective-C a los objetos que derivan de `NSObject` (o de `Object`).

¿Cuándo tiene sentido crear clases Objective-C que no tengan clase base? Una razón es para crear objetos Objective-C que sean usados como estructuras de datos. El Listado 3.3 muestra un ejemplo de este tipo de clases.

```
@interface Fecha {
    @public
    NSInteger dia;
    NSInteger mes;
    NSInteger anno;
}
@end
```

Listado 3.3: Ejemplo de clase sin clase base

Debido a que la clase no tiene métodos, no necesitamos crear la implementación de la clase. El uso del modificador de acceso `@public` evita que se produzcan warnings al intentar acceder a las variables de instancia del objeto desde fuera del objeto.

Ahora podemos instanciar el objeto de la forma:

```
Fecha* f = [Fecha new]; // Warning de compilación
```

Se producirá un warning durante la compilación avisando de que el objeto podría no responder al mensaje `new`, esto se debe a que `new` es un método de la clase `NSObject`, y de hecho en tiempo de ejecución se producirá una excepción.

Sin embargo, sí que podemos crear el objeto `Fecha` de la forma:

```
Fecha* f = (Fecha*) malloc(sizeof(Fecha)); // Correcto
```

```
f->x = 3; // Correcto
free(d); // Correcto
```

Luego el objeto Objective-C se está comportando como una estructura de datos C.

Tenga en cuenta que los objetos Objective-C sin clase base son composiciones más limitadas que las estructuras C, en el sentido de que sólo pueden ser creados y destruidos en memoria dinámica, si intentamos crear el objeto en la pila:

```
Fecha f; // Error de compilación
```

El compilador nos impide crearlo indicando que los objetos Objective-C sólo se pueden crear en memoria dinámica.

6.2. Paso de objetos por valor

Los métodos Objective-C sólo pueden recibir como parámetro punteros a estructuras y punteros a objetos, nunca pueden recibir por valor un objeto o estructura. Sin embargo Objective-C mantiene compatibilidad con C, y esto hace que podamos crear funciones que reciban objetos Objective-C por valor. Por ejemplo, el Listado 3.4 muestra un ejemplo de función C que recibe objetos Objective-C por valor.

```
Punto* Suma(Punto p1, Punto p2) {
    Punto* sol = [Punto new];
    [sol setX:[(&p1) x]+[(&p2) x]];
    [sol setY:[(&p1) y]+[(&p2) y]];
    return sol;
}
```

Listado 3.4: Función C que recibe objetos por referencia

Observe para mandar mensajes a los objetos dentro de la función, obtenemos su dirección de memoria con el operador `&`.

Ahora podríamos llamar a esta función de la forma:

```
Punto* s = Suma(*p, *p2);
[s release];
```

Donde para obtener el objeto a partir de su puntero necesitamos indirigir con el operador `*`. Como la implementación de la función crea un objeto en memoria dinámica, el programa que llama a la función debe acordarse de liberarlo con el mensaje `release`.

7. Objetos cadena

En Objective-C, además de poder usar el tipo `char*` de C para trabajar con cadenas acabadas en cero, podemos usar la clase `NSString` y `NSMutableString` para trabajar con **objetos cadena**.

La clase `NSString` se usa para representar cadenas invariantes, lo cual permite al compilador optimizar las operaciones de gestión de cadenas. Siempre que no vaya a modificar el contenido de la cadena (la mayoría de las veces) debe instanciar objetos de esta clase. La clase `NSMutableString` se usa sólo para representar cadenas variables, es decir, cuyo contenido va a ser modificado por el programa en tiempo de ejecución. Como muestra la Figura 3.1, la clase `NSMutableString` es una derivada de `NSString` que añade métodos que modifican el contenido del objeto cadena.

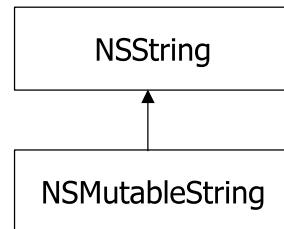


Figura 3.1: Clases para gestión de cadenas

Los objetos de tipo `NSString` (y de su tipo derivado `NSMutableString`) internamente siempre almacenan el contenido de las cadenas en Unicode, pero podemos ejecutar sobre estos objetos métodos para convertir desde, y hasta, otros formatos (p.e. ASCII de 7 bits, ISO Latin 1, UTF-8). También podemos ejecutar sobre `NSString` el método de clase `availableStringEncoding`s para obtener una lista de formatos de codificación que soporta el objeto. También podemos usar el método de clase `defaultCStringEncoding` para obtener el formato de codificación por defecto.

7.1. Crear y manipular objetos cadena

La forma más sencilla de crear objetos cadena en Objective-C es usar la directiva del compilador `@"..."`, por ejemplo:

```
NSString* mensaje = @"Hola";
```

Cuando creamos un objeto cadena de esta forma sólo deberíamos de usar ASCII de 7 bits, ya que el objeto se crea en tiempo de compilación, y no conocemos el formato de codificación por defecto de la plataforma donde lo vayamos a compilar.

Es importante destacar que esta forma de crear un objeto `NSString` hace que el compilador cree un objeto constante y único para cada módulo en el

segmento de datos (y no en el heap), con lo cual este objeto nunca es liberado. El objeto puede recibir los mensajes `retain` y `release`, pero los ignora.

También es posible enviar a estos objetos cualquier mensaje que `NSString` acepte, por ejemplo:

```
BOOL encontrado = [@"Fernando" isEqualToString: nombre];
```

Si esta llamada se hiciera dentro de un bucle, usaría siempre el mismo objeto constante, al que enviaría el mensaje `isEqualToString:`.

En lenguaje C realiza la concatenación automática de cadenas consecutivas. De esta forma podemos dividir una cadena larga en varias líneas de la forma:

```
char* msg = "Este tutorial ha sido escrito por Fernando"  
           " López Hernández para MacProgramadores";
```

Objective-C también realiza la concatenación automática de objetos cadena literales consecutivos, es decir:

```
NSObject* msg = @"Este tutorial ha sido escrito por Fernando"  
                 @" López Hernández para MacProgramadores";
```

Daría lugar a un único objeto cadena que se almacenaría en `msg`.

Un detalle que en ocasiones podría resultar útil es saber que cuando creamos un objeto cadena con la directiva `@"..."`, obtenemos una instancia de la clase que pasemos a la opción del compilador `-fconstant-string-class`. En el runtime de NeXTSTEP por defecto se instancia un `NSString`, por el contrario, en el runtime de GNU por defecto se instancia un `NXConstantString`.

7.1.1. Crear objetos cadena a partir de cadenas C

Podemos crear un objeto cadena a partir de una cadena C usando el método de clase `stringWithCString:`. Aunque si la cadena tiene caracteres fuera del ASCII de 7 bits debemos de usar `stringWithCString:encoding:` para indicar el formato de codificación. Por ejemplo:

```
char* str = "López";  
NSString* apellido = [NSString stringWithCString:str  
                           encoding:NSUTF8StringEncoding];
```

Debido a que el formato de codificación UTF-8 es muy utilizado, `NSString` también proporciona el método `stringWithUTF8String:` para crear un objeto cadena a partir de una cadena codificada en UTF-8:

```
char* strUTF8 = ...;
NSString* nombreUTF8 =
    [NSString stringWithUTF8String:strUTF8];
```

7.1.2. Obtener la cadena C de un objeto cadena

Podemos obtener cualquiera de los caracteres Unicode de una cadena con el método `characterAtIndex:`: el cual recibe un índice y devuelve una variable de tipo `unichar`¹⁶, que no es más que una variable de dos bytes donde se almacena un carácter Unicode. Para obtener la longitud de una cadena podemos usar el método de instancia `length`. También podemos obtener todos los caracteres Unicode de una cadena con el método `getCharacters:`: el cual recibe un buffer de tipo `unichar*` y lo rellena con los caracteres de la cadena.

Si preferimos obtener los caracteres transformados a algún otro formato de codificación podemos usar el método de instancia `UTF8String` que devuelve un `const char*` con la cadena convertida a UTF-8. El método nos devuelve un array de caracteres constante para indicar que no debemos de modificar su contenido. El array que nos devuelve se liberará cuando se destruya el objeto cadena, con lo que debemos de copiarlo a otro buffer si queremos conservarlo.

Si queremos obtener la cadena convertida a otro formato de codificación debemos usar `cStringUsingEncoding:`: indicando el formato de codificación deseado. El método nos devuelve un `const char*` que también se libera al destruirse el objeto. Si preferimos mantener la cadena podemos pedir al objeto cadena que nos copie la cadena en un buffer usando `getCString:maxLength:encoding:`.

7.1.3. Objetos cadena variable

Para crear objetos cadena en los que podamos modificar su contenido usamos `NSMutableString` que – como muestra la Figura 3.1 – es una derivada de `NSString` que añade métodos para modificar su contenido como puedan ser `appendString:`, para añadir caracteres al final de la cadena, `insertString: atIndex:` para insertar otra cadena en medio de la cadena, `deleteCharactersInRange:` que recibe un objeto `NSRange` con el rango de caracteres a borrar, o `setString:` que cambia todo el contenido de la cadena.

¹⁶ El tipo `unichar` es un tipo de dato definido en `NSString.h` como un `typedef` para `unsigned short`, y que es similar al tipo estándar `wchar_t`, definido en `wchar.h`.

Para inicializar el objeto `NSMutableString` podemos utilizar métodos de la base `NSString` que han sido redefinidos (overriden) en la derivada, como por ejemplo `initWithString:`. Un ejemplo de creación y uso de estos objetos podría ser:

```
NSMutableString* cliente =
    [[NSMutableString alloc] initWithString: @"Fernando"];
[cliente appendString: @" Lopez"];
```

7.2. Formatear cadenas

Podemos usar el método de instancia `initWithFormat:format...` de la clase `NSString` para crear objetos cadena formateados. Este método soporta especificadores de formato similares a los de `printf()`. La Tabla 3.1 describe estos especificadores.

Especificador de formato	Descripción
%@	Objeto sobre el que se debe ejecutar <code>description</code> para obtener una descripción textual del objeto.
%%	Carácter %.
%hi	Entero con signo de 16 bits (<code>short</code>).
%hu	Entero sin signo de 16 bits (<code>unsigned short</code>).
%d, %D, %i	Entero con signo de 32 bits (<code>long</code>).
%u, %U	Entero sin signo de 32 bits (<code>unsigned long</code>).
%qi	Entero con signo de 64 bits (<code>long long</code>).
%qu	Entero sin signo de 64 bits (<code>unsigned long long</code>)
%x	Entero sin signo de 32 bits (<code>unsigned long</code>), escrito en hexadecimal con letras de la a a la f.
%X	Entero sin signo de 32 bits (<code>unsigned long</code>), escrito en hexadecimal con letras de la A a la F.
%qx	Entero sin signo de 64 bits (<code>unsigned long long</code>), escrito en hexadecimal con letras de la a a la f.
%qX	Entero sin signo de 64 bits (<code>unsigned long long</code>), escrito en hexadecimal con letras de la A a la F.
%o, %O	Entero sin signo de 32 bits (<code>unsigned long</code>), escrito en octal.
%f	Número en punto flotante de 64 bits (<code>double</code>).
%e	Número en punto flotante de 64 bits (<code>double</code>), escrito en notación científica con e minúscula.
%E	Número en punto flotante de 64 bits (<code>double</code>), escrito en notación científica con E mayúscula.
%c	Carácter ASCII (<code>char</code>)
%C	Carácter Unicode (<code>unichar</code>)

%s	Cadena de caracteres ASCII acabada en cero.
%S	Cadena de caracteres Unicode acabada en cero.
%p	Puntero. Escribe la dirección de memoria en hexadecimal precedida por 0x.

Tabla 3.1: Especificadores de formato de `NSString`

Por ejemplo, podemos crear un objeto cadena así:

```
NSString* pedido = [[NSString alloc] initWithFormat:
    @"El cliente %s pide %d unidades del producto %s"
    , "Pedro", 2, "Zapatos blancos"];
```

Además de los especificadores de formato de `printf()`, podemos usar el especificador de formato "%@" para indicar que su parámetro correspondiente es un objeto. Todos los objetos Objective-C heredan de la clase raíz el método `description`, que por defecto devuelve una descripción del objeto (con el nombre de la clase del objeto y dirección de memoria del objeto) en un `NSString`. Cuando `initWithFormat:format,...` encuentra el especificador de formato "%@" ejecuta sobre el objeto correspondiente el método `description`, para obtener así una descripción textual del objeto.

7.2.1. Imprimir cadenas formateadas

Podemos imprimir una cadena formateada con la función:

```
void NSLog (NSString* format, ...);
```

La función antepone la fecha e imprime en `stderr` el mensaje formateado. Por ejemplo el siguiente trozo de programa:

```
NSLog(@"El cliente %s pide %d unidades del producto %s"
    , "Pedro", 2, "Zapatos blancos");
```

Produciría la salida:

```
2008-08-01 10:02:11.124 [543:813] El cliente Pedro pide 2
unidades del producto Zapatos blancos
```

Si preferimos imprimir el mensaje en `stdout` y sin el informe de fecha podemos usar el método `UTF8String` de `NSString` para obtener el correspondiente array de caracteres. Por ejemplo:

```
NSString* pedido = [[NSString alloc] initWithFormat:
    @"El cliente %s pide %d unidades del producto %s"
    , "Pedro", 2, "Zapatos blancos"];
printf("%s\n", [pedido UTF8String]);
```

Produciría la salida:

El cliente Pedro pide 2 unidades del producto Zapatos blancos

Alternativamente también podemos hacer que `printf()` formatee la salida de la forma:

```
printf("El cliente %s pide %d unidades del producto %s\n",
    @"Pedro" UTF8String], 2, @"Zapatos blancos" UTF8String));
```

7.3. Leer y escribir cadenas de ficheros y URLs

La clase `NSString` proporciona métodos que nos permiten muy fácilmente leer o escribir de un fichero. En este apartado vamos a ver cómo se usan estos métodos.

7.3.1. Leer cadenas

Para leer texto de un fichero, suponiendo que conocemos el formato de codificación, la clase `NSString` dispone del método de clase `stringWithContentsOfFile:encoding:error:`, el cual devuelve un objeto `NSString` con el contenido del fichero. Por ejemplo:

```
NSString* path = @"/Users/fernando/.bash_history";
NSError* error;
NSString* texto = [NSString stringWithContentsOfFile:path
                                                encoding:NSUTF8StringEncoding error:&error];
if (texto==nil) {
    NSLog(@"Error leyendo fichero %@: %@", path
        , [error localizedFailureReason]);
}
```

Al método le pasamos en `encoding` el formato de codificación, y nos devuelve en un objeto `NSError`¹⁷ información de error, en caso de que se produzca. En caso de no querer obtener información de error podemos pasar `nil` en el parámetro `error`.

También podemos leer de una URL con el método de clase `stringWithContentsOfURL:encoding:error:` de la clase `NSString`. Por ejemplo:

```
NSURL* url = [NSURL
    URLWithString:@"http://www.macprogramadores.org"];
NSError* error;
NSString* texto = [NSString stringWithContentsOfURL:url
                                                encoding:NSUTF8StringEncoding]
```

¹⁷ Los objetos de tipo `NSError` se explican con más detalle en el apartado 6.5 del Tema 5.

```
error:&error];
if (texto==nil) {
    NSLog(@"Error leyendo URL %@: %@",url
          , [error localizedFailureReason]);
}
```

En caso de que no conozcamos el formato de codificación usado por un fichero o URL podemos usar los métodos:

```
stringWithContentsOfFile:usedEncoding:error:
stringWithContentsOfURL:usedEncoding:error:
```

los cuales, mediante técnicas estadísticas, intentan adivinar el formato utilizado, y en `usedEncoding` nos devuelven el formato que creen más probable para el fichero.

7.3.2. Escribir cadenas

Escribir cadenas es más sencillo, debido a que siempre conocemos el formato de codificación a usar. Para escribir en un fichero disponemos del método `writeToFile:atomically:encoding:error:`, donde el parámetro `atomically` es un booleano donde indicamos si queremos que se grabe primero en un fichero intermedio para asegurar la atomicidad, es decir, asegurar que el fichero queda bien grabado o no se graba en absoluto.

8. Tipos de datos de 32 y 64 bits

Existen varios modelos para el tamaño de los tipos de datos fundamentales C que usan los compiladores. Objective-C también utiliza los tipos de datos fundamentales de C.

El compilador GCC permite generar código tanto de 32 bits como de 64 bits y sigue los convenios ILP32 y LP64 para los tamaños de sus tipos de datos fundamentales. Como Objective-C usa GCC, Objective-C por defecto utiliza estos convenios. El convenio ILP32 significa que al generar binarios de 32 bits, los tipos **Integer**, **Long** y **Pointer** son de 32 bits. El convenio LP64 significa que al generar binarios de 64 bits, los tipos **Long** y **Pointer** son de 64 bits. La Tabla 4.1 recopila los tamaños de los tipos fundamentales para estos convenios.

	ILP32	LP64	ILP64
Tipo	Tamaño	Tamaño	Tamaño
char	8 bits	8 bits	8 bits
short	16 bits	16 bits	16 bits
int	32 bits	32 bits	64 bits
long	32 bits	64 bits	64 bits
long long	64 bits	64 bits	64 bits
puntero	32 bits	64 bits	64 bits

Tabla 4.1: Tamaños de los tipos fundamentales para los convenios ILP32 y LP64

Aunque el compilador de GCC sigue el modelo LP64, Apple ha elegido el modelo ILP32 para los binarios Objective-C de 32 bits y el modelo ILP64 para los binarios de 64 bits de Objective-C. Debido a que GCC sigue el modelo LP64 Apple ha propuesto el uso de los tipos **NSInteger** y **NSUInteger** para representar las variables enteras de forma que puedan tener su tamaño correcto tanto en binarios de 32 bits como en binarios de 64 bits. Para ello, en el fichero **NSObjCRuntime.h** han definido los tipos **NSInteger** y **NSUInteger** como:

```
#if __LP64__
typedef long NSInteger;
typedef unsigned long NSUInteger;
#else
typedef int NSInteger;
typedef unsigned int NSUInteger;
#endif
```

En general, Apple recomienda utilizar los tipos **NSInteger** y **NSUInteger** (en lugar de **int** y **unsigned int**) con el fin de que el mismo código fuente compile correctamente tanto en binarios de 32 bits como en binarios de 64 bits.

Tenga en cuenta que la mayoría de los métodos Objective-C han sido cambiados para recibir y devolver tipos `NSInteger` y `NSUInteger`, pero nuestro programa todavía puede seguir usando el tipo `int` de GCC, en cuyo caso los enteros, tanto en binarios de 32 bits como en binarios de 64 bits, serán siempre de 32 bits.

Apple recomienda utilizar el nombre *int* para referirse a variables que en ambos binarios sean de 32 bits, y el nombre *integer* para referirse a variables que serán de 32 bits en binarios de 32 bits y de 64 bits en binarios de 64 bits. De hecho, algunas clases proporcionan ambos métodos. Por ejemplo, `NSString` tiene los métodos:

- `(int)intValue`
- `(NSInteger)integerValue`

El primero devuelve el contenido de la cadena representado como `int` y el segundo devuelve el contenido de la cadena representado como un `NSInteger`.

9. Declaraciones adelantadas

Normalmente un fichero de interfaz, antes de declarar la interfaz de la clase, importa la clase de la que deriva. De esta forma tiene información suficiente sobre los miembros (variables de instancia y métodos) que va a heredar.

Cuando una clase contiene punteros a objetos de los que no deriva (es decir, hace una agregación de objetos de otra clase), en el fichero de interfaz se puede hacer una **declaración adelantada** de la clase de los objetos agregados, y en el fichero de implementación importar el fichero de interfaz de las clases agregadas.

Para hacer una declaración adelantada de una clase se usa la palabra reservada `@class` como muestra el Listado 3.5.

```
/* Segmento.h */
#import <Foundation/NSObject.h>

@class Punto;

@interface Segmento : NSObject {
    Punto* desde;
    Punto* hasta;
}
- initDesde:(Punto*)paramDesde hasta:(Punto*)paramHasta;
- (double)longitud;
- (void)dealloc;
@end
```

Listado 3.5: Ejemplo de declaración adelantada

Después, en el fichero de implementación se podrá importar la interfaz de la declaración adelantada como muestra el Listado 3.6.

Estrictamente hablando en Objective-C nunca es necesario que una interfaz importe la interfaz de sus clases agregadas ya que las variables de instancia y parámetros de los métodos en Objective-C siempre son punteros, con lo que siempre se pueden hacer declaraciones adelantadas.

Sin embargo en el caso de C++, como además de la relación de agregación existe la relación de composición, es decir las variables de instancia de la clase pueden no ser punteros a objetos, no siempre es cierto que se puedan hacer declaraciones adelantadas en la interfaz de la clase. En consecuencia en C++ a veces pueden surgir relaciones de dependencia, entre los ficheros de cabecera, difíciles de resolver.

A pesar de todo, en Objective-C es menos recomendable usar declaraciones adelantadas de las clases con las que existe una relación de agregación, que

importar el fichero de cabecera. Esto se debe a que elimina la independencia del fichero de interfaz donde hacemos la declaración adelantada, es decir, si ahora otro programador importa el fichero de interfaz del Listado 3.5, posiblemente encontrará extraños errores de compilación debido a que no está definida la clase `Punto`.

```
/* Segmento.m */
#import <math.h>
#import "Segmento.h"
#import "Punto.h"

@implementation Segmento
- initDesde:(Punto*)paramDesde hasta:(Punto*)paramHasta {
    if (self = [super init]) {
        desde = [paramDesde retain];
        hasta = [paramHasta retain];
    }
    return self;
}
- (double)longitud {
    NSInteger dist_x = [hasta x] - [desde x];
    NSInteger dist_y = [hasta y] - [hasta x];
    return sqrt(dist_x*dist_x + dist_y*dist_y);
}
- (void)dealloc {
    if (desde!=nil)
        [desde release];
    if (hasta!=nil)
        [hasta release];
    [super dealloc];
}
@end
```

Listado 3.6: Fichero de implementación que importa una clase con declaración adelantada

```
/* Segmento.h */
#import <Foundation/NSObject.h>
#import "Punto.h"

@interface Segmento : NSObject {
    Punto* desde;
    Punto* hasta;
}
- initDesde:(Punto*)paramDesde hasta:(Punto*)paramHasta;
- (double)longitud;
- (void)dealloc;
@end
```

Listado 3.7: Declaración adelantada de clase agregada con importación

En general, cuando la relación de agregación sea intrínseca (véase apartado 3 del Tema 2), es decir, proceda de relaciones de objetos contenidos donde hay un **objeto contenedor** (p.e. `Segmento`), y un **objeto contenido** (p.e.

Punto) es mejor no hacer una declaración adelantada de la clase contenida, sino únicamente importar su fichero tal como muestra el Listado 3.7.

Por el contrario, cuando la relación entre clases sea extrínseca es mejor usar declaraciones adelantadas, pero debemos importar el fichero de cabecera de la otra clase una vez que declaramos la interfaz de la nuestra. El Listado 3.8 y Listado 3.9 muestran la forma correcta de declarar los ficheros de cabecera de dos clases Punto y Complejo entre las que existen métodos para transformar de una a otra, pero entre ellas ninguna domina sobre la otra, es decir, la relación es extrínseca.

```
/* Punto.h */
#import <Foundation/NSObject.h>

@class Complejo;

@interface Punto : NSObject {
    @public
    NSInteger x;
    NSInteger y;
}
-initWith: (NSInteger)paramX Y: (NSInteger)paramY;
-(Complejo*)comoComplejo;
@end

#import "Complejo.h"
```

Listado 3.8: Declaración adelantada de clase agregada con relación extrínseca

```
/* Complejo.h */
#import <Foundation/NSObject.h>

@class Punto;

@interface Complejo : NSObject {
    @public
    NSInteger real;
    NSInteger imaginario;
}
- initReal: (NSInteger)paramReal
imaginario: (NSInteger)paramImaginario;
-(Punto*)comoPunto;
@end

#import "Punto.h"
```

Listado 3.9: Declaración adelantada de clase agregada con relación extrínseca

Tema 4

Profundizando en el lenguaje

Sinopsis:

En este tema comenzaremos viendo cómo implementa la herencia Objective-C. Después veremos con más detalle qué son las clases y metaclasses, y cómo las modela Objective-C en forma de objetos.

En la segunda parte del tema veremos cuál es el ciclo de vida de un objeto Objective-C. Para acabar este tema veremos algunas estructuras del lenguaje más avanzadas, como son las categorías, los protocolos, las extensiones y los clusters de clases.

1. Herencia

La **herencia** es una operación aditiva. Cuando definimos una **clase derivada** nos basamos en otra **clase base** de la que heredamos sus métodos y variables de instancia. La nueva clase solamente añade o modifica lo heredado, pero no duplica el código heredado.

Un ejemplo de cómo realizar la herencia en Objective-C lo podemos ver en el Listado 3.5. En este caso estamos indicando que la clase `Segmento` hereda de la clase `NSObject` (la clase base de todas las clases Cocoa). Para ello, el nombre de la clase derivada en la interfaz se precede por dos puntos, y se pone después el nombre de la clase base. Tenga en cuenta que la clase base se indica sólo en el fichero de interfaz, no en el de implementación (véalo por ejemplo en el Listado 3.6).

1.1. La clase raíz

En Objective-C todas las clases derivadas, directa o indirectamente, derivan o bien de la clase base de Cocoa `NSObject`, o bien de la clase base de GNU `Object`.

Estas clases, llamadas **clases raíz**, implementan el runtime de Objective-C. En principio nada nos impide crear nuestra propia clase raíz personalizada, pero en general la creación de una clase raíz es una operación complicada que no se recomienda.

1.2. Redefinir métodos y variables de instancia

Objective-C nos permite **redefinir** (override) métodos en la clase derivada volviendo a definir un método con el mismo prototipo. Para redefinir un método heredado no hace falta declararlo en la interfaz de la clase derivada (aunque es conveniente por claridad), basta con declararlo en la implementación. Esta es una diferencia con C++ donde para poder redefinir un método es necesario declararlo tanto en la interfaz como en la implementación.

En Objective-C no se pueden redefinir las variables de instancia, es decir, no podemos volver a declarar una variable de instancia con el mismo nombre (aunque tenga distinto tipo) en la clase derivada. A diferencia de C++, esta regla es cierta incluso si la variable de instancia que intentamos redefinir tenía el modificador de acceso privado en la clase base.

1.3. Los receptores especiales `self` y `super`

Un **receptor** es un objeto al cual se envía un mensaje. Cuando se envía un mensaje a un receptor, primero se busca el método en los métodos de instancia del objeto, y luego en los métodos de instancia de las clases base hasta llegar a la raíz. Objective-C tiene dos receptores especiales llamados `self` y `super` que vamos a comentar en este apartado.

El receptor `self` no es más que un puntero a objeto estático tipificado con la clase sobre la que se ejecuta el método (el equivalente a `this` en C++ o Java). En consecuencia, el receptor `self` sólo tiene sentido utilizarlo dentro de un método.

El receptor `super` proporciona una forma de saltarse los métodos redefinidos e ir directamente al método de la clase base, es decir, evita que se busque el método en la clase donde nos encontramos, y comienza la búsqueda del método en la clase base.

Una de las aplicaciones de `super` es repartir la implementación de un método redefinido entre varias clases, por ejemplo podemos implementar el método `describe` de forma que primero se describan las variables de instancia de la clase derivada, y luego se llame a `describe` de la clase base para que se describan las variables de instancia de la base así:

```
- describe {
    .....
    return [super describe];
}
```

Sin embargo es más común llamar primero al método redefinido de la base y luego al de la derivada. Esto es lo que por ejemplo se acostumbra a hacer en `init` para que se inicialicen las variables de instancia de la base y de la derivada:

```
- init {
    if (self = [super init]) {
        .....
    }
    return self;
}
```

El receptor `super` es un l-value, es decir, no podemos asignarle un valor, pero por el contrario el receptor `self` sí que se puede modificar. Esta es una diferencia con C++ o Java donde `this` es de sólo lectura. En el apartado 6.2.3 veremos que es muy común modificar `self` dentro de `init` de la forma:

```
- init {
    if (self = [super init]) {
        .....
    }
    return self;
}
```

Donde `init` puede devolver `nil` en caso de que falle la construcción de la base, o bien el método `init` de la base puede cambiar el valor de `self`, por ejemplo por un proxy. En el apartado 8.9 estudiaremos el patrón de diseño proxy.

Otra diferencia importante entre `self` y `super` es que `self` se resuelve en tiempo de ejecución, mientras que `super` se resuelve en tiempo de compilación.

Es decir, `self` se refiere al objeto que está ejecutando el método actual. En tiempo de ejecución la clase de `self` puede ser la clase en la cual aparece la llamada a `self`, o una subclase.

Por el contrario, `super` se resuelve en tiempo de compilación, con lo que la búsqueda empieza en la clase base del método donde aparece `super`, y no en la clase base del objeto receptor (que podría ser una subclase).

Para aclarar esto supongamos que tenemos una jerarquía de clases como la de la Figura 4.1 e implementamos las clases A, B, C como muestra el Listado 4.1. Es decir, el método `saluda` está redefinido en B para llamar al de A, y en C no está redefinido.

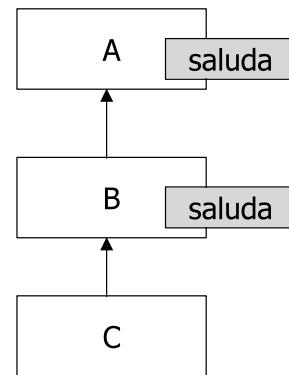


Figura 4.1: Ejemplo de jerarquía de clases

En este ejemplo, si `super` se resolviera en tiempo de ejecución, y en consecuencia `super` empezara la búsqueda en la clase base del receptor, el siguiente programa se metería en un bucle infinito:

```
C* c = [C new];
[c saluda];
```

Es decir, la segunda sentencia haría que el runtime empezara a buscar `saluda` en la base de C que es B, con lo que se ejecutaría el método de B, con lo que el runtime volvería a empezar la búsqueda de `saluda` a partir de B.

Como `super` se resuelve en tiempo de compilación, lo que ocurre realmente al ejecutar la segunda sentencia es que `super` hace que el runtime empiece a buscar en la base de `B` que es `A`, y se imprime el mensaje de saludo del ejemplo.

```
@implementation A
- (void)saluda {
    printf("Hola\n");
}
@end
@implementation B
- (void)saluda {
    [super saluda];
}
@end
@implementation C
@end
```

Listado 4.1: Ejemplo de redefinición de métodos

2. Objetos clase

2.1. Los objetos clase

La definición de una clase contiene varios tipos de información, como puedan ser: El nombre de la clase y de su superclase, los nombres de los métodos, su retorno, sus parámetros, y su implementación. Toda esta información es recopilada por el compilador y guardada en estructuras de datos que son usadas por el runtime de Objective-C.

El compilador crea un **objeto clase** por cada clase que tenga el programa, donde en tiempo de ejecución se almacena información sobre la clase. Estos objetos son inicializados por el runtime antes de que el programa empiece su ejecución, con lo que están siempre disponibles.

Para acceder al objeto clase disponemos de los métodos de clase y de instancia:

```
+ (Class)class
- (Class)class
```

El método de clase se ejecuta sobre el nombre de la clase, y se limita a devolver `self`, con lo que para obtener el objeto clase de `Punto` podemos hacer:

```
id c = [Punto class];
```

El método de instancia se puede ejecutar tanto sobre un **objeto de instancia**, como sobre un objeto clase. Por ejemplo, para obtener el objeto clase de un `Punto` podemos hacer:

```
Punto* p = [Punto new];
id c = [p class];
```

Pero también podemos ejecutar el método `class` directamente sobre el objeto clase:

```
id c2 = [c class];
```

Donde se cumple que `c==c2`. Esto se debe a que al recibir el mensaje `class` el objeto clase, primero busca su implementación en sus métodos (los métodos de clase), y luego en los métodos de la clase base (los métodos de instancia de la clase raíz). Como el método de clase `class` es un método del objeto clase, se ejecuta y devuelve `self`.

Hemos apuntado al objeto clase usando el puntero a objeto dinámico `id`, también podemos usar un puntero a objeto estático del tipo `Class` de la forma:

```
Class c = [p class];
Class c2 = [c class]; // c==c2
```

Observe que no debemos declarar las variables como `Class*`, sino como `Class`, ya que `Class` es un `typedef` definido como muestra el Listado 4.2.

Enviar mensaje a objetos clase es tan común que el lenguaje proporciona el atajo de escribir en el código fuente el nombre de la clase directamente. Esto es lo que hacemos cuando, por ejemplo, enviamos el mensaje de clase `new` a una clase:

```
Punto* p = [Punto new];
```

Que sería equivalente a haber hecho:

```
Punto* p = [[Punto class] new];
```

El nombre de la clase sólo se puede usar en el código fuente para enviar mensajes al objeto clase, no podemos usarlo para obtener referencias al objeto clase. Por ejemplo la siguiente sentencia produciría un error de compilación:

```
Class c = Punto; // Error de compilación
```

Es importante remarcar que los objetos clase son objetos Objective-C normales a los que podemos enviar mensajes. La única diferencia con el resto de los objetos Objective-C es que los crea el compilador.

```
struct objc_class {
    struct objc_class *isa; // Metaclass pointer
    struct objc_class *super_class;
    const char *name;
    long version;
    long info;
    long instance_size;
    struct objc_ivar_list *ivars;
    struct objc_method_list **methodLists;
    struct objc_cache *cache;
    struct objc_protocol_list *protocols;
};

typedef struct objc_class* Class;
typedef struct objc_object {
    Class isa; // Class pointer
} *id;
```

Listado 4.2: Declaración del tipo `Class` en `objc/objc.h`

2.2. La variable de instancia `isa`

En el apartado 2.2 del Tema 3 vimos que el puntero a objeto dinámico `id` era totalmente no restrictivo en el sentido de que podíamos apuntar a cualquier objeto. Pero en realidad no todos los objetos son iguales: Un `Punto` no tiene los mismos métodos que un `Rectángulo`. En consecuencia en algún momento el puntero a objeto dinámico necesitará conocer información más específica del objeto al que apunta. Ya que el tipo `id` no proporciona esta información al compilador, la información sobre el objeto se deberá descubrir en tiempo de ejecución.

Esto es posible gracias a que todo objeto Objective-C derivado de `NSObject` (o de `Object` de GNU) tiene la variable de instancia `isa` que apunta al objeto clase del objeto en cuestión. De esta forma un objeto `Rectángulo` podrá acceder a su objeto clase en tiempo de ejecución. En Objective-C los objetos apuntados con punteros a objeto dinámicos son tipificados en tiempo de ejecución.

Realmente `id` aparece en los ficheros de cabecera de Objective-C declarado como un `typedef` a un puntero a estructura con un único campo: La variable de instancia `isa`, la cual apunta al objeto clase (véase el Listado 4.2).

Como veremos en el apartado 2.5, el puntero `isa` también se usa para introspección, es decir para encontrar en tiempo de ejecución, entre otras co-

sas, el tamaño ocupado por las variables de instancia del objeto, los métodos, o la clase base del objeto.

En el apartado 4 veremos que los métodos de un objeto de instancia están almacenados en su objeto clase, de forma que para ejecutar un método sobre un objeto de instancia se usa el puntero `isa` para acceder al objeto clase y buscar en él (y después en sus clases base) el método a ejecutar.

2.3. Crear instancias de una clase

A los objetos clase también se les llama **factory object**, porque una de las funciones principales del objeto clase es crear nuevas instancias. Por ejemplo, para decir al objeto clase de `Punto` que cree un nuevo punto le mandamos el mensaje `alloc`:

```
Punto* p = [Punto alloc];
```

Este mensaje crea el objeto, y deja todas sus variables de instancia a cero, excepto la variable de instancia `isa` que apunta al objeto clase que creó al objeto. Para que el objeto sea útil necesitaremos enviar también al nuevo objeto el mensaje `init`:

```
Punto* p = [[Punto alloc] init];
```

Todas las clases tienen al menos el método de clase `alloc` y el método de instancia `init`, ya que lo heredan de la clase raíz.

2.4. Personalización con objetos clase

El hecho de que Objective-C cree objetos que representen las clases no es un capricho, proporciona grandes ventajas a la hora de crear patrones de diseño.

Podemos, por ejemplo, personalizar un objeto con una clase. Esto es lo que ocurre en la clase `NSMatrix`, una clase del Application Kit usada para crear tablas y a la cual la podemos pasar la clase de los objetos que va a contener. Por ejemplo, para crear un `NSMatrix` que contenga objetos de tipo `NSButtonCell` podemos usar el método `setCellClass:` de `NSArray` así:

```
[miarray setCellClass: [NSButtonCell class]];
```

2.5. Introspección

Como adelantamos en el apartado 2 del Tema 2, la **introspección** permite que en tiempo de ejecución descubramos información sobre la clase de un objeto. Para ello podemos preguntar a su objeto clase.

Las clases `Object` y `NSObject` proporcionan el método:

- `(BOOL)isMemberOfClass:(Class)aClass`

Que nos permite comprobar si el receptor es una instancia de la clase `aClass`.

Las clases `Object` y `NSObject` también proporcionan el método:

- `(BOOL)isKindOfClass:(Class)aClass`

Que de forma más general nos permite saber si el receptor es de la clase `aClass` o de alguna de sus clases base. Por ejemplo, para comprobar si un objeto `p` es de tipo `Punto` podemos hacer:

```
id p = ...;
if ([p isKindOfClass: [Punto class] ]) {
    .....
}
```

La clase `Object` de GNU, además del método `isKindOfClass:`, proporciona el método:

- `(BOOL)isKindOfClass:aClassObject;`

Que es equivalente a `isKindOfClass:.` Y el método:

- `(BOOL)isKindOfClassNamed:(const char*)aClassName;`

Que sirve para determinar si la clase del objeto receptor tiene el nombre dado.

En el apartado 4.4 del Tema 5 veremos cómo obtener información de introspección sobre los métodos de una clase.

2.6. Variables de clase

Ya comentamos, en el apartado 1.1 del Tema 3, que en Objective-C, a diferencia de C++ o Java, no existen variables de clase. Por esta razón los obje-

tos clase no tienen variables de instancia¹⁸. En consecuencia, igual que ocurre en C++ o Java, los objetos clase tampoco tienen acceso a las variables de instancia de las instancias de su clase.

En el apartado 1.1 del Tema 3 también vimos que la forma de implementar el equivalente a una variable de clase en Objective-C era crear una variable global o de módulo (con el modificador `static`) donde almacenar estos posibles valores.

2.7. Inicializar un objeto clase

Si una clase hace uso de variables globales o estáticas, puede necesitar que éstas estén inicializadas antes de que se empiece a utilizar la clase. C++ no dispone de una ayuda estándar para realizar estas inicializaciones¹⁹, con lo que se suelen hacer llamando a una función `InicializaXXX()` al principio de la función `main()`. En el caso de Java, estas inicializaciones se realizan en los bloques estáticos.

En Objective-C, aunque es el runtime (y no las aplicaciones) el encargado de crear los objetos clase, el runtime de Objective-C permite a las aplicaciones realizar inicializaciones en las clases antes de utilizarlas por primera vez. Para ello las aplicaciones deben implementar en la clase que quieran inicializar un método de clase con el nombre `initialize`, el cual es ejecutado por el runtime de Objective-C en cada objeto clase antes de que la clase reciba ningún otro mensaje. Lógicamente, si no necesitamos inicializar un objeto clase, no es necesario implementar este método en su clase.

El runtime garantiza que si existe jerarquía de clases, se enviará el mensaje primero a la clase base y luego a la derivada. Esta regla produce un efecto lateral debido a la relación de herencia entre clases, ya que si una clase derivada no implementa el método de clase `initialize`, este mensaje se buscará en la clase base, incluso aunque la base haya recibido ya el mensaje `initialize`. En consecuencia, es responsabilidad de toda clase que implemente `initialize` el garantizar que las inicializaciones se realicen sólo una vez. Para ello es habitual usar un flag de la forma:

```
+ (void)initialize {
    static BOOL inicializada = NO;
    if (inicializada==NO) {
        // Realizar inicializaciones
        .....
    }
}
```

¹⁸ En el apartado 4 veremos que realmente los objetos clase tienen dos variables de instancia internas: `isa` y `super_class`.

¹⁹ En el caso de las GCC se pueden definir las funciones `_init()` y `_fini()` que se ejecutan respectivamente antes y después de ejecutar la función `main()`.

```
}
```

Al igual que en los bloques estáticos Java, el runtime de Objective-C sólo ejecuta el método `initialize` si hacemos uso de la clase, en caso contrario el método `initialize` nunca se ejecutará. Para que el runtime considere que hemos hecho uso de una clase debemos de, o bien enviar un mensaje a la clase, o bien instanciar un objeto de esta clase. Si por ejemplo, sólo hacemos un `sizeof()` de la clase no se ejecuta la inicialización.

Es importante destacar que lo único que garantiza el runtime es que el método `initialize` se ejecuta antes de usar cualquier otro método de la clase, pero no se garantiza que los métodos `initialize` se ejecuten antes que la función `main()`. De hecho en la implementación actual de Objective-C el método `initialize` sólo se ejecuta cuando el programa pasa por primera vez por una zona de programa que hace uso de la clase.

3. Otros receptores especiales

Sabemos que un receptor es el objeto al que podemos enviar mensajes. En el apartado 1.3 vimos que, además de los objetos receptores normales, existían receptores especiales como `self` y `super`. En este apartado vamos a comentar otros dos receptores especiales.

Un receptor especial es el nombre de una clase. Por ejemplo:

```
Class c = [Punto class];
```

Hemos visto en el apartado 2.1 que cuando enviamos un mensaje a un nombre de clase, realmente estamos enviando el mensaje al objeto clase que representa a la clase en tiempo de ejecución.

El otro receptor especial es `nil`. En Objective-C no es un error enviar un mensaje a `nil`, que es equivalente a `NULL`, es decir, un objeto sin inicializar.

La forma en que se comporta el runtime de Objective-C cuando enviamos un mensaje a `nil` es la siguiente²⁰:

- En el caso de que el método receptor del mensaje tenga retorno `void`, no ocurre nada.
- En el caso de que el método receptor del mensaje tenga retorno `NSInteger`, `NSUInteger` o `float`, se devuelve cero.

²⁰ En el apartado 2.6 del Tema 5 veremos que cuando enviamos un mensaje a un objeto, el runtime de Objective-C realmente lo que hace es buscar el receptor del mensaje usando la función de runtime `objc_msgSend()`, la cual puede detectar que ha recibido `nil` como receptor, y no fallar.

- En el caso de que el método receptor del mensaje retorne un puntero, se devuelve `NULL`.
- En el caso de que el método retorne un tipo más grande (p.e. `double`, `long long`, `struct`, `union`) el retorno está indefinido y debemos de evitar que el código que precede a la llamada dependa de este retorno.

Objective-C implementó esta peculiaridad para evitar que se produjera un error en tiempo de ejecución si enviábamos un mensaje a `nil` (como ocurre por ejemplo en C++ o Java). De esta forma no tenemos que comprobar el valor de un puntero a objeto antes de enviar un mensaje al objeto, lo cual es especialmente útil en las cadenas de mensajes, como por ejemplo:

```
id p = [[Punto alloc] init];
```

Ahora, si `alloc` devuelve `nil`, el método `init` no se ejecuta, y sólo tenemos que comprobar que la variable `p` vale `nil`, para saber que la llamada falló.

4. Ruta de un mensaje durante su ejecución

Ya vimos en el apartado 2.2 que todos los objetos derivados de `NSObject` (o de `Object`) tenían la variable de instancia `isa` para indicar su objeto clase. Como muestra la Figura 4.2, una de las funciones de esta variable de instancia es encontrar la **dispatch table** con los métodos a los que puede responder un objeto de esa clase.

En el caso de los objetos clase, éstos además tienen otra variable de instancia llamada `super_class` (véase Listado 4.2) que apunta a el objeto clase de la base. En el caso del objeto clase de `NSObject`, esta variable de instancia estará puesta a `nil` para indicar que no tiene clase base.

En el apartado 5 veremos qué valores toma la variable de instancia `isa` en el caso de los objetos clase.

Esta información nos es suficiente para estudiar ahora cuál es la ruta que sigue un mensaje desde que se recibe por parte de un objeto de instancia, hasta que se encuentra el método sobre el que ejecutarlo.

En el apartado 2 del Tema 2 adelantamos que una diferencia entre Objective-C y C++ es que en Objective-C cada método a ejecutar se busca en una tabla, en vez de disponer de un puntero al método a ejecutar (métodos no virtuales de C++), o usar una v-table a nivel de clase para buscarlo (métodos virtuales C++). También dijimos que, aunque buscar un método en una tabla aumentaba el coste de ejecución de un método, la búsqueda mantenía un coste lineal que era fácilmente tolerable, y más gracias a los mecanismos de caché que implementa el sistema de búsqueda de métodos del runtime de Objective-C.

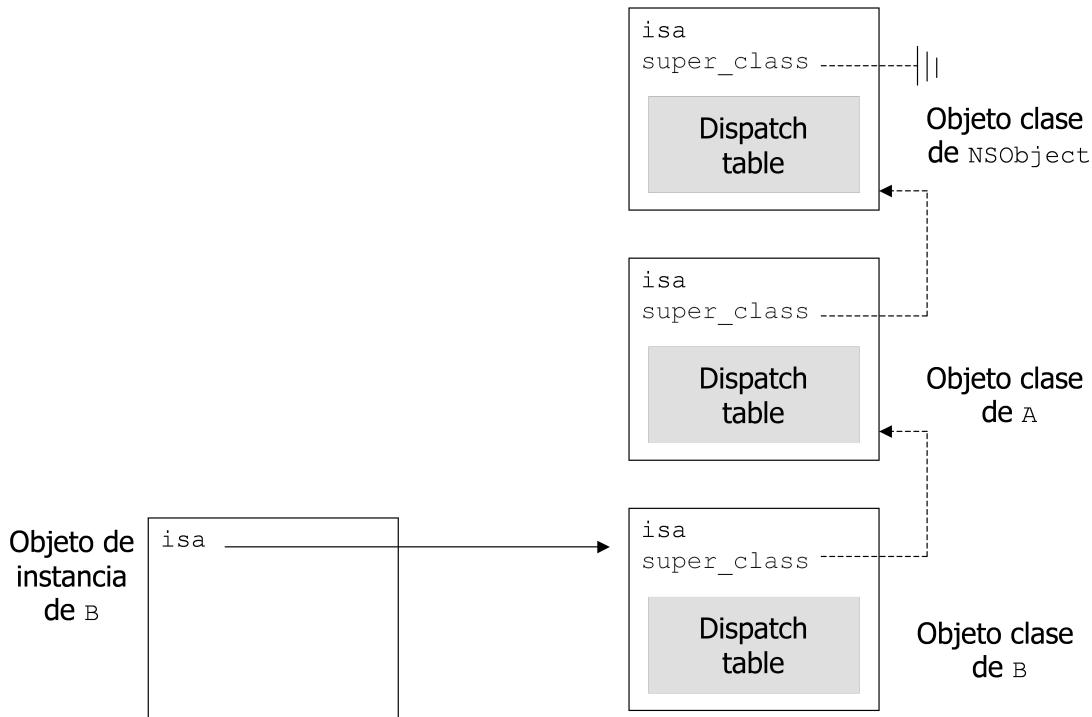


Figura 4.2: `isa` y `super_class` en objetos de instancia y de clase

A continuación vamos a detallar cuál es el algoritmo de búsqueda que sigue el runtime de Objective-C para encontrar el método que debe ejecutarse como respuesta a un mensaje que reciba un objeto de instancia. El mecanismo es idéntico cuando se trata de un objeto clase, pero vamos a posponer su explicación hasta que veamos los objetos metaclasses en el apartado 5:

1. El runtime accede al objeto clase del objeto de instancia sobre el que se ejecuta el método. Para ello usa la variable de instancia `isa` del objeto de instancia.
2. El runtime busca el método en la dispatch table del objeto clase.
 - a. Si encuentra el método lo ejecuta.
 - b. En caso contrario, el runtime usa la variable de instancia `super_class` del objeto clase para encontrar el objeto clase base, y repite el proceso de búsqueda del paso 2.
3. En caso de que el objeto clase no tenga base (es decir, `super_class` valga `Nil`)²¹, es que estamos en el objeto clase de `NSObject`, y como se explica en el apartado 3 del Tema 10, se hace forwarding del mensaje llamando al método `forward:::`.
4. Si el forwarding también falla se produce una excepción que, si no se captura, termina el programa.

²¹ En el apartado 2.2 del Tema 3 vimos que el símbolo `Nil` se acostumbra a usar para referirnos a punteros a objetos clase y objetos metaclasses, mientras que `nil` se suele usar para referirnos a objetos de instancia.

En el apartado 2.6 del Tema 5 veremos que la función encargada de realizar esta búsqueda en Objective-C se llama `objc_msgSend()`, y forma parte del runtime de Objective-C. Además esta función implementa un sistema de caché que acelera la búsqueda de métodos cuando se llama varias veces al mismo método.

5. Objetos metaclasses

5.1. Objetos de instancia, de clase, y de metaclas

Si los métodos de instancia están almacenados en el objeto clase: ¿Dónde están almacenados los métodos de clase?. La respuesta es que los objetos clase tienen otros objetos encargados de almacenar sus métodos a los que se conoce como **objetos metaclasses**.

Al igual que pasa con los objetos de instancia, los objetos clase utilizan su variable de instancia `isa` para apuntar a sus objetos metaclasses (véase Figura 4.3). Este diseño permite al runtime de Objective-C procesar llamadas a métodos de clase de la misma forma que se procesan las llamadas a métodos de instancia sobre objetos de instancia: Buscando el método a partir de la variable de instancia `isa` del objeto receptor.

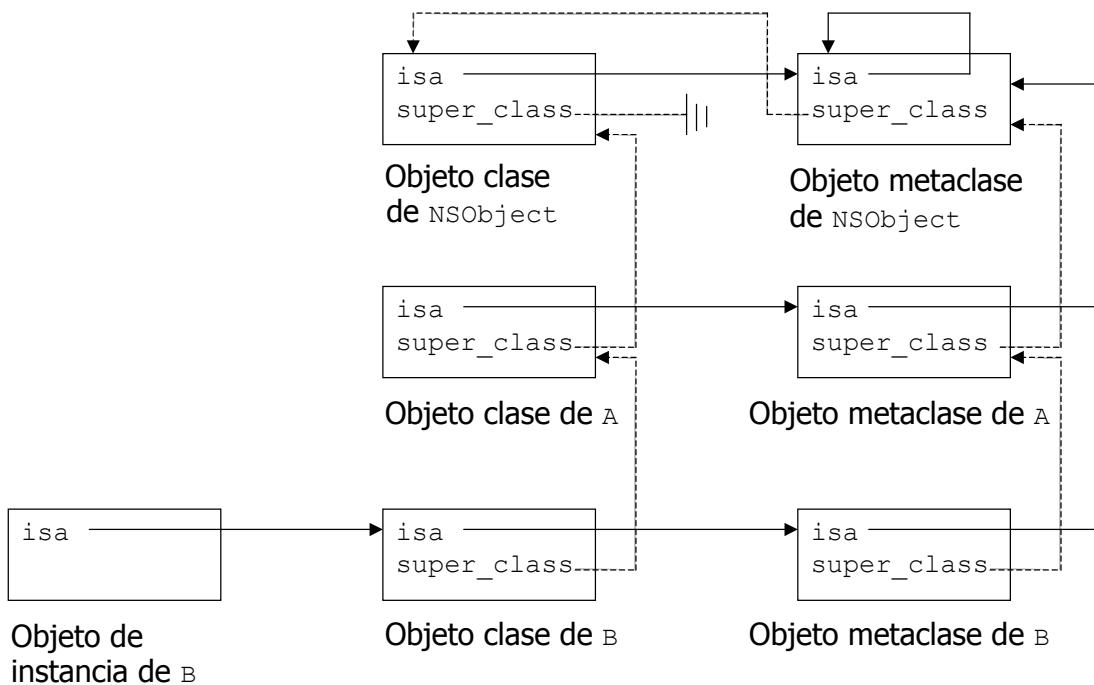


Figura 4.3: Objetos de instancia, de clase y de metaclas

Los objetos metaclasses son idénticos en estructura a los objetos clase, sin embargo su variable de instancia `isa` apunta siempre al objeto metaclase

raíz. Esta asimetría evita que se produzca una explosión infinita de meta-metaclasses.

Gracias a esta asimetría la metaclass raíz se puede interpretar como la meta-metaclass de todos los objetos. Debido a que las aplicaciones no trabajan nunca directamente con la metaclass, éste no es un problema práctico.

5.2. Obtener la metaclass de una clase

Reacuérdate que en el apartado 2.1 vimos que el método `class`, cuando lo ejecutábamos tanto sobre un objeto de instancia como sobre un objeto clase, devolvía el objeto clase correspondiente. Luego si intentamos obtener el objeto metaclass así:

```
Class clase_punto = [Punto class];
Class metaclass_punto = [clase_punto class];
```

En `metaclass_punto` acabaremos obteniendo el objeto clase (y no el objeto metaclass) de la clase `Punto`. Esto se debe a que al existir el método de clase `class`, se ejecuta el método de clase (que devuelve `self`), y no llega a ejecutarse el método de instancia `class` (que devuelve `isa`).

Luego para obtener el objeto metaclass vamos a tener que usar una de estas dos formas:

La primera es acceder directamente a la variable de instancia `isa` del tipo `objc_class` del Listado 4.2 de la forma:

```
Class clase_punto = [Punto class];
Class metaclass_punto = clase_punto->isa;
```

Si ahora volviéramos a ejecutar:

```
metaclass_punto = metaclass_punto->isa;
```

Obtendríamos el objeto metaclass raíz, y si lo ejecutáramos una vez más volveríamos a obtener de nuevo el objeto metaclass raíz (véase Figura 4.3).

La otra forma de obtener tanto la clase como la metaclass de una clase es usando respectivamente las funciones del runtime de Objective-C:

```
id objc_getClass(const char* aClassName);
id objc_getMetaClass(const char* aClassName);
```

Ambas funciones reciben como parámetro una cadena de caracteres con el nombre de la clase, y devuelven respectivamente un puntero a la clase y un puntero a la metaclass.

Por ejemplo, para obtener la metacalse de `Punto` hacemos:

```
Class metaclase_punto = objc_getMetaClass("Punto");
```

5.3. La variable de instancia `super_class` en los objetos clase y metacalse

La Figura 4.3 muestra otra simetría entre los objetos clase y los objetos metacalse respecto a la variable de instancia `super_class` (recuérdese que los objetos de instancia no tienen la variable de instancia `super_class` por no ser objetos que representen clases).

La diferencia está en que mientras que la superclase del objeto clase raíz es `Nil`, la superclase del objeto metacalse raíz es el objeto clase raíz, es decir, esto se puede interpretar como que la clase raíz (`NSObject`) actúa como clase base de todas las clases²², incluida la metacalse raíz.

5.4. Métodos de la clase raíz en los objetos clase

Como muestra la Figura 4.3, tanto los objetos de instancia, de clase y de metacalse derivan de la clase raíz. En consecuencia, cuando cualquiera de estos objetos recibe un mensaje que no entiende, el runtime empieza a buscar una implementación por la jerarquía de clases hasta llegar a la clase raíz. Esto hace que todos los métodos implementados en la clase raíz sean especiales respecto a que pueden ser ejecutados tanto sobre objetos de instancia, como sobre objetos clase, es decir, su programación está preparada para aceptar el ser llamados como métodos de instancia y como métodos de clase.

El método `class` que vimos en el apartado 2.1 podía ser llamado sobre una instancia, en cuyo caso devuelve el puntero `isa`, como sobre una clase, en cuyo caso devuelve `self`. Dijimos que esto se debía a que al recibir el objeto clase el mensaje `class`, el método de clase `class` se encuentra antes que el método de instancia `class`, ya que el método de instancia está en la dispatch table del objeto metacalse de `NSObject`.

En el apartado 4 vimos cuál era la ruta que se seguía para la ejecución de un método. Comentamos que el runtime sigue un proceso similar tanto en los objetos de instancia como en los objetos clase:

²² No se confunda con el hecho (que comentamos en el apartado 5.1) de que la metacalse raíz es la meta-metacalse de todas las clases.

1. Lo primero que hace el runtime es usar la variable de instancia `isa` para subir un nivel y acceder al objeto que actuaba como clase del que recibe el mensaje.
2. El runtime usa la dispatch table del objeto clase para buscar el método.
 - a. Si el runtime encuentra el método en la dispatch table lo ejecuta.
 - b. En caso contrario, el runtime usa la variable de instancia `super_class` del objeto clase para encontrar su objeto clase base, y se repite el paso 2.

Si observamos la Figura 4.3 podemos estudiar con más detenimiento cuál es el proceso de búsqueda que sigue el runtime para ejecutar la siguiente sentencia de ejemplo:

```
A class;
```

El proceso de búsqueda es el siguiente:

- Como `A` representa al objeto clase de `A`, lo primero que hace el runtime (paso 1) es usando el puntero `isa` acceder al objeto metaclas de `A`.
- El runtime busca en la dispatch table del objeto metaclas de `A` el método `class`, y no lo encuentra, ya que los objetos metaclas (excepto el objeto metaclas de `NSObject`) no implementan `class` en su dispatch table, sólo lo implementan las dispatch tables de los objetos clase.
- El runtime usa el puntero `super_class` del objeto metaclas de `A` para subir un nivel.
- El runtime busca ahora en la dispatch table del objeto metaclas de `NSObject` el método `class`, y esta vez encuentra la implementación de `class` como método de clase.
- Al haberlo encontrado lo ejecuta sobre el objeto clase de `A`.
- El método `class` se limita a devolver `self`, con lo que devuelve un puntero al objeto clase `A`.

En caso de no haber encontrado el método `class` en la dispatch table del objeto metaclas de `NSObject`, el runtime hubiera vuelto a usar el puntero `super_class` para acceder al objeto clase de `NSObject`, y hubiera encontrado otra implementación distinta de `class`. En concreto hubiera encontrado la implementación de `class` como método de instancia.

6. Ciclo de vida de un objeto

Todas las clases Objective-C deben implementar métodos que manejen el ciclo de vida del objeto: Creación, inicialización, copia y destrucción. Este apartado describe los métodos usados durante el ciclo de vida de un objeto Objective-C, bien sean objetos de librerías, o bien sean nuevos objetos que quiera crear el programador.

Tanto la clase `Object` como `NSObject` proporcionan los siguientes métodos para gestionar su ciclo de vida:

```
+ initialize  
+ alloc  
+ new  
- init  
- copy
```

Además, la clase `Object` añade los siguientes métodos:

```
- shallowCopy  
- deepCopy  
- deepen  
- free
```

Mientras que la clase `NSObject` añade el método:

```
- dealloc
```

Estos serían los métodos estándar que una clase debe proporcionar, pero además, como veremos en el siguiente apartado, las clases pueden tener otros métodos de creación e inicialización que reciban parámetros que guíen la creación o inicialización del objeto.

Para gestionar la vida de un objeto es importante, tanto saber llamar a los métodos, como saber implementarlos. Estos dos aspectos se tratan por separado en los siguientes subapartados.

6.1. Creación e inicialización

En C++ y Java la reserva de memoria la realiza el operador `new`²³, mientras la inicialización de un objeto se realiza en el constructor del objeto. En Objective-C, cada paso de la operación de construcción de un objeto es realizada por un método distinto. La creación es realizada por el método de clase `alloc`, y

²³ En C++ la reserva de memoria también se puede hacer en la pila (variables locales), o en el segmento de datos (variables globales).

la inicialización es realizada por el método de instancia `init`. Hasta que ambas operaciones no se han realizado el objeto no se considera completamente funcional:

```
id obj1 = [[Rectangulo alloc] init];
```

El método `alloc` reserva memoria suficiente para todas las variables de instancia del objeto, y se encarga de poner todas las variables de instancia a cero, excepto la variable de instancia `isa` que apunta al objeto clase que creó el objeto de instancia.

Como ya hemos visto, debido a que es muy común combinar la creación e inicialización del objeto, tanto `Object` como `NSObject` proporcionan el método de clase:

```
+ (id) new
```

que realiza ambas operaciones a la vez.

Aunque no forman parte de la clase raíz, muchas clases añaden nuevos métodos de creación e inicialización que deben tener nombres de la forma `allow...` y `init....`. Estos métodos permiten parametrizar la creación o inicialización del objeto. Por ejemplo, la clase `Object` proporciona el método de creación:

```
+ allocFromZone: (void*) zone
```

Mientras que `NSObject` proporciona el método:

```
+ allocWithZone: (void*) zone
```

Como veremos en el apartado 2 del Tema 10, estos métodos permiten indicar la zona de memoria donde crear el objeto. Otro ejemplo es el método `initWithFrame:`, de la clase `NSView`, que permite indicar que zona de la ventana se asocia a la vista.

6.1.1. El objeto devuelto

El método de inicialización es responsable de devolver un objeto inicializado, pero el objeto devuelto puede ser diferente al que recibió desde el método de creación. Esto ocurre al menos en dos circunstancias:

Un caso es cuando la inicialización del objeto no se pudo llevar a cabo. En este caso el método de inicialización debe devolver `nil`. Por ejemplo, la clase `NSString` tiene el método:

```
- (id)initWithContentsOfFile:(NSString*)path
    encoding:(NSStringEncoding)enc
    error:(NSError**)error
```

El cual devuelve `nil` en caso de que el fichero pasado en `path` no se pueda leer. En este caso es responsabilidad del método `initWithContentsOfFile:encoding:error:` liberar la memoria reservada por `alloc`.

El otro caso es cuando el método de inicialización sustituye el objeto por un proxy o por otro tipo especial de objeto.

Debido a que el método de inicialización puede devolver un objeto distinto al creado por el método de creación, es importante que el programa use el objeto devuelto por el método de inicialización (y no por el de creación). Por ejemplo el siguiente programa es inseguro:

```
id c = [Circulo alloc];
[c init];
[c pinta];
```

Ya que estamos usando el objeto devuelto por del método de creación, y no por el de inicialización. La forma correcta de llamarlo sería:

```
id c = [Circulo alloc];
c = [c init];
[c pinta];
```

O Bien:

```
id c = [[Circulo alloc] init];
[c pinta];
```

Además siempre que un método de inicialización pueda devolver `nil`, debemos de comprobar su retorno de la forma:

```
id c = [[Circulo alloc] init];
if (c)
    [c pinta];
else
    ...
```

6.1.2. Métodos factory

En Objective-C, algunas clases definen métodos de clase que combinan ambos pasos (creación e inicialización) para devolver un objeto ya inicializado. Estos métodos deben tener el nombre `+clase....`, donde `clase` es el nombre sin prefijo de la clase sobre la que se ejecutan, y también de la clase del

objeto que devuelven. Por ejemplo, `NSString` tiene los métodos (entre otros):

```
+ (id)stringWithCString:(const char*)cString
                  encoding:(NSStringEncoding)enc
+ (id)stringWithFormat:(NSString*)format, ...
+ (id)stringWithContentsOfFile:(NSString*)path
                           usedEncoding:(NSStringEncoding*)enc
                           error:(NSError**)error
```

O la clase `NSArray` también combina creación e inicialización en métodos como:

```
+ (id)array
+ (id)arrayWithObject:(id)anObject
+ (id)arrayWithObjects:(id)firstObj, ...
```

Como volveremos a recordar en el apartado 3.6 del Tema 5, no debemos liberar (con `release`) las instancias devueltas por los **métodos factory** sino que las libera automáticamente el sistema de gestión de memoria cuando acabemos de trabajar con el objeto.

Estos métodos factory también son muy útiles cuando la creación depende de la inicialización. Por ejemplo, si los datos a crear dependen del contenido de un fichero, puede no conocerse el tamaño de memoria a reservar hasta que no se conoce qué fichero se quiere leer.

Otro caso donde resultan muy útiles los métodos factory es cuando la memoria reservada por el método de creación puede tener que ser liberada inmediatamente por el método de inicialización, como ocurría con el método `initWithContentsOfFile:encoding:error:` en el apartado anterior. En este caso no tiene sentido reservar la memoria si los parámetros de inicialización son inválidos, y por eso es más recomendable usar el método factory:

```
+ (id)stringWithContentsOfFile:(NSString*)path
                           usedEncoding:(NSStringEncoding*)enc
                           error:(NSError**)error
```

6.1.3. Objetos singleton

Los **objetos singleton** son objetos que sólo se pueden instanciar una vez. Algunas clases de Cocoa, como `NSApplication` o `NSWorkspace` son ejemplos de objetos singleton.

Para almacenar un objeto singleton, se suele usar una variable global estática que inicialmente está a `nil`. Para crear el objeto se proporciona un método factory que comprueba que el objeto no se instancie más de una vez. A este método se le suele llamar **método factory singleton**.

Para evitar que el objeto se instancie más de una vez, el lenguaje debe de proporcionar un mecanismo de control. En el caso de C++ o Java, se declara como privado el constructor, con lo que el usuario se ve obligado a crear el objeto a través de un método factory singleton. En Objective-C lo que se hace es redefinir `allocWithZone:`, como vamos a ver a continuación, para que compruebe si el objeto ya está instanciado, si no está creado lo crea, y si ya está creado devuelve `nil` indicando que el intento de instanciar el objeto ha fallado.

Obsérvese que en C++ o Java se oculta el constructor, con lo que la única forma que tiene el usuario de crear el objeto es a través de su método factory. En Objective-C no se oculta el constructor, sólo se redefine el método `allocWithZone:`, con lo que, en principio, el usuario puede crear el objeto tanto usando `alloc` como usando un método factory. En cualquier caso, el método factory singleton siempre debería de existir en los objetos singleton, y es la forma recomendada de crear el objeto singleton en Objective-C.

En concreto, para crear una clase singleton necesitamos:

1. Declarar una variable estática inicializada a `nil`.
2. Declarar un método factory singleton que sólo cree el objeto durante la primera llamada.
3. Redefinir `allocWithZone:` para que sólo devuelva una instancia del objeto la primera vez que se le llame, y `nil` si ya se ha llamado previamente.
4. Redefinir `copyWithZone:` para que devuelva la instancia ya creada como copia.
5. Redefinir `retain`, `release` y `autorelease` para que no modifiquen la cuenta de referencias del objeto. Los objetos singleton siempre tienen su cuenta de referencias a uno.

El Listado 4.3 y Listado 4.4 muestran una clase `Supervisor` donde hemos seguido estos pasos para que `Supervisor` sea singleton. Cuando el objeto singleton no está creado, el método factory singleton, acaba llamando a `allocWithZone:` realiza la actualización de la variable global. El método `allocWithZone:` es el único punto donde se realiza la creación del objeto singleton, y la actualización de la correspondiente variable global. Debido a que varios hilos podrían intentar crear el objeto singleton a la vez, debemos proteger su creación en un bloque sincronizado mediante la directiva del compilador `@synchronized()`. En el apartado 7 del Tema 5 se explicará esta directiva.

```
#import <Foundation/Foundation.h>

@interface Supervisor : NSObject {
}
+ (Supervisor*)supervisor;
+ (id)allocWithZone:(NSZone*)zona;
- (id)copyWithZone:(NSZone*)zona;
- (id)retain;
- (void)release;
- (id)autorelease;
@end
```

Listado 4.3: Interfaz de objeto singleton

```
#import "Supervisor.h"

static Supervisor* singletonSupervisor = nil;

@implementation Supervisor
+ (Supervisor*)supervisor {
    if (singletonSupervisor==nil)
        return [[self alloc] init];
    else
        return singletonSupervisor;
}
+ (id)allocWithZone:(NSZone*)zona {
    @synchronized(self) {
        if (singletonSupervisor==nil)
            return singletonSupervisor =
                [super allocWithZone:zona];
    }
    return nil;
}
- (id)copyWithZone:(NSZone*)zona {
    return self;
}
- (id)retain {
    return self;
}
- (void)release {
    // No hace nada
}
- (id)autorelease {
    return self;
}
@end
```

Listado 4.4: Implementación de objeto singleton

Tenga en cuenta que de acuerdo al protocolo de creación de objetos singleton definido por Apple, la segunda llamada a `alloc` devolvería `nil` (indicando que la creación ha fallado), mientras que la segunda llamada al correspondiente método factory singleton devolvería el objeto singleton. Es decir:

```
Supervisor* s = [Supervisor new]; // Objeto singleton  
s = [Supervisor new]; // nil
```

Mientras que :

```
Supervisor* s = [Supervisor supervisor]; // Objeto singleton  
s = [Supervisor supervisor]; // Objeto singleton
```

Este comportamiento es poco homogéneo, y es la razón por la que se recomienda usar sólo el método factory singleton para acceder a objetos que sean singleton. Cuando `alloc` nos devuelve `nil` la segunda vez que lo llamamos, esto es un síntoma de que nos estamos equivocando en su uso.

Por último, obsérvese que los objetos singleton siempre tienen su cuenta de referencias a cero, y sin embargo nunca se eliminan de memoria ya que `release` ha sido redefinido para no llamar a `dealloc`.

6.2. Implementar la inicialización

Cuando existen relaciones de herencia pueden surgir problemas difíciles de detectar a la hora de inicializar los objetos. En este apartado vamos a ver qué problemas son éstos, y cómo evitar que se produzcan.

En lenguajes como C++ y Java las reglas que guían la ejecución de los constructores ayudan a evitar estos problemas, sin embargo en Objective-C, al tener un mayor control del proceso de construcción, es responsabilidad del programador evitar que estos problemas se produzcan.

Tanto en C++ como en Java, cuando se ejecuta el constructor de una clase derivada, primero se llama al constructor de la clase base, en concreto:

- Si el constructor de la clase derivada no indica qué constructor base ejecutar, se ejecuta el constructor por defecto de la base, y luego el de la derivada.
- Si el constructor de la clase derivada indica qué constructor base ejecutar, se ejecuta el constructor base especificado, y luego el de la derivada.

Esta forma de ejecutarse los constructores garantiza que siempre las variables de instancia de la clase base estarán correctamente inicializados cuando comience la inicialización de las variables de instancia de la clase derivada.

Aunque C++ o Java están diseñados de forma que esta condición siempre se garantiza, en Objective-C es responsabilidad del programador seguir una serie de reglas, que vamos a ver a continuación, los cuales garantizan esta condición.

6.2.1. El inicializador de máxima genericidad

En Objective-C toda clase debe tener uno o más métodos de instancia que actúen como inicializadores, y cuyo nombre debe empezar por `init....`. Como mínimo una clase tendrá el método `init` que hereda de la clase raíz.

Independientemente de que una clase tenga uno o más métodos de inicialización, siempre habrá uno de ellos, al que vamos a llamar **inicializador de máxima genericidad**, que será el que más parámetros reciba, y que va a tener un tratamiento especial en los patrones de diseño que vamos a estudiar a continuación.

En la Figura 4.4 vemos un ejemplo de relación de herencia. En el caso de la clase `NSObject` el inicializador de máxima genericidad es `init`, ya que es el único que existe. En el caso de la clase `Edificio` es `initWithDireccion:`, pues recibe más parámetros que `init`, y en el caso de `Vivienda` el inicializador de máxima genericidad es `initWithDireccion:propietarios:`, ya que es el que más parámetros recibe.

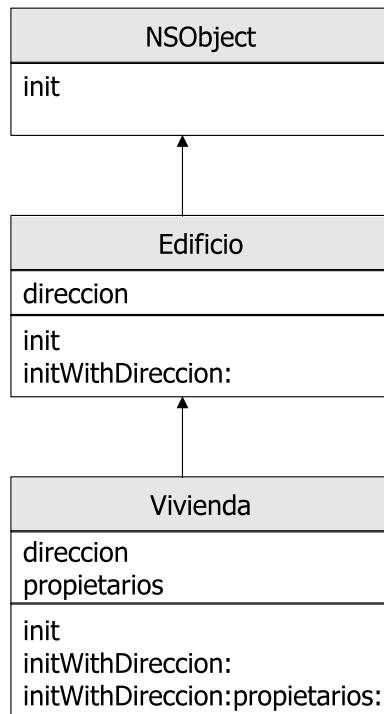


Figura 4.4: Ejemplo de herencia

6.2.2. Cómo implementar los inicializadores

En Objective-C, cuando implementemos inicializadores sólo debemos de tener en cuenta dos sencillas reglas:

1. El inicializador de máxima genericidad se debe implementar de forma que antes de inicializar las variables de instancia de la clase, llame – mediante `super` – al inicializador de máxima genericidad de la clase base.
2. El resto de inicializadores deben redefinirse de forma que lo único que hagan sea, mediante `self` llamar a un inicializador más general de su clase, o bien al inicializador de máxima genericidad de su clase.

Basta con seguir estas dos reglas para garantizar que en Objective-C las variables de instancia de la clase base estén correctamente inicializadas cuando empiece la inicialización de la derivada.

La Figura 4.5 muestra gráficamente qué inicializador debe llamar a cuál. Aquí resulta importante destacar tres observaciones: La primera observación es que el inicializador de máxima genericidad pasa las llamadas hacia arriba de la jerarquía, mientras que el resto de inicializadores llaman al de máxima genericidad de su clase. La segunda observación es que al seguir estas reglas, el inicializador de máxima genericidad es el que más trabajo realiza, ya que tiene que encargarse de asignar valores a las variables de instancia, mientras que el resto de inicializadores delegan en inicializadores más genéricos la tarea de inicializar el objeto. La tercera observación es que `init` de `Vivienda` llama al siguiente inicializador más general (y no al inicializador de máxima genericidad), aunque también podría haber llamado al de máxima genericidad si hubiera sido más fácil implementarlo así.

Veamos que ocurriría si no cumpliéramos alguna de estas reglas en nuestro programa Objective-C:

Si no cumpliéramos con la primera regla en, por ejemplo, `initWithDireccion:proprietarios:` de la clase `Vivienda`, no se ejecutaría el inicializador de máxima genericidad de `Edificio`, y podrían quedar sin realizar inicializaciones de variables de instancia públicas o privadas de la base.

Si por el contrario no cumpliéramos con la segunda regla en, por ejemplo, `init` de `Edificio`, cuando el usuario intente inicializar un objeto `Edificio` con `init` se ejecutaría `init` de `NSObject`, y la variable de instancia `direccion` quedaría sin inicializar.

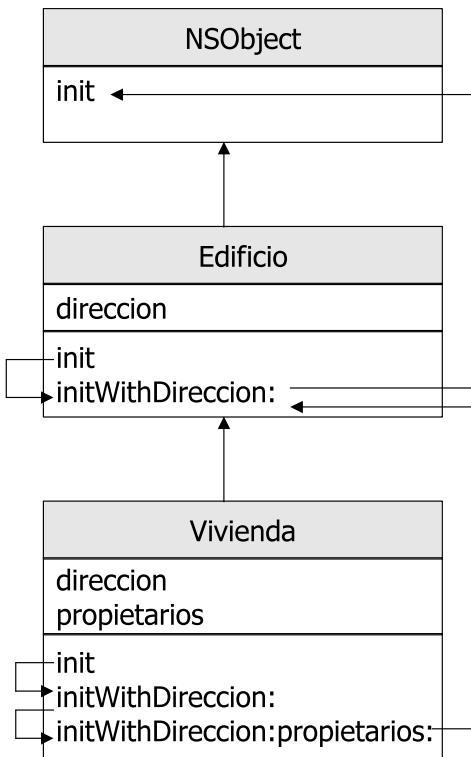


Figura 4.5: Orden correcto de inicialización

6.2.3. Redefinir self

En el apartado 1.3 vimos que, a diferencia de lo que ocurre en C++ y en Java, es posible asignar un valor a `self`. También vimos que la forma correcta de implementar un inicializador es:

```
- init {
    if (self = [super init]) {
        .....
    }
    return self;
}
```

Donde lo que hacemos es comprobar que el inicializador de la clase base no falle (devuelva `nil`). Además, como el inicializador de la clase base puede cambiar el objeto (por ejemplo por un proxy), conviene guardar su retorno en la variable `self`.

Debido a que la forma correcta de implementar un inicializador (método de instancia) es redefiniendo `self`, existe una tendencia a implementar así también los métodos factory (métodos de clase). Este es un error común que conviene destacar aquí para evitar que el lector lo cometa: No debemos de asignar a `self` en los métodos factory, ya que `self` en los métodos de clase se refiere al objeto clase, y no al objeto de instancia. Es decir, es un error hacer cosas como:

```
+ (Rectangulo*) rectanguloDeColor:(NSInteger)color {
    self = [[Rectangulo alloc] init]; // Mal
    if (self!=nil) {
        [self setColor:color];
    }
    return [self autorelease];
}
```

En este caso estamos guardando en `self` – que es un puntero al objeto clase – un objeto de instancia. En el apartado 3 del Tema 5 veremos que `autorelease` se usa para que la variable sea liberada por el sistema de gestión de memoria de Cocoa.

Una forma correcta de implementarlo sería guardar el resultado en una variable local de la forma:

```
+ (Rectangulo*) rectanguloDeColor:(NSInteger)color {
    id r = [[Rectangulo alloc] init]; // Mejor
    if (r!=nil) {
        [r setColor:color];
    }
    return [r autorelease];
}
```

Pero en este caso todavía podemos mejorar la implementación del método de clase haciendo que el mensaje `alloc` se envíe a `self` (en vez de a `Rectangulo`). De esta forma, si se crean subclases de `Rectangulo` (p.e. `Cuadrado`), el mensaje `alloc` será recibido por la subclase, y se devolverá

una instancia de la subclase donde `init` habrá asignado a las variables de instancia de la subclase valores por defecto.

6.3. Desinicialización y liberación

En C++ y Java se separa el proceso de desinicialización de variables de instancia (realizado por el destructor) del proceso de liberación de memoria (realizado por el operador `delete` en C++, y por el recolector de basura en Java). Por el contrario, en Objective-C ambas operaciones se realizan en el mismo método, aunque este método tiene distinto nombre en la clase `Object` que en la clase `NSObject`.

La clase `Object` proporciona el método de instancia `free` para realizar la desinicialización y liberación de memoria del objeto. Este método está implementado en la clase raíz de forma que sólo libera la memoria ocupada por las variables de instancia del objeto. Los objetos apuntados por variables de instancia de tipo puntero no son liberados, y es responsabilidad del programador del objeto redefinir el método `free` para liberar los objetos agregados. La liberación se debe implementar al revés que la inicialización, es decir, de forma que primero se liberen los recursos de la clase derivada y luego los de la base. Por ejemplo:

```
- free {
    // Libera los recursos de la derivada
    .....
    return [super free];
}
```

Observe que `free` devuelve un tipo `id`, en el caso de `free` de `Object`, éste siempre devuelve `nil`.

La clase `NSObject` proporciona otro método análogo llamado `dealloc`. Es parecido a `free`, pero añade la mejora de que tiene retorno `void`, ya que no es necesario conocer el retorno. Luego la forma de implementar este método debe ser algo así como:

```
- (void)dealloc {
    // Libera los recursos de la derivada
    .....
    [super dealloc];
}
```

En Cocoa el método `dealloc` no se suele llamar directamente sino que, al usarse el sistema de gestión de memoria que veremos en el apartado 3 del Tema 5, el programador llama al método `release`, el cual cuando la cuenta de referencias llega a cero libera la memoria llamando a `dealloc`.

Dentro de `dealloc` la forma correcta de liberar los objetos agregados no es llamar a `dealloc` sobre ellos, sino a `release`, ya que otros objetos podrían tener referencias a ellos. Es decir, la forma correcta de implementar `dealloc` es la que vimos en Listado 3.6, y que volvemos a escribir aquí:

```
- (void) dealloc {
    if (desde!=nil)
        [desde release];
    if (hasta!=nil)
        [hasta release];
    [super dealloc];
}
```

Téngase en cuenta que cuando un programa termina es mejor no enviar el mensaje `dealloc` a todos los objetos de la aplicación, ya que es más eficiente que la memoria dinámica asociada al proceso la libere el sistema operativo. Sin embargo, si el objeto debe cerrar ficheros, u otro tipo de recursos, entonces sí que deberíamos de enviarle el mensaje `dealloc`.

7. Categorías

Las clases son la forma más importante de asociar métodos a objetos, pero no la única. En los siguientes apartados discutiremos otras tres formas de declarar métodos y asociárselos a objetos: 1) Las categorías, que nos permiten dividir una clase en partes, o bien extender su funcionalidad, 2) los protocolos, que nos permiten declarar métodos que pueden ser implementados por varias clases, y 3) las extensiones, que nos permiten añadir métodos a una clase para uso exclusivo del framework al que pertenece la clase.

7.1. Qué son las categorías

Las **categorías** nos permiten modificar una clase ya existente aunque no dispongamos de su código fuente. La diferencia que hay entre la herencia y la **categorización** es que la herencia sólo nos permite crear nuevas hojas en la jerarquía de clases, mientras que la categorización nos permite modificar nodos interiores de la jerarquía de clases.

Al igual que la herencia tiene varias finalidades (reutilización, clasificación, ...), la categorización también tiene al menos cuatro finalidades:

1. Las categorías se pueden usar para implementar una clase en distintos ficheros de código fuente. Esto nos permite particionar una clase en grupos de métodos relacionados, lo cual por ejemplo, reduce el tiempo de compilación de una clase grande al sólo compilar el fichero de la categoría modificada. Dividir la clase en varias categorías también podría permitir

- repartir el trabajo de implementación de la clase entre varios programadores.
2. Las categorías permiten añadir a las clases de librerías de otros fabricantes métodos que nos hubiese resultado especialmente útil que tuviesen, pero que le fabricante de la librería no metió. Por ejemplo, con las categorías podemos añadir métodos a clases de la librería Cocoa sin necesidad de crear una derivada. Los métodos añadidos a la clase son automáticamente heredados por las subclases, y en tiempo de ejecución son indistinguibles de los métodos de la clase. Es muy típico crear categorías que añaden nuevos métodos (p.e. métodos de búsqueda) a las clases que implementan colecciones como `NSArray`.
 3. Otra ventaja de las categorías es que permiten diferenciar entre un API pública y un API para el framework. De esta forma podemos crear dos versiones de nuestra clase: una con un conjunto de métodos públicos que todo el mundo puede usar, y otra con métodos añadidos destinados a uso exclusivo para nuestro framework.
 4. Una cuarta finalidad de las categorías es declarar protocolos informales, tal como veremos en el apartado 8.8.

7.2. Declarar la interfaz de una categoría

La declaración de la interfaz de una categoría es parecida a la declaración de la interfaz de una clase, excepto que el nombre de la categoría se pone entre paréntesis después del nombre de la clase a la que se añade la categoría. De esta forma indicamos que los métodos de la categoría son añadidos a una clase que ya existe en algún otro lugar.

Suponiendo que tenemos la clase `Punto` del Listado 3.1, podemos declarar la interfaz de una categoría `Pinta` tal como muestra el Listado 4.5. Esta categoría engloba dos métodos relacionados con la forma de pintar un objeto, o bien en consola con `imprime`, o bien en pantalla con `pinta`.

Apple recomienda nombrar los ficheros de la categoría de la forma `Clase+Categoría`. Por ejemplo, para añadir la categoría `Pinta` a la clase `Punto` usaríamos los nombres de fichero `Punto+Pinta.h` y `Punto+Pinta.m`.

Una diferencia entre la herencia y la categorización es que las categorías no indican la clase base de la que deriva la clase a la que se añade la categoría. Otra diferencia está en que las categorías no pueden añadir nuevas variables de instancia a la clase, sólo nuevos métodos (razón por la que no pueden ponerse variables de instancia encerradas entre llaves en la declaración de la interfaz de la categoría). Esto se debe a que las categorías añaden nuevos métodos a la información de runtime de la clase, pero las categorías no pueden modificar la organización del objeto en memoria. Si necesita añadir nuevas variables de instancia para sus métodos, la única solución es la herencia.

Por el contrario una categoría puede acceder a todas las variables de instancia de su clase, incluso a las marcados como privadas.

La declaración de la interfaz de la categoría se pueden crear o bien en el mismo fichero donde se declara la interfaz de la clase, o bien en un fichero aparte (como en el caso del Listado 4.5). En cualquier caso, la declaración de la interfaz de la clase debe ser conocida por el compilador cuando éste encuentra la categoría. En el caso de que la interfaz de la categoría se declare en un fichero aparte, lo recomendable es importar el fichero con la interfaz de la clase antes de declarar la interfaz de la categoría.

```
/* Punto+Pinta.h */
#import "Punto.h"

@interface Punto (Pinta)
- (void)imprime;
- (void)pinta;
@end
```

Listado 4.5: Interfaz de una categoría

Los nombres de las categorías se guardan en un namespace distinto al de las clases, con lo que una categoría puede tener el mismo nombre que una clase.

7.3. Implementación de una categoría

El Listado 4.6 muestra un ejemplo de implementación de la categoría anterior. El lenguaje Objective-C no nos obliga a implementar todos los métodos de la interfaz de la categoría, pero nos avisa con un warning en caso de encontrar en la implementación un método de la interfaz de la categoría sin implementar. De esta forma se facilita en la medida de lo posible que un nombre de método mal escrito sea detectado. En el apartado 8.8 veremos que podemos no implementar ningún método de la interfaz de la categoría. Esto nos va a permitir crear lo que se llama un protocolo informal.

```
/* Punto+Pinta.m */
#import "Punto+Pinta.h"

@implementation Punto (Pinta)
- (void)imprime {
    printf("[%d,%d]", x, y);
}
- (void)pinta {
    .....
}
@end
```

Listado 4.6: Implementación de una categoría

La interfaz de la categoría se puede declarar tanto en un fichero de cabecera, como en un fichero de implementación. Si declaramos la interfaz de la categoría en un fichero de interfaz, cualquier código que importe el fichero de interfaz que contiene la categoría podrá ejecutar los métodos de ésta. En caso de encontrarse la interfaz de la categoría en un fichero de implementación, o bien en caso de que no se importara el fichero de interfaz de la categoría, los métodos que añade a la clase la categoría van a ser encontrados en tiempo de ejecución, pero el compilador producirá un warning avisando de que el método de la categoría al que estamos llamando podría no existir.

En principio podemos añadir cuantas categorías queramos a una clase, pero cada categoría debe tener un nombre distinto. Sin embargo, dos categorías pueden tener el mismo nombre siempre que pertenezcan a clases distintas. El Listado 4.7 y el Listado 4.8 muestran un ejemplo de dos categorías con el mismo nombre creadas para las clases `Punto` y `Complejo`. El crear categorías con el mismo nombre es muy común cuando ambas tienen la misma funcionalidad aplicada a objetos distintos.

```
/* Pinta.h */
#import "Punto.h"

@interface Punto (Pinta)
- (void)imprime;
- (void)pinta;
@end

#import "Complejo.h"

@interface Complejo (Pinta)
- (void)imprime;
- (void)pinta;
@end
```

Listado 4.7: Interfaz de categorías con el mismo nombre

```
/* Pinta.m */
#import "Pinta.h"

@implementation Punto (Pinta)
- (void)imprime {
    printf("[%d,%d]",x,y);
}
- (void)pinta {
    .....
}
@end

@implementation Complejo (Pinta)
- (void)imprime {
    printf("R:%d I:%d",real,imaginario);
}
- (void)pinta {
```

```
    . . . . .
}
```

Listado 4.8: Implementación de categorías con el mismo nombre

7.4. Sobrescribir métodos con categorías

Si creamos una categoría con un método que tiene el mismo nombre que algún método de la clase, el método de la categoría sobrescribe al de la clase, pero de una forma especial: Cuando el nuevo método envía mensajes a `super`, éstos van directamente a la clase base de la clase, y no al método sobrescrito. Esto se debe a que – como indicamos en el apartado 1.3 – `super` es resuelto en tiempo de compilación. De hecho no hay forma de enviar mensajes al método sobrescrito de la clase desde el método de la categoría²⁴.

Si varias categorías de una clase definen un método con el mismo nombre, uno de ellos será el que perdurará, pero cuál de ellos sea no está definido por el lenguaje y depende del compilador, con lo que no se recomienda crear varias categorías que definan el mismo nombre de método para la misma clase.

Conviene tener en cuenta que las categorías no son una forma alternativa de herencia. En general, si se van a redefinir métodos se recomienda utilizar herencia. Las categorías se recomiendan principalmente para añadir nuevos métodos a las clases.

7.5. Categorías en la clase raíz

Una categoría puede añadir métodos a cualquier clase, incluida la clase raíz. Esto hace que los métodos añadidos a `NSObject` (o a `Object`) estén disponibles en todas las clases que deriven de ella. Lógicamente, en este caso los mensajes a `super` no serán válidos.

Es importante tener en cuenta que aunque añadir métodos a la clase raíz puede ser útil, también puede resultar peligroso si los cambios que realiza un programador no son conocidos por otros programadores, los cuales pueden no entender que es lo que está ocurriendo.

También hay que tener en cuenta que los objetos clase también tendrían disponibles los métodos añadidos a la clase raíz. Normalmente los objetos clase ejecutan sólo métodos de clase, pero al añadir métodos a la clase raíz, los métodos de instancia añadidos por la categoría serían también métodos

²⁴ En el apartado 3.4 del Tema 10 veremos que este efecto sí que se puede conseguir con el `posing`.

de clase que podríamos ejecutar sobre un objeto clase. Por ejemplo, hay que tener en cuenta que en el cuerpo del método, el receptor `self` se estaría refiriendo a un objeto clase y no a un objeto de instancia.

8. Protocolos

La interfaz de una clase o categoría declara métodos asociados con una clase en particular. Por el contrario, los **protocolos** declaran métodos no asociados con ninguna clase, sino que cualquier clase o conjunto de clases pueden implementar. Según esto, los protocolos nos permiten indicar que un conjunto de clases no relacionadas comparten un conjunto de métodos comunes. Por ejemplo, un conjunto de clases que representen objetos de interfaz gráfica podrían compartir los métodos:

- `(void)mouseDown:(NSEvent*)theEvent;`
- `(void)mouseDragged:(NSEvent*)theEvent;`
- `(void)mouseUp:(NSEvent*)theEvent;`

Se dice que una clase **cumple, adopta o implementa** un protocolo si dispone de métodos que reciben los mensajes del protocolo²⁵. De esta forma los protocolos nos permiten separar la jerarquía de clases del hecho de compartir funcionalidad común. El objetivo es que los objetos se puedan agrupar, no sólo por su jerarquía, sino por los protocolos que adoptan.

Al igual que las clases y las categorías se podían usar con varias finalidades, los protocolos también resultan útiles en al menos seis escenarios:

1. Para declarar métodos que otros deben implementar. El protocolo puede enumerar la interfaz de los métodos que deben implementar las clases que adoptan el protocolo.
2. Para capturar similitudes entre clases que no están jerárquicamente relacionadas. Cuando varias clases no jerárquicamente relacionadas adoptan un protocolo, sabemos que comparten un conjunto común de operaciones.
3. Para declarar la interfaz de un objeto ocultando su clase. Los protocolos se pueden usar para declarar los métodos de un **objeto anónimo**, es decir, un objeto del que no conocemos su clase. Lógicamente, los objetos no son anónimos para su programador, pero sí que pueden serlo para otros programadores que lo usan a través de la interfaz de un protocolo²⁶.

²⁵ En el resto de este documento procuraremos usar el término *adoptar* o *implementar*, aunque en la literatura puede encontrar cualquiera de estos tres términos.

²⁶ Aunque el usuario del objeto sólo conozca el protocolo del objeto, siempre puede obtener su clase en tiempo de ejecución con el método `class`, sin embargo, el objetivo de los objetos anónimos es que el usuario del objeto no tenga que recurrir a preguntar por la clase del objeto para poder usarlo.

4. Para indicar a varios fabricantes qué interfaz común de operaciones debe tener un objeto. Por ejemplo, si varios fabricantes tienen distintas bases de datos, un protocolo `BaseDatos` puede estandarizar los métodos que debería tener un objeto para ser considerado una base de datos. De esta forma el programador se puede conectar a distintas bases de datos con sólo obtener un puntero a un objeto que adopte este protocolo.
5. Para reducir la complejidad de un objeto. El objetivo de la encapsulación es reducir la complejidad de los objetos, es decir, el número de métodos que debe conocer el usuario del objeto para poder trabajar con él. Otra forma de reducir la complejidad de la interfaz de un objeto es usando un protocolo que adoptan las clases que proporcionan esa funcionalidad. De esta forma el número de métodos que da a conocer el protocolo en su interfaz es menor al número de métodos que proporciona la interfaz de la clase.
6. Para hacer comprobación estática de tipos. En el apartado 8.3 veremos que podemos usar el protocolo para que el compilador compruebe en tiempo de compilación que un objeto dispone de los métodos que queremos ejecutar.

8.1. Declarar un protocolo

Los protocolos se dividen en protocolos formales y protocolos informales. Las interfaces Java, o las clases abstractas puras de C++, corresponderían a los protocolos formales de Objective-C. C++ y Java no proporcionan protocolos informales, lo cual limita la implementación de patrones de diseño importantes como el patrón de delegación que veremos en el apartado 8.9. Cuando se utiliza el término protocolo sin calificar, se asume que nos estamos refiriendo a un protocolo formal, en caso de tratarse de un protocolo informal, debe indicarse explícitamente. Vamos a empezar estudiando los protocolos formales, y los protocolos informales los analizaremos en el apartado 8.8.

Los **protocolos formales**, permiten declarar una lista de métodos que una clase adopta, y tanto en tiempo de compilación como en tiempo de ejecución vamos a poder comprobar si un objeto está implementando los métodos del protocolo.

En el apartado 8.3 veremos que los protocolos permiten realizar una tipificación estática del objeto, donde el compilador comprueba el tipo de un objeto basándose en protocolos. La forma en que se comprueba en tiempo de ejecución si un objeto, al que tenemos un puntero, adopta un protocolo la veremos en el apartado 8.6. En el apartado 8.8 veremos los protocolos informales, que son agrupaciones de métodos que una clase puede implementar, pero que no se obliga a que lo haga ni en tiempo de compilación, ni en tiempo de ejecución.

El Listado 4.9 y Listado 4.10 muestra dos ejemplos de declaración de protocolos. Los protocolos – a diferencia de las clases y de las categorías – sólo tienen un fichero .h, ya que los ficheros .m donde sus métodos son implementados serían las clases o categorías que los adoptan.

Para declarar un protocolo se usa la directiva del compilador `@protocol`, y dentro se enumeran los métodos que componen el protocolo. Los protocolos no pueden tener variables de instancia, sólo métodos.

```
/* Imprimible.h */
#import <Foundation/Foundation.h>

@protocol Imprimible
- (void)imprime;
- (void)imprime:(char*)buffer;
@end
```

Listado 4.9: Declaración de un protocolo `Imprimible`

```
/* Dibujable.h */
#import <Foundation/Foundation.h>

@protocol Dibujable
- (void)pinta;
@end
```

Listado 4.10: Declaración de un protocolo `Dibujable`

Los nombres de los protocolos se guardan en un namespace distinto al de las clases y al de las categorías, con lo que en principio podríamos tener una clase, una categoría y un protocolo con el mismo nombre. Por ejemplo, `NSObject` es tanto un nombre de clase como un nombre de protocolo, que a su vez es implementado por la clase `NSObject`. En el apartado 8.5 veremos qué utilidad tiene esta separación entre clase y protocolo.

8.2. Adoptar un protocolo

Si una clase quiere adoptar uno o más protocolos, debe indicar los protocolos que adopta en su interfaz entre paréntesis angulares, y separando por comas los nombres de los protocolos que adopta.

El Listado 4.11 muestra un ejemplo de una clase `Punto` que adopta tanto el protocolo `Imprimible`, definido en el Listado 4.9, como el protocolo `Dibujable`, definido en el Listado 4.10. Cuando una clase adopta un protocolo, en su implementación debe implementar todos los métodos del protocolo, pero no es necesario (aunque se puede hacer por claridad) escribir los nombres de los métodos del protocolo en la interfaz de la clase.

Si la implementación de la clase no implementara algún método del protocolo adoptado, el compilador emitiría un warning, y en tiempo de ejecución produciría una excepción al no encontrar el método. Como regla general, si una clase adopta un protocolo debería de implementar todos sus métodos. En el apartado 3 del Tema 10 veremos que esto podría no ser así en caso de usar delegación.

Cuando una clase adopta un protocolo, normalmente en su fichero de interfaz importa el fichero del protocolo.

```
/* Punto.h */
#import "Imprimible.h"
#import "Dibujable.h"

@interface Punto : NSObject <Imprimible, Dibujable> {
    NSInteger x;
    NSInteger y;
}
- (NSInteger)x;
- (NSInteger)y;
- (void)setX:(NSInteger)paramX;
- (void)setY:(NSInteger)paramY;
// Metodos del protocolo Imprimible
- (void)imprime;
- (void)imprime:(char*)buffer;
// Métodos del protocolo Dibujable
- (void)pinta;
@end
```

Listado 4.11: Clase que adopta dos protocolos

```
/* Punto+Pinta.h */
#import "Punto.h"
#import "Imprimible.h"

@interface Punto (Pinta) <Imprimible>
// Metodos del protocolo Imprimible
- (void)imprime;
- (void)imprime:(char*)buffer;
// metodo para dibujar en pantalla
- (void)pinta;
@end
```

Listado 4.12: Categoría que adopta un protocolo

Las categorías también pueden adoptar protocolos, en cuyo caso la clase de la categoría también adopta el protocolo (a pesar de que no se indique en la interfaz de la clase). El Listado 4.12 muestra un ejemplo de categoría que adopta el protocolo `Imprimible`. Cuando es la categoría la que adopta el protocolo, como regla general la clase no debe adoptar también este mismo protocolo. Es decir, al crear la categoría del Listado 4.12, deberíamos de

eliminar la declaración de que adopta el protocolo del Listado 4.11. De no hacerlo, los métodos de la categoría sobrescribirían a los de la clase.

8.3. Tipificación estática de prototipo

En el apartado 2.2 del Tema 3 vimos que a los objetos nos podemos referir tanto con tipos estáticos, que se representaban como punteros a objetos, en cuyo caso el compilador comprobaba la existencia de los métodos ejecutados en la clase, como con tipos dinámicos, representados por el tipo `id`, en cuyo caso no se comprueba en tiempo de compilación que el objeto pueda responder a los mensajes que le enviamos.

Realmente existen dos formas de tipificación estática: La **tipificación estática de clase**, que es la que vimos que nos permitía que el compilador comprobase que el método llamado exista en la clase. Por ejemplo:

```
Punto* p = [Punto new];
[p setX: 5];
```

Y la **tipificación estática de prototipo** la cual permite al compilador comprobar que el método llamado existe en el prototipo. Para realizar una tipificación estática de prototipo se indica entre paréntesis angulares el nombre del prototipo. Por ejemplo:

```
id <Imprimible> p;
```

Al igual que en la tipificación estática de clase, a la variable tipificada también se la llamaría **puntero a objeto estático**, y podemos asignarle objetos cuya clase está en distintas partes de la jerarquía con la condición de que la clase adopte el protocolo. Por ejemplo, como en el Listado 4.11 la clase `Punto` adoptó el protocolo `Imprimible`, podemos hacer:

```
id <Imprimible> p = [Punto new];
[p imprime];
```

Para poder realizar tipificación estática de prototipo no es necesario que la clase indique que adopta el protocolo, podría ser que quien lo indicase fuera una categoría de la clase, como por ejemplo la del Listado 4.12.

Con los punteros a objeto estáticos el compilador realiza las siguientes comprobaciones:

- El compilador permite ejecutar sobre un puntero a objeto estático sus métodos.
- El compilador emite un warning si usamos un puntero a objeto estático para ejecutar un método que no existe en el tipo estático.

- El compilador emite un warning cuando intentamos asignar un objeto que no cumple con un protocolo a un puntero a objeto estático.
- El compilador permite asignar el valor de un puntero a objeto dinámico (un puntero de tipo `id`) a un puntero a objeto estático. Si en tiempo de ejecución ejecutamos sobre el puntero estático un método que no existe, se producirá una excepción.
- El compilador emite un warning si intentamos asignar el valor de un puntero a objeto estático a un puntero a otro tipo estático que no adopta el protocolo.
- El compilador permite asignar el puntero a objeto estático a un puntero a objeto dinámico. Si en tiempo de ejecución intentamos ejecutar un método que no existe en el puntero dinámico se producirá una excepción.

Por ejemplo, si `Punto` adopta `Imprimible` y `Complejo` no lo adopta:

```
id <Imprimible> p = [Punto new];
Complejo* c1 = [Complejo new];
id c2 = [Complejo new];
[p imprime]; // Correcto
[c2 imprime]; // No warning. Excepcion en tiempo de ejecucion
p = c1; // Warning.
[p imprime]; // No warning. Excepcion en tiempo de ejecucion
p = c2; // No warning
[p imprime]; // No warning. Excepcion en tiempo de ejecucion
c1 = p; // Warning
c2 = p; // No warning
```

La tipificación estática de protocolo también se puede hacer en los parámetros y retornos de las funciones y métodos. Por ejemplo:

```
void Imprime(id <Imprimible> obj) {
    [obj imprime];
}
```

También podemos hacer tipificación estática de varios protocolos de la forma:

```
id <Imprimible, Dibujable> obj;
```

En Java el nombre de la clase y el nombre del protocolo están en el mismo namespace, en consecuencia, al crear una referencia sólo podemos usar el nombre de la clase, o el nombre del protocolo, por el contrario Objective-C permite hacer tipificación estática de clase y de protocolo de la forma:

```
Punto <Imprimible> *p = [Punto new];
```

En este caso el compilador comprueba tanto los métodos de la clase como los del protocolo.

Podemos pensar que, al ser los métodos de `Imprimible` un subconjunto de los métodos de `Punto`, no existe diferencia con haber hecho:

```
Punto* p = [Punto new];
```

Pero, como muestra el ejemplo de la Figura 4.7, la interfaz `Imprimible` podría no estar implementada por todos los objetos `Punto`, sino sólo por un subtipo de estos llamado `PuntoImprimible` en nuestra figura de ejemplo.

En este caso hacer tipificación estática de clase y protocolo sobre `p` indica que el objeto apuntado derivada de `Punto`, y que además implementa el protocolo `Imprimible`.

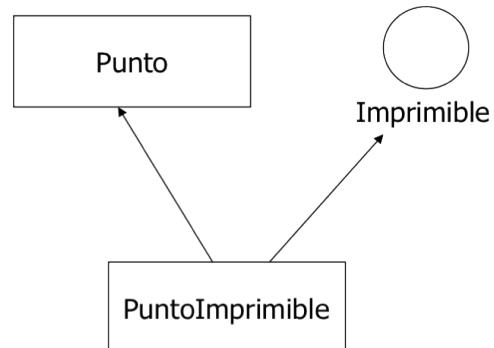


Figura 4.7: Ejemplo de clases y protocolos

8.4. Jerarquía de protocolos

Un protocolo puede adoptar los métodos de otro protocolo usando la misma sintaxis que usan las clases o categorías para adoptar protocolos, es decir, ponemos entre paréntesis angulares la lista de protocolos que adopta el nuevo protocolo. Por ejemplo, el Listado 4.13 muestra un protocolo `Formateable` que adopta los métodos de `Imprimible` y añade un nuevo método.

Cuando una clase o categoría adopta el protocolo `Formateable` debe implementar tanto los métodos que adopta directamente del protocolo `Formateable`, como los que adopta indirectamente de `Imprimible`.

```
/* Formateable.h */
#include "Imprimible.h"

@protocol Formateable <Imprimible>
- (void)imprime:(char*)buffer
           conFormato:(const char*)formato;
@end
```

Listado 4.13: Jerarquía de protocolos

8.5. El protocolo `NSObject`

En el apartado 8.1 vimos que `NSObject` es tanto un nombre de clase como un nombre de protocolo, donde al estar en namespaces distintos no hay conflictos entre ellos. La clase `NSObject` adopta el protocolo `NSObject` e implementa todos sus métodos.

Apple proporciona dos razones para crear esta distinción. La primera es permitir que existan otras clases raíz que cumplan con el protocolo `NSObject`, y que en consecuencia puedan servir como clases raíz del runtime de Apple. De esta forma, los métodos que aparezcan en la clase `NSObject`, pero no en el protocolo `NSObject` no serían necesarios para poder usar el runtime de Apple. Téngase en cuenta que los métodos del protocolo `NSObject` son un subconjunto de los métodos de la clase `NSObject`.

La segunda razón de esta distinción es permitir ejecutar métodos de la clase raíz sobre objetos apuntados con un puntero a objeto estático tipificado como un protocolo. Para ver que significa esto, supongamos que hacemos:

```
id <Imprimible> p = [Punto new];
[p release]; // Warning
```

El warning se debe a que no existe el método `release` en el protocolo `Imprimible`. Debido a que es muy común ejecutar métodos de la raíz sobre punteros a objetos estáticos, quizá una mejor forma de declarar `Imprimible` sería la que se muestra en el Listado 4.14, es decir, hacer que `Imprimible` adopte el protocolo `NSObject`. Ahora la llamada a `release` anterior no produciría el warning.

```
/* Imprimible.h */
@protocol Imprimible <NSObject>
- (void)imprime;
- (void)imprime:(char*)buffer;
@end
```

Listado 4.14: Protocolo que adopta el protocolo `NSObject`

8.6. Objetos protocolo

Al igual que las clases están representadas en tiempo de ejecución por objetos clase, los protocolos formales están representados en tiempo de ejecución por **objetos protocolo**.

También, al igual que el runtime de Objective-C crea un objeto clase por cada definición de clase que usa el programa, el runtime de Objective-C crea un

objeto protocolo por cada declaración de protocolo que se usa en el programa.

Los objetos protocolos se pueden referenciar con la clase `Protocol`, y a diferencia de los objetos clase, la única forma de obtener un objeto protocolo es usando la directiva del compilador `@protocol(nOMBRE)`. Por ejemplo, para obtener el objeto protocolo de la declaración del protocolo `Imprimible` hacemos:

```
Protocol* imprimible = @protocol(Imprimible);
```

El compilador crea para el runtime un objeto protocolo, pero sólo cuando el protocolo es adoptado por alguna clase del programa, o bien el programa se refiere al protocolo en algún punto con la directiva del compilador `@protocol()`. Los protocolos que se declaran en las cabeceras, pero no se usan no dan lugar a objetos protocolo. Tampoco se crea un objeto protocolo si el protocolo sólo es usado para tipificación estática.

En el apartado 2.5 vimos que el objeto clase se podía usar para extraer información de introspección con métodos como `isKindOfClass:`, el cual servía para saber si un objeto era de una determinada clase. Otro método que podemos usar para extraer información de introspección es el método de instancia de la clase raíz:

- `(BOOL)conformsToProtocol:(Protocol*)aProtocol`

Que sirve para saber si un objeto adopta un protocolo. El método recibe como parámetro un objeto protocolo. Por ejemplo:

```
if ([p conformsToProtocol:@protocol(Imprimible)])
    ....
```

8.7. Declaración adelantada de protocolos

Al igual que vimos en el apartado 9 del Tema 3 que podíamos realizar una declaración adelantada de clase con la directiva del compilador `@class`, también podemos realizar una declaración adelantada de protocolo usando la directiva `@protocol`. El Listado 4.15 muestra un ejemplo de declaración adelantada del protocolo `Huevo`.

```
/* Gallina.h */
@protocol Huevo;

@protocol Gallina
- (Huevo*)pone;
@end
```

Listado 4.15: Ejemplo de declaración adelantada de protocolo

8.8. Protocolos informales

Un **protocolo informal** es una agrupación de métodos que otras clases podrían *opcionalmente implementar*, pero que a diferencia de los protocolos formales no reciben soporte del compilador, es decir, el compilador no emite un warning cuando una clase no implementa el protocolo informal. Esto se debe a que, a diferencia de las categorías, no creamos un fichero de implementación para la interfaz de la categoría. Como indicamos en el apartado 7.3, si creáramos una implementación para la interfaz de la categoría, tendríamos que implementar todos los métodos de la categoría o obtendríamos un warning.

El Listado 4.16 muestra un ejemplo de cómo declarar un protocolo informal llamado `CuentaReferencias`. Para ello declaramos la interfaz de una categoría en un fichero de cabecera, pero no creamos su correspondiente fichero de implementación.

```
/* NSObject+CuentaReferencias.h */
@interface NSObject (CuentaReferencias)
- (NSInteger)cuenta;
- (void)incrementarCuenta;
- (void)decrementarCuenta;
@end
```

Listado 4.16: Ejemplo de protocolo informal

Debido a que no implementamos los métodos del protocolo informal, normalmente la categoría se pone en la clase raíz, pero podría ponerse en otra clase para limitar su ámbito.

El hecho de que el compilador no compruebe si una clase implementa los métodos del protocolo informal resulta útil para evitar warnings en escenarios como la delegación. Por ejemplo, si con un objeto `Punto`, que no implementa los métodos del protocolo informal `CuentaReferencias`, hacemos:

```
Punto* p = [Punto new];
[p incrementarCuenta]; // Warning
```

Se produce un warning ya que el compilador desconoce el método `incrementarCuenta`. Sin embargo, si incluimos el fichero de cabecera del protocolo informal:

```
#import "NSObject+CuentaReferencias.h"

Punto* p = [Punto new];
[p incrementarCuenta]; // Desaparece el warning
```

Ahora no se produce mensaje de warning, a pesar de que `Punto` no implemente `incrementarCuenta`, ya que el método está declarado en una categoría a nivel de `NSObject`.

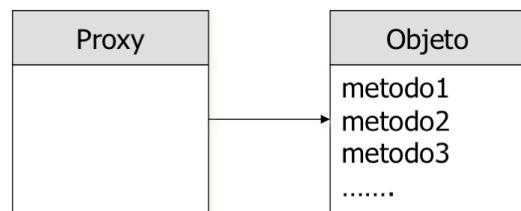
8.9. Proxies y delegados

Un objeto **proxy** (procurador) es un objeto encargado de gestionar el trabajo de otro objeto. Existen ocasiones es conveniente que un objeto proxy se haga pasar por otro objeto. En este caso, el objeto proxy puede resolver o bien pasar las peticiones al objeto al que representa. La Figura 4.8 (a) muestra esta idea.

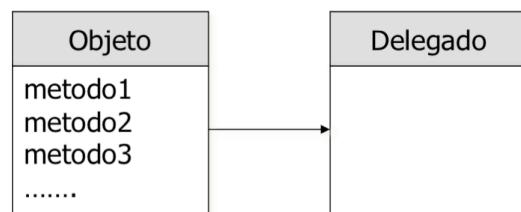
En otras ocasiones, resulta conveniente un patrón de diseño donde un objeto puede delegar parte de su trabajo en otro objeto llamado objeto **delegado**. La Figura 4.8 (b) muestra este patrón.

Existen librerías de programación como MFC o wxWidgets que proporcionan un conjunto de clases (p.e. elementos de la interfaz gráfica) de las que debemos crear derivadas para extender su funcionalidad. Crear derivadas suele implicar escribir un fichero por cada clase derivada, lo cual reduce la productividad del programador. Cocoa por su parte no suele recomendar crear derivadas de las clases de librerías sino que tiende a usar el patrón de delegación como alternativa a la herencia. En vez de crear una derivada de una clase de librería (p.e. `NSWindow`), se crea un objeto delegado de una clase personalizada. Cuando el objeto de librería recibe una llamada a uno de sus métodos, éste pregunta al delegado si implementa ese método, en cuyo caso ejecuta el método del delegado. Por ejemplo, al pulsar el usuario el botón de cerrar ventana, la ventana puede preguntar a su delegado si implementa este método con un comportamiento específico. Si no lo implementa, por defecto la ventana se cierra. El objeto singleton de la clase `NSApplication` también sigue este patrón. La clase `NSApplication` no se suele derivar, sino que el objeto delegado de la implementa métodos como `applicationWillTerminate:` para ejecutar determinadas acciones cuando este evento se produce.

Cuando un objeto no tiene delegado asociado, se acostumbra a hacer que el objeto sea delegado de sí mismo. Esta convención hace que sí una clase



(a) Patrón de diseño proxy



(b) Patrón de diseño delegado

Figura 4.8: Proxies y delegados

derivada lo desea, pueda actuar como delegado de su clase base. En caso contrario la clase base de librería proporciona un comportamiento por defecto para los métodos de delegación.

Dado que el delegado no tiene porqué implementar todos sus métodos, la forma de implementar los método del delegado es mediante un protocolo informal. El objeto de librería puede preguntar al delegado si implementa un determinado método enviándole el mensaje `respondsToSelector:` que estudiaremos en el apartado del Tema 5.

En este sentido los protocolos informales son un patrón de programación muy potente para implementar la delegación, y en consecuencia incrementar la productividad del programador. Por desgracia, ni C++ ni Java permiten crear protocolos informales.

8.10. Protocolos formales con métodos opcionales

En Objective-C 2.0 se añadió la posibilidad de marcar a uno o más métodos de un protocolo formal como métodos opcionales con la directiva `@optional`. Podemos reescribir el protocolo informal del Listado 4.16 usando un protocolo formal con sus métodos opcionales tal como muestra el Listado 4.17.

```
/* CuentaReferencias.h */
#import <Foundation/Foundation.h>

@protocol CuentaReferencias
@optional
- (NSInteger)cuenta;
- (void)incrementarCuenta;
- (void)decrementarCuenta;
@end
```

Listado 4.17: Protocolo formal con métodos opcionales

El protocolo formal del Listado 4.17 declara todos los métodos como opcionales. En general un protocolo formal puede tener parte de sus métodos opcionales y parte no opcionales. En este caso, la forma más parecida de declarar estos métodos antes de Objective-C 2.0 hubiera sido declarar los métodos no opcionales en un protocolo formal, y los métodos opcionales en un protocolo informal.

Tenga en cuenta que los protocolos formales con métodos opcionales no tienen el mismo ámbito de aplicación que los protocolos informales. Los protocolos informales (al ser categorías) permiten extender los métodos de cualquier clase (como en el Listado 4.16), mientras que los protocolos formales deben de ser adoptados explícitamente por una clase. Por ejemplo, para declarar nombres de los métodos de delegación es mejor usar protocolos

informales, ya que de esta forma todas las clases pueden hacer uso de estos métodos sin declarar que los adoptan.

9. Extensiones

En el apartado 5 del Tema 3 indicamos que los métodos en Objective-C sólo pueden ser de dos tipos: Públicos cuando se declaran en la interfaz de la clase (y se implementan en su implementación), y privados cuando sólo aparecen en la implementación de la clase. Los métodos privados sólo deben ser llamados desde la clase donde se sitúan. En el apartado 5 del Tema 3 también indicamos que los métodos privados de una clase debían declararse en el fichero de implementación siempre antes que los métodos públicos, o en caso contrario obtendríamos un warning al llamar al método privado desde el público. Las extensiones (introducidas en Objective-C 2.0) nos permiten implementar métodos privados ignorando esta regla²⁷.

Las extensiones son parecidas a las categorías pero con tres diferencias:

1. Son "anónimas", es decir, no se les asigna un nombre. Una clase puede tener varias extensiones, pero todas ellas serán anónimas.
2. Se aconseja declarar la interfaz de la extensión en el fichero de implementación de la clase a la que extiende, en vez de en un fichero de interfaz como ocurre con las categorías.
3. Sus métodos se implementan en la implementación de la clase a la que extienden, en vez de usar una sección `@implementation` distinta como ocurre con las categorías.

El Listado 4.18 muestra la interfaz de la una clase `Numero`. En el Listado 4.19 vemos que tanto la interfaz como la implementación de la extensión aparecen en el fichero de implementación de la clase `Numero` a la que estamos extendiendo.

```
/* Numero.h */
#import <Foundation/Foundation.h>

@interface Numero : NSObject {
    double numero;
}
- (double)numero;
@end
```

Listado 4.18: Interfaz de una clase con una extensión

²⁷ En el apartado 5 del Tema 3 también indicamos que si se intentan ejecutar desde fuera de la clase se producirá un warning (y no un error), y en caso de que realmente exista el método se ejecutará aunque sea privado. Por esta razón debemos ver la encapsulación como una ayuda al programador para simplificar la interfaz de una clase, y no como una restricción de seguridad.

```
/* Numero.m */
#import "Numero.h"

@interface Numero ()
- (void)setNumero:(double)param;
@end

@implementation Numero
- (double)numero {
    return numero;
}
- (void)setNumero:(double)param {
    numero = param;
}
@end
```

Listado 4.19: Implementación de una clase con una extensión

A pesar de que las extensiones son anónimas, podemos crear varias extensiones para una clase. Por ejemplo, al Listado 4.19 le podríamos añadir la extensión:

```
@interface Numero ()
- (NSInteger)parteEntera;
@end
```

A la hora de decidir si queremos usar categorías o extensiones para implementar el API de una clase debemos tener en cuenta que:

1. Las categorías representan métodos públicos mientras que las extensiones representan método privados.
2. Se recomienda que las categorías se creen en ficheros aparte, mientras que las extensiones se suelen escribir en el fichero de implementación de la clase a la que extienden.
3. Las extensiones se ponen en el fichero de implementación de la clase con lo que el compilador nos dará un mensaje de warning si no implementamos sus métodos. Por el contrario, si no implementamos los métodos de una categoría, no obtendremos ningún feedback del compilador hasta el tiempo de ejecución, momento en el cual obtendremos un error en tiempo de ejecución. En consecuencia, las categorías tienen la ventaja de que nos permiten crear protocolos informales (ya que no obtendremos warnings si no implementamos sus métodos). Por su parte las extensiones realizan un control estático más estricto.

10. Clases abstractas

Las clases abstractas son clases que no se pueden instanciar ya que existen exclusivamente para que otras clases deriven de ellas. Tanto C++ como Java disponen de clases abstractas. En el caso de C++ la razón por la que una clase es abstracta es porque tiene un método virtual puro (método sin implementar). En el caso de Java una clase podría ser abstracta incluso sin tener ningún método sin implementar, ya que en Java una clase es abstracta cuando tiene el modificador `abstract`.

```
/* Figura.java */

public abstract class Figura {
    int lados;
    public int getLados() {
        return lados;
    }
    public void setLados(int lados) {
        this.lados = lados;
    }
    public abstract void pinta();
}
```

Listado 4.20: Clase Java abstracta

Los protocolos comparten con las clases abstractas la característica de que no se pueden instanciar. En cierto sentido una clase abstracta es una combinación entre clase normal y protocolo, ya que puede tener variables de instancia y métodos implementados, aunque también puede tener otros métodos sin implementar. Desde este punto de vista la clase Java del Listado 4.20 podría descomponerse en una clase normal con la variable de instancia `lados` y los métodos `getLados()` y `setLados()`, y una interfaz Java con el método `pinta()`.

Aunque en Objective-C no existe ningún modificador que permita declarar una clase como abstracta (p.e. `abstract` de Java), podemos crear una clase abstracta no proporcionando ningún método de clase para su creación ni ningún método de instancia para su inicialización. Observe que esto no impide que el usuario pueda ejecutar los métodos base `alloc` e `init` sobre la clase, con lo que si la clase es abstracta deberemos indicarlo en su documentación.

Objective-C también permite crear una clase en la que no implementemos todos los métodos, pero a diferencia de lo que ocurre con las clases abstractas:

1. El compilador producirá un warning avisando de que hay métodos sin implementar.

2. Podremos instanciar objetos de esta clase aunque no estén todos los métodos implementados.

11. Cluster de clases

Un **cluster de clases** es un patrón de diseño que agrupa un conjunto de subclases privadas bajo una única clase abstracta pública. Dependiendo del método que usemos para crear el objeto se nos devuelve una instancia de una clase privada optimizada para nuestro problema concreto. La Figura 4.9 muestra un cluster de clases existente en Objective-C donde las clases privadas aparecen en gris.

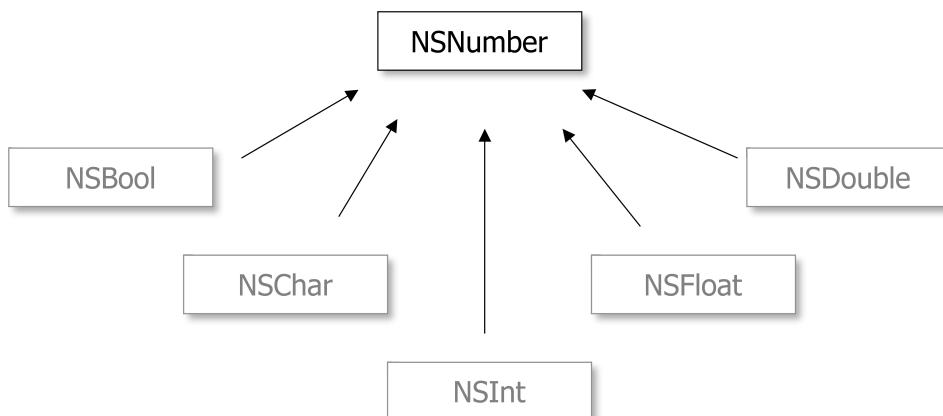


Figura 4.9: Cluster de clases

La clase `NSNumber` tiene los métodos para trabajar con números, pero no tiene una variable de instancia para almacenar el valor. Las clases privadas derivadas son las que realizan la reserva de memoria dependiendo del tipo del número a almacenar.

Dado que Objective-C no proporciona al programador las clases privadas (`NSBool`, `NSChar`, etc). ¿Cómo podemos instanciar un objeto `NSNumber`? La respuesta es usando los métodos de clase de `NSNumber` que están preparados para devolver objetos de clase privada, es decir:

```

NSNumber* booleano = [NSNumber numberWithBool:YES];
NSNumber* caracter = [NSNumber numberWithChar:'a'];
NSNumber* entero = [NSNumber numberWithInt:3];
NSNumber* flotante = [NSNumber numberWithFloat:3.6f];
  
```

Recuérdese que dado que estos métodos de clase son métodos factory, no debemos de decrementar su cuenta de referencias cuando dejemos de usar el objeto devuelto. De hecho, estos métodos factory devuelven instancias singleton (ver apartado 6.1.3) para cada los valores más comunes. Por ejemplo,

se crean objetos singleton para los valores booleano YES, NO y para los enteros más comunes -1,0,1,2.

Para acceder a los valores almacenados en `NSNumber` tenemos métodos de instancia como:

- `(BOOL)boolValue`
- `(char)charValue`
- `(int)intValue`
- `(float)floatValue`
- `(double)doubleValue`

Estos métodos realizan conversiones implícitas (casting) en caso de que sea necesario.

Tema 5

El runtime de Objective-C

Sinopsis:

Este tema está dedicado a estudiar los servicios que presta el runtime a las aplicaciones Objective-C.

Empezaremos analizando en profundidad el sistema de paso de mensajes, para pasar luego a estudiar los sistemas de gestión de memoria con que cuenta Objective-C. También repasaremos los métodos de la clase raíz que actúan de interfaz con el runtime, y que en consecuencia heredan todos los objetos. Después veremos cómo implementar la copia de objetos. Para acabar el tema estudiaremos el sistema de gestión de excepciones y de sincronización entre hilos de Objective-C.

1. Interactuar con el runtime de Objective-C

En el apartado 2 del Tema 2 explicamos que una característica que diferencia al lenguaje Objective-C de otros lenguajes orientados a objetos es que, siempre que resulta posible, aplaza las decisiones que otros lenguajes toman en tiempo de compilación para el tiempo de ejecución. Esto implica que el lenguaje necesita no sólo un compilador, sino también un runtime que proporcione servicios al lenguaje en tiempo de ejecución.

En Objective-C los programas interactúan con el runtime al menos a tres niveles:

1. A través del código fuente. La mayoría de las veces el runtime proporciona servicios al programa de forma transparente. El compilador traduce instrucciones del lenguaje en llamadas al runtime. También el compilador crea estructuras de datos a partir de la información encontrada en el código fuente, como pueda ser por ejemplo los objetos clase o los objetos protocolo.
2. A través de los métodos de la clase raíz. Los objetos heredan métodos de la clase raíz que sirven para acceder al runtime. Por ejemplo, la información de introspección (que estudiamos en el apartado 2.5 del Tema 4) usando métodos de instancia como `isKindOfClass:` o `isMemberOfClass:`, que nos permiten determinar la clase de un objeto, o métodos como `conformsToProtocol:`, que nos permite saber si un objeto cumple con un protocolo.
3. A través de llamadas a las funciones de runtime. En los ficheros de cabecera del directorio `/usr/include/objc` encontramos un conjunto de funciones C (p.e. `objc_msgSend()`) que permiten al programa interactuar directamente con el runtime de Objective-C.

En este apartado vamos a detallar cómo el runtime ayuda a Objective-C a implementar su dinamismo, cómo aprovechar al máximo los servicios del runtime, y también detallaremos la forma en que están implementados estos servicios.

2. El sistema de paso de mensajes

En el apartado 4 del Tema 3 aprendimos que una diferencia entre las llamadas a funciones y el envío de mensajes a objetos es que en el primer caso la dirección de memoria de la función a ejecutar se conoce en tiempo de compilación, mientras que, cuando mandamos un mensaje a un objeto, el método a ejecutar no se conoce hasta el tiempo de ejecución. Esto se debe a que el método a ejecutar depende del objeto que reciba el mensaje. De esta forma diferentes receptores pueden tener diferentes implementaciones para el mismo mensaje (polimorfismo). Tendremos que esperar hasta el tiempo de ejecución para que el receptor indique qué método se debe ejecutar.

En el apartado 4 del Tema 4 vimos que el receptor de un mensaje buscaba en la dispatch table de su objeto clase, y luego en las dispatch table de los objetos clase base, hasta encontrar qué método debía ejecutarse. Al recibir un mensaje, el nombre del método sirve para *seleccionar* la implementación del método a ejecutar. Por esta razón en Objective-C a los nombres de métodos se les llama **selectores**.

En el apartado 2.1 vamos a ver cómo se implementan de forma eficiente los selectores. En los apartados 2.2 y 2.3 veremos que Objective-C es un lenguaje tan dinámico que nos permite almacenar en variables, tanto el objeto receptor de un evento, como el selector a ejecutar. Finalmente, en los apartados 2.5 y 2.6 terminaremos de concretar aspectos de implementación del sistema de paso de mensajes a bajo nivel.

2.1. Los selectores

Como vimos en el apartado 4.1 del Tema 3, en Objective-C los métodos tienen un name-mangling basado en el nombre completo del método. El nombre completo estaba formado por el nombre del método, sus etiquetas y los dos puntos. Un ejemplo de nombre completo sería `stringWithCString:encoding:`.

Por eficiencia, para referirnos a un método, o para buscarlo en la dispatch table, no se usa el nombre completo del método, sino un entero de treinta y dos bits único para cada nombre completo de método, el cual está representado por el tipo `SEL`, y que se conoce como el **selector**.

Al igual que en Objective-C las clases estaban representadas en tiempo de ejecución por objetos clase, y los protocolos por objetos protocolos, los métodos en tiempo de ejecución se representan por selectores. Cuando una llamada a método es compilada, el nombre del método a llamar se indica con su selector. En tiempo de ejecución el selector se usa para buscar en la dispatch table del objeto clase el método a ejecutar.

El programador también puede conocer el selector asociado a cada nombre completo de método con la directiva del compilador `@selector()`. Por ejemplo:

```
SEL sel_imprime = @selector(imprime);
```

Para asegurar que cada nombre completo de método tenga uno, y sólo un valor de selector, el compilador genera una **tabla de métodos** con el nombre completo de todos los métodos que aparecen en el programa. La dirección de memoria de cada nombre completo de método se usa para garantizar que dos métodos con etiquetas distintas tengan distintos selectores. El valor especial `nil` se utiliza para indicar que el selector no está asignado.

Si lo que tenemos es un objeto cadena con un nombre de método también podemos obtener su selector con la función:

```
SEL NSSelectorFromString(NSString *aSelectorName);
```

Es importante tener en cuenta que los selectores representan nombres de métodos, no implementaciones de métodos. Por ejemplo, el método `imprime` tiene el mismo selector en la clase `Punto` que en la clase `Rectangulo`, sin embargo la implementación de estos métodos debe ser distinta. Esto permite que se dé el polimorfismo. También un método de clase y un método de instancia con el mismo nombre tienen asignado el mismo selector, sin embargo sus implementaciones son distintas.

2.2. Ejecutar métodos a través de selectores

Para ejecutar un método a través de su selector la clase `NSObject` tiene el método de instancia:

```
- (id)performSelector:(SEL)aSelector
```

Por ejemplo para ejecutar el método `imprime` sobre un `Punto` podríamos hacer:

```
Punto* p = [Punto new];
[p performSelector: @selector(imprime)];
```

La clase `NSObject` también proporciona otras variantes del método de instancia `performSelector:` que permiten enviar uno o dos parámetros al método ejecutado²⁸:

²⁸ En la clase `Object` de GNU estos métodos son similares, excepto que en vez de llamarse `performSelector` se llaman `perform`.

```
- (id)performSelector:(SEL)aSelector withObject:(id)anObject
- (id)performSelector:(SEL)aSelector withObject:(id)anObject
    withObject:(id)anotherObject
```

Aunque los parámetros están marcados con el tipo `id`, podemos pasar al método cualquier variable del tamaño de palabra (32 bits ó 64 bits dependiendo del runtime para el que esté compilada la aplicación). Por ejemplo punteros o enteros. Aunque los métodos `performSelector:` devuelven un `id`, al retorno se le puede hacer casting a otro tipo, pero sólo si este tipo es de tamaño menor o igual al tamaño de palabra.

Aunque estos métodos cubren la mayoría de los casos, claramente resultan insuficientes para poder ejecutar otros tipos de métodos (métodos con más de dos parámetros, o métodos que no reciben parámetros del tamaño de la palabra). En el apartado 3.2.2 del Tema 10 veremos cómo tratar estos casos.

Si, por ejemplo, queremos ejecutar el método `imprime:`, es decir, el que recibe un buffer como parámetro, sobre un objeto `Punto` podemos hacer:

```
Punto* p = [Punto new];
char buffer[255];
[p performSelector:@selector(imprime:) withObject:(id)buffer];
```

Es importante tener en cuenta que el runtime de Objective-C lo único que conoce de los métodos es el selector. Tanto los parámetros a pasar, como si estos deben ser recogidos lo deja preparado el compilador, que es el que conoce el tipo y número de parámetros, así como el tipo de retorno.

2.3. El patrón de diseño target-action

Los métodos `performSelector:` del apartado anterior reciben como parámetro una variable de tipo `SEL` con el selector a ejecutar sobre el objeto. Cuando un programa tiene una llamada a método de la forma:

```
[p imprime];
```

El compilador la transforma en algo así como:

```
[p performSelector: @selector(imprime)];
```

Pero al igual que es posible variar el receptor en tiempo de ejecución, también es posible variar el selector en tiempo de ejecución. Para ello tanto receptor como selector se pueden almacenar en variables de la forma:

```
id receptor = [Punto new];
SEL selector = @selector(imprime);
[receptor performSelector:selector];
```

El patrón de diseño target-action es un patrón de diseño muy usado por el Application Kit Framework para variar tanto el receptor como el mensaje.

Por ejemplo, tanto la clase `NSControl` como la clase `NSCellControl` pueden ser inicializadas con:

- Un receptor al que enviar un mensaje cuando el control reciba un evento externo.
- Un selector a ejecutar sobre el receptor. Esto permite indicar el tipo de evento recibido (pulsar un botón, escribir en una caja de texto, desplegar una lista).

Por ejemplo, podemos hacer:

```
[mi_boton setTarget: receptor];
[mi_boton setAction:@selector(pulsadoBoton:)];
```

Si Objective-C no permitiese que el mensaje a enviar al receptor fuese una variable todos los controles tendrían que enviar el mismo mensaje, y el mensaje a enviar estaría cableado dentro de la implementación del control. Por ejemplo, en Java, donde no se pueden almacenar selectores en variables, para implementar un receptor de eventos, lo que se hace es implementar una interfaz con nombres de métodos preparados para recibir cada posible evento: `mouseDown()`, `mouseMove()`, `mouseUp()`, ...

2.4. Evitar errores en el envío de mensajes

Un inconveniente que presenta el fuerte dinamismo de Objective-C (y que no encontramos ni en C++ ni en Java) es que si un objeto recibe un mensaje para ejecutar un método que no existe se produce una excepción en tiempo de ejecución. Este error sería parecido al error de intentar ejecutar una función que no exista, pero cuando se llama a una función que no existe es el compilador el que detecta el problema, mientras que cuando se llama a un método que no existe es el runtime el que detecta el problema.

Una forma de evitar este problema es usar punteros a objetos estáticos (que son los únicos que puede usar C++ y Java), de esta forma la detección de que el método no existe la realiza el compilador. Ante esta situación, el compilador de Objective-C genera un warning avisando de que el método llamado no está entre los métodos que tiene el puntero a objeto estático. Por el contrario en C++ y Java se genera un error de compilación.

Sin embargo, si bien el objeto receptor, o bien el selector varían en tiempo de ejecución, resulta necesario posponer este control hasta el tiempo de ejecución. Un programa Objective-C puede comprobar en tiempo de ejecución si

un receptor responde a un mensaje utilizando el método de introspección de la clase raíz:

- (BOOL) respondsToSelector: (SEL)aSelector

El método recibe como argumento el selector, y devuelve un booleano indicando si el objeto responde a este mensaje. Luego siempre que no sepamos hasta el tiempo de ejecución si un objeto va a disponer de un método debemos previamente comprobarlo de la forma:

```
if ([p respondsToSelector:@selector(imprime)])
    [p imprime];
else
    .....
```

El uso de `respondsToSelector:` es especialmente importante cuando enviamos mensajes a objetos que en tiempo de compilación no conocemos su implementación. Por ejemplo, si escribimos código que envía un mensaje a un objeto representado por una variable que un programa externo debe fijar, debemos asegurarnos de que el objeto que nos han fijado responde a los mensajes que le vayamos a enviar. En el apartado 8.9 del Tema 4 explicamos la utilidad de este método en el caso de los objetos delegados. En el apartado 3 del Tema 10 veremos que un objeto puede hacer forwarding de los mensajes que no reconoce, en este caso el objeto podría ser capaz de responder a un mensaje a pesar de que `respondsToSelector:` de la clase raíz diga que el objeto no responde a este mensaje. Para evitarlo podemos redefinir `respondsToSelector:` en una clase derivada del objeto receptor.

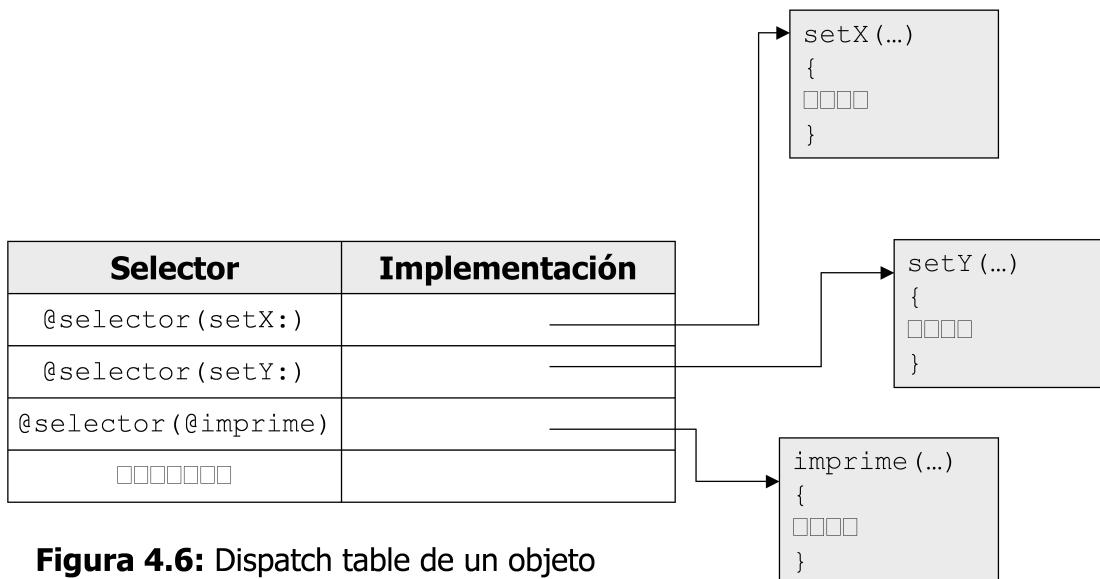
2.5. Parámetros implícitos

Como muestra la Figura 4.6, cuando el runtime encuentra un selector en la dispatch table de un objeto, la dispatch table del objeto también sirve para encontrar la dirección de memoria de la función que implementa ese método. Entonces el runtime ejecuta la función correspondiente pasándola también los parámetros (previamente depositados en la pila por el compilador).

Al llamar a la función, además de pasarla los parámetros del mensaje, se la pasan dos parámetros implícitos correspondientes a las dos partes del mensaje:

- El objeto receptor del mensaje
- El selector del mensaje

Estos parámetros se llaman **parámetros implícitos** porque no aparecen en la declaración de los métodos, a pesar de que siempre se pasan.

**Figura 4.6:** Dispatch table de un objeto

Dentro del método podemos referirnos al primer parámetro implícito (el que almacena el receptor del objeto) usando `self`, y al segundo parámetro implícito (el que almacena el selector) usando `_cmd`. Es decir, si estamos dentro del método `setX:` para obtener su selector basta con escribir `_cmd`, si estamos fuera del método `setX:` para obtener su selector deberemos escribir `@selector(setX:)`.

Dentro del método también se puede usar `super`, pero realmente éste es igual a `self`, excepto que la búsqueda del método a ejecutar comienza en la clase base del método donde se ejecuta, y la búsqueda se hace durante la compilación en vez de hacerse durante la ejecución.

A diferencia de Objective-C, en C++ y en Java sólo se pasan un parámetro implícito al método a ejecutar, que es el parámetros implícito `this`.

2.6. Cómo se envían los mensajes

En este apartado vamos a detallar cómo hace el runtime de Objective-C para enviar mensajes a los objetos. En Objective-C un mensaje no se asocia a un método hasta el tiempo de ejecución. El compilador convierte una llamada de la forma:

```
char buffer[255];
[p imprime: buffer];
```

En una llamada a la función del runtime:

```
id objc_msgSend(id theReceiver, SEL theSelector, ...)
```

Esta función recibe como primer parámetro el objeto receptor, como segundo parámetro el selector, y después recibe una lista variable de parámetros con los parámetros del objeto. Por ejemplo en el caso anterior el compilador generaría la llamada²⁹:

```
char buffer[255];
objc_msgSend(p, @selector(imprime:), buffer);
```

La función de runtime `objc_msgSend()` sólo se usa cuando la llamada a método no tiene retorno, o retorna un tipo de tamaño menor o igual al tamaño de palabra. En caso de retornar un valor más grande (p.e. un `double`) se usa la función:

```
void objc_msgSend_stret(void* stretAddr, id theReceiver,
                        SEL theSelector, ...);
```

En cuyo caso en `stretAddr` se devuelve por referencia el valor de retorno.

El runtime también proporciona otras variantes como:

```
id objc_msgSendSuper(struct objc_super* superContext,
                      SEL theSelector, ...);
```

Que se usa cuando queremos empezar la búsqueda del selector en la clase base.

Estas funciones de runtime se encargan de hacer todas las operaciones necesarias para que actúe el enlace dinámico:

- Primero usan la dispatch table de los objetos clase para, a partir del selector, encontrar el método a ejecutar.
- Despues llaman al método pasándole los parámetros adecuados.
- Por último retornan el valor resultante de ejecutar el método.

Para que las funciones de runtime puedan hacer su trabajo es necesario que los objetos de instancia tengan un puntero `isa`, y que los objetos clase tengan una dispatch table y un puntero a su superclase tal como muestra la Figura 4.2.

Para acelerar el proceso de envío de mensajes estas funciones de runtime cachean los selectores y direcciones de métodos recientemente usados. Existe una caché por cada clase, aunque una clase puede cachear tanto las llamadas a sus métodos como las llamadas a los métodos de sus clases base. Antes de que las funciones de runtime busquen un selector en las dispatch table, buscan en la caché del objeto. El uso de cachés se basa en el principio de que un método recientemente ejecutado es más probable que se vuelva a ejecutar

²⁹ El compilador genera llamadas a la funciones del runtime por nosotros, con lo que normalmente el programador nunca tendrá que llamar a estas funciones.

de nuevo en un futuro cercano. Las cachés crecen dinámicamente para acomodarse a los mensajes que va recibiendo el objeto. Si el selector está en la caché, enviar un mensaje a un objeto es sólo ligeramente más lento que ejecutar directamente la función que implementa el método. Una vez que el programa ha estado en ejecución durante algún tiempo casi todas las llamadas a método se encuentran en la caché.

3. Gestión de memoria

En C++ la reserva y liberación de memoria dinámica es responsabilidad exclusiva del programador. En Java el recolector de basura se encarga de gestionar automáticamente la liberación de la memoria dinámica reservada. Objective-C se encuentra en medio de ambas aproximaciones: El programador puede decidir entre gestionar manualmente la reserva y liberación de memoria dinámica, usar el runtime para que le ayude a llevar la cuenta de la memoria reservada, o bien olvidarse de la gestión de memoria y dejarla en manos de un recolector de basura.

3.1. Técnicas de gestión de memoria

En Objective-C actualmente existen tres técnicas de gestión de memoria:

- **Gestión de memoria manual.** Donde el programador se encarga de crear el objeto, y de indicar cuando la memoria usada por el objeto ya no se necesita, con lo que el objeto se puede liberar. Esta es la técnica de gestión de memoria típica de las clases derivadas de `Object`.
- **Gestión de memoria por cuenta de referencias.** Donde el objeto se libera cuando la cuenta de referencias llega a cero. Esta es la técnica de gestión de memoria usada por las clases derivadas de `NSObject`.
- **Gestión de memoria con recolector de basura.** Esta técnica permite que el programador se olvide de la gestión de memoria. Un recolector de basura se encarga de llevar la cuenta de los objetos referenciados desde el programa que se está ejecutando, y libera la memoria de los objetos que ya no se pueden alcanzar desde el programa.

3.1.1. Gestión de memoria manual

En la gestión de memoria manual es responsabilidad del programador el liberar la memoria de los objetos cuando éstos no se van a usar más. Ésta es la técnica utilizada por los objetos derivados de `Object`. El programador debe llamar al método `free` de la clase `Object` cuando no necesita más el objeto. Este método se puede redefinir para que también libere otros objetos o recur-

sos. La gestión de memoria manual es similar al uso de los operadores `new` y `delete` de C++.

3.1.2. Recolector de basura

El recolector de basura automatiza completamente la liberación de la memoria de los objetos sin que el programador tenga que preocuparse de realizar la gestión de memoria. Por otro lado, dejar que un recolector de basura realice la gestión de memoria tiene una penalización en el rendimiento de la aplicación. En este sentido, esta técnica sería equivalente a la gestión de memoria que realiza Java.

La gestión de memoria con recolector de basura es una característica introducida en Mac OS X 10.5, y se explica en el apartado 2 del Tema 6.

3.2. Mantener la cuenta de referencias de un objeto

En este y los siguientes apartados detallaremos paso a paso la técnica de gestión de memoria por cuenta de referencias, que es la técnica actualmente utilizada por la clase `NSObject` de Cocoa.

La técnica de cuenta de referencias asocia a cada objeto un número que indica cuantas referencias existen a este objeto³⁰. Mientras que este número es positivo el objeto está en uso. Cuando la cuenta llega a cero, la memoria ocupada por el objeto se libera.

A diferencia de la técnica del recolector de basura, la técnica de cuenta de referencias necesita de la cooperación del programador. Si el programador se olvida de decrementar la cuenta de referencias de un objeto cuando no lo va a usar más, esta memoria quedará reservada hasta que se cierre el proceso. Si por el contrario el programador decrementa la cuenta de referencias, ésta llega a cero, la memoria del objeto se libera, y luego se indirecciona un puntero al objeto, se producirá un acceso a memoria inválido que lanza una excepción. Para evitar estas circunstancias el programador debe acostumbrarse a usar una serie de patrones de programación que vamos a ir detallando en las siguientes secciones.

La clase `NSObject` proporciona cinco métodos de instancia para gestionar la cuenta de referencia a los objetos:

³⁰ Este número no se guarda como una variable de instancia del objeto, sino que es el runtime de Objective-C quien lleva a cuenta del número de referencias que existen a cada puntero a objeto. Esta forma de llevar la cuenta de referencias permite a los ingenieros de Apple cambiar en un futuro de forma transparente el mecanismo de gestión de memoria de sus objetos.

- `(id) retain`
- `(NSUInteger) retainCount`
- `(oneway void) release`
- `(id) autorelease`
- `(void) dealloc`

El método `retain` incrementa la cuenta de referencias de un objeto, `retainCount` devuelve el número de referencias que existen a un objeto, `release` y `autorelease` decrementan esta cuenta, y `dealloc` libera la memoria del objeto. Este último método sería el equivalente a `free` en la clase `Object`.

La cuenta de referencias a un objeto cambia de la siguiente forma:

- Cuando se crea un objeto (p.e. con `alloc` o `copy`) la cuenta de referencias al objeto se inicializa a uno.
- Cuando el objeto recibe un mensaje `retain` se incrementa su cuenta de referencias en una unidad. Debemos usar este método para indicar que nuestro código está usando el objeto.
- Cuando el objeto recibe un mensaje `autorelease` se almacena un mensaje `release` que deberá ser ejecutado por el framework con posterioridad. En el caso del Application Kit Framework este decremto se hace cuando se devuelve la pila de llamadas al gestor de eventos. El método `autorelease` se suele usar en dos situaciones:
 - Para decrementar la cuenta de referencias de un objeto pero poder seguir usándolo durante la ejecución del método. Esto resulta útil cuando el método que estamos implementando puede producir una excepción que impidiese llegar a las últimas instrucciones del método.
 - Para retornar un objeto y asegurar que este objeto sea liberado después de que lo use el método que lo recibe. Si el método que nos llama quiere mantener el objeto durante más tiempo que la ejecución de su cuerpo deberá enviar un mensaje `retain` al objeto que le devolvemos.
- Cuando el objeto recibe un mensaje `release` su cuenta de referencias se decremente en uno. Debemos llamar a este método para indicar que una parte del programa ya no necesita más el objeto. Este método también comprueba el valor de la cuenta de referencias, y cuando es cero llama al método `dealloc` para que libere la memoria del objeto.
- El método `dealloc` es un método que nunca debe llamar el programador directamente, sino que es `release` quien lo llama cuando la cuenta de referencias llega a cero. Sin embargo `dealloc` puede ser redefinido para liberar recursos asociados al objeto, y después usando `super` llamar al método `dealloc` de la base.

La regla general es que el número de mensajes `release` y `autorelease` que recibe un objeto durante la ejecución del programa debe ser uno más que el número de mensajes `retain` que recibe. Esto se debe a que la cuenta de

referencias de un objeto empieza en uno. La principal excepción son los métodos factory donde el número de retenciones debe de ser igual al número de liberaciones.

3.3. Política de gestión de la cuenta de referencias

En un programa Objective-C los objetos están continuamente creándose y destruyéndose. En muchas ocasiones un método o función crea un objeto, lo usa, y cuando acaba de trabajar con él lo libera. En este caso el responsable de gestionar la cuenta de referencias del objeto es el propio método o función. Sin embargo, cuando un método pasa como parámetro (o retorna) un objeto a otro método, la responsabilidad de gestionar la cuenta de referencias se diluye.

Por ejemplo, supongamos que un método de un objeto `Polígono` retorna un array con las coordenadas que lo forman:

- `(NSArray*) coordenadas;`

Este método no dice nada respecto a quién debe liberar la memoria de este array y de sus objetos. Para hacer frente a estas situaciones se han definido una serie de **políticas de gestión de cuentas de referencias**, las cuales deben ser seguidas por todos los programadores Objective-C.

Las dos reglas básicas que resumen la política de gestión de cuentas de referencias son:

- Si un trozo de programa ha creado un objeto usando un método de tipo `alloc...`, `copy...` o `mutableCopy...` (p.e. `alloc`, `copy`, `allocWithZone:`, `copyWithZone:`, `mutableCopy`, `mutableCopyWithZone:`), entonces este trozo de programa es el responsable de liberar el objeto.
- Si el programa no ha creado el objeto, entonces el programa no debería de modificar su cuenta de referencias una vez usado el objeto.

Es decir, quien crea el objeto es el responsable de liberarlo.

Hay que tener en cuenta que la liberación de los recursos y objetos agregados a nuestro objeto no se debe implementar nunca redefiniendo `release`, ya que este método se ejecuta tantas veces como se decremente la cuenta de referencias, sino que debe hacerse en el método `dealloc`, tal como se explicó en el apartado 6.3 del Tema 4.

3.4. Retornar un objeto

Cuando un método que crea un objeto lo retorna, es responsabilidad de este método el liberar la memoria del objeto. Para ello tiene dos posibilidades:

La primera es devolver un puntero a el objeto que mantiene en una variable de instancia, en cuyo caso simplemente devuelve un puntero al objeto. Por ejemplo el `Rectangulo` puede devolver uno de los puntos agregados de la forma:

```
- (Punto*) desde {
    return desde;
}
```

La segunda forma de devolver un objeto es devolviendo una copia del objeto. Para hacer una copia del objeto `Punto` puede ejecutarse el método `copy` sobre el objeto. Al ser el `Rectangulo` quien crea el objeto, él es el responsable de liberarlo. Pero ¿cómo lo libera si debe retornarlo?: La solución es usar el método `autorelease` sobre el objeto a retornar. Esto hace que el objeto retornado siga estando creado hasta que al principio de la pila de llamadas a funciones se libere la memoria. Luego ahora la forma de implementar el método `desde` sería:

```
- (Punto*) desde {
    return [[desde copy] autorelease];
}
```

En ambos casos el método que recibe el objeto sabe que el objeto será válido durante su ejecución, pero si desea guardarla (p.e. como una variable de instancia) deberá ejecutar primero `retain` sobre el objeto.

El método que recibe el objeto puede incluso retornar el objeto, ya que sabe que éste será válido hasta que se llegue al principio de la pila de llamadas.

3.5. Recibir un objeto de un ámbito superior

Cuando estemos implementando una función o método que reciba como parámetro un objeto, nuestra función no habrá creado el objeto recibido, con lo que no debe liberarlo, sino que será la función que llame a la nuestra la que se encargue de hacerlo.

Además, en este caso no debemos de suponer que la vida del objeto recibido será mayor a lo que dure la ejecución de nuestra función. Si queremos guardar el objeto (p.e. usando una variable de instancia), deberemos de ejecutar

primero `retain` sobre el objeto, y encargarnos de ejecutar `release` o `autorelease` sobre el objeto cuando ya no lo necesitemos más.

3.6. Métodos factory

En el apartado 6.1.2 del Tema 4 vimos que algunas clases disponían de métodos `factory`, que eran métodos de clase cuyo nombre tenía la forma `+clase... donde clase` es el nombre sin prefijo de la clase sobre la que se ejecuta, y también la clase del objeto que devuelve.

Los métodos `factory` se encargan de crear objetos y devolverlos pero, como es el método `factory` (y no nosotros) quien crea el objeto, corresponde al método `factory` la responsabilidad de liberarlo. En algunas ocasiones el método `factory` devuelve un objeto que simplemente no se libera nunca, y en otras devuelve un objeto sobre el que se ha ejecutado un `autorelease`. En cualquier caso nosotros no debemos de suponer que la vida del objeto devuelto será mayor a la de la ejecución del método o función desde donde lo llamamos. Si queremos almacenar el objeto, deberemos enviar al objeto el mensaje `retain`, y cuando ya no lo necesitemos más accordarnos de enviarle el mensaje `release` o `autorelease`.

En las aplicaciones iPhone OS no se recomienda usar métodos `factory` ya que la memoria de un objeto sobre el que se ha hecho `autorelease` podría tardar bastante tiempo en ser liberada. iPhone OS dispone de bastante menos memoria que Mac OS X. Sin embargo, en iPhone OS sí que se permite usar métodos `factory` en métodos `getter` (como explicamos en el apartado 3.4) que crean y retornan un método objeto sobre el que se ha hecho `autorelease`, ya que en ambos casos la eficiencia es la misma.

En el apartado 7.1 del Tema 3 indicamos que cuando usamos la directiva del compilador `@"..."` para crear un objeto `NSString`, éste se crea en el segmento de datos (y no en el heap), con lo cual este objeto nunca es liberado. Sin embargo, no debemos eludir el incremento y decremento de la cuenta de referencias de los objetos `NSString` que guardamos en variables de instancia de nuestros objetos. Esto se debe a que hay objetos `NSString` que no son creados por nuestro programa, sino que son creados por el runtime de Objective-C o de Cocoa, y estos objetos sí que son creados en el heap. En consecuencia su memoria es liberada cuando la cuenta de referencias llega a cero.

3.7. Métodos setter

En el apartado 3.4 vimos dos formas distintas de implementar un método `getter`: Retornando directamente el objeto almacenado, y retornando una

copia del objeto almacenado. Ahora vamos a ver cómo se debe implementar un método setter.

Si lo que queremos es guardar una copia del objeto recibido podemos ejecutar `copy` sobre el objeto recibido, pero antes deberemos de ejecutar `release` o `autorelease` sobre la posible copia que ya tengamos. Según esto la forma de implementar un método setter con copia sería:

```
- (void)setDesde:(Punto*)p {
    if (desde!=nil)
        [desde autorelease];
    desde = [p copy];
}
```

Debido a que Objective-C permite ejecutar métodos sobre `nil` (véase apartado 3 del Tema 4), la comprobación anterior se podría haber omitido:

```
- (void)setDesde:(Punto*)p {
    [desde autorelease];
    desde = [p copy];
}
```

Si lo que queremos es mantener un puntero al objeto recibido, podemos ejecutar `autorelease` sobre el antiguo valor almacenado, y luego `retain` sobre el nuevo valor a almacenar:

```
- (void)setDesde:(Punto*)p {
    [desde autorelease];
    desde = [p retain];
}
```

En este caso es importante ejecutar sobre el antiguo valor almacenado `autorelease`, y no `release`, ya que sino, en caso de que el nuevo puntero pasado en `p` sea igual al puntero almacenado en `desde`, podríamos estar liberando el objeto y al volver a ejecutar sobre el `retain` se produciría un acceso a memoria inválido. Otra razón por la que debemos de ejecutar `autorelease` se explicará en el apartado 3.10.

3.8. Retenciones cíclicas

En algunas ocasiones puede ocurrir que dos objetos tengan **referencias cíclicas**, es decir, cada uno tenga una referencia al otro. P.e. una ventana tiene una referencia a un control y el control tiene a su vez una referencia a la ventana. Este hecho es completamente normal y no debemos de evitarlo.

Cuando se producen referencias cíclicas entre objetos lo que no debemos de permitir es que se produzcan **retenciones cíclicas**, es decir, que cada objeto ejecute `retain` sobre la referencia del otro, ya que en este caso estos dos

objetos podrían pasar a ser inalcanzables por el programa, y sin embargo seguir teniendo una cuenta de referencias positiva. En consecuencia estos objetos nunca serían liberados.

Para evitar las retenciones cíclicas, cuando encontremos una referencia cíclica debemos de decidir cuál de los dos objetos actúa como **maestro** y cuál como **esclavo**, y sólo el maestro ejecutará `retain` sobre el esclavo. De esta forma, cuando el programa deje de tener referencias a este par de objetos el maestro alcanzará la cuenta de referencias cero y al liberarse ejecutará `release` sobre el esclavo.

3.9. Referencias débiles

Ejecutar `retain` sobre un objeto crea una **referencia fuerte (strong reference)** al objeto. La memoria de un objeto no puede ser liberada hasta que todas sus referencias fuertes hayan sido liberadas. En consecuencia el ciclo de vida de un objeto está condicionado por el número de maestros que tiene el objeto, es decir, el número de objetos que mantienen referencias fuertes al objeto.

En ocasiones este comportamiento no es el deseado. Podemos querer tener referencias a un objeto que no le impidan autodestruirse. En este caso debemos de mantener **referencias débiles (weak references)**³¹ al objeto, es decir, referencias que no ejecutan `retain` sobre el objeto. Este es el rol que desempeñaba el esclavo sobre el maestro en las referencias circulares. Es decir, el maestro mantiene una referencia fuerte a su esclavo, pero el esclavo mantiene una referencia débil a su maestro.

Un ejemplo práctico donde se producen referencias cíclicas es la jerarquía de vistas donde la vista padre tiene referencias fuertes a las vistas hijas, y las vistas hijas mantienen referencias débiles a la vista padre.

Cuando tenemos una referencia débil a un objeto debemos de tener cuidado a la hora de enviar mensajes al objeto, ya que si la memoria del objeto ha sido liberada la aplicación producirá un acceso inválido a memoria. Para evitar que esto ocurra, se usa el patrón de diseño consistente en que el objeto maestro debe informar al esclavo cuando el maestro vaya a ser liberado. De esta forma el esclavo sabe que no debe enviar más mensajes al maestro. Por ejemplo, cuando registramos un objeto observer (maestro) en un notification center (esclavo), el notification center mantiene referencias débiles a los objetos observer, y les envía mensajes cuando ocurre un determinado even-

³¹ No debemos confundir las referencias débiles que aparecen cuando estamos usando gestión de memoria por cuenta de referencias con las referencias débiles que aparecen cuando estamos usando el recolector de basura (y que veremos en el apartado 2 del Tema 6). El modificador `__weak` sólo se usa para referirse a estas últimas.

to. Cuando un objeto observer va a ser destruido, primero debe des registrarse del notification center, para que éste no le envíe más mensajes.

3.10. Validez de los objetos compartidos

La política de gestión de memoria por cuenta de referencias especifica que los objetos retornados deben permanecer válidos durante toda la ejecución del cuerpo del método que lo recibe. También debe ser posible que el método que recibe el objeto lo retorne sin temor a que deje de ser válido hasta el principio de la pila de llamadas.

Sin embargo hay una excepción a esta regla que se produce cuando modificamos en el objeto contenedor un objeto agregado que antes nos había devuelto el contenedor. Por ejemplo:

```
Rectangulo* r = [Rectangulo new];
Punto* p = [r desde];
[r setDesde: [Punto new]];
NSInteger x = [p getX]; // Posible acceso a memoria invalido
```

En este caso el rectángulo puede haber ejecutado `release` sobre el objeto Punto que nos devolvió en `p`, y al enviar el mensaje `setX:` a este objeto producirse un acceso a memoria inválido.

En el apartado 3.7 dijimos que al liberar un objeto agregado siempre debíamos de hacerlo con `autorelease`, y no con `release`. Si en la implementación de `setDesde:` de `Rectangulo` hemos seguido esta recomendación, el acceso a memoria inválido no se producirá.

Por desgracia no todas las implementaciones de objetos contenedores siguen esta recomendación, con lo que si vamos a modificar un objeto agregado de un objeto contenedor que antes nos había devuelto un objeto agregado, es recomendable primero asegurar la vida de este objeto agregado ejecutando `retain` y `autorelease` sobre él.

Es decir, una mejor forma de implementar el programa anterior sería

```
Rectangulo* r = [Rectangulo new];
Punto* p = [r desde];
[[p retain] autorelease]; // Asegura que no se destruya
[r setDesde: [Punto new]];
NSInteger x = [p getX];
```

3.11. Autorelease pools

Un **autorelease pool** es una instancia de `NSAutoreleasePool` que almacena referencias a objetos que han recibido el mensaje `autorelease`. Cuando se destruye un autorelease pool, éste envía el mensaje `release` a todos los objetos a los que almacena referencias. Si un mismo objeto ha sido puesto varias veces en el autorelease pool (enviándole varias veces el mensaje `autorelease`) el objeto recibirá ese mismo número de veces el mensaje `release`. En consecuencia, enviar un mensaje `autorelease` a un objeto, en vez de un mensaje `release`, extiende su vida hasta que se destruye el autorelease pool que lo referencia.

Para crear un autorelease pool, al igual que para crear cualquier otro objeto Objective-C, enviamos a la clase `NSAutoreleasePool` los mensajes `alloc` e `init`. Para destruir el autorelease pool le enviamos el mensaje `release`. El mensaje `drain` llama a `release` en el caso de la gestión de memoria por cuenta de referencias, y inicia un proceso de recogida de basura en caso de estar usando gestión de memoria con recogida de basura.

Se recomienda que un autorelease pool sea siempre liberado en el mismo ámbito (método, función o bloque de llaves) en que fue creado. En consecuencia normalmente no debemos de guardar un autorelease pool en una variable de instancia de un objeto, sino que se debe mantener sólo en una variable local. Esta es la razón por la que no se permite enviar a un autorelease pool los mensajes `retain` ni `autorelease`. En caso de enviar alguno de estos dos mensajes a un autorelease pool se produce una excepción en tiempo de ejecución.

También se recomienda que un programa no tenga sólo un autorelease pool, ya que si no todos los objetos que han recibido el mensaje `autorelease` no serían liberados hasta que se destruyera el único autorelease pool (normalmente al final del programa).

3.11.1. La pila de autorelease pools

Igual que la cuenta de referencias de cada objeto es mantenida por el runtime de Objective-C, el runtime también mantiene la cuenta de los autorelease pools existentes en la **pila de autorelease pools**. Cuando un objeto recibe el mensaje `autorelease` el runtime guarda una referencia a este objeto en el autorelease pool que está en la cima de la pila. Cuando un autorelease pool se destruye es eliminado de la cima de la pila. Aunque los autorelease pool son guardados por el runtime en una pila, muchas veces se dice que los autorelease pools están *anidados* para indicar que gestionan la liberación de objetos de distintos ámbitos, o niveles de profundidad, del programa.

La posibilidad de anidar autorelease pools nos permite crear autorelease pool en cualquier método o función. Por ejemplo, la función `main()` puede crear un autorelease pool, y llamará a otras funciones que crean otros autorelease pools. Incluso un mismo método o función puede tener varios autorelease pools, uno para el exterior de un bucle y otro para el interior del bucle (en el apartado 3.11.2 veremos un ejemplo de este tipo de anidamiento). Existe un efecto lateral con el anidamiento de autorelease pools cuando se produce una excepción que debemos de contemplar en nuestro programa como explicaremos en el apartado 3.11.4.

Cocoa siempre espera que exista al menos un autorelease pool. Si no existe ningún autorelease pool los objetos que reciban el mensaje `autorelease` no serán nunca liberados y se producirá un memory leak en el programa. Para ayudarnos a detectar estos problemas, cuando enviamos un mensaje `autorelease` a un objeto y no existe un autorelease pool, el runtime de Objective-C escribe un mensaje de log por la salida de errores estándar.

El Application Kit Framework crea siempre un autorelease pool al principio de cada evento (por ejemplo mover el ratón) y lo destruye al final del evento. En consecuencia, normalmente los programas hechos dentro de este framework no tienen que preocuparse de crear sus propios autorelease pools. Sin embargo existen otras ocasiones donde el programador tiene que ocuparse de crear sus propios autorelease pools:

- Si la aplicación no está basada en el AppKit Framework, por ejemplo en el caso de aplicaciones de consola.
- Si se crea un segundo hilo, el programador debe crear un autorelease pool al principio de este hilo. En caso contrario se producirán memory leaks. En el apartado 3.11.3 detallaremos este aspecto.
- Si se tiene un bucle con gran carga computacional que crea muchos objetos temporales. En este caso podemos reducir el número de objetos pendientes de liberar si creamos un autorelease pool por cada iteración del bucle, y el autorelease pool es destruido al final de cada iteración.

3.11.2. Autorelease pools fuera del AppKit Framework

En una aplicación de consola que va a usar las librerías de Apple (las derivadas de `NSObject`) normalmente debemos empezar creando un autorelease pool al principio de la función `main()`, y destruirlo al final con `release` (o `drain`).

En el apartado 3.1 del Tema 1 vimos que cuando creamos una aplicación de consola con Xcode, por defecto nos genera una función `main()` que lo único que tiene es un autorelease pool. Sin embargo esta forma de construir el programa hace que los objetos que han recibido el mensaje `autorelease` no

sean destruidos hasta el final del programa. Si la aplicación de consola que estamos construyendo consume mucha memoria, sería recomendable crear otros autorelease pools en ámbitos interiores.

El Listado 5.1 muestra un ejemplo de comando implementado con varios autorelease pools, uno a nivel de función `main()`, y otro que se crea cada vez que se repite el bucle. De esta forma si el procesado de cada fichero que recibe el comando implica crear muchos objetos, estos objetos serán liberados de memoria antes de empezar a procesar el siguiente fichero.

```
/* autoreleasepools.m */
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) {
    // Crea un autorelease pool global
    NSAutoreleasePool* pool = [NSAutoreleasePool new];
    NSInteger i;
    for (i=1;i<argc;i++) {
        // Crea un autorelease pool por cada iteracion
        NSAutoreleasePool* pool_bucle =
            [NSAutoreleasePool new];
        // Procesa un fichero
        NSString* fichero =
            [NSString stringWithCString:argv[i]];

        // Libera los objetos creados durante esta iteracion
        [pool_bucle release];
    }
    // Destruye el autorelease pool global
    [pool drain];
    return EXIT_SUCCESS;
}
```

Listado 5.1: Ejemplo de comando con varios autorelease pools

3.11.3. Aplicaciones con varios hilos

El runtime de Objective-C mantiene una pila de autorelease pools por cada hilo. Cuando un hilo termina, éste debe liberar todos los autorelease pools que ha creado.

Cuando creamos un hilo que no realiza ninguna llamada a Cocoa, este hilo no necesita crear un autorelease pool.

En las aplicaciones que usan el AppKit Framework el programador no tiene que crear un autorelease pool en la función `main()`, ya que es el propio framework el que se encarga de crearlo. Además el AppKit Framework crea un autorelease pool por cada evento que lanza.

3.11.4. Retroceder en la pila de autorelease pools

Sabemos que cuando un objeto recibe el mensaje `autorelease`, el runtime lo añade al autorelease pool de la cima de la pila. Si luego se crea un nuevo autorelease pool, el autorelease pool anterior mantiene sus referencias a objetos sin liberar, pero deja de almacenar nuevos objetos hasta que el autorelease pool que se puso encima de él se destruye.

Es posible enviar el mensaje `release` a un autorelease pool que no está en la cima de la pila, en cuyo caso primero se destruyen los autorelease pool que tiene encima, y luego se destruye él. Esta propiedad resulta útil en dos ocasiones:

- Si olvidamos enviar el mensaje `release` a un autorelease pool, cuando se envía el mensaje `release` al autorelease pool que tiene debajo, también se libera el autorelease pool olvidado.
- Si se produce una excepción se retrocede por la pila de llamadas a funciones hasta que una función captura la excepción. Si durante este retroceso se dejó sin ejecutar la liberación de algún autorelease pool, el siguiente autorelease pool que se libere también liberará los autorelease pool que tenga encima. Hay que tener en cuenta que en este caso sólo se liberarán los objetos que recibieron el mensaje `autorelease` antes de producirse la excepción, pero objetos que no recibieron ni el mensaje `release` ni el mensaje `autorelease` pueden quedar inaccesibles, aunque ocupando memoria, es decir, producirse un memory leak. Para solucionar este problema, en el apartado 6.4 veremos que deberemos prevenirlo realizando las operaciones `release` correspondientes en un bloque `@finally`.

Otro problema con el que se puede encontrar el programador es el de querer pasar un objeto que está marcado para autorelease en un autorelease pool, a otro autorelease pool que esté situado una posición más abajo en la pila. Para ello simplemente ejecutamos `retain` sobre el objeto a pasar a un autorelease pool inferior, destruimos el autorelease pool de la cima, y ejecutamos `autorelease` sobre el objeto para pasarlo al nuevo autorelease de la cima. Por ejemplo, en el Listado 5.2 tenemos un ejemplo de una función que busca un objeto a partir de un `código`. En cada iteración del bucle se crea un autorelease pool que limpia los objetos temporales que se generan durante la iteración. Sin embargo, en caso de encontrarse el objeto (en la variable `buscado`) para pasarlo al autorelease pool de nivel inferior ejecutamos sobre él `retain`, destruimos el autorelease de la cima, al haber ejecutado `retain` sobre el objeto a bajar de nivel, éste no se destruye, y una vez que cambiamos de autorelease pool activo volvemos a ejecutar `autorelease` sobre el objeto para que su referencia se añada al nuevo autorelease pool de la cima.

```
id EncuentraObjeto(NSInteger código) {  
    id buscado = nil;  
    while (buscado==nil) {
```

```

NSAutoreleasePool* subpool = [NSAutoreleasePool new];
// Busqueda que produce muchos objetos temporales
buscado = BusquedaCostosa(codigo);
if (buscado!=nil)
    [buscado retain];
[subpool release];
}
return [buscado autorelease];
}

```

Listado 5.2: Ejemplo de objeto que cambia de autorelease pool

4. Las clases raíz

Ya sabemos que Objective-C proporciona al menos dos clases raíz: La clase raíz `Object` de GNU y la clase raíz `NSObject` de Cocoa. Conocer en profundidad estas clases es importante porque tienen la funcionalidad común de todos los objetos, incluyendo características como la gestión de memoria o la introspección.

Tanto `Object` como `NSObject` declaran sólo la variable de instancia `isa` que apunta al objeto clase del objeto.

Cocoa, además de la clase `NSObject` proporciona el protocolo `NSObject`, que a su vez está implementado por la clase `NSObject`. En el apartado 8.5 del Tema 4 explicamos la utilidad de este protocolo.

Los métodos de las clases raíz son heredados tanto por los objetos de instancia como por los objetos clase, y algunos de ellos están pensados para ser redefinidos, mientras que otros no deberían de ser redefinidos.

En los siguientes apartados vamos a clasificar por grupos funcionales, y a detallar los principales métodos de cada una de estas clases raíz. Para cada grupo funcional vamos a realizar una comparativa entre los métodos que proporciona `Object` y los que proporciona `NSObject`. Cuando el método no exista en ambas clases raíz, o no tenga el mismo nombre, se indicará.

4.1. Creación, copia y liberación de objetos

En general `NSObject` soporta la mayoría de las características de `Object` y añade otras nuevas. Además los métodos de `NSObject` suelen estar más estáticamente tipificados que los de `Object`. Por ejemplo, el método `class` de `NSObject` retorna un objeto de tipo `Class`, mientras que el método `class` de `Object` retorna un `id`.

Para la creación y destrucción de objetos Objective-C nunca debemos de usar las funciones `malloc()` y `free()`, sino que debemos usar los métodos que vamos a comentar en este apartado.

```
+ (id)initialize  
+ (void)initialize
```

La primera forma existe sólo en `Object` y la segunda sólo en `NSObject`. Este método de clase lo ejecuta el runtime sobre cada clase antes de que se cree ninguna instancia de la clase. Podemos sobrescribirlo para hacer inicializaciones a nivel de clase tal como se explicó en el apartado 2.7 del Tema 4. En el caso de `Object` el método devuelve `self`, es decir, devuelve el objeto clase sobre el que se ejecuta.

```
+ (id)alloc
```

Reserva memoria para un objeto del tamaño de las variables de instancia de la clase sobre la que se ejecuta, y pone toda la memoria a cero excepto el primer campo `isa` que apunta al objeto clase. Retorna el objeto creado, o `nil` si no se pudo reservar memoria. Las subclases no deben redefinir este método, sino que la inicialización del objeto se hace en un método de instancia `init`. En el caso de `NSObject`, el objeto devuelto tiene la cuenta de referencias a uno, y debe ser liberado con `release`.

```
- (id)init
```

Es el inicializador de máxima genericidad de `Object` y de `NSObject`, y por defecto se limita a retornar `self`. Debe ser redefinido por las derivadas para realizar inicializaciones adicionales, tal como se explica en el apartado 6.2 del Tema 4.

```
+ (id)new
```

Llama a `alloc` sobre la clase, y sobre el objeto de instancia devuelto ejecuta `init`.

```
- (id)copy
```

Como se explica en el apartado 5, este método permite crear una copia del objeto sobre el que se ejecuta.

```
- (id)shallowCopy
```

Este método existe sólo en `Object`. Como se explica en el apartado 5.1, este método permite crear una copia plana del objeto sobre el que se ejecuta.

```
- (id)deepCopy
```

Este método existe sólo en `Object`. Como se explica en el apartado 5.1, este método permite crear una copia en profundidad del objeto sobre el que se ejecuta.

- `(id)deepen`

Este método existe sólo en `Object`. Como se explica en el apartado 5.1, podemos sobrescribir este método para que cree una copia de los objetos agregados al objeto.

- `(id)mutableCopy`

Este método existe sólo en `NSObject`. Como se explica en el apartado 5.2, permite crear un objeto copia mutable (variable) del objeto sobre el que se ejecuta.

- `(id)free`

Este método existe sólo en `Object`. Libera la memoria del objeto sobre el que se ejecuta. Además pone la variable de instancia `isa` a cero, con lo que cualquier mensaje que se intentara enviar a este objeto produciría un acceso a memoria inválido.

- `(void)dealloc`

Este método existe sólo en `NSObject`. Libera la memoria del objeto sobre el que se ejecuta. Normalmente nunca debería de ser ejecutado por el programador, sino que lo ejecuta `release` cuando la cuenta de referencias llega a cero. El programador puede redefinirlo para liberar los recursos asociados al objeto.

4.2. Identificar objetos y clases

Los métodos que vamos a comentar en este apartado sirven para obtener información sobre un objeto y sobre la clase a la que pertenece, así como para comparar objetos entre sí.

- + `(id)class`
- `(id)class`
- + `(Class)class`
- `(Class)class`

El método `class` está implementado tanto a nivel de clase como a nivel de instancia. Los dos primeros sólo existen en `Object`, y los dos segundos sólo en `NSObject`. El método de clase que se ejecuta sobre el nombre de la clase devuelve `self`. Por el contrario el método de instancia que se ejecuta sobre un objeto instancia devuelve el puntero `isa`. Recuérdese que si enviábamos

el mensaje `class` a un objeto clase, el método de clase se encontraba antes que el método de instancia de la clase raíz, con lo que se devuelve `self`. Además, enviar un mensaje a un objeto clase es lo mismo que enviar un mensaje al nombre de la clase correspondiente.

```
+ (id)superclass
- (id)superclass
+ (Class)superclass
- (Class)superclass
```

El método `superclass` está implementado tanto a nivel de clase como a nivel de instancia. Los dos primeros sólo existen en `Object`, y los dos segundos sólo en `NSObject`. El método de clase se ejecuta sobre el nombre de la clase, y devuelve el objeto clase de su clase base. Para ello devuelve el valor al que apunta su variable de instancia `super_class` (véase Figura 4.3). El método de instancia también devuelve el objeto clase de su clase base. Para ello se ejecuta sobre el objeto de instancia, accede a su objeto clase usando la variable de instancia `isa`, y luego usa la variable de instancia `super_class` para encontrar el objeto clase a devolver (véase Figura 4.3). De nuevo, recuérdese que si enviamos el mensaje `superclass` a un objeto de clase se encontrará el método de clase antes que el método de instancia.

```
- (id)self
```

Retorna el receptor del mensaje. No debemos confundir el método `self` con la palabra reservada `self` que se usa dentro de un método para referirse a su objeto. Aunque tanto [Punto `class`] como [Punto `self`] devuelven el objeto clase de `Punto`, posiblemente la segunda forma resulte más clara.

```
+ (const char*)name
- (const char*)name
- (NSString*)className
```

Los dos primeros métodos están implementados sólo en `Object`, y el tercer método en `NSObject`. Todos ellos devuelven el nombre de la clase. El método de clase devuelve el nombre de la clase del objeto clase sobre el que se ejecuta, y el método de instancia el nombre de la clase del objeto de instancia sobre el que se ejecuta.

```
- (BOOL)isEqual:(id)anObject
```

Devuelve YES si el puntero al receptor tiene la misma dirección de memoria que `anObject`. Podemos sobrescribir este método para que compruebe la relación de igualdad en base a las variables de instancia del objeto.

```
- (NSUInteger)hash
```

Devuelve un entero que sirve para indexar un objeto en una tabla hash. Por defecto devuelve la dirección de memoria del objeto. Si dos objetos son iguales (desde el punto de vista de `isEqual:`), `hash` debe devolver el mismo número, con lo que si redefinimos `isEqual:` debemos de redefinir también `hash`.

- `(NSString*)description`

Devuelve una descripción textual del objeto. Por defecto devuelve una cadena con el nombre de la clase del objeto y la dirección de memoria del objeto. Los objetos pueden redefinirlo para devolver una mejor descripción de su contenido. El comando `print-object` de `gdb` envía el mensaje `description` a un objeto Objective-C para obtener una descripción textual de su contenido e imprimirlo.

4.3. Introspección de jerarquía y protocolos

Podemos usar los siguientes métodos para comprobar la posición de la clase de un objeto en la jerarquía de clases, así como consultar qué protocolos adopta.

- `(BOOL)isMemberOf:aClass`
- `(BOOL)isMemberOfClass:(Class)aClass`

Estos métodos indican si un objeto es exactamente de la clase `aClass`. El primero existe sólo en la clase `Object`, y el segundo existe sólo en la clase `NSObject`. Obsérvese que el primero recibe un parámetro de tipo `id` mientras que el segundo recibe un parámetro de tipo `Class`.

- `(BOOL)isKindOfClass:aClass`
- `(BOOL)isKindOfClass:(Class)aClass`

Estos métodos indican si un objeto es de la clase `aClass`, o de alguna de sus superclases. El primero existe sólo en la clase `Object` y el segundo existe sólo en la clase `NSObject`. Obsérvese que el primero recibe un parámetro de tipo `id` mientras que el segundo recibe un parámetro de tipo `Class`.

- + `(BOOL)conformsTo:(Protocol*)aProtocol`
- `(BOOL)conformsTo:(Protocol*)aProtocol`
- + `(BOOL)conformsToProtocol:(Protocol*)aProtocol`
- `(BOOL)conformsToProtocol:(Protocol*)aProtocol`

Estos métodos comprueban si la clase de un objeto adopta un determinado protocolo. Los dos primeros existen sólo en la clase `Object`, y los dos segundos sólo en `NSObject`.

4.4. Introspección de métodos

Los métodos de este apartado permiten preguntar por los mensajes a los que responde un objeto, y obtener información sobre la forma y parámetros de estos métodos.

- `(BOOL) respondsTo: (SEL)aSelector`
- `(BOOL) respondsToSelector: (SEL)aSelector`

El primero sólo existe en la clase `Object` y el segundo sólo existe en la clase `NSObject`. Ambos métodos permiten preguntar si el objeto receptor implementa o hereda el método cuyo selector se da como parámetro.

- + `(BOOL) instancesRespondsToSelector: (SEL)aSelector`
- + `(BOOL) instancesRespondToSelector: (SEL)aSelector`

El primero sólo existe en la clase `Object` y el segundo sólo existe en la clase `NSObject`. Ambos preguntan a una clase si sus instancias responden a un mensaje (implementado o heredado) cuyo selector se da como parámetro. Para preguntar a un objeto clase cuando él, y no sus instancias, responde a un mensaje debemos enviar `respondToSelector:` al objeto clase.

- `(IMP) methodFor: (SEL)aSelector;`
- `(IMP) methodForSelector: (SEL)aSelector`

El primero sólo existe en la clase `Object` y el segundo sólo existe en la clase `NSObject`. Si el receptor responde al selector indicado como parámetro, devuelve la dirección de memoria donde está implementado el método. Estos métodos se usan para acelerar la ejecución de métodos como se explica en el apartado 6 del Tema 10.

- + `(IMP) instanceMethodFor: (SEL)aSelector`
- + `(IMP) instanceMethodForSelector: (SEL)aSelector`

El primero sólo existe en la clase `Object` y el segundo sólo existe en la clase `NSObject`. Sirven para preguntar a un objeto clase por la implementación de un método de instancia. Para preguntar a un objeto clase por la implementación de un método de clase deberemos enviar al objeto clase el mensaje `methodFor:` o `methodForSelector:,` dependiendo de si la raíz es `Object` o `NSObject`.

- + `(struct objc_method_description*) descriptionForInstanceMethod: (SEL)aSel`
- `(struct objc_method_description*) descriptionForMethod: (SEL)aSel`

Estos métodos sólo existen en la clase `Object` y permiten obtener una descripción de un método dado su selector.

```
- (NSMethodSignature*)
methodSignatureForSelector:(SEL)aSelector
```

Este método sólo existe en `NSObject` y permite obtener una descripción de un método dado su selector. Cuando se trate de un método de instancia el receptor debe ser un objeto de instancia, y cuando se trate de un método de clase el receptor debe ser un objeto clase. En el apartado 3.2.2 del Tema 10 se darán más detalles sobre su funcionamiento.

5. Copia de objetos

Cuando un objeto tiene sólo variables de instancia de tipo fundamental (aparte de la variable de instancia `isa`), hacer una copia del objeto consiste sólo en duplicar la memoria del objeto en otro trozo de memoria. Cuando alguna variable de instancia es un puntero, se pueden realizar dos tipos de copias del objeto:

- **Copia plana** que duplica los punteros de forma que los objetos agregados son compartidos por el nuevo objeto y por el original.
- **Copia en profundidad** que duplica los objetos apuntados por los punteros con el fin de no compartirlos entre el nuevo objeto y el original. La copia en profundidad se ejecuta de forma recursiva, de forma que si un objeto agregado tiene otros objetos agregados, éstos también se duplican.

Los objetos Objective-C también se pueden clasificar en objetos mutables e inmutables:

- Un **objeto mutable** es un objeto cuyo contenido puede cambiar una vez creado.
- Un **objeto inmutable** es aquel cuyo contenido no puede cambiar una vez creado. Por ejemplo los objetos `NSString` son inmutables. Su versión mutable está representada por `NSMutableString`.

Cuando una función o método de copia (p.e. `copy`) devuelve una copia de un objeto, devuelve su cuenta de referencias a uno, de forma que el receptor deberá ejecutar `release` sobre el objeto cuando ya no lo necesite más. Una importante optimización que se puede llevar a cabo cuando se copia un objeto inmutable es que, en vez de duplicar la memoria que éste ocupa, se puede simplemente ejecutar `retain` sobre él. Cuando un objeto agregado sea mutable debemos hacer una copia de él, y devolverlo con su cuenta de referencias a uno. En ambos casos, cuando no necesitemos más el objeto, debemos ejecutar el método `release` sobre él.

La forma de realizar una copia de un objeto depende de si estamos usando la clase raíz `Object` o `NSObject`. Aunque ambas clases raíz comparten el méto-

do `copy`, la forma de implementar la copia varía considerablemente, con lo que en los próximos dos apartados vamos a describir cómo se implementa la copia con cada una de las clases raíz.

5.1. Copia de objetos Object

La clase `Object` proporciona cuatro métodos para realizar la copia de un objeto:

- `(id) copy`

Realiza una copia en profundidad del objeto. Por defecto está implementado de forma que realiza una copia plana del objeto, lo cual es suficiente si el objeto sobre el que lo ejecutamos no tiene objetos agregados. Si el objeto tiene objetos agregados se recomienda redefinir `copy` para realizar una copia en profundidad de los objetos agregados.

- `(id) shallowCopy`
- `(id) deepen`
- `(id) deepCopy`

Estos tres métodos están por defecto sin implementar, con lo que cualquier intento de llamarlos acaba produciendo una excepción. Podemos implementarlos con la siguiente semántica:

- `(id) shallowCopy`

Devuelve una copia plana del objeto.

- `(id) deepen`

Reemplaza todos los objetos agregados con copias de éstos.

- `(id) deepCopy`

Devuelve una copia en profundidad del objeto.

En caso de que el objeto sea inmutable podemos redefinir `copy`, `shadowCopy` y `deepCopy` para que se limiten a llamar a `retain` sobre el objeto, en vez de copiarlo. En este caso no debemos incrementar la cuenta de referencias de los objetos agregados. Además en este caso `deepen` no debe, ni incrementar la cuenta de referencias, ni modificar los objetos agregados.

En caso de que el objeto sea mutable `shadowCopy` debe devolver un objeto con la cuenta de referencias a uno. A los métodos `copy` y `deepCopy` se pueden implementar de forma que primero llaman a `shadowCopy` y luego a `deepen`. El método `deepen` se implementaría de forma que incrementa la

cuenta de referencias de los objetos agregados inmutables y hace copia de los objetos agregados mutables.

5.2. Copia de objetos `NSObject`

En caso de estar usando `NSObject` como clase raíz, debemos usar el método de instancia:

- `(id)copy`

definido en la clase `NSObject` para copiar un objeto.

5.2.1. La interfaz `NSCopying`

El método `copy` se limita a llamar a:

- `(id)copyWithZone:(NSZone*)zone`

con zona por defecto (es decir `nil`), y devuelve el resultado de llamar a este método.

El método `copyWithZone:` no está definido en la clase `NSObject`. El protocolo formal `NSCopying` es el que declara como único método el método `copyWithZone::`. En consecuencia no todos los objetos derivados de `NSObject` pueden copiarse, sólo aquellos que adopten el protocolo `NSCopying`. Si enviamos el mensaje `copy` a un objeto, y `copy` no encuentra el método `copyWithZone:` en el objeto, se lanza una excepción.

El protocolo `NSCopying` lo pueden implementar tanto objetos mutables como inmutables. Si el objeto es inmutable se puede implementar llamando a `retain` sobre el objeto a copiar. Si el objeto es mutable, pero alguno de sus objetos agregados es inmutable, se puede copiar los objetos agregados mutables y ejecutar `retain` sobre los objetos agregados inmutables.

5.2.2. Objetos mutables e inmutables

En este apartado vamos a ver cómo se crean objetos Cocoa que pueden ser copiados de dos formas, una mutable (creando otra copia en memoria) y otra inmutable (incrementando la cuenta de referencias).

El protocolo `NSMutableCopying` es un protocolo que dispone de un único método:

```
- (id)mutableCopyWithZone: (NSZone*) zone
```

Sólo cuando un objeto puede ser copiado de dos formas: Una mutable y otra inmutable, es cuando se debe implementar el protocolo `NSMutableCopying`. Si el objeto sólo es mutable o inmutable (la mayoría de los casos) se debe implementar sólo `NSCopying`.

Un objeto que implementa `NSCopying` y `NSMutableCopying` está indicando que se puede copiar de dos formas: Una mutable y otra inmutable. Si queremos crear una copia del objeto que no vamos a modificar (inmutable) ejecutaremos sobre el objeto el método `copy`, el cual llamará a `copyWithZone:`. Si por el contrario queremos crear una copia que pretendemos modificar, ejecutaremos sobre el objeto el método de la clase `NSObject`:

```
- (id)mutableCopy
```

Al igual que `copy` está implementado de forma que llama sobre sí mismo a `copyWithZone:`, el método `mutableCopy` está implementado de forma que llama sobre sí mismo a `mutableCopyWithZone:`. Análogamente, en caso de no estar implementado `mutableCopyWithZone:` en la clase del objeto, se produce una excepción.

Un ejemplo de clase que adopta los protocolos `NSCopying` y `NSMutableCopying` es la clase `NSString`. De esta forma podemos crear una copia inmutable con `copy`, o una copia mutable con `mutableCopy`³².

En el apartado 1.4 del Tema 7 se verá un ejemplo más práctico con `NSArray`. Esta clase también adopta los protocolos `NSCopying` y `NSMutableCopying`.

5.3. Método getter/setter y liberación de objetos agregados

Cuando un objeto implementa `NSCopying` y tiene objetos agregados podemos implementar los métodos `setter` que acceden a estos objetos de la forma:

```
- (void)setDesde:(Punto*) p {
    [desde autorelease];
    desde = [p copy];
}
```

³² No es necesario que la clase del objeto que implemente el comportamiento inmutable/mutable sea diferente. De hecho, actualmente tanto el objeto que devuelve `copy` como el objeto que devuelve `mutableCopy` es una instancia de la clase derivada `NSCFString`, la cual se comporta como un objeto mutable o inmutable dependiendo de su estado interno.

De esta forma, en caso de que el objeto agregado sea inmutable el método `copy` simplemente ejecutará un `retain` sobre el nuevo objeto recibido. Por el contrario, si el objeto es mutable se hará una copia independiente del objeto `p`. Análogamente podemos implementar el método `getter` de la forma:

```
- (Punto*) desde {
    return [[desde copy] autorelease];
}
```

De nuevo, si el objeto `desde` es inmutable, estaremos devolviendo un puntero a nuestro objeto agregado con la cuenta de referencias incrementada, mientras que si el objeto `desde` es mutable, estaremos devolviendo una copia de nuestro objeto agregado.

Cuando un objeto se libera, debe ejecutarse sobre todos sus objetos agregados (excepto que sean punteros `weak`) el método `release`. En caso de ser inmutables los objetos agregados, estaremos decrementando la cuenta de referencias sobre ellos. En caso de ser objetos mutables (de los cuales tendremos una copia), estaremos liberando la copia.

6. Gestión de excepciones

En este apartado vamos a explicar cómo tratar excepciones en el lenguaje Objective-C.

El mecanismo que vamos a explicar es el nuevo sistema de gestión de excepciones basado en bloques `@try-@catch` introducido en Mac OS X 10.3. Anteriormente existía otro mecanismo basado en crear bloques `NS_DURING`, `NS_HANDLER`, que no vamos a explicar aquí. Para utilizar este nuevo sistema de gestión de excepciones debe usarse el flag `-fobjc-exceptions`³³ durante la compilación. Este flag hace que la aplicación no pueda ser ejecutada en sistemas anteriores a Mac OS X 10.3.

6.1. El bloque `@try-@catch`

El nuevo sistema de gestión de excepciones es parecido al de C++ o Java. El Listado 5.3 muestra un ejemplo de cómo usar el bloque `@try-@catch`. El `autorelease pool` es necesario, ya que cuando el runtime lanza una excepción, el objeto que se pasa durante la excepción (el objeto `NSEException`) recibe un mensaje `autorelease` antes de ser lanzada la excepción.

³³ En Xcode 3.0 la opción se llama `Enable Objective-C Exceptions`, y está habilitada por defecto.

```
/* trycatch.h */
#import <stdio.h>

int main (int argc, const char * argv[]) {
    NSAutoreleasePool* pool = [NSAutoreleasePool new];
    @try {
        NSString metodoInexistente];
    }
    @catch(NSEException* ex) {
        printf("Se ha producido la excepcion: %s\n"
               , [[ex reason] UTF8String]);
    }
    [pool release];
    return EXIT_SUCCESS;
}
```

Listado 5.3: Ejemplo de bloque @try-@catch

6.2. Lanzar excepciones con @throw

Además de las excepciones que lanza el runtime cuando detecta algún problema en tiempo de ejecución, nuestro programa también puede lanzar excepciones usando la directiva del compilador `@throw`. El Listado 5.4 muestra un ejemplo de cómo lanzar una excepción usando la directiva del compilador `@throw`.

```
/* throw.h */
#import <stdio.h>

int main (int argc, const char * argv[]) {
    NSAutoreleasePool* pool = [NSAutoreleasePool new];
    @try {
        @throw [[NSEException alloc] initWithName: @"Prueba"
               reason:@"Algo va mal" userInfo:nil] autorelease];
    }
    @catch(NSEException* ex) {
        printf("Se ha producido la excepcion: %s\n"
               , [[ex reason] UTF8String]);
    }
    [pool release];
    return EXIT_SUCCESS;
}
```

Listado 5.4: Ejemplo de lanzar una excepción con `@throw`

Observe que ejecutamos `autorelease` sobre la excepción para que se libere la memoria una vez procesada la excepción.

En C++ se permite lanzar con `throw` cualquier tipo de variable (enteros, cadenas de caracteres, punteros, ...). Por el contrario en Java sólo se permite

lanzar objetos derivados de `Throwable`, y normalmente se recomienda que sean de tipo `Exception` o derivados. En Objective-C sólo se permite lanzar con `@throw` punteros a objetos. Aunque el lenguaje permite lanzar cualquier objeto, se recomienda lanzar sólo objetos de tipo `NSEException` o derivados³⁴. Esto se debe a que los objetos `NSEException` tienen variables de instancia donde se puede almacenar una explicación de la causa de la excepción.

Las excepciones se pueden volver a lanzar una vez que han sido capturadas (en el bloque `@catch`) de la forma:

```
@catch(NSEException* ex) {
    @throw ex;
}
```

En cuyo caso se vuelve a retroceder por la pila de llamadas a funciones hasta que otro bloque `@catch` capture la excepción.

6.3. Usar varios bloques `@catch`

A un bloque `@try` le podemos poner varios bloques `@catch`. Por ejemplo, el siguiente bloque `@try` tiene cuatro bloques `@catch`, cada uno preparado para capturar excepciones de un tipo. El runtime va buscando secuencialmente hasta que encuentra un bloque `@catch` cuyo tipo acepta la excepción lanzada. El último de ellos actúa como comodín, es decir, captura todo tipo de objetos, y es donde va a parar el objeto lanzado, ya que su tipo no coincide (desde el punto de vista de `isKindOfClass:`) con ningún bloque anterior.

```
@try {
    @throw [[[NSObject alloc] init autorelease]];
}
@catch(NSEException* ex) {
    printf("Se ha producido una excepcion %s\n"
          , [[ex reason] UTF8String]);
}
@catch(NSError* err) {
    printf("Se ha producido error con codigo %i"
          , [err code]);
}
@catch(NSString* msg) {
    printf("Se ha lanzado el mensaje %s"
          , [msg UTF8String]);
}
@catch(id obj) {
    printf("Se ha lanzado el objeto %s"
          , [[obj description] UTF8String]);
}
```

³⁴ En Objective-C no es tan común como en Java crear derivadas de `NSEException`.

6.4. El bloque @finally

El bloque `@catch` sólo se ejecuta si se produce una excepción, luego si queremos decrementar la cuenta de referencias de un objeto tendremos que contemplar el caso de que se produzca, y el de que no se produzca la excepción, más o menos de la forma:

```
NSObject* obj = [[NSObject alloc] init];
@try {
    // Operaciones con obj
    .....
    [obj release];
}
@catch(NSEException* ex) {
    printf("Se ha producido una excepcion %s\n"
          , [[ex reason] UTF8String]);
    [obj release];
}
```

En caso de no producirse ninguna excepción se ejecutará sólo el primer `release`, y si se produce excepción se ejecutará sólo el segundo `release`.

Sin embargo, resulta más cómodo indicar en el bloque `@finally` liberaciones de recursos que se deban de realizar independientemente de que se produzca, o no, una excepción. Para ello podemos escribir el decreimiento de la cuenta de referencias anterior de la forma:

```
@try {
    // Operaciones con obj
    .....
}
@catch(NSEException* ex) {
    printf("Se ha producido una excepcion %s\n"
          , [[ex reason] UTF8String]);
}
@finally {
    [obj release];
}
```

En caso de producirse la excepción, primero se ejecutará el bloque `@catch` y luego el bloque `@finally`.

También es posible escribir un bloque `@finally` sin bloque `@catch` de la forma:

```
@try {
    // Operaciones con obj
    .....
}
@finally {
```

```
[obj release];  
}
```

En cuyo caso, si se produce la excepción primero se ejecuta el bloque `@finally`, y luego se empieza a retroceder por la pila de llamadas a funciones buscando un bloque `@catch`.

6.5. Excepciones y errores

En Java normalmente cualquier error en tiempo de ejecución se modela como una excepción. Sin embargo en lenguajes como C++ u Objective-C es más común que los errores en tiempo de ejecución se traten con la técnica tradicional de devolver un código de error.

De hecho, en Objective-C no se recomienda usar excepciones para informar de un error en tiempo de ejecución. Esto se debe principalmente a dos razones:

- No existe una forma clara de que el lenguaje indique que un método o función lanza un tipo de excepción (como ocurre en Java con las lista de `throws`). Con lo que la única forma de saber si un método o función lanza una excepción es consultando la documentación.
- Tradicionalmente no se han usado excepciones para informar de los errores en tiempo de ejecución.

Por todo ello, en Objective-C sólo se recomienda usar excepciones para representar problemas en tiempo de ejecución no esperados por el programador³⁵. Por ejemplo salirse fuera de los límites de un array, agotarse la memoria, o enviar un mensaje desconocido a un objeto.

Siempre que un error en tiempo de ejecución sea previsible (por ejemplo que un fichero no exista, que una conexión de red no se pueda abrir, o que se produzca un error parseando un fichero XML), deberemos de devolver un código de error, y opcionalmente llenar con información del error un puntero a objeto `NSError`, que nos pasen por referencia.

En definitiva: Los objetos `NSError` o derivados se usan para representar errores inesperados en tiempo de ejecución. Los objetos `NSError` se usan para representar errores que se sabe podrían producirse en tiempo de ejecución.

Se acostumbra a permitir que el parámetro `NSError` sea opcional, y la función o método que produce el error lo rellene con información del error sólo si

³⁵ En Java este tipo de excepciones son los derivadas de la clase `RuntimeException`

lo recibe. Es decir, la forma de implementar una función o método que podría producir un error sería más o menos ésta:

```
int ProcesaFichero (char* fichero, NSError** error) {
    if /*fichero no existe*/ {
        if (error!=NULL) {
            // Rellena error con una descripción del error
            *error = [[NSError alloc] init];
            .....
            [error autorelease];
        }
        return ERROR;
    }
    .....
}
```

Para inicializar un objeto `NSError` normalmente se usa su método de instancia:

- `(id)initWithDomain:(NSString*)domain
 code:(NSInteger)code
 userInfo:(NSDictionary*)dict`

Tradicionalmente los errores de Mac OS X se han agrupado en dominios. Por ejemplo, los errores POSIX pertenecen a un dominio, o los errores de Carbon pertenecen a otro dominio. La Tabla 5.1 muestra los dominios que actualmente tiene Mac OS X. Cada dominio está representado por un objeto `NSString`. El parámetro `domain` debe ser uno de estos valores que están almacenados en constantes dentro de `NSError.h`:

```
const NSString *NSPOSIXErrorDomain;
const NSString *NSOSStatusErrorDomain;
const NSString *NSMachErrorDomain;
const NSString *NSCocoaErrorDomain;
```

Dominio	Fichero de cabecera
NSMachErrorDomain	/usr/include/mach/kern_return.h
NSPOSIXErrorDomain	/usr/include/sys/errno.h
NSOSStatusErrorDomain	<CarbonCore/MacErrors.h>
NSCocoaErrorDomain	<Foundation/FoundationErrors.h> <AppKit/AppKitErrors.h> <CoreData/CoreDataErrors.h>

Tabla 5.1: Dominios de los errores Mac OS X y ficheros de cabecera con sus códigos

```
/*
 * Error codes
 */
#define EPERM          1    /* Operation not permitted */
#define ENOENT         2    /* No such file or directory */
#define ESRCH          3    /* No such process */
#define EINTR          4    /* Interrupted system call */
```

```

#define EIO 5 /* Input/output error */
#define ENXIO 6 /* Device not configured */
#define E2BIG 7 /* Argument list too long */
#define ENOEXEC 8 /* Exec format error */
#define EBADF 9 /* Bad file descriptor */
#define ECHILD 10 /* No child processes */
#define EDEADLK 11 /* Resource deadlock avoided */
/* 11 was EAGAIN */
#define ENOMEM 12 /* Cannot allocate memory */
#define EACCES 13 /* Permission denied */
#defineEFAULT 14 /* Bad address */
.....

```

Listado 5.5: Listado de códigos de error en `errno.h`

Además de un dominio, los errores tienen asociado un código (parámetro `code`), que dentro del dominio especifica la causa del error. Varios dominios pueden tener el mismo código con significados diferentes, luego antes de consultar el código es necesario conocer el dominio. La Tabla 5.1 también muestra en qué ficheros de cabecera se resumen los códigos de cada dominio. El Listado 5.5 muestra un ejemplo de los códigos de error POSIX que se encuentran dentro de `errno.h`.

Por último un `NSError` lleva asociado un diccionario (parámetro `userInfo`) que opcionalmente puede llevar varias entradas con mensajes textuales para describir el error, y por ejemplo, mostrar este mensaje textual al usuario. Consulte la documentación de referencia para ver cómo obtener, usar, e internacionalizar estos mensajes.

6.6. El handler de excepciones no capturadas

Si se produce una excepción y ningún bloque `@catch` la captura, el programa terminará con la señal `SIGTRAP`. Sin embargo es posible fijar una función que queremos que se ejecute antes de terminar el programa llamada **handle de excepciones no capturadas**. Para fijar el handler de excepciones no capturadas se usa la función³⁶:

```
void NSSetUncaughtExceptionHandler(
    NSUncaughtExceptionHandler *handler);
```

Donde `NSUncaughtExceptionHandler` está definido como un puntero a la función que actúa como handler de excepciones no capturadas, y que tiene la forma:

```
typedef volatile void
NSUncaughtExceptionHandler(NSException *exception);
```

³⁶ El handle de excepciones no capturadas no funciona en las aplicaciones gráficas que usan el Application Kit Framework porque `NSApplication` captura todas las excepciones.

El Listado 5.6 muestra cómo usar un handle de excepciones no capturadas para liberar un autorelease pool. Para ello declaramos el autorelease pool como una variable de módulo, y fijamos la función que actúa como handle de excepciones no capturadas con `NSSetUncaughtExceptionHandler()`. La llamada a `exit(0)` hace que el programa termine con el código de estado 0 (`EXIT_SUCCESS`) en vez de lanzarse la señal `SIGTRAP`.

```
/* handle.h */
#import <stdio.h>

static NSAutoreleasePool* pool;

void HandleExcepcionesNoCapturadas(NSException* ex) {
    printf("Se produjo la excepcion no capturada: %s\n"
           , [[ex reason] UTF8String]);
    printf("Liberando el autorelease pool\n");
    [pool release];
    exit(EXIT_SUCCESS);
}

int main (int argc, const char * argv[]) {
    pool = [NSAutoreleasePool new];
    NSSetUncaughtExceptionHandler(
        HandleExcepcionesNoCapturadas);
    @throw [[[NSEException alloc] initWithName: @"Prueba"
                                         reason:@"Algo va mal" userInfo:nil] autorelease];
    [pool release];
    return EXIT_SUCCESS;
}
```

Listado 5.6: Ejemplo de handler de excepciones no capturadas

6.7. El handle de excepciones por defecto

Si no modificamos el handle de excepciones no capturadas se ejecuta el **handle de excepciones no capturadas por defecto**. Este handle por defecto se limita a imprimir un mensaje de log en la salida estándar con la excepción capturada, y luego deja que se produzca la `SIGTRAP`.

Podemos obtener una instancia del objeto `NSEExceptionHandler`³⁷ para personalizar el comportamiento del handle de excepciones por defecto de acuerdo a la Tabla 5.2.

Existen tres tipos de excepciones ante las que puede responder el handle de excepciones por defecto:

³⁷ Para poder usar esta clase debemos de enlazar con `ExceptionHandling.framework`, e importar el fichero de cabecera `<ExceptionHandling/NSEExceptionHandler.h>`

- Excepciones `NSEException` no capturadas.
- Excepciones del sistema (p.e. un acceso inválido a memoria).
- Excepciones del runtime (p.e. un mensaje enviado a un objeto liberado).

Acción	Constante (entre paréntesis valor)
Genera un log cuando se produce una excepción <code>NSEException</code> no capturada.	NSLogUncaughtExceptionMask (1)
Captura las excepciones <code>NSEException</code> para que no produzcan una señal <code>SIGTRAP</code> .	NSHandleUncaughtExceptionMask (2)
Genera un log cuando se produce una excepción de sistema no capturada.	NSLogUncaughtSystemExceptionMask (4)
Captura las excepciones de sistema para que no produzcan una señal <code>SIGTRAP</code> .	NSHandleUncaughtSystemExceptionMask (8)
Genera un log cuando el runtime produce una excepción <code>NSEException</code> no capturada.	NSLogUncaughtRuntimeErrorMask (16)
Captura las excepciones producidas por el runtime para que no produzcan una señal <code>SIGTRAP</code> .	NSHandleUncaughtRuntimeErrorMask (32)

Tabla 5.2: Tipos de acciones para el handle de excepciones por defecto

Por ejemplo, si queremos que se capturen las excepciones (para que no se produzca la señal `SIGTRAP`), pero no queremos que se haga log de ninguna excepción no capturada podemos hacer:

```
[ [NSEExceptionHandler defaultExceptionHandler]
    setExceptionHandlingMask:2+8+32];
```

Además de modificar este comportamiento para un programa, podemos modificarlo para todos los programas Objective-C con el comando `defaults`. Para ello debemos ejecutar desde el terminal el siguiente comando:

```
$ defaults write NSGlobalDomain NSEexceptionHandlingMask 42
```

7. Bloques sincronizados

Objective-C permite que se ejecuten varios hilos en una misma aplicación. Esto significa que dos hilos podrían intentar modificar el mismo objeto al mismo tiempo lo cual da lugar a una serie de problemas conocidos como **race conditions**. Para evitar estos problemas hay que identificar qué partes del código son **secciones críticas** y protegerlas mediante el uso de un **mutex**. Objective-C proporciona un mecanismo de sincronización parecido al de Java basado en el uso de **bloques sincronizados**.

La directiva del compilador `@synchronized()` permite marcar secciones críticas³⁸. La directiva del compilador recibe como parámetro un objeto Objective-C que actúa como mutex, es decir, como objeto que sólo puede ser poseído por un hilo a vez. Si un hilo posee el mutex y otro lo solicita, el que lo solicita tiene que esperar a que el que lo posea lo libere.

En Objective-C (a diferencia de Java) la directiva `@synchronized` no se puede poner a nivel de método, sólo se pueden crear bloques sincronizados. Aun así, se usa el término **método sincronizado** para referirse a un método donde toda su implementación está dentro de un bloque sincronizado.

Aunque cualquier objeto Objective-C puede actuar como mutex, hay tres objetos típicos que se utilizan como mutex dependiendo del ámbito de sincronismo que se desea: A nivel de método, a nivel de objeto, o a nivel de clase. A continuación vamos a explicar estos ámbitos de sincronismo.

En el apartado 2.5 indicamos que el parámetro implícito `_cmd` es el selector del método en el que estamos situados, con lo que podemos usar `_cmd` para obtener un mutex que sólo bloquee la sección crítica contenida en un determinado método. Debido a que `_cmd` es una variable de tipo `SEL` (y no un objeto Objective-C), podemos usar la función `NSStringFromSelector()` para obtener un objeto singleton `NSString` correspondiente a el selector dado como parámetro. Luego la forma de implementar un **mutex a nivel de método** sería:

```
- (void)metodoCritico {
    @synchronized(NSStringFromSelector(_cmd)) {
        // Sección crítica
        ...
    }
}
```

³⁸ Para poder utilizar esta directiva debemos de pasar al compilador la opción `-fobjc-exceptions`, la cual existe sólo desde Mac OS X 10.3 y no permite crear aplicaciones con versiones de Mac OS X anteriores. Por defecto Xcode 3.0 crea proyectos con esta opción activada.

El mutex más usado es el **mutex a nivel de objeto**, el cual hace que una vez que un hilo entra en algún método de un objeto, ningún otro hilo puede entrar en ningún otro método del mismo objeto. El parámetro implícito `self` es el mutex a nivel de objeto que se suele usar dentro de los métodos de un objeto de la forma:

```
- (void)metodoCritico {
    @synchronized(self) {
        // Sección crítica
        ...
    }
}
```

Por último, si queremos implementar un **mutex a nivel de clase**, el cual es útil para los métodos de clase, podemos usar el objeto clase de la forma:

```
+ (void)metodoClaseCritico {
    @synchronized(self) {
        // Sección crítica
        ...
    }
}
```

Obsérvese que ahora, al ser un método de clase, `self` se refiere al objeto clase.

Existen dos diferencias entre los bloques sincronizados de Objective-C y Java:

- En Java no existe el parámetro implícito `_cmd`, con lo que si queremos implementar un sincronismo a nivel de método en Java deberíamos de implementar un mecanismo alternativo.
- En Java podemos poner el modificador `synchronized` en el prototipo de un método sincronizado, con lo que se utilizaría `this` para implementar su correspondiente bloque sincronizado. En Objective-C la directiva del compilador `@synchronized` no se puede poner como un modificador de método.

Objective-C soporta accesos reentrantes a una sección crítica con lo que un método con un bloque sincronizado puede llamarse de forma recursiva desde el mismo hilo, aunque el número de entradas en el bloque sincronizado deberá ser igual al número de salidas para que se libere el mutex.

Cuando se lanza una excepción en un bloque sincronizado, se retrocede por la pila de llamadas y cuando se sale del bloque sincronizado se libera automáticamente el mutex asociado al bloque sincronizado, con lo que otros hilos podrían entrar en el bloque sincronizado. Sería posible colocar un bloque `@finally` dentro del bloque sincronizado para que antes de liberar el mutex se restaure el estado de los recursos del bloque sincronizado, con el fin de que otros hilos que entren en el bloque sincronizado encuentren estos recursos en un estado coherente.

Tema 6

Objective-C 2.0

Sinopsis:

Mac OS X 10.5 (Leopard) introdujo nuevas características en el lenguaje Objective-C que hicieron que al lenguaje se le pasase a llamar Objective-C 2.0. En este tema vamos a comentar estas características. En concreto veremos dos nuevas características: las propiedades y la gestión de memoria con recolector de basura. Las enumeraciones rápidas son otra característica introducida en Objective-C 2.0 que se explicará en el apartado 3.2 del Tema 7, ya que su ámbito de aplicación se limita a las colecciones.

Tenga en cuenta que el uso de estas nuevas características implica generar aplicaciones que sólo ejecutarán en el runtime de Mac OS X 10.5 o posterior.

1. Las propiedades

Los métodos getter/setter ayudan a encapsular las variables de instancia de un objeto, pero su implementación es una tarea tediosa y repetitiva. Las propiedades ayudan a mejorar la productividad del programador realizando esta tarea de forma más rápida. Además, las propiedades permiten personalizar la forma en que queremos realizar estos accesos.

Debemos tener en cuenta que las propiedades sólo están destinadas a encapsular el acceso a variables de instancia. En Objective-C las clases no tienen variables de clase (a excepción de las variables de instancia `isa` y `super_class`) y en consecuencia las propiedades no se pueden usar para implementar el acceso a las variables de clase.

El uso de propiedades se puede dividir en tres pasos: declaración, implementación y acceso. En los siguientes apartados vamos a describir cada uno de estos pasos.

1.1. Declarar propiedades

La declaración de una propiedad se hace con la directiva del compilador `@property`. Esta directiva debe de aparecer después de las variables de instancia en la interfaz de la clase tal como muestra el Listado 6.1. Obsérvese que el nombre de la propiedad coincide con el nombre de su variable de instancia correspondiente.

```
/* Punto.m */
#import <Foundation/Foundation.h>

@interface Punto : NSObject {
    NSInteger x;
    NSInteger y;
}
@property NSInteger x;
@property NSInteger y;
@end
```

Listado 6.1: Declaración de la clase `Punto` con propiedades

Las propiedades no sólo se pueden usar en la interfaz de las clases, sino que también se pueden usar en categorías, extensiones y protocolos. En caso de usar una categoría, extensión o protocolo, la clase que implemente la correspondiente categoría, extensión o protocolo deberá disponer de las variables de instancia correspondientes, ya que las categorías, extensiones y protocolos no pueden reservar memoria para variables de instancia.

1.2. Implementar propiedades

En la implementación de la clase podemos pedir al compilador que implemente los métodos getter/setter para acceder a la propiedad. Para ello usamos la directiva `@synthesize` tal como muestra el Listado 6.2.

```
/* Punto.m */
#import "Punto.h"

@implementation Punto
@synthesize x;
@synthesize y;
@end
```

Listado 6.2: Implementación de la clase `Punto` con propiedades

Cuando el compilador encuentra las directivas `@property` y `@synthesize`, las expande respectivamente en una declaración y una implementación de métodos getter/setter tal como muestra el Listado 6.3 y Listado 6.4.

```
/* Punto.h */
#import <Foundation/Foundation.h>

@interface Punto : NSObject {
    NSInteger x;
    NSInteger y;
}
- initWithX:(NSInteger)paramX y:(NSInteger)paramY;
- (NSInteger)x;
- (void)setX:(NSInteger)param;
- (NSInteger)y;
- (void)setY:(NSInteger)param;
@end
```

Listado 6.3: Declaración de la clase `Punto` sin propiedades

```
/* Punto.m */
#import "Punto.h"

@implementation Punto
- initWithX:(NSInteger)paramX y:(NSInteger)paramY {
    if (self = [super init]) {
        x = paramX;
        y = paramY;
    }
    return self;
}
- (NSInteger)x {
    return x;
}
- (void)setX:(NSInteger)param {
```

```

    x = param;
}
- (NSInteger)y {
    return y;
}
- (void)setY:(NSInteger)param {
    y = param;
}
@end

```

Listado 6.4: Implementación de la clase `Punto` sin propiedades

De hecho, en vez de haber usado la directiva del compilador `@synthesize`, podríamos haber implementado nosotros mismos los métodos getter/setter tal como se implementan en el Listado 6.4.

1.3. Acceso a propiedades

Si tenemos la clase `Punto` del Listado 6.1 y Listado 6.2, podemos usar los métodos getter/setter para acceder a los valores de las propiedades como normalmente.

```
[p setX:3];
[p setY:7];
printf("Los valores del punto son [%i,%i]\n", [p x], [p y]);
```

1.4. Modificadores de la propiedad

El formato general de la declaración de una propiedad es:

```
@property(modificadores) tipo nombre;
```

Donde *tipo* y *nombre* son el tipo de dato y nombre de la variable que actúa como propiedad. Los *modificadores* son opcionales, y en caso de no proporcionar modificadores se deben omitir los paréntesis. En caso de proporcionar varios modificadores, se separan por comas. La Tabla 6.1 recopila los modificadores de propiedad existentes.

Modificador	Descripción
assign	Especifica que el método setter hace una asignación simple y que el método getter se limita a retornar el valor de la variable de instancia correspondiente. Éste es el modificador por defecto si no se especifica modificador. Se recomienda para tipos de datos simples y para punteros a objetos Objective-C cuando se está usando el recolector de basura.
retain	El modificador es válido sólo cuando la variable de instancia

	que representa la propiedad es un puntero a objeto Objective-C. Especifica que el método getter debe comprobar que la variable de instancia correspondiente no sea <code>nil</code> , en cuyo caso ejecuta <code>retain</code> seguido de <code>autorelease</code> sobre la variable de instancia antes de devolverla. El método setter por su parte deberá comprobar que la variable de instancia no sea <code>nil</code> , en cuyo caso ejecutará <code>release</code> sobre la variable de instancia para después ejecutar <code>retain</code> sobre el nuevo valor recibido que va a almacenar en la variable de instancia. Éste es el modificador que se recomienda para los punteros a objetos Objective-C cuando se usa la gestión de memoria por cuenta de referencias.
<code>copy</code>	Este modificador es válido sólo cuando la variable de instancia que representa la propiedad es un puntero a objeto Objective-C. El método getter se comporta exactamente igual que con el modificador <code>retain</code> , es decir, el método getter debe comprobar que la variable de instancia no sea <code>nil</code> , en cuyo caso ejecuta <code>retain</code> seguido de <code>autorelease</code> sobre el objeto a devolver. Por el contrario, el método setter hace una copia del valor recibido durante la asignación. En concreto, el método setter debe comprobar que la variable de instancia no sea <code>nil</code> , y en este caso ejecutar <code>release</code> sobre la variable de instancia, para luego ejecutar <code>copy</code> sobre el objeto recibido, con el fin de guardar una copia del objeto. Además este objeto deberá implementar la interfaz <code>NSCopying</code> . En caso contrario no se producirá un warning, pero en tiempo de ejecución se producirá un error al no encontrarse el método <code>copyWithZone:</code> .
<code>readonly</code>	Indica que la propiedad debe disponer de métodos getter y setter. Es el modificador por defecto. En caso de usar la directiva del compilador <code>@synthesize</code> en la implementación, ésta generará tanto el método getter como el setter.
<code>readwrite</code>	Indica que la propiedad debe disponer sólo de método getter. En caso de usar la directiva del compilador <code>@synthesize</code> en la implementación, ésta generará sólo el método getter.
<code>nonatomic</code>	Hace que los métodos getter y setter no se implementen de forma atómica. Por defecto los métodos getter/setter de las propiedades los genera <code>@synthesize</code> atómicos, es decir, encerrados en un bloque sincronizado.
<code>getter=</code>	Indica el método que debe actuar como getter de la propiedad.
<code>setter=</code>	Indica el método que debe actuar como setter de la propiedad.

Tabla 6.1: Modificadores de propiedad

En caso de no estar usando el recolector de basura para la gestión de memoria, independientemente de que estemos usando el modificador de propiedad `assign`, `retain` o `copy`, será responsabilidad del objeto contenedor el hacer `release` o `autorelease` de los objetos almacenados en sus variables de instancia al acabar la vida del objeto contenedor. Normalmente este decremento en la cuenta de referencias se hace en el método `dealloc`.

Los modificadores `assign`, `retain` y `copy` son excluyentes, su finalidad se describe en la Tabla 6.1, y el compilador sigue distintas políticas con ellos dependiendo de si estamos usando el recolector de basura. En concreto:

- Si no estamos usando el recolector de basura, cuando la propiedad represente un puntero a objeto Objective-C deberemos indicar un modificador: `assign`, `retain` o `copy`. En caso contrario el compilador emitirá un warning. Esto nos ayuda a pensar sobre el comportamiento que queremos que tengan los métodos getter y setter respecto a la gestión de memoria.
- Si estamos usando el recolector de basura no obtendremos un warning si no indicamos ninguno de estos modificadores y por defecto se usará el modificador `assign`. La excepción es cuando el objeto Objective-C apuntado por la propiedad implemente la interfaz `NSCopying`, ya que en ese caso se producirá un warning avisando de que el objeto implementa la interfaz `NSCopying` y deberíamos considerar el uso del modificador `copy`.

Los modificadores `readonly` y `readwrite` también son mutuamente excluyentes, y su finalidad se describe en la Tabla 6.1.

La directiva del compilador `@synthesize` genera métodos getter y setter atómicos, es decir, que son robustos ante el acceso concurrente de varios hilos al mismo método. Estos métodos corresponden conceptualmente a encerrar los métodos getter y setter en un bloque `@synchronized`, tal como se explica en el apartado 7 del Tema 5. Aunque conceptualmente los métodos getter y setter están encerrados en un bloque `@synchronized`, en la práctica los métodos que produce `@synthesize` utilizan bloqueo optimista, que es estadísticamente más eficiente³⁹. En caso de no necesitar esta funcionalidad podemos usar el modificador de propiedad `nonatomic` para que los métodos getter y setter no se implementen de forma atómica.

³⁹ En el bloqueo pesimista se adquiere un cerrojo para la sección crítica y es el que implementan los bloques `@synchronized`. En el bloqueo optimista no se usan cerrojos para la sección crítica, sino que se usan instrucciones del procesador para actualización atómica de variables, llamadas Compare And Swap (CAS), junto con el hecho de que normalmente dos hilos no ejecutan a la vez el mismo método (razón por la que es estadísticamente más eficiente). Con este mecanismo se guarda una copia de la variable a actualizar antes de ejecutar la operación atómica y después de ejecutar la operación atómica se comprueba que la variable actualizada contiene el mismo valor que la variable que devolvió la operación CAS. Si son distintos significa que otro hilo ha actualizado concurrentemente la variable y se repite el intento.

En ocasiones podemos preferir no seguir la convención de nombrado de los métodos getter y setter asociados a una propiedad. En este caso podemos usar los modificadores `getter=nombreMetodo` y `setter=nombreMetodo` para indicar el método que queremos que actúe como getter o setter de la propiedad.

El Listado 6.5 muestra un ejemplo de declaración con modificadores en las propiedades que es más completo que el ejemplo del Listado 6.1. El ejemplo empieza definiendo una interfaz `Fecha` con tres propiedades. El tipo `NSNumber` es un tipo inmutable, y en consecuencia es seguro utilizar el modificador de propiedad `retain`, ya que si un método getter devuelve un objeto de tipo `NSNumber`, éste no podrá ser modificado por el receptor.

La clase `Fecha` implementa el protocolo `Fecha` y añade dos propiedades más: La propiedad `bisiesto`, es de sólo lectura y su método getter no se llamará `bisiesto` sino `esBisiesto` ya que hemos usado el modificador `getter=` para personalizar este método getter. La propiedad `mannana` también es de sólo lectura y el modificador `copy` hace que la propiedad devuelva una copia de un objeto de tipo `Fecha`, con lo que el receptor podría modificar este objeto sin afectar al objeto `Fecha` original.

```
/* Fecha.h */
#import <Foundation/Foundation.h>

@protocol Fecha
@property(retain) NSNumber* dia;
@property(retain) NSNumber* mes;
@property(retain) NSNumber* anno;
@end

@interface Fecha : NSObject <Fecha, NSCopying> {
    NSNumber* dia;
    NSNumber* mes;
    NSNumber* anno;
}
@property(nonatomic,readonly,getter=esBisiesto) BOOL bisiesto;
@property(nonatomic,readonly,copy) Fecha* mannana;
- init;
- initWithDia:(NSNumber*)d mes:(NSNumber*)m anno:(NSNumber*)a;
- (BOOL)esBisiesto;
- (id)copyWithZone: (NSZone*) zone;
- (NSString*)description;
- (void)dealloc;
@end
```

Listado 6.5: Ejemplo de declaración con modificadores en las propiedades

1.5. Personalizar la implementación

Cuando en la interfaz de una clase, categoría o extensión aparece una propiedad, o bien se hereda una propiedad de un protocolo (como ocurre en el Listado 6.5), en la implementación correspondiente es necesario hacer una de estas tres cosas:

1. Usar la directiva del compilador `@synthesize` para que el compilador genere automáticamente sus métodos `getter` y `setter`.
2. Implementar los métodos `getter` y `setter` (o bien sólo el método `getter` en caso de que la propiedad tenga el modificador `readonly`).
3. Usar la directiva del compilador `@dynamic`.

La directiva del compilador `@synthesize` sólo genera métodos `getter` y `setter` si éstos no existen, en caso de que alguno de estos métodos exista, `@synthesize` sólo generaría el que no exista.

Además, la directiva del compilador `@synthesize` nos permite indicar el nombre de la variable de instancia que se asocia a la propiedad, en caso de que estos nombres no coincidan usando la forma:

```
@synthesize nombrePropiedad = nombreVarInstancia;
```

El Listado 6.6 muestra la implementación correspondiente al Listado 6.5. Observe que el nombre de la propiedad `anno` no coincide con el nombre de la variable de instancia `ano` con lo que la directiva del compilador asocia la variable de instancia a la propiedad de la forma:

```
@synthesize anno=ano;
```

La directiva del compilador `@dynamic` tiene el formato:

```
@dynamic nombrePropiedad;
```

Y sirve para indicar al compilador que cumpliremos con el contrato que implica la propiedad `nombrePropiedad` utilizando técnicas dinámicas como el forwarding. Estas técnicas se verán en el apartado 3 del Tema 10.

```
/* Fecha.m */
#import "Fecha.h"

@implementation Fecha
@synthesize dia;
@synthesize mes;
@synthesize anno=ano;
- init {
    return [self initWithDia:[NSNumber numberWithInt:1]
                  mes:[NSNumber numberWithInt:1]
                 anno:[NSNumber numberWithInt:1900]];
}
- initWithDia:(NSNumber*)d mes:(NSNumber*)m anno:(NSNumber*)a{
```

```

    if (self = [super init]) {
        [self setDia: d];
        [self setMes: m];
        [self setAnno: a];
    }
    return self;
}
- (BOOL)esBisiesto {
    if ( ([ano intValue]%4==0 && [ano intValue]%100!=0)
        || [ano intValue]%400==0 )
        return YES;
    return NO;
}
- (Fecha*)mannana {
    Fecha* m = [self copy];
    [m setDia:[NSNumber numberWithInt:[[m dia] intValue]+1]];
    // Para simplificar el algoritmo suponemos que todos
    // los meses tienen 30 dias
    if ([[m dia] intValue]==31) {
        [m setDia:[NSNumber numberWithInt:1]];
        [m setMes:[NSNumber numberWithInt:
                    [[m mes] intValue]+1]];
        if ([[m mes] intValue]==13) {
            [m setMes:[NSNumber numberWithInt:1]];
            [m setAnno:[NSNumber numberWithInt:
                        [[m anno] intValue]+1]];
        }
    }
    return m;
}
- (id)copyWithZone: (NSZone*)zone {
    return [[Fecha allocWithZone:zone] initWithDia:dia
                                         mes:mes
                                         anno:anno];
}
- (NSString*)description {
    return [NSString stringWithFormat:@"%@:%@:%@"
                           ,dia,mes,ano];
}
- (void)dealloc {
    [dia release];
    [mes release];
    [ano release];
    [super dealloc];
}
@end

```

Listado 6.6: Ejemplo de implementación con modificadores en las propiedades

Los métodos inicializadores `init` y `initWithDia:mes:anno` se han creado de acuerdo a las reglas que estudiamos en el apartado 6.2 del Tema 4. En el método `initWithDia:mes:anno` hemos copiado los parámetros a las variables de instancia del objeto usando la forma:

```
[self setDia: d];
```

Y no la forma:

```
dia = d;
```

Porque queremos que se ejecute `retain` sobre los objetos recibidos, y de esta forma controlar la cuenta de referencias a los objetos almacenados en el objeto `Fecha`.

La implementación del método `mannana` devuelve una copia del objeto `Fecha` con el fin de cumplir con el modificador de propiedad `copy`. Obsérvese que el modificador de propiedad `copy` en este caso se puede omitir sin obtener un warning, ya que no es `@synthesize` quién lo implementa. Sin embargo, lo hemos puesto para se ayude a documentar el comportamiento del método.

La implementación del método `dealloc` es necesaria para liberar la memoria de los objetos `NSNumber` asociados al objeto `Fecha`, cuando el objeto `Fecha` se destruya.

1.6. El operador punto

El operador punto se usa en C para acceder a los campos de una estructura. Por ejemplo:

```
struct Fecha {
    int dia;
    int mes;
    int anno;
} f1;
f1.dia = 23;
printf("Hoy estamos a día %i\n", f1.dia);
```

Objective-C 2.0 ha extendido el operador punto para poder usarlo en el acceso a las propiedades de un objeto Objective-C de forma más cómoda y concisa. Por ejemplo, si tenemos la clase `Fecha` del Listado 6.5, podemos hacer:

```
Fecha* f1 = [[Fecha alloc] init];
f1.dia = [NSNumber numberWithInt:23];
f1.mes = [NSNumber numberWithInt:12];
f1.anno = [NSNumber numberWithInt:2007];
NSLog(@"%@", "Hoy es dia %@ del mes %@ del año %@", f1.dia, f1.mes, f1.anno);
```

El compilador traduce las llamadas al operador punto por llamadas a sus correspondientes métodos `getter` y `setter`, con lo que las sentencias anteriores serían equivalentes a:

```
Fecha* f1 = [[Fecha alloc] init];
[f1 setDia: [NSNumber numberWithInt:23]];
[f1 setMes: [NSNumber numberWithInt:12]];
[f1 setAnno: [NSNumber numberWithInt:2007]];
NSLog(@"%@", "Hoy es dia %d del mes %d del año %d",
       [f1 dia], [f1 mes], [f1 anno]);
```

De hecho, en Objective-C 2.0 aunque no hayamos declarado las propiedades usando la directiva del compilador `@property`, siempre que existan los correspondientes métodos getter y setter que sigan la convención de nombres de las propiedades podemos usar el operador punto para ejecutar los métodos getter y setter.

También es posible combinar el operador punto con llamadas a métodos getter:

```
int d = [[f1 manana] dia].intValue;
```

1.6.1. Acceso a variables de instancia y propiedades

Hay que tener en cuenta que el operador punto nunca accede directamente a las variables de instancia, sino que accede a los métodos getter y setter de la propiedad. En caso de que las variables de instancia sean públicas podríamos acceder a las variables de instancia directamente usando el operador flecha como vimos en el apartado 3 del Tema 3:

```
Punto* p = [Punto new];
p->x=3;
p->y=7;
```

Lógicamente esta forma de acceso sólo se recomienda para variables de instancia de tipo fundamental, ya que si las variables de instancia son objetos (como ocurre en el Listado 6.5) debemos evaluar si conviene hacer un acceso con modificador `assign`, `retain` o `copy`.

Cuando estamos dentro de un objeto y queramos acceder a una propiedad del propio objeto debemos usar explícitamente el operador `self` de la forma:

```
self.dia = [NSNumber numberWithInt:5];
```

Ya que si no estamos accediendo directamente a la variable de instancia asociada a la propiedad:

```
dia = [NSNumber numberWithInt:5];
```

Este es un despiste muy común que da lugar a errores en la gestión de memoria por cuenta de referencias. La razón es que si la propiedad `dia` tiene el modificador de propiedad `retain`, al asignar a `self.dia` estamos incrementando la cuenta de referencias, mientras que al asignar a `dia` no. Este pro-

blema sólo surge dentro de la implementación de un método, con lo que siempre que acceda a una propiedad dentro de un método, piense si quiere acceder usando `self` o no.

1.6.2. Comprobación estática o dinámica

El operador punto y los métodos getter/setter son equivalentes a la hora de acceder a propiedades, aunque el compilador realiza una comprobación estática más estricta cuando usamos el operador punto para acceder a propiedades que cuando enviamos un mensaje al método setter. En concreto el compilador produce un error cuando escribimos en una propiedad con el modificador `readonly`, mientras que si enviamos un mensaje al método setter (que no exista) de esta misma propiedad lo más que podemos es obtener un warning durante la compilación, aunque durante la ejecución obtendríamos un error de ejecución. Es decir, con el objeto `Fecha` anterior:

```
f1.bisiesto = true; // Error de compilación
[f1 setBisiesto:true]; // Warning, y error en ejecución
```

Por el contrario, si estamos trabajando con un puntero a objeto dinámico no podemos usar el operador punto para acceder a las propiedades, mientras que sí que podemos usar sus correspondientes métodos getter y setter, es decir:

```
id p = [Punto new];
p.x = 5; // Error de compilación
[p setX:5]; // Correcto
```

1.6.3. Operadores de asignación con propiedades

Los operadores de asignación compuestos de C: `+=`, `-=`, `*=`, `/=`, `++`, etc., pueden ejecutarse directamente sobre propiedades, siempre que las propiedades sean tipos que el lenguaje C acepte, es decir, sean tipos fundamentales o punteros C, pero lógicamente no pueden usarse si las propiedades son puntero a objetos Objective-C. Por ejemplo, podemos hacer:

```
Punto* p = [Punto new];
p.x++;
p.y += p.x;
```

Que sería equivalente a:

```
Punto* p = [Punto new];
[p setX:[p x]+1];
[p setY:[p y]+[p x]];
```

Cuando usamos el operador de asignación en cadena como en el siguiente ejemplo:

```
Punto* p1 = [Punto new];
Punto* p2 = [Punto new];
p1.x = p2.x = 3;
```

El compilador realiza una llamada al método setter de cada objeto por cada operación de asignación, con lo que estas llamadas se traducirían por:

```
Punto* p1 = [Punto new];
Punto* p2 = [Punto new];
[p1 setX:3];
[p2 setX:3];
```

Recuérdese que los métodos setter devuelven `void`, y en consecuencia no se pueden anidar.

1.6.4. Valores nulos

En el apartado 3 del Tema 4 comentamos que en Objective-C es posible enviar un mensaje a `nil`, lo cual era útil para evitar errores en tiempo de ejecución. También es posible ejecutar el operador punto cuando el objeto sobre el que lo ejecutamos es `nil`. En este caso el comportamiento es el mismo que si ejecutáramos el método getter o setter correspondiente sobre el objeto `nil`. En concreto:

- Si estamos escribiendo en una propiedad de un objeto que vale `nil`, simplemente no se hace nada.
- Si estamos leyendo una propiedad de un objeto que vale `nil`, si la propiedad es de tipo `int`, `NSInteger`, `NSUInteger` o `float` se devuelve `0`, y si la propiedad es de un tipo más grande (p.e. `double`, `long long`, `struct`, `union`) el retorno está indefinido.

Por ejemplo:

```
Punto* p = nil;
p.x = 3; // No falla ni hace nada
NSInteger valor = p.x; // No falla, valor valdrá 0
```

1.6.5. Abusar del operador punto

No se recomienda utilizar el operador punto para ejecutar métodos de un objeto, aunque la sintaxis del lenguaje lo permita. Es decir, es posible hacer cosas como:

```
pl.retain;
```

También se recomienda respetar el modificador `readonly` de las propiedades no creando métodos setter para estas propiedades. Por ejemplo, en principio sería posible declarar en el Listado 6.5 un método setter para la propiedad `bisiesto` (que recuérdese que tenía el modificador `readonly`):

```
@property(readonly,getter=esBisiesto) BOOL bisiesto;
- (BOOL)esBisiesto;
- (void)setBisiesto:(BOOL)param;
```

En este caso no obtendríamos ningún warning al ejecutar un acceso de escritura a la propiedad:

```
f1.bisiesto = YES; // Funciona
```

Recuérdese que en el apartado 1.4 comentamos que si un método estaba marcado con `readonly`, la directiva del compilador `@synthesize` sólo generaba su método getter. Además, en el apartado 1.6.2 indicamos que si intentamos escribir en la propiedad se producía un error de compilación. Pero este error de compilación desaparece si implementamos el método setter de la propiedad, con lo que realmente estamos engañando al compilador. Luego, si una propiedad tiene método setter, no debemos marcarla con el modificador `readonly`.

1.7. Redefinir modificadores de propiedad

Podemos redefinir los modificadores de una propiedad en una subclase, para lo cual volvemos a usar la directiva del compilador `@property` e indicamos sus nuevos modificadores.

En caso de cambiar los modificadores de una propiedad deberemos volver a implementar (redefinir) sus métodos getter y setter (o bien volver a utilizar la directiva del compilador `@synthesize`). Esto se debe a que la forma en que estos métodos están implementados depende de los modificadores de su propiedad. La excepción a esta regla es cuando sólo cambiamos el modificador `readonly` por `readwrite`. En este caso sólo es necesario implementar el método setter. Implementar una clase con el modificador `readonly` y una derivada con el modificador `readwrite` es un patrón de diseño muy usado donde los objetos de la clase base actúan como objetos inmutables, y los objetos de la clase derivada actúan como objetos mutables.

```
/* Numero.h */
#import <Foundation/Foundation.h>

@interface Numero : NSObject {
    NSInteger numero;
```

```

}
@property(nonatomic) NSInteger numero;
@end

@interface Numero ()
@property(nonatomic) NSInteger numero;
@end

```

Listado 6.7: Declaración de una extensión con propiedades redefinidas

```

/* Numero.h */
#import "Numero.h"

@implementation Numero
@synthesize numero;
@end

```

Listado 6.8: Implementación de una extensión con propiedades redefinidas

También es posible redefinir las propiedades dadas en un protocolo, o bien usar una categoría o extensión para redefinir las propiedades de una clase.

En caso de usar una extensión para redefinir las propiedades de una clase, la implementación de los métodos sólo se podrá proporcionar una vez, y el único modificador que se podrá cambiar es `readonly` por `readwrite`, tal como muestra el Listado 6.7 y Listado 6.8.

1.8. Diferencias en el runtime

Existe una diferencia entre el runtime de Objective-C para 32 bits y el runtime de Objective-C para 64 bits. La diferencia radica en que al usar la directiva del compilador `@synthesize` en el runtime de 32 bits las variables de instancia que almacenan los valores de las propiedades deben estar declaradas en la clase, mientras que en el runtime de 64 bits su declaración es opcional, y en caso de no hacerse, la directiva del compilador `@synthesize` genera esta variable de instancia en la clase. Es decir, el siguiente ejemplo compilaría tanto en el runtime de 32 bits como en el de 64 bits.

```

@interface Numero : NSObject {
    NSInteger numero;
}

@property(nonatomic) NSInteger numero;
@end

@implementation Numero
@synthesize numero; // Usa la variable de instancia numero
@end

```

Sólo en runtime de 32 bits sería obligatorio declarar la variable de instancia numero. En el runtime de 64 bits la declaración de la variable de instancia numero es opcional, y si no se hace la directiva @synthesize lo generaría. Luego el siguiente ejemplo sólo compilaría en el runtime de 64 bits.

```
@interface Numero : NSObject {  
}  
@property(nonatomic) NSInteger numero;  
@end  
  
@implementation Numero  
@synthesize numero; // Genera la variable de instancia numero  
@end
```

En el futuro se pretende que la directiva @synthesize genere automáticamente la variable de instancia donde almacenar las propiedades, pero por compatibilidad con el ABI del runtime de 32 bits se ha tenido que mantener la primera forma en el runtime de 32 bits.

Apple usa el término **variable de instancia frágiles** para referirse a las variables de instancia del runtime de 32 bits, donde todas las variables de instancia deben de aparecer en la @interface de la clase, y el término **variables de instancia no frágiles** para referirse a las variables de instancia del runtime de 64 bits, donde las variables de instancia también se pueden crear en la @implementation.

La existencia de variables de instancia no frágiles en el runtime de 64 bits abre la puerta a que en un futuro se puedan declarar variables de instancia privadas en la @implementation.

2. El recolector de basura

Objective-C 2.0 introduce la posibilidad de gestionar la memoria usando un recolector de basura, el cual automáticamente identifica la memoria reservada en el heap que ya no es alcanzable y la libera sin la intervención del programador.

2.1. Activar el recolector de basura

Por defecto GCC y Xcode tienen desactivado el recolector de basura, es decir, la gestión de memoria se realiza por cuenta de referencias, tal como explicamos en el apartado 3 del Tema 5.

Existen tres opciones de compilación que indican el modelo de gestión de memoria que queremos usar:

- Sin flags. El código objeto generado puede gestionar la memoria únicamente por cuenta de referencias, es decir, el código generado no puede ejecutarse cuando la gestión de memoria con recolector de basura está activada.
- `-fobjc-gc-only` El código objeto generado con este flag sólo puede ejecutarse con el recolector de basura activado.
- `-fobjc-gc` El código objeto generado con este flag puede ejecutarse en ambos modelos de gestión de memoria.

Cuando generamos un binario a partir de varios ficheros de código objeto, todos los ficheros deberán usar el mismo flag de compilación.

Cuando compilamos usando la opción `-fobjc-gc-only` las llamadas a operaciones de gestión de memoria por cuenta de referencias (`retain`, `release` y `autorelease`) son ignoradas por el compilador. Cuando compilamos usando la opción `-fobjc-gc` las llamadas a los métodos de cuenta de referencias son ignoradas por el runtime sólo si el recolector de basura está activado.

En caso de estar desarrollando una librería debemos usar la opción `-fobjc-gc`, ya que de esta forma (aunque la librería estará usando gestión de memoria con recolector de basura) la librería podrá ser usada tanto por aplicaciones que usan la gestión de memoria por cuenta de referencias como por aplicaciones que usan la gestión de memoria con recolector de basura.

Si generamos una librería usando la gestión de memoria por cuenta de referencias (sin flags), ésta librería no podrá ser usada por una aplicación que generemos con la opción `-fobjc-gc-only`. Del mismo modo, si generamos una librería con la opción `-fobjc-gc-only`, ésta no podrá ser usada por una aplicación que haga uso de la gestión de memoria por cuenta de referencias.

Antes de empezar a desarrollar una aplicación, Apple recomienda elegir entre el modo de cuenta de referencias (sin flag) y el modo `-fobjc-gc-only`, y dejar la opción `-fobjc-gc` únicamente para el desarrollo de librerías.

Idealmente el modo de gestión de memoria se debe elegir al empezar a desarrollar una aplicación, con lo que Apple no recomienda activar el recolector de basura recompilando una aplicación que empezó usando gestión de memoria por cuenta de referencias.

2.2. Técnicas de recolección de basura

En este apartado pretendemos recopilar las bases teóricas en las que se basan los recolectores de basura de distintos lenguajes de programación. En los siguientes apartados nos centraremos en el recolector de basura de Objective-C 2.0.

Los recolectores de basura realizan dos tareas principales:

1. Determinan qué objetos son **alcanzables** (también llamados **live objects**) y qué objetos son **inalcanzables** (también llamados **garbage**) por el programa.
2. Eliminan la memoria de los objetos que ya no son alcanzables.

El término *objeto* se usa en la literatura científica que estudia las técnicas de recogida de basura para referirse a todo tipo de variables, no sólo a los objetos de la programación orientada a objetos. En el caso de Objective-C coincide que sólo los objetos Objective-C son susceptibles a que su ciclo de vida sea gestionados por el recolector de basura.

La mayoría de los recolectores de basura son sistemas **cerrados**, es decir, el lenguaje, compilador y runtime colaboran para identificar todas las referencias a objetos alcanzables e inalcanzables. Normalmente los sistemas cerrados (p.e. Java) introducen restricciones en el lenguaje para el uso directo de punteros a memoria con el fin de poder gestionar más fácilmente la vida de los objetos. Una ventaja que suelen tener los sistemas cerrados es que se pueden mover bloques de memoria asignados con el fin de compactar la memoria. El recolector de basura de Objective-C es un recolector de basura **abierto**, es decir, no toda la memoria es gestionada por el recolector de basura.

Por otro lado, existen dos técnicas ampliamente utilizadas para la recolección de basura:

- **Cuenta de referencias.** Consiste en que cada objeto tiene asociado un contador. Cuando la referencia al objeto es asignada a otra referencia el

contador se incrementa. Cuando la referencia al objeto se elimina el contador se decrementa. Esta es la técnica que se suele usar en los recolectores de basura cerrados. En Objective-C se utiliza la cuenta de referencias para gestionar el ciclo de vida de un objeto de forma manual (tal como vimos en el apartado 3 del Tema 5), pero es responsabilidad del programador el incrementar y decrementar esta cuenta.

- **Barrido de memoria.** Esta es la técnica que se suele usar en los recolectores de basura abiertos, y es la técnica utilizada por el recolector de basura de Objective-C. Consiste en recorrer la memoria identificando qué trozos de memoria reservados ya no son alcanzables por el programa.

2.2.1. Referencias fuertes y débiles

En el apartado 3.9 del Tema 5, cuando veíamos la gestión de memoria por cuenta de referencias introdujimos el concepto de referencias débiles, que eran referencias sobre las que no ejecutábamos `retain`.

En el caso de estar utilizando el recolector de basura para la gestión de memoria, la semántica de las referencias fuertes y débiles cambia ligeramente: Cuando activamos el recolector de basura, éste pasa a controlar la liberación de memoria de todos los objetos Objective-C, pero no la memoria dinámica de variables que no sean objetos Objective-C. Es decir, cuando reservemos memoria dinámica con `malloc()` seguimos necesitando liberarla con `free()`. Se usa el término **referencia controlada** para referirse a una referencia cuya memoria es controlada por el recolector de basura.

La memoria controlada por el recolector de basura se divide en referencias fuertes y referencias débiles. Se llama **referencia fuerte** a un puntero a objeto que impide que el objeto sea liberado por el recolector de basura. Por el contrario, se llama **referencia débil** a un puntero a objeto que no impide que el objeto sea liberado por el recolector de basura.

Por defecto se consideran referencias fuertes a todos los punteros a objeto Objective-C almacenados como variables globales, como variables de pila (incluidas las variables locales), como variables de instancia de un objeto o como campos de una estructura. Todas las variables que no sean puntero a objetos Objective-C por defecto son consideradas referencias débiles.

Podemos indicar si una referencia controlada es fuerte o no usando los modificadores `__strong` y `__weak` respectivamente. Estos modificadores se pueden poner en cualquier lugar de la declaración del tipo de dato, es decir, las siguientes declaraciones son equivalentes:

```
__weak Punto* p;
Punto __weak* p;
Punto* __weak p;
```

Los objetos apuntados con referencias `_weak` son objetos cuya memoria puede ser liberada por el recolector de basura en cualquier momento, pero el recolector de basura nos garantiza que asignará el valor `nil` a la referencia controlada antes de eliminar su memoria. De esta forma el programa puede saber que esa memoria ha sido eliminada por el recolector de basura, y no intentará indireccionala.

Para acceder a las referencias débiles debemos usar siempre métodos (nunca acceder directamente a las variables de instancia) ya que la recogida de basura puede ejecutarse en cualquier momento y producirse un error en tiempo de ejecución. Cosa que (como vimos en el apartado 3 del Tema 4) no ocurre cuando enviamos un mensaje a `nil`. Por ejemplo, si tenemos la siguiente variable global:

```
_weak Punto* p;
```

El siguiente programa podría fallar si el objeto `p` es liberado por el recolector de basura inmediatamente después de evaluarase la sentencia condicional:

```
int valor;
if (p!=nil)
    valor = p->x;
```

La forma correcta de acceder a la referencia débil hubiera sido:

```
valor = [p x];
if (p==nil) {
    // El objeto ha sido liberado (valor valdrá 0)
}
```

En general sólo debemos usar referencias débiles para las variables globales, para las variables de instancia o para los campos de las estructuras, pero no es recomendable usar punteros débiles para las variables locales. De hecho, el compilador emite un warning cuando intentamos declarar un puntero a objeto débil como variable local.

2.2.2. El root set

En los sistemas donde se implementa la recogida de memoria por barrido se utiliza el concepto de **root set** para referirse a los objetos que son alcanzables por el programa sin seguir ningún puntero. En el caso de Objective-C el root set está formado por los punteros a objetos globales y los punteros a objetos en la pila.

Para realizar el barrido de memoria el recolector de basura:

1. Marca como alcanzables a todos los objetos en el root set.

2. Recorre de forma recursiva todas las referencias controladas que aparezcan en las variables de instancia de los objetos del root set para identificar nuevos objetos alcanzables.
3. Todos los demás objetos son considerados objetos inalcanzables y se libera su memoria.

Obsérvese que los punteros a objeto que aparezcan en las variables de instancia de una clase o estructura no forman parte del root set, pero son referencias controladas, con lo que se recorren en el paso 2.

Tenga en cuenta que forman parte del root set tanto los objetos Objective-C marcados como `__strong` como los marcados con `__weak`.

2.2.3. Recogida de basura incremental

El principal problema que tiene la recogida de basura por barrido es que mientras que se está realizando el barrido la ejecución de la aplicación queda interrumpida. Para evitar estas pausas, muchos sistemas de recogida de basura (incluido el de Objective-C) implementan la recogida de basura en un hilo distinto al de la aplicación, que es lo que se denomina **recogida de basura incremental**.

La recogida de basura incremental tiene el efecto lateral de que los hilos de aplicación, a los que se suele denominar **mutator**, pueden modificar el estado de los objetos de la memoria antes de que el hilo del recolector de basura haya terminado el barrido, lo cual introduce dos problemas: 1) El recolector de basura podría no encontrar todos los objetos que realmente podrían ser liberados y 2) que el recolector de basura podría liberar erróneamente un objeto que no debería de ser liberado. El primer problema no es importante, ya que en la próxima ejecución del recolector de basura se encontrará el objeto no liberado y se liberará. Los recolectores de basura suelen ser conservadores y tienen a ignorar el primer problema. El segundo problema es más grave, porque el recolector liberaría memoria que cuando intentase usar el mutator produciría un error en tiempo de ejecución.

Para evitar el segundo problema se usa un mecanismo de sincronismo llamado **barreras de escritura** (write barrier)⁴⁰ que consiste en que todas las escrituras en referencias controladas son monitorizadas por el recolector de basura. En concreto, cuando el compilador encuentra una asignación a una referencia controlada, ésta es sustituida por una llamada a una función del

⁴⁰ En general, el término **barrera de memoria** (memory barrier) se usa para forzar al procesador (tenga uno o más cores) a terminar todas las operaciones de lectura y escritura en memoria situadas antes de la barrera antes de ejecutar las operaciones situadas después de la barrera. Las barreras de escritura serían aquellas que sólo fuerzan a terminar las escrituras. En el caso del recolector de basura, sus barreras de escritura no controlan la escritura en toda la memoria del sistema, sólo controlan los accesos a las referencias controladas.

runtime que controla y sincroniza los accesos a las referencias. Dependiendo del tipo de referencia en la que estamos asignando un valor, estas funciones son:

- `objc_assign_global()` para referencias globales.
- `objc_assign_ivar()` para referencias en variables de instancia.
- `objc_assign_strongCast()` para otras referencias fuertes.
- `objc_assign_weak()` para otras referencias débiles.

2.3. La librería auto

El recolector de basura de Objective-C está implementado en una librería reutilizable **llamada auto**. Como muestra la Figura 6.1, el runtime de Objective-C es un cliente de la librería auto, y da servicio tanto al Foundation Framework y AppKit de Objective-C, como a la librería Core Foundation.

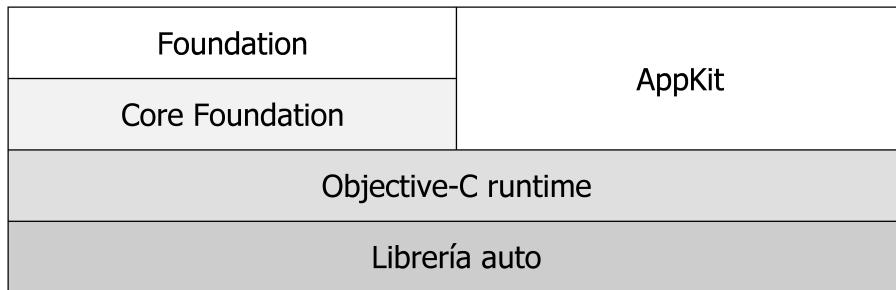


Figura 6.1: Arquitectura en capas del runtime de Objective-C

El recolector de basura usa como root set los punteros a objetos en la pila y la memoria global. Como muestra la Figura 6.2, el heap está dividido en dos zonas: La **auto zone** donde se almacenan los objetos Objective-C y la **zona de memoria estándar**, que es donde `malloc()` realiza la reserva de memoria. El recolector de basura sólo escanea la memoria en la auto zone, la zona de memoria estándar nunca es escaneada por el recolector de basura.

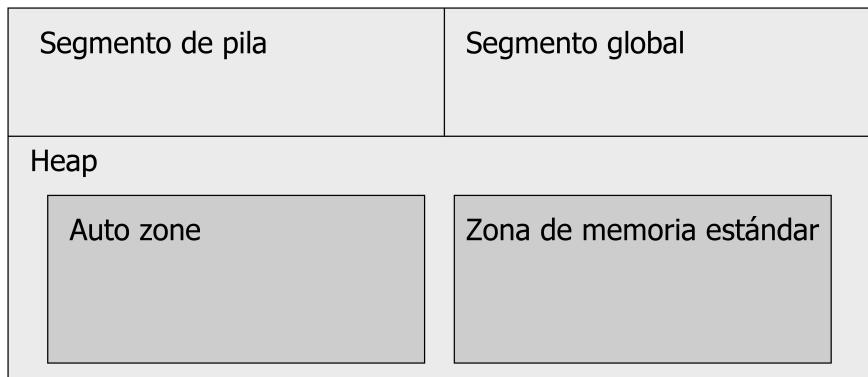


Figura 6.2: Zonas de memoria en una aplicación Objective-C

2.4. Recolector de memoria con tipos fundamentales

Es posible crear un buffer de memoria donde almacenar tipos de datos fundamentales (p.e. enteros) y dejar que sea el recolector de basura quien se encargue de gestionar esta memoria. En este caso no debemos de usar la función `malloc()` para asignar esta memoria, sino que debemos usar la función:

```
void * __strong
NSAllocateCollectable(NSUInteger size, NSUInteger options);
```

Obsérvese que la función devuelve una referencia fuerte, con lo que un error común es hacer:

```
int* buffer;
buffer = NSAllocateCollectable(100*sizeof(int), 0);
```

En Objective-C los punteros a tipos fundamentales son referencias no controladas, y estaríamos asignando el retorno de la función a una referencia no controlada. Dado que no existen referencias controladas apuntando al buffer, el recolector de basura puede liberar el buffer en cualquier momento. Para evitarlo deberíamos haber hecho:

```
__strong int* buffer;
buffer = NSAllocateCollectable(100*sizeof(int), 0);
```

También podríamos haber declarado la variable `buffer` como `__weak`, y en este caso el recolector de basura pondría esta variable a `NULL` en caso de liberar la memoria a la que apunta.

El parámetro `options` normalmente se pone a 0. En caso de que el buffer vaya a almacenar referencias controladas debemos poner en este parámetro el valor `NSScannedOption` para indicarlo:

```
__strong char** palabras;
palabras = NSAllocateCollectable(100*sizeof(int),
                                 NSScannedOption);
```

En el primer caso la memoria se reserva en la zona de memoria estándar y en el segundo caso en la auto zone. Los punteros pueden apuntar a objetos o a buffers creados con `NSAllocateCollectable()`.

2.5. Gestión de memoria de objetos puente

Existe un conjunto de objetos, llamados **objetos puente** (bridged objects) los cuales se pueden usar tanto desde Cocoa como desde librerías C como Carbon, Core Services o Core Foundation. Normalmente estos objetos se

pueden pasar de una librería a otra con sólo hacer casting entre ellos. Por ejemplo `CFString` de Core Foundation y `NSString` de Cocoa, son tipos compatibles entre sí y los punteros a objetos de estos tipos se pueden usar desde ambas librerías con sólo hacer un casting a sus punteros. En este apartado vamos a ver cómo gestionar los objetos puente de un entorno desde el otro entorno.

La política de gestión de memoria de objetos Core Foundation dice que los objetos devueltos por funciones cuyo nombre comienza por `Create` o `Copy` deben ser liberados por el receptor. Los objetos devueltos por otras funciones no deben ser liberados por el receptor.

La política de gestión de objetos de Cocoa dice que los objetos devueltos por métodos cuyo nombre comienza por `alloc` o `copy` deben ser liberados por el receptor. Los objetos devueltos por otros métodos no deben ser liberados por el receptor.

Como la convención usada por ambos entornos es muy parecida, y debido a que el sistema de gestión de memoria de ambos entornos es compatible, los objetos puente de ambos entornos pueden ser manejados de forma intercambiable. Por ejemplo:

```
NSString* str = [[NSString alloc] initWithCString:@"Hola"];
[str release];
```

Es equivalente a:

```
NSString* str = [[NSString alloc] initWithCString:@"Hola"];
CFRelease(str);
```

Y equivalente a:

```
NSString* str = CFStringCreateWithCString(...);
[str autorelease];
```

Aunque tengamos activado el recolector de basura, por defecto el recolector de basura no controla la vida de las variables Core Foundation, es decir, son referencias no controladas. Para que el recolector de basura gestione la memoria de las variables Core Foundation deberemos pedírselo explícitamente con el modificador `__strong` (o con el modificador `__weak`). Por ejemplo:

```
@interface Festivo
    __strong CFDateRef dia;
@end
```

Aunque tengamos activado el recolector de basura, en caso de no marcar una variable Core Foundation con estos modificadores, deberemos de realizar la gestión de memoria del objeto Core Foundation mediante cuenta de referencias con las funciones `CFRetain()` y `CFRelease()`.

Si queremos que una referencia Core Foundation sea controlada por el recolector de basura debemos pedírselo pasando la referencia a la función:

```
CFTyperef CFMakeCollectable(CFTyperef cf);
```

Además en este caso debemos de recoger la referencia devuelta por la función en una referencia controlada, que deberá estar marcada explícitamente con `__strong` o `__weak`.

Es muy común convertir referencias Core Foundation en referencias controladas cuando retornamos una referencia Core Foundation. Por ejemplo:

```
- (id) diaInteresante {
    CFDateRef dia = CFDateCreate(NULL, 0);
    .....
    return NSMakeRangeable(dia);
}
```

La función `NSMakeRangeable()` es igual a `CFMakeCollectable()`, solo que en vez de devolver un `CFTyperef` devuelve un `id`, lo cual es útil para eludir el casting.

2.6. Clases para recolección de basura

La clase `NSGarbageCollector` permite interactuar con el recolector de basura de Objective-C. Podemos obtener una referencia al recolector de basura usando su método de clase `defaultCollector` de la forma:

```
NSGarbageCollector* gc = [NSGarbageCollector
                           defaultCollector];
```

Una vez que tengamos el recolector de basura, podemos activarlo o desactivarlo con los métodos:

- `(void) disable`
- `(void) enable`

También podemos preguntar si el recolector de basura está activado con:

- `(BOOL) isEnabled`

Como vimos en el apartado 2.2.3, el recolector de basura de Objective-C es incremental, y podemos preguntar cuándo se está ejecutando la recolección de basura con el método:

- `(BOOL) isCollecting`

La recogida de basura la puede iniciar tanto el runtime de Objective-C cuando detecta que se ha alcanzado un umbral, como el programador llamando a los métodos:

- `(void)collectIfNeeded`
- `(void)collectExhaustively`

El primer método indica al runtime que estamos en un buen momento para iniciar la recolección de basura si así lo considera. El segundo pide ejecutar una recogida exhaustiva de todos los objetos inalcanzables que existan en el proceso.

Podemos desactivar la recogida de basura para un determinado objeto usando el método:

- `(void)disableCollectorForPointer:(void*)ptr`

En este caso `ptr` pasa a formar parte del root set y no podrá ser liberado (aunque llegase a ser inalcanzable). Esto es especialmente útil en el caso de las referencias débiles. Podemos volver a activar la recogida de basura de la memoria de ese objeto con:

- `(void)enableCollectorForPointer:(void*)ptr`

En el apartado 2.2.1 indicamos que los punteros débiles son punteros que pueden ser eliminados por el recolector de basura en cualquier momento. El Listado 6.9 muestra un programa que dispone de un puntero débil y de otro fuerte. En el ejemplo llamados a `collectExhaustively` y estudiamos qué ocurre con los punteros.

Tras ejecutar el programa obtenemos:

```
punteroDebil = 16848992 ('debil')
punteroFuerte = 16849008 ('fuerte')
punteroDebil = 0 ('(null)')
punteroFuerte = 16849008 ('fuerte')
```

Es decir, el puntero débil ha sido recolectado y puesto a cero, mientras que el puntero fuerte no.

```
/* gc-test.m */
#import <Foundation/Foundation.h>

__weak NSString * punteroDebil;
__strong NSString * punteroFuerte;

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool =
        [[NSAutoreleasePool alloc] init];
    punteroDebil = [[NSString alloc]
```

```

        initWithCString: "debil"];
punteroFuerte = [[NSString alloc]
                  initWithCString: "fuerte"];
NSLog(@"%@", punteroDebil = %d ('%@'), (int)punteroDebil,
        punteroDebil);
NSLog(@"%@", punteroFuerte = %d ('%@'), (int)punteroFuerte,
        punteroFuerte);
[[NSGarbageCollector defaultCollector]
 collectExhaustively];
NSLog(@"%@", punteroDebil = %d ('%@'), (int)punteroDebil,
        punteroFuerte);
NSLog(@"%@", punteroFuerte = %d ('%@'), (int)punteroFuerte,
        punteroFuerte);
[pool drain];
return EXIT_SUCCESS;
}

```

Listado 6.9: Ejemplo de recogida de basura

2.7. Finalizar un objeto

Cuando usamos gestión de memoria por cuenta de referencias el método `dealloc` sirve para liberar los objetos agregados, y de esta forma liberar su memoria. Sin embargo cuando usamos gestión de memoria con recolector de basura el método `dealloc` no se llama nunca ya que el programador no debe preocuparse de liberar esta memoria, sino que es el recolector de basura quien determina cuando los objetos agregados son inalcanzables y los libera.

El método `dealloc` también se utiliza para liberar recursos asociados al objeto como puedan ser ficheros o sockets. Cuando estamos usando gestión de memoria por cuenta de referencias el método `dealloc` no se ejecuta nunca, pero en su lugar la clase `NSObject` proporciona el método:

- `(void)finalize`

Este método es ejecutado por el recolector de basura inmediatamente antes de liberar la memoria de un objeto inalcanzable. Es decir, es un método callback y nosotros no debemos ejecutarlo nunca directamente, excepto al llamarlo con `super` en su propia implementación.

Debido a que este método es llamado por el hilo de recogida de basura, y no por el mutador, Apple recomienda limitar su uso lo más posible. Por ejemplo, Apple recomienda proporcionar métodos que cierren explícitamente los recursos del objeto y que deban ser llamados por el programador en vez de implementar el cierre de los recursos en `finalize`. En ocasiones esto no es posible, por ejemplo si creamos un objeto global `Log` destinado a almacenar mensajes de log desde distintas partes del programa. En este caso el recurso fichero asociado habrá que cerrarlo en `finalize`.

A diferencia de lo que ocurría con `dealloc`, el runtime de Objective-C no garantiza el orden en que se ejecuta `finalize` sobre los objetos con lo que no deberíamos nunca acceder a otros objetos desde `finalize`, ya que no podemos saber si los objetos agregados han sido ya liberados o no.

En caso de que un objeto Objective-C agregue objetos Core Foundation, los objetos Core Foundation no son liberados por defecto por el recolector de basura. Apple recomienda no llamar a `CFRelease()` sobre un objeto Core Foundation, sino marcarlo para que sea liberado por el recolector de basura usando la función `NSMakeCollectable()` tal como se explico en el apartado 2.5.

Por último, Apple indica que en caso de implementar `finalize` tenemos que tener cuidado de no **resucitar un objeto**. Resucitaríamos un objeto cuando dentro de `finalize` hacemos que el objeto vuelva a ser alcanzable. Por ejemplo, una forma trivial de producir este error sería:

```
- (void) finalize {
    [arrayGlobal addObject:self];
    [super finalize];
}
```

2.8. Ventajas e inconvenientes

Tanto la gestión de memoria por cuenta de referencias como la gestión de memoria con recolector de basura tienen ventajas e inconvenientes. En este apartado vamos a analizar estos aspectos.

Las principales ventajas de la gestión de memoria con recolector de basura son:

- Mejora la productividad ya que el programador puede olvidarse de cumplir con el protocolo de cuenta de referencias
- Evita que posibles errores en la programación den lugar a fugas de memoria.
- Evita las retenciones cíclicas (que explicamos en el apartado 3.8 del Tema 5) ya que el recolector de basura puede detectar que dos objetos que se referencian entre ellos de forma cíclica no son alcanzables desde el root set, y los elimina.

Por su parte, la gestión de memoria por cuenta de referencias presenta como principales ventajas:

- En general consume menos ciclos de CPU ya que no existe un hilo auxiliar de recogida de basura encargado de determinar periódicamente los objetos inalcanzables a liberar.
- En general consume menos memoria ya que el mutator no deja un rastro de objetos sin liberar hasta que se ejecuta la recogida de basura.
- Cuando usamos un recolector de basura la latencia de la aplicación suele ser mayor debido al hilo de recogida de basura. El hecho de que el recolector de basura de Objective-C sea incremental evita que se produzcan grandes latencias.
- Permite que los recursos (p.e. ficheros) se asocien a la vida de un objeto de forma que el recurso se libera cuando la memoria del objeto se libera. Por su parte, la gestión de memoria con recolector de basura obliga a que el objeto proporcione un método explícito para cerrar los recursos.

2.9. Trazar la recolección de basura

Podemos trazar el comportamiento del recolector de basura definiendo la variable de entorno `OBJC_PRINT_GC` con el valor `YES`.

```
/* gc-test.m */
#import "Punto.h"

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool =
        [[NSAutoreleasePool alloc] init];
    Punto* p = [[Punto alloc] initWithX:1 y:1];
    p = [[Punto alloc] initWithX:2 y:2];
    [[NSGarbageCollector defaultCollector]
        collectExhaustively];
    [pool drain];
    return EXIT_SUCCESS;
}
```

Listado 6.10: Programa que usa el recolector de basura

Por ejemplo, si compilados el programa del Listado 6.10 con recolección de basura y lo ejecutamos con la variable de entorno `OBJC_PRINT_GC` puesta a `YES` obtenemos:

```
$ gcc Punto.m gc-test.m -fobjc-gc -framework Foundation -o gc-test
$ OBJC_PRINT_GC=YES ./gc-test
objc[676]: GC: executable 'gc-test' supports GC
objc[676]: GC: library '/usr/lib/libobjc.A.dylib' supports GC
objc[676]: GC: library
'/System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation' supports GC
objc[676]: GC: library
'/System/Library/Frameworks/Foundation.framework/Versions/C/Foundation' supports GC
```

```
objc[676]: GC: is ON
gc-test[676:10b] finalize [1,1]
```

Vemos que `collectExhaustively` ha eliminado al primer objeto `Punto`, pero el segundo objeto no es eliminado por `drain`. Esto se debe a que el recolector de basura no se ejecuta al finalizar el programa, lo cual implica una mejora en el tiempo de terminación de los procesos, pero cualquier operación puesta en `finalize` del objeto no se ejecutaría. Esta es una de las razones por las que se recomienda usar métodos explícitos del cierre de recursos cuando se usa la gestión de memoria con recolector de basura.

Tema 7

Las objetos de colección

Sinopsis:

En este tema estudiaremos las clases de librería que proporciona Objective-C para crear objetos de colección, es decir, objetos usados para agrupar otros objetos.

En particular, estudiaremos los objetos arrays, los objetos conjunto y los objetos diccionario. También aprenderemos a recorrer, ordenar y filtrar los elementos de un objeto de colección.

1. Objetos arrays

Los **objetos de colección** Objective-C están pensados para agregar punteros a objetos Objective-C, y en caso de estar usando gestión de memoria por cuenta de referencias realizan la gestión de memoria de estos objetos agregados mediante las operaciones `retain` y `release`. Los objetos de colección de Objective-C no tienen porqué ser homogéneos, es decir, de la misma clase, sino que podemos crear un objeto de colección que agregue objetos de distinta clase con la única restricción de que deriven de `NSObject`. Si desea almacenar colecciones de otro tipo de elementos, en vez de usar objetos de colección, utilice estructuras de datos C, como por ejemplo arrays C.

Un **objeto array** es un objeto de colección que almacena objetos indexados. En Objective-C existen tres clases para gestión de arrays:

- `NSArray` representa objetos array inmutables, es decir, objetos array que una vez creados no podemos cambiar el número de elementos que contienen, pero sí que podemos modificar el contenido de sus objetos.
- `NSMutableArray` es una derivada de `NSArray` que se usa cuando el número de elementos agregados en el objeto array puede variar una vez creado el objeto array. Si el número de elementos del objeto array no va a variar es más eficiente usar `NSArray`.
- `NSPointerArray` es otra variante de objeto array mutable que es más flexible que las dos anteriores.

En los siguientes apartados vamos a estudiar cómo funciona cada una de estas clases.

1.1. Objetos arrays inmutables

`NSArray` representa objetos arrays estáticos, que son objetos array donde el número de elementos no cambia una vez inicializado el objeto array, es decir, no podemos añadir o eliminar objetos, tan sólo acceder a los objetos y leer o modificar su contenido.

1.1.1. Creación y liberación

Para crear un objeto `NSArray` usamos métodos de inicialización, es decir métodos de instancia cuyo nombre empieza por `init...` (véase apartado 6.1.1 del Tema 4) como por ejemplo:

- `(id)initWithObjects:(id)firstObj, ...`

O bien usamos métodos factory (véase apartado 6.1.2 del Tema 4), es decir, métodos de clase cuyo nombre empieza por `array...` como por ejemplo:

```
+ (id)arrayWithObjects:(id)firstObj, ...
```

Ambos reciben una lista de objetos Objective-C acabada en `nil`. Suponiendo que estamos usando la gestión de memoria por cuenta de referencias, los objetos de colección incrementan la cuenta de referencias de los objetos almacenados al introducirlos en la colección, y la decrementan (llamando a `release`)⁴¹ cuando el objeto de colección se destruye.

Por otro lado, como indicamos en el apartado 3.6 del Tema 5, la diferencia entre usar un método factory (los métodos `array...`) y un método de inicialización (los métodos `init...`) está en que los primeros devuelven un objeto array sobre el que se ha ejecutado `autorelease`, con lo que no debemos de ejecutar `release` sobre el objeto array al acabar de utilizarlo, mientras que los segundos devuelven un objeto array sobre el que tendremos que ejecutar `release` al acabar de utilizarlo. Es decir, suponiendo que tenemos cuatro objetos creados:

```
Punto* p1 = [[[Punto alloc] initWithX:-2 y:2] autorelease];
Punto* p2 = [[[Punto alloc] initWithX:2 y:2] autorelease];
Punto* p3 = [[[Punto alloc] initWithX:2 y:-2] autorelease];
Punto* p4 = [[[Punto alloc] initWithX:-2 y:-2] autorelease];
```

En el primer caso crearíamos el objeto array y éste sería liberado automáticamente por `NSAutoreleasePool`:

```
NSArray* cuadrilatero = [NSArray
                           arrayWithObjects:p1,p2,p3,p4,nil];
// Operaciones con el array
.....
// Cuando no lo necesitemos más no debemos ejecutar release
```

Con lo que si no queremos que se libere el objeto array deberemos ejecutar `retain` sobre el objeto array.

Sin embargo, en el segundo caso debemos de ejecutar `release` o `autorelease` sobre el objeto array cuando ya no lo necesitemos más:

```
NSArray* cuadrilatero2 = [[NSArray alloc]
                           initWithObjects:p1,p2,p3,p4,nil];
// Operaciones con el array
.....
// Liberamos la cuenta de referencias
[cuadrilatero2 release];
```

⁴¹ Creemos que hubiera sido mejor idea que los objetos de colección llamasen a `autorelease` sobre los objetos agregados, ya que así se permitiría acceder a los objetos agregados hasta el final del método o función donde se están usando.

También podemos crear un objeto `NSArray` a partir del contenido de un array C con los métodos:

```
+ (id)arrayWithObjects: (const id*)objects
                  count: (NSUInteger)count
- (id)initWithObjects: (const id*)objects
                  count: (NSUInteger)count
```

O bien crear un objeto `NSArray` a partir de otro objeto `NSArray` (o objeto derivado `NSMutableArray`) con los métodos:

```
+ (id)arrayWithArray: (NSArray*)anArray
- (id)initWithArray: (NSArray*)anArray
```

1.1.2. Acceso a los elementos del objeto array

Podemos preguntar por el número de elementos en el objeto array con el método:

- `(NSUInteger)count;`

O bien acceder a un determinado elemento proporcionando su índice al método:

- `(id)objectAtIndex: (NSUInteger)index`

Los objetos array se indexan empezando en el índice 0. También podemos buscar un objeto en el objeto array con los métodos:

- `(NSUInteger)indexOfObject: (id)anObject`
- `(NSUInteger)indexOfObjectIdenticalTo: (id)anObject`

Ambos devuelven el valor `NSNotFound` en caso de no encontrarse el objeto en el objeto array. La diferencia está en que el primero ejecuta `isEqual:` sobre cada elemento del objeto array para determinar si son idénticos, y el segundo ejecuta el operador `==` sobre el puntero para determinar si son punteros idénticos. Es decir, en la clase `NSObject` tenemos el método `isEqual:` (véase apartado 4.2 del Tema 5), que por defecto compara el puntero `self` con el puntero `anObject`, pero que nosotros podemos redefinir para que compare las variables de instancia que representan a nuestro objeto. Por ejemplo, suponiendo que hemos redefinido `isEqual:` de esa forma:

```
Punto* p1 = [[[Punto alloc] initWithX:2 y:3] autorelease];
Punto* p2 = [[[Punto alloc] initWithX:2 y:3] autorelease];
NSArray* array = [NSArray arrayWithObjects:p1,nil];
NSUInteger pos = [array indexOfObject: p2]; // pos==0
pos = [array indexOfObjectIdenticalTo: p2]; // pos==NSNotFound
```

1.2. Objetos array mutables

Los objetos arrays mutables están representados por la clase `NSMutableArray`, y derivan de `NSArray` con lo que heredan toda su funcionalidad, e introduce funcionalidad para añadir o eliminar elementos del objeto array. En concreto introducen métodos como:

- `(void) addObject:(id) anObject`
- `(void) insertObject:(id) anObject atIndex:(NSUInteger) index`
- `(void) replaceObjectAtIndex:(NSUInteger) index withObject:(id) anObject`
- `(void) removeAllObjects`
- `(void) removeLastObject`
- `(void) removeObjectAtIndex:(NSUInteger) index`
- `(void) removeObject:(id) anObject`
- `(void) removeObjectIdenticalTo:(id) anObject`

Los métodos `removeObject:` y `removeObjectIdenticalTo:` usan respectivamente para buscar el objeto a eliminar el método `isEqual:` y el operador `==`.

El objeto array ejecuta `release` sobre los objetos agregados cuando los elimina. En consecuencia, si queremos conservar el objeto, deberemos ejecutar `retain` sobre los objetos a eliminar del objeto array antes de eliminarlos. En caso contrario podríamos obtener un error en tiempo de ejecución por intentar acceder a un objeto eliminado, es decir:

```
id obj = [array objectAtIndex:0];
[array removeObjectAtIndex:0];
[obj unMensaje]; // Posible error en tiempo de ejecución
```

Y para evitar el error en tiempo de ejecución debemos hacer:

```
id obj = [[array objectAtIndex:0] retain];
[array removeObjectAtIndex:0];
[obj unMensaje]; // Esto es seguro
```

Tanto `NSArray` como `NSMutableArray` no pueden almacenar valores `nil`. Si queremos almacenar este valor podemos almacenar una instancia del objeto singleton de la clase `NSNull` de la forma:

```
[array addObject: [NSNull null]];
```

1.3. Rangos y concatenaciones

Podemos obtener un objeto array con un rango de elementos de otro array usando el método:

- `(NSArray*) subarrayWithRange: (NSRange) range`

Donde `NSRange` es una estructura C que representa un rango de enteros que se empieza a indexar en cero.

```
typedef struct _NSRange {
    unsigned int location;
    unsigned int length;
} NSRange;
```

Por ejemplo para obtener los elementos en el rango [2..4] de un objeto array haríamos:

```
NSRange r = {2,2};
NSArray* subarray = [array subarrayWithRange:r];
```

También podemos obtener un array resultado de concatenar a los elementos de un arrays los elementos de otro array con el método:

- `(NSArray*) arrayByAddingObjectsFromArray:
 (NSArray*) otherArray`

O bien añadir los elementos del segundo array al final del primer array usando el método de la clase `NSMutableArray`:

- `(void) addObjectsFromArray: (NSArray*) otherArray`

1.4. Copia de objetos array

Tanto la clase `NSArray` como `NSMutableArray` implementan los protocolos `NSCopying` y `NSMutableCopying`. En el apartado 5.2 del Tema 5 vimos que cuando un objeto implementa `NSCopying` y `NSMutableCopying` está indicando que se puede copiar de dos formas: Una mutable y otra inmutable. Recuérdese que vimos que cuando queríamos crear una copia de un objeto que no íbamos a modificar (inmutable) ejecutaremos sobre el objeto el método `copy`. Si por el contrario queremos crear una copia que pretendemos modificar, ejecutaremos sobre el objeto el método `mutableCopy`.

En el caso de los objetos array esto significa que cuando copiamos un objeto array con `copy` obtenemos un objeto array inmutable (de la clase `NSArray`),

mientras que cuando copiamos un objeto array con `mutableCopy` obtenemos un array mutable (de la clase `NSMutableArray`).

Es importante tener en cuenta que en Objective-C cuando copiamos un array nunca se ejecuta `copy` o `mutableCopy` sobre los objetos agregados, sino que o bien se copia directamente los punteros a objetos agregados al nuevo array, o bien se ejecuta `retain` sobre los objetos agregados cuyo puntero se copia al nuevo array. A continuación vamos a ver cómo funciona la copia de objetos en los distintos casos.

Cuando copiamos un objeto `NSArray` con `copy`, el objeto array puede limitarse a incrementar su propia cuenta de referencias (ejecutar `retain` sobre el objeto array) ya que ni él ni su copia serán modificables. Sin embargo si copiamos un objeto `NSArray` con `mutableCopy` deberá crearse un objeto de tipo `NSMutableArray` y copiar todos sus elementos agregados al nuevo objeto array, con lo que se ejecutará `retain` sobre cada objeto agregado al nuevo objeto array.

Por el contrario, cuando copiamos un objeto `NSMutableArray` (con `copy` o `mutableCopy`), debido a que tanto el objeto array original como el objeto array copia son modificables, siempre habrá que copiar los elementos al nuevo objeto array, con lo que siempre se ejecuta `retain` sobre los objetos agregados a copiar al nuevo array.

Finalmente, cuando usamos el método factory `arrayWithArray:` para crear un array a partir del contenido de otro, siempre se ejecuta `retain` sobre los objetos agregados que se deben de copiar al nuevo array. Cuando ejecutamos el método factory `arrayWithArray:` sobre la clase `NSArray` obtenemos un array inmutable y cuando ejecutamos el método factory `arrayWithArray:` sobre la clase `NSMutableArray` obtenemos un array mutable.

1.5. Objetos array de punteros

La clase `NSPointerArray` implementa objetos array mutables con dos características adicionales que las clases `NSArray` y `NSMutableArray` no proporcionan:

1. Las clases `NSArray` y `NSMutableArray` no pueden almacenar punteros `nil`, lo más que pueden hacer es almacenar el objeto singleton de la clase `NSNull` (que vimos en el apartado 1.2), que es una clase singleton usada en los objetos de colección para indicar que un elemento es nulo. Por el contrario la clase `NSPointerArray` permite almacenar el valor `NULL` en sus elementos.

2. La clase `NSPointerArray` puede almacenar referencias fuertes o débiles (que explicamos en el apartado 3.9 del Tema 5 y apartado 2.2.1 del Tema 6).

Cuando las referencias almacenadas en un objeto `NSPointerArray` son fuertes, entonces:

- Si estamos usando gestión de memoria por cuenta de referencias se ejecutan los métodos `retain` y `release` sobre los objetos apuntados.
- Si estamos usando gestión de memoria con recolector de basura, nunca se liberan los objetos agregados ya que el objeto array tiene referencias fuertes a ellos.

Cuando las referencias almacenadas en un objeto `NSPointerArray` son débiles, entonces:

- Si estamos usando gestión de memoria por cuenta de referencias no se ejecutan los métodos `retain` y `release` sobre los objetos apuntados.
- Si estamos usando gestión de memoria con recolector de basura los objetos agregados pueden ser eliminados por el recolector de basura si nadie más los usa. Cuando el elemento es eliminado por el recolector de basura, en su posición dentro del objeto array se almacena el valor `NULL`.

2. Objetos conjunto

Otro tipo de objeto de colección son los **objetos conjunto**, los cuales representan objetos no indexados y sin repetición. Los objetos conjunto se usan cuando la posición de los elementos en la colección no importa. A cambio, los objetos conjunto pueden comprobar si un objeto pertenece a la colección más rápido que los objetos array.

Existen tres clases para representar objetos conjunto:

- `NSSet` representa conjuntos inmutables, es decir, una vez creado el conjunto no se puede modificar el número de objetos agregados, pero sí el contenido de los objetos agregados.
- `NSMutableSet` deriva de `NSSet` y representa objetos conjunto mutables, es decir, donde el número de elementos que forman el conjunto se puede modificar.
- `NSCountedSet` deriva de `NSMutableSet` y además de permitir modificar su contenido, permite almacenar repeticiones de un mismo objeto agregado.

2.1. Objetos conjunto inmutables

Podemos crear una objeto `NSSet` usando los métodos factory:

```
+ (id) set
+ (id) setWithObjects: (id) anObject ...
+ (id) setWithSet: (NSSet*) aSet
```

El primero crea un objeto `NSSet` vacío mientras que el segundo crea un objeto `NSSet` con los objetos agregados que le pasamos como parámetro. Al igual que en el caso de los objetos array, cuando usamos gestión de memoria por cuenta de referencias los objetos conjunto incrementan la cuenta de referencias de los objetos agregados.

Con vista a poder almacenar correctamente objetos agregados en un objeto `NSSet` o derivado debemos redefinir los métodos `isEqual:` y `hash` de los objetos agregados tal como se indicó en el apartado 4.2 del Tema 5. En concreto, si dos objetos son iguales (desde el punto de vista de `isEqual:`), `hash` debe devolver el mismo número, luego si queremos almacenar objetos `Punto` en un objeto conjunto deberemos redefinir estos métodos. El Listado 7.1 y Listado 7.2 muestran un ejemplo de como hacerlo.

```
/* Punto.h */
#import <Foundation/Foundation.h>

@interface Punto : NSObject {
    NSInteger x;
    NSInteger y;
}
@property NSInteger x;
@property NSInteger y;
- initWithTitle:(NSInteger)paramX y:(NSInteger)paramY;
- (BOOL)isEqual:(id)otro;
- (NSUInteger) hash;
- (NSString*)description;
@end
```

Listado 7.1: Interfaz de objetos `Punto` para a almacenar en un objeto conjunto

```
/* Punto.m */
#import "Punto.h"

#define MAX_COORDENADA 50000

@implementation Punto
@synthesize x;
@synthesize y;
- initWithTitle:(NSInteger)paramX y:(NSInteger)paramY {
    if (self = [super init]) {
        x = paramX;
```

```

        y = paramY;
    }
    return self;
}
- (BOOL)isEqual:(id)otro {
    return (x==[otro x] && y==[otro y]);
}
- (NSUInteger)hash {
    return x+MAX_COORDENADA*y;
}
- (NSString*)description {
    return [NSString stringWithFormat:@"%@[%i,%i]",x,y];
}
@end

```

Listado 7.2: Implementación de objetos `Punto` para almacenar en un objeto conjunto

Ahora podemos crear un objeto `NSSet` con dos puntos y comprobar la existencia de objetos `Punto` en el conjunto de la forma:

```

Punto* p1 = [[[Punto alloc] initWithX:4 y:6] autorelease];
Punto* p2 = [[[Punto alloc] initWithX:6 y:4] autorelease];
Punto* p3 = [[[Punto alloc] initWithX:6 y:4] autorelease];
NSSet* conjunto = [NSSet setWithObjects:p1,p2,p3,nil];
NSUInteger n = [conjunto count]; // 2
Punto* p4 = [[[Punto alloc] initWithX:5 y:5] autorelease];
id ps = [conjunto member:p4]; // ps == nil
Punto* p5 = [[[Punto alloc] initWithX:6 y:4] autorelease];
ps = [conjunto member:p5]; // ps == p5

```

Obsérvese que el objeto `p2` y `p3` son idénticos desde el punto de vista de `hash`, con lo que el objeto `p3` no se añade al conjunto.

Cuando metemos objetos en un objeto conjunto no deberíamos modificar variables de instancia del objeto de las que dependa el método `hash`, ya que si no el objeto conjunto puede perder su consistencia.

2.2. Objetos conjunto mutables

Los objetos conjunto mutables están representados por la clase `NSMutableSet`. Para crear un objeto conjunto mutable podemos usar los métodos factory de `NSSet`, los cuales han sido convenientemente redefinidos para devolver instancias de `NSMutableSet`:

```

+ (id)set
+ (id)setWithObjects:(id)anObject ...
+ (id)setWithSet:(NSSet*)aSet

```

El tercero de ellos recibe un objeto conjunto que puede ser de tipo `NSSet` o derivado, y devuelve otro objeto de tipo `NSMutableSet`.

Los objetos `NSMutableSet` permiten variar el número de objetos agregados que contienen, para lo cual proporcionan métodos como:

- `(void) addObject:(id) anObject`
- `(void) removeObject:(id) anObject`

Si estamos usando gestión de memoria por cuenta de referencias, cuando agregamos un objeto al objeto conjunto se incrementa su cuenta de referencias con `retain`, y cuando eliminamos un objeto agregado del conjunto se decrementa su cuenta de referencias con `release`. Cuando ejecutamos `release` sobre el objeto conjunto se ejecuta `release` sobre todos sus objetos agregados.

2.3. Operaciones con conjuntos

Podemos realizar álgebra de conjuntos con objetos conjunto. La clase `NSSet` proporciona algunas operaciones con conjuntos como:

- `(BOOL) isSubsetOfSet:(NSSet*) otherSet`
- `(BOOL) isEqualToSet:(NSSet*) otherSet`

La clase `NSMutableSet` añade operaciones que implican modificar el contenido del objeto conjunto como:

- `(void) unionSet:(NSSet*) otherSet`
- `(void) intersectSet:(NSSet*) otherSet`
- `(void) minusSet:(NSSet*) otherSet`

2.4. Objetos conjunto con repeticiones

Los **objetos conjunto con repeticiones**, también llamados **objetos bolsa**, permiten almacenar varias veces el mismo objeto. En Objective-C se representan con la clase `NSCountedSet` (la cual deriva de `NSMutableSet`).

Para almacenar varias veces el mismo objeto agregado, `NSCountedSet` en vez de almacenar varias instancias del mismo objeto, lo que hace es incrementar un **contador de repeticiones** asociado a cada objeto agregado. Después `NSCountedSet` exige que ese objeto sea eliminado tantas veces como es añadido.

Cuando se introduce un objeto agregado en el objeto conjunto con repeticiones, el objeto agregado recibe el mensaje `retain` sólo la primera vez, las

demás veces se incrementa su contador de repeticiones, pero no se vuelve a enviar el mensaje `retain` al objeto. Análogamente, cuando se elimina un objeto agregado del objeto conjunto con repeticiones sólo se le envía el mensaje `release` cuando su contador de repeticiones llega a cero.

Tenga en cuenta que de cara a identificar repeticiones el objeto conjunto con repeticiones lo que tiene en cuenta es el valor devuelto por el método `hash` del los objetos agregados, no su dirección de memoria. Es decir, en el siguiente ejemplo sólo recibirán los mensajes `retain` y `release` los objetos `p1` y `p2`. Además `p2` tendría su cuenta de repeticiones a dos:

```
Punto* p1 = [[[Punto alloc] initWithX: 4 y:6] autorelease];
Punto* p2 = [[[Punto alloc] initWithX: 6 y:4] autorelease];
Punto* p3 = [[[Punto alloc] initWithX: 6 y:4] autorelease];
NSCountedSet* bolsa = [NSCountedSet
                        setWithObjects:p1,p2,p3,nil];
```

3. Recorrer los elementos de una colección

Los objetos de colección proporcionan mecanismos que facilitan el recorrer sus elementos.

3.1. Enumeraciones

Para recorrer los elementos de un objeto de colección podemos pedir al objeto de colección un objeto de tipo `NSEnumerator`, el cual nos permite recorrer cómodamente sus elementos en un bucle. La clase `NSEnumerator` actúa como una clase abstracta, y sólo tiene dos métodos de instancia:

- `(id)nextObject`
- `(NSArray*)allObjects`

Para obtener un objeto de tipo `NSEnumerator` usamos el método `objectEnumerator`, el cual tienen las clases `NSArray`, `NSMutableArray`, `NSSet`, `NSMutableSet` y `NSCountedSet` pero no `NSPointerArray`. Es decir, objetos agregados en un objeto `NSPointerArray` sólo se pueden acceder usando el método `pointerAtIndex:`.

En método `nextObject` se suele llamar en un bucle de la siguiente forma. El bucle termina cuando `nextObject` devuelve `nil`:

```
Punto* p1 = [[[Punto alloc] initWithX:-2 y:2] autorelease];
Punto* p2 = [[[Punto alloc] initWithX:2 y:2] autorelease];
NSArray* array = [NSArray arrayWithObjects:p1,p2,nil];
NSEnumerator* e = [array objectEnumerator];
Punto* p;
```

```

while ((p = [e nextObject])) {
    // Hacer algo con p
    .....
}

```

Obsérvese que `nextObject` devuelve un `id` mientras que `p` es de tipo `Punto*`. Debido a que (como vimos en el apartado 2.2 del Tema 3) existe conversión implícita entre un puntero a objeto dinámico y un puntero a objeto estático no necesitamos hacer casting en la asignación. El doble paréntesis evita el warning que se produce cuando se usa el resultado de una asignación como condición.

Las clases `NSArray` y `NSMutableArray` disponen también del método `reverseObjectEnumerator`, que permite recorrer los elementos desde el último índice al primer índice. Las clases `NSSet`, `NSMutableSet` y `NSCountedSet` sólo disponen del método `objectEnumerator` porque sus objetos agregados no tienen orden, es decir, no se garantiza el orden en que se nos devuelven.

No debemos añadir o eliminar elementos a un objeto de colección cuando lo estamos recorriendo. En caso de añadir un elemento al objeto de colección cuando se está recorriendo los efectos son impredecibles. Apple indica en su documentación que en el futuro un intento de modificación de un objeto de colección que se esté recorriendo podría impedirse con un error en tiempo de ejecución. Si una aplicación tiene varios hilos que puedan acceder concurrentemente a un objeto de colección deberemos plantearnos el uso de un mecanismo de sincronización como los que se estudiaron en el apartado 7 del Tema 5.

3.2. Enumeraciones rápidas

A partir de Objective-C 2.0 los objetos de colección implementan la interfaz `NSFastEnumeration`, la cual permite recorrer los elementos de la colección de forma más cómoda. Para ello usamos la siguiente sintaxis en el bucle `for`:

```

Punto* p1 = [[[Punto alloc] initWithX:4 y:6] autorelease];
Punto* p2 = [[[Punto alloc] initWithX:6 y:4] autorelease];
NSSet* conjunto = [NSSet setWithObjects:p1,p2,nil];
for (Punto* p in conjunto) {
    // Hacer algo con p
    .....
}

```

La variable `p` se puede declarar dentro del bucle `for`, o antes del bucle `for`.

También podemos utilizar una enumeración rápida sobre un objeto `NSEnumerator`, es decir, la clase `NSEnumerator` implementa la interfaz `NSFastEnumeration`. Por ejemplo:

```
NSEnumerator* e = [conjunto objectEnumerator];
for (Punto* p in e) {
    // Hacer algo con p
    .....
}
```

Las enumeraciones rápidas proporcionan al menos tres ventajas respecto a las enumeraciones convencionales:

- Ejecutan más rápido que una enumeración convencional
- La sintaxis es más concisa.
- Las enumeraciones rápidas son seguras frente a intentos de cambio en el objeto de colección que estamos enumerando.

Si intentamos modificar un objeto de colección cuando estamos haciendo una enumeración rápida, el método de modificación del objeto de colección produciría una excepción en tiempo de ejecución.

3.3. Ejecutar un selector

En muchas ocasiones queremos recorrer los objetos agregados a una colección para enviarles un único mensaje. Los objetos de colección disponen de los métodos:

- `(void)makeObjectsPerformSelector:(SEL)aSelector`
- `(void)makeObjectsPerformSelector:(SEL)aSelector
withObject:(id)anObject`

Los cuales permiten enviar de forma eficiente el mensaje `aSelector` a cada uno de los objetos de la colección. En la primera forma el método `aSelector` no debe de tener parámetros, y en la segunda forma el método `aSelector` recibe como argumento el objeto `anObject`. Por ejemplo, para ejecutar el método `imprime` (de la categoría del Listado 4.5) sobre los objetos agregados de un objeto conjunto podemos hacer:

```
[conjunto makeObjectsPerformSelector:@selector(imprime)];
```

3.4. Ordenar elementos

Los objetos conjunto no se pueden ordenar porque no tienen orden, pero los objetos array sí. Para tal fin podemos usar métodos de instancia como:

```
- (NSArray*)sortedArrayUsingSelector:(SEL)comparator
```

Este método se pueden ejecutar sobre objetos array mutables e inmutables, y siempre devuelve un nuevo objeto array de tipo `NSArray`.

El método `sortedArrayUsingSelector:` ejecuta el método `comparator` sobre sus objetos agregados pasándoles como parámetro otro objeto con el que compararlo. Los objetos agrados deberán disponer de un método como el que muestra el Listado 7.3 y Listado 7.4 de forma que recibe un objeto `otro` con el que comparar con `self`, y devuelven un valor de tipo `NSComparisonResult` el cual podrá tomar tres valores:

- `NSOrderedDescending` si `self` es mayor a `otro`.
- `NSOrderedAscending` si `self` es menor a `otro`.
- `NSOrderedSame` si son iguales.

```
/* Punto.h */
#import <Foundation/Foundation.h>

@interface Punto : NSObject {
    NSInteger x;
    NSInteger y;
}
@property NSInteger x;
@property NSInteger y;
- initWithTitle:(NSInteger)paramX y:(NSInteger)paramY;
- (double) modulo;
- (double) fase;
- (NSComparisonResult) compararModulo:(Punto*)otro;
- (NSComparisonResult) compararFase:(Punto*)otro;
- (NSString*)description;
@end
```

Listado 7.3: Interfaz de `Punto` ordenable

```
/* Punto.m */
#import "Punto.h"

@implementation Punto
@synthesize x;
@synthesize y;
- initWithTitle:(NSInteger)paramX y:(NSInteger)paramY {
    if (self = [super init]) {
        x = paramX;
        y = paramY;
    }
    return self;
}
- (double) modulo {
    return sqrt(x*x+y*y);
}
- (double) fase {
```

```

        return atan(((double)x)/y);
    }
- (NSComparisonResult)compararModulo:(Punto*)otro {
    if ([self modulo] > [otro modulo])
        return NSOrderedDescending;
    else if ([self modulo] < [otro modulo])
        return NSOrderedAscending;
    else
        return NSOrderedSame;
}
- (NSComparisonResult)compararFase:(Punto*)otro {
    if ([self fase] > [otro fase])
        return NSOrderedDescending;
    else if ([self fase] < [otro fase])
        return NSOrderedAscending;
    else
        return NSOrderedSame;
}
- (NSString*)description {
    return [NSString stringWithFormat:@"%@[%i,%i]",x,y];
}
@end

```

Listado 7.4: Implementación de Punto ordenable

Ahora para obtener un array ordenado de acuerdo a los dos criterios de ordenación implementados en `Punto` (módulo y fase) podríamos hacer:

```

Punto* p1 = [[[Punto alloc] initWithX:-4 y:2] autorelease];
Punto* p2 = [[[Punto alloc] initWithX:2 y:2] autorelease];
Punto* p3 = [[[Punto alloc] initWithX:-3 y:-1] autorelease];
Punto* p4 = [[[Punto alloc] initWithX:-3 y:2] autorelease];
NSArray* array = [NSArray arrayWithObjects:p1,p2,p3,p4,nil];
NSArray* array_modulo = [array sortedArrayUsingSelector:
                         @selector(compararModulo:)];
NSArray* array_fase = [array sortedArrayUsingSelector:
                       @selector(compararFase:)];

```

3.5. Filtrar elementos

Las clases de colección disponen de los métodos:

- (NSArray*)filteredArrayUsingPredicate:(NSPredicate*)predicate
 - (NSSet*)filteredSetUsingPredicate:(NSPredicate*)predicate

Los cuales permiten obtener el subconjunto de objetos agregados que cumplen con un **predicado**. Los predicados están representados por instancias de la clase `NSPredicate` y son expresiones que pueden acceder a las propiedades del objeto sobre el que se evalúa el predicado. Por ejemplo, el siguiente

te predicado permite obtener el subconjunto de objetos agregados de tipo Punto donde sus coordenadas x e y no son mayores a 10.

```
NSPredicate* predicado = [NSPredicate predicateWithFormat:
                           @"x<=10 and y<=10"];
NSSet* subconjunto = [conjunto
                      filteredSetUsingPredicate:predicado];
[subconjunto makeObjectsPerformSelector:@selector(imprime)];
```

4. Objetos diccionario

Un **diccionario** es una colección de pares clave-valor. A un par clave-valor también se le llama **entrada**. Objective-C proporciona tres clases para representar los objetos diccionario:

- `NSDictionary` representa un diccionario que una vez creado no podemos modificar sus claves ni sus valores asociados. Tampoco podemos modificar el objeto que actúa como clave, pero sí el objeto que actúa como valor.
- `NSMutableDictionary` es una subclase de `NSDictionary` que añade funcionalidad para añadir o borrar entradas.
- `NSMapTable` implementa un diccionario más flexible que los dos anteriores y la estudiaremos en el apartado 4.3.

4.1. Objetos diccionario inmutables

Vamos a empezar viendo cómo se trabaja con objetos diccionario de la clase `NSDictionary`.

4.1.1. Creación de un objeto diccionario

Para crear un objeto diccionario se usan métodos factory como:

```
+ (id)dictionaryWithObjectsAndKeys:(id)firstObject ...
```

El cual recibe una lista con un número par de elementos acabada en `nil`. El primer elemento de la lista representa el primer valor, el segundo elemento de la lista la primera clave, y así sucesivamente.

```
+ (id)dictionaryWithObjects:(NSArray*)objects
                     forKeys:(NSArray*)keys
```

El cual recibe un objeto array con los valores y otro objeto array con sus correspondientes claves.

Por ejemplo, podemos crear un diccionario de días festivos de la forma:

```

Fecha* f1 = [[[Fecha alloc] initWithDia:1 mes:1 anno:2008]
              autorelease];
Fecha* f2 = [[[Fecha alloc] initWithDia:6 mes:1 anno:2008]
              autorelease];
Fecha* f3 = [[[Fecha alloc] initWithDia:3 mes:4 anno:2008]
              autorelease];
Fecha* f4 = [[[Fecha alloc] initWithDia:4 mes:4 anno:2008]
              autorelease];
NSDictionary* festivos = [NSDictionary
    dictionaryWithObjectsAndKeys:
        f1, @"Año nuevo"
        ,f2, @"Reyes"
        ,f3, @"Jueves santo"
        ,f4, @"Viernes santo"
        ,nil
];

```

Dentro de un objeto diccionario la clave debe de ser única, es decir, los objetos agregados que se usan para representar la clave deben ser distintos desde el punto de vista de `isEqual:`. Es muy común que para la clave se use un objeto `NSString`, pero se puede usar como clave cualquier objeto que adopte el protocolo `NSCopying`.

Cuando creamos un objeto diccionario, el método `factory` que se encarga de crear el objeto diccionario sigue los siguientes pasos:

1. Crea copias de las claves enviando a los objetos agregados que actúan como clave el mensaje `copy`. Ésta es la razón por la que las claves deben de implementar el protocolo `NSCopying`. A partir de este momento el diccionario usa como claves las copias, y la búsqueda de claves se hace con el método `isEqual:` del objeto copia que mantiene el diccionario.
2. Se ejecuta `retain` sobre los objetos que actúan como valor.

Cuando el diccionario se libera se ejecuta `release` tanto sobre los objetos copia que actúan como claves, como sobre los objetos agregados que actúan como valor.

Ni la clave ni el valor pueden ser `nil`, si queremos representar el valor `nil` podemos usar el objeto singleton de la clase `NSNull`. Ya que `NSNull` adopta el protocolo `NSCopying`, el objeto singleton de su clase también puede usarse como clave.

4.1.2. Acceso a los elementos del diccionario

Podemos preguntar por el número de entradas en un diccionario usando el método:

- `(NSUInteger)count`

O bien obtener las claves o valores que componen el diccionario usando los métodos:

- `(NSArray*)allKeys`
- `(NSArray*)allValues`

El número de elementos devueltos por `allKeys` siempre va a coincidir con el número de elementos devueltos por `allValues`, pero la posición de los elementos del objeto array devuelto por `allKeys` no tiene porqué coincidir con la posición de los elementos de cada entrada en `allValues`.

Si queremos buscar un determinado elemento por clave o por valor podemos usar:

- `(id)objectForKey:(id)aKey`
- `(NSArray*)allKeysForObject:(id)anObject`

El primer método siempre devuelve un sólo valor para la clave dada (que será `nil` si no lo encuentra), mientras que el segundo método puede devolver cero, una o varias claves que tengan ese valor.

4.1.3. Recorrer los elementos de un diccionario

Si queremos recorrer los elementos de un diccionario podemos obtener un objeto `NSEnumerator` tanto para las claves como para los valores usando respectivamente:

- `(NSEnumerator*)keyEnumerator`
- `(NSEnumerator*)objectEnumerator`

Si queremos acceder a ambos valores lo normal es obtener todas las claves y a partir de la clave obtener su correspondiente valor de la forma:

```
NSEnumerator* e = [festivos keyEnumerator];
NSString* k;
while ((k=[e nextObject])) {
    NSLog(@"%@", k, [festivos objectForKey:k]);
}
```

En Objective-C 2.0 `NSDictionary` implementa `NSFastEnumerator` con lo que es más fácil obtener sus claves de la forma:

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

4.2. Objetos diccionario mutables

La clase `NSMutableDictionary` añade métodos a `NSDictionary` para poder modificar las entradas del objeto diccionario.

Podemos modificar las entradas del objeto diccionario con los métodos:

- `(void)setObject:(id)anObject forKey:(id)aKey`
- `(void)removeObjectForKey:(id)aKey`

El primer método añade una entrada al diccionario, para ello copia (con `copy`) `aKey` e incrementa la cuenta de referencias de `anObject`. Si `aKey` ya existe (desde el punto de vista de `isEqual:`) el anterior valor de `anObject` recibe el mensaje `release` y es eliminado del diccionario.

El segundo método envía los mensajes `release` tanto al objeto copia que usa como clave como a su correspondiente objeto valor.

4.3. Objetos diccionario de punteros

La clase `NSMapTable` es un diccionario en el que además de poder modificar sus entradas podemos almacenar referencias débiles y los punteros no están limitados a almacenar objetos Objective-C.

Dependiendo de si queremos almacenar claves y valores fuertes o débiles debemos usar uno de estos métodos factory para crear el objeto `NSMapTable`:

- + `(id)mapTableWithStrongToStrongObjects`
- + `(id)mapTableWithStrongToWeakObjects`
- + `(id)mapTableWithWeakToStrongObjects`
- + `(id)mapTableWithWeakToWeakObjects`

Cuando estamos usando gestión de memoria por cuenta de referencias sólo los objetos agregados fuertes reciben los mensajes `retain` y `release`. En caso de usar gestión de memoria con recolector de basura las referencias débiles que son eliminadas por el recolector de basura aparecen marcadas como `NULL` en el objeto diccionario.

5. Tipos fundamentales en colecciones

Los objetos de colección están diseñados para almacenar objetos Objective-C, sin embargo, gracias a la clase `NSValue`, también es posible almacenar tipos de datos fundamentales envueltos en un objeto `NSValue`. Para crear un objeto de este tipo podemos usar el método de clase:

```
+ (NSValue*)valueWithBytes:(const void*)value  
                      objCType:(const char*)type
```

El cual envuelve el valor apuntado por `value` junto con su tipo que se obtiene mediante la directiva del compilador `@encode()`, la cual veremos en el apartado 1.2.1 del Tema 10.

Por ejemplo, para crear un `NSValue` que envuelva un entero haríamos:

```
int n = 5;  
NSValue* numero = [NSValue valueWithBytes:&n  
                      objCType:@encode(int)];
```

Después podemos obtener el valor con el método:

```
- (void*)pointerValue  
- (void)getValue:(void*)buffer
```

El primer método devuelve un puntero al valor mientras que el segundo copia en `buffer` el valor. Podemos preguntar por el tipo del valor envuelto con:

```
- (const char*)objCType
```

También podemos usar `NSValue` para almacenar objetos Objective-C sobre los que no se ejecute `retain` con:

```
+ (NSValue*)valueWithNonretainedObject:(id)anObject
```

Tema 8

Key-Value Coding

Sinopsis:

En este tema introduciremos un mecanismo para asignar nombres a las propiedades de los objetos Objective-C. Este mecanismo de acceso es menos eficiente que los accesos convencionales, pero es más flexible a la hora de interactuar con objetos de los cuales no tenemos su información de introspección.

1. Qué es KVC

Key-Value Coding (KVC) es un mecanismo para acceder a las propiedades de los objetos indirectamente. En vez de acceder a las propiedades de un objeto directamente mediante sus variables de instancia o métodos getter/setter, KVC nos permite acceder a las propiedades indirectamente a través de una cadena (clave) que identifica a la propiedad a acceder.

KVC nos permite acceder a propiedades que sean objetos, devolviéndonos el puntero al objeto. La propiedad accedida también puede ser una colección, en cuyo caso KVC nos permite acceder a toda la colección o a un determinado elemento de la colección. KVC también nos permite acceder a propiedades escalares, en este caso KVC envuelve automáticamente el escalar en un `NSNumber` o `NSValue` según proceda. Los valores `nil` nunca se devuelven directamente, sino que se representan como instancias de la clase singleton `NSNull`.

La tecnología KVC se usa a menudo indirectamente en otras tecnologías como AppleScript, Cocoa Bindings y Core Data. En general el acceso a las propiedades de un objeto mediante KVC es ligeramente más lento que su acceso directo mediante métodos getter/setter, con lo que su uso debe limitarse a los escenarios donde resulta necesaria y beneficiosa esta forma de acceso.

Los métodos para KVC están implementados en la categoría `NSKeyValueCoding`, la cual añade métodos a la clase raíz `NSObject` con lo que todos los objetos Objective-C pueden acceder fácilmente a esta funcionalidad.

Los dos métodos más importantes que implementa `NSKeyValueCoding` para el acceso KVC son:

- `(id)valueForKey:(NSString*)key`
- `(void)setValue:(id)value forKey:(NSString*)key`

El primero permite obtener el valor de una propiedad dado el nombre de la propiedad y el segundo permite cambiar el valor de una propiedad dado su nombre.

1.1. Tecnologías relacionadas

Dentro del patrón de diseño Modelo-Vista-Controlador, Core Data es la tecnología que permite representar y guardar fácilmente en almacenamiento persistente los **objetos modelo** (también llamados **entidades**) así como las **relaciones** entre las entidades. Core Data utiliza propiedades KVC para representar tanto los atributos de las entidades como sus relaciones basándose

en el conocido **modelo entidad-relación**. Las relaciones pueden ser de dos tipos, relaciones uno a uno, y relaciones uno a muchos.

AppleScript permite acceder a los objetos de las aplicaciones Cocoa desde fuera del proceso de la aplicación. Estos objetos accesibles mediante AppleScript corresponden con **objetos modelo** del patrón de diseño Modelo-Vista-Controlador⁴². Los objetos modelo están descriptos en la llamada **descripción de clase** que veremos en el apartado 5. AppleScript usa KVC para acceder a las propiedades de los objetos modelo. Ejemplos de propiedades de un objeto modelo AppleScript serían `words`, `font`, `documents` o `color`. Normalmente los objetos modelo forman un árbol jerárquico, en cuyo caso la aplicación AppleScript debe recorrer este árbol para acceder a las propiedad que le interese leer o modificar.

1.2. Terminología

En KVC se usa el término **propiedad KVC**, o simplemente **propiedad** (no confundir con las propiedades del lenguaje Objective-C 2.0 que introdujimos en el Tema 6) para referirse a las variables de instancia de un objeto. Estas variables pueden ser de tres tipos:

1. **Atributos.** Son propiedades simples, como escalares, cadenas de caracteres o booleanos que corresponden con las variables que describen una entidad.
2. **Propiedades KVC uno a uno.** Son punteros a objetos que corresponden con las relaciones uno a uno del modelo entidad-relación.
3. **Propiedades KVC uno a muchos.** Son colecciones de objetos, como por ejemplo una instancia de `NSArray`, que corresponden con las relaciones uno a muchos del modelo entidad relacióñ. Las propiedades uno a muchos se dividen a su vez en **propiedades KVC uno a muchos ordenadas**, que corresponden con la idea de array, y **propiedades KVC uno a muchos no ordenadas**, que corresponden con la idea de conjunto.

Esta distinción será importante en las explicaciones que se dan en los siguientes apartados.

En KVC se llama **clave** (key) a una cadena con el nombre de una propiedad KVC. Normalmente la clave corresponde con el nombre del método getter (o en su defecto el de la variable de instancia) para la propiedad. Las claves suelen empezar con una minúscula y no deben de contener espacios (p.e. `pagador`, `sumaTotales`, `importe`).

⁴² No es estrictamente necesario usar el patrón de diseño Modelo-Vista-Controlador para hacer una aplicación accesible mediante AppleScript, pero en este caso la implementación de los objetos modelo se vuelve mucho más trabajosa.

En KVC se llama **camino de claves** (key path) a una o más claves separadas por puntos que indica la secuencia de claves a recorrer. La primera clave indica la propiedad KVC del objeto receptor, y las siguientes claves se evalúan respecto al valor de la clave anterior.

2. Métodos de acceso

En el apartado 1 indicamos que la categoría `NSKeyValueCoding` implementaba los métodos de acceso KVC en la clase raíz `NSObject`, e indicamos que sus dos métodos más importantes eran `valueForKey:` y `setValue:forKey:`. En este apartado vamos a describir con más detalle los métodos de acceso con que cuenta la categoría `NSKeyValueCoding`.

En los siguientes apartados es importante diferenciar entre los **métodos KVC** que son los métodos que implementa la categoría `NSKeyValueCoding`, y los **métodos de patrón**, que son métodos que debe implementar el programador del objeto y cuyo nombre tiene que seguir un determinado patrón para que puedan encontrarlos y usarlos los métodos KVC.

2.1. Lecturas simples

Para leer una propiedad KVC se ejecuta sobre el objeto en cuestión el método KVC:

- `(id)valueForKey:(NSString*)key`

Este método KVC usa la información de introspección para buscar un método de patrón cuyo nombre coincide con el valor del parámetro `key`. En concreto:

1. Si `key` vale @"colorFondo", se buscan los métodos de patrón llamados `colorFondo` o `getColorFondo`. Si alguno de estos métodos de patrón existe, `valueForKey:` lo ejecuta para obtener el valor de la propiedad KVC.
2. En caso contrario `valueForKey:` busca una variable de instancia cuyo nombre coincide con el valor del parámetro `key`. En concreto, si `key` vale @"colorFondo", se buscaría una variable de instancia llamada `colorFondo` o `_colorFondo`. Si alguna de estas variables de instancia existiese se devolvería su valor.
3. En caso de que tampoco exista se ejecutará el método KVC `valueForUndefinedKey:`, el cual por defecto, o sea si no lo redefinimos, lanza una `NSUndefinedKeyException`. Sin embargo, las subclases pueden redefinir este comportamiento.

Como ejemplo, supongamos que tenemos la clase `Punto` definida tal como muestran el Listado 8.1 y Listado 8.2. En este caso podríamos crear un `Punto` y usar los métodos KVC para acceder a sus propiedades KVC de la forma:

```
Punto* p = [[Punto alloc] initWithX:3 Y:4];
NSNumber* y = [p valueForKey:@"y"];
 NSLog(@"El valor de y es: %@", y);
```

En este ejemplo, al no existir método de patrón getter en `Punto`, `valueForKey:` accede directamente a la variable de instancia `y`. Además `valueForKey:` es capaz de detectar que `y` es una variable de tipo `NSInteger` y la envuelve en un objeto `NSNumber` antes de devolverla.

```
#import <Foundation/Foundation.h>

@interface Punto : NSObject {
    @public
    NSInteger x;
    NSInteger y;
}
- (id)initWithX:(NSInteger)x Y:(NSInteger)y;
@end
```

Listado 8.1: Interfaz de la clase `Punto`

```
#import "Punto.h"

@implementation Punto
- (id)initWithX:(NSInteger)paramX Y:(NSInteger)paramY {
    if (self = [super init]) {
        x = paramX;
        y = paramY;
    }
    return self;
}
@end
```

Listado 8.2: Implementación de la clase `Punto`

Otros métodos KVC implementados en la categoría `NSKeyValueCoding` son:

- `(id)valueForKeyPath:(NSString*)keyPath`
- `(NSDictionary*)dictionaryWithValuesForKeys:(NSArray*)keys`

El primer método recibe un camino de claves en `keyPath`, lo recorre, y acaba devolviendo su valor. El segundo método permite obtener varias propiedades KVC a la vez, para lo cual en `keys` le pasamos un array con las propiedades KVC a leer, y devuelve un diccionario con las claves y su correspondiente valor.

Por ejemplo, si creamos un Segmento formado por dos objeto Punto tal como muestra el Listado 8.3 y Listado 8.4, vamos a poder usar el camino de claves "desde.x" de la siguiente forma:

```
Punto* p1 = [[Punto alloc] initWithX:3 Y:4];
Punto* p2 = [[Punto alloc] initWithX:8 Y:-2];
Segmento* s = [[Segmento alloc] initWithDesde:p1 hasta:p2];
NSNumber* x = [s valueForKeyPath:@"desde.x"];
 NSLog(@"%@", x);
[p1 release];
[p2 release];
[s release];
```

```
#import <Foundation/Foundation.h>
#import "Punto.h"

@interface Segmento : NSObject {
    Punto* desde;
    Punto* hasta;
}
@property(retain) Punto* desde;
@property(retain) Punto* hasta;
-(id)initWithDesde:(Punto*)paramDesde
              hasta:(Punto*)paramHasta;
-(void) dealloc;
@end
```

Listado 8.3: Interfaz de la clase Segmento

```
#import "Segmento.h"

@implementation Segmento
@synthesize desde;
@synthesize hasta;
-(id)initWithDesde:(Punto*)paramDesde
              hasta:(Punto*)paramHasta {
    if (self = [super init]) {
        desde = [paramDesde retain];
        hasta = [paramHasta retain];
    }
    return self;
}
-(void) dealloc {
    [desde release];
    [hasta release];
    [super dealloc];
}
@end
```

Listado 8.4: Implementación de la clase Segmento

Es importante no confundir las propiedades del lenguaje Objective-C 2.0 que aparecen en la clase Segmento con las propiedades KVC. El primer grupo de

propiedades sirven para definir métodos getter/setter para acceder a las variables de instancia. El segundo grupo define un mecanismo de acceso a propiedades de un objeto modelo mediante una cadena que actúa como clave. En nuestro ejemplo, como `Segmento` define propiedades del lenguaje, el método KVC `valueForKeyPath:` encontrará el método de patrón getter de la propiedad `desde` del objeto `Segmento`. Sin embargo, no encontrará el método de patrón getter de la propiedad `x` (pero sí su variable de instancia). Observe que el método de patrón getter de la propiedad `desde` no está implementado explícitamente en el [Listado 8.4](#) sino que lo implementa la directiva del compilador `@synthesize` sobre la propiedad del lenguaje `desde`.

2.2. Representación dinámica de objetos

Una ventaja de KVC es que no requiere conocer la interfaz de una clase para acceder a sus propiedades. Por ejemplo, si no importamos la interfaz `Persona` en el siguiente ejemplo obtendríamos un warning:

```
Persona* p = ...  
NSString* nombre = [p nombre];
```

Sin embargo, con KVC podemos acceder a las propiedades de un objeto sin conocer su interfaz concreta:

```
id p = ...  
NSString* nombre = [p valueForKey:@"nombre"];
```

Ya indicamos en el apartado 1.1 que Core Data utiliza propiedades KVC para representar tanto los atributos de las entidades como sus relaciones. Core Data utiliza la clase `NSManagedObject` para representar las entidades del repositorio. La ventaja de usar `NSManagedObject` junto con KVC es que no es necesario crear clases para representar cada entidad del almacenamiento persistente. Una instancia de `NSManagedObject` puede representar cualquier entidad.

```
NSManagedObject p = ...  
NSString* nombre = [p valueForKey:@"nombre"];
```

Aunque no es necesario, muchas veces sí que se crear clases derivadas de `NSManagedObject` para representar las distintas entidades. El [Listado 8.5](#) muestra la interfaz de una clase `Persona` tal como la generaría Core Data. Observe que las propiedades no tienen variables de instancia. Además las propiedades están declaradas en el [Listado 8.6](#) como `@dynamic`. Esto se debe a que `NSManagedObject` se encarga de acceder a estas propiedades, nosotros no necesitamos hacer nada más.

```
#import <CoreData/CoreData.h>

@interface Persona : NSManagedObject {
}
@property (nonatomic, retain) NSString* nombre;
@property (nonatomic, retain) NSNumber* edad;
@end
```

Listado 8.5: Interfaz de Persona

```
#import "Persona.h"

@implementation Persona
@dynamic nombre;
@dynamic edad;
@end
```

Listado 8.6: Implementación de Persona

2.3. Escrituras simples

Para modificar una propiedad KVC se ejecuta sobre el objeto en cuestión el método KVC:

- `(void)setValue:(id)value forKey:(NSString*)key`

Este método es capaz de determinar el tipo de la propiedad a modificar. En caso de que la propiedad sea de un tipo fundamental, el método convierte `value` a este tipo antes de escribirlo en la propiedad correspondiente.

La forma de operar de `setValue:forKey:` es similar a la de `valueForKey:..`. En concreto:

1. En base al valor del parámetro `key` busca un método de patrón con el nombre `setKey:..`. Por ejemplo, si `key` vale `@"colorFondo"`, busca el método `setColorFondo:..`. Si el método existe, `setValue:forKey:` lo ejecuta para asignar el valor a la propiedad.
2. Si no lo encuentra, `setValue:forKey:` busca una variable de instancia con el nombre de la clave. Por ejemplo, si la clave es `@"colorFondo"`, busca la variable de instancia `colorFondo`, y también busca la variable de instancias `_colorFondo`.
3. Si la variable de instancia anterior tampoco existe, `setValue:forKey:` ejecuta el método KVC `setValue:forUndefinedKey:` el cual por defecto lanza una `NSUndefinedKeyException`.

Es importante recordar que nunca debemos de pasar `nil` en el parámetro `value`. Cuando queramos poner a `nil` una propiedad KVC debemos de pasar una instancia de la clase singleton `NSNull`.

Si intentamos poner a `nil` una propiedad KVC que no sea un objeto (p.e. un escalar) `setValue:forKey:` ejecutará el método KVC:

```
- (void)setNilValueForKey:(NSString*)key
```

Y por defecto este método lanza una excepción de tipo `NSInvalidArgumentException`. El programador puede redefinir el método `setNilValueForKey:` para que tenga un comportamiento distinto.

La categoría `NSKeyValueCoding` también define los métodos KVC setter:

```
- (void)setValue:(id)value forKeyPath:(NSString*)keyPath  
- (void)setValuesForKeysWithDictionary:  
    (NSDictionary*)keyedValues
```

Los cuales permiten respectivamente recorrer un camino de claves y asignar a la vez un conjunto de propiedades KVC.

El método KVC `setValuesForKeysWithDictionary:` resulta especialmente útil cuando cargamos un conjunto de propiedades de un fichero donde estén serializadas (almacenadas en fichero), y los nombres de las propiedades serializadas coinciden con los nombres de las propiedades de nuestra clase.

2.4. Soporte para escalares y estructuras

Tanto `valueForKey:` como `setValue:forKey:` son capaces de detectar propiedades escalares y estructuras para envolverlas automáticamente en tipos `NSNumber` o `NSValue`. El tipo `NSNumber` permite envolver enteros, caracteres y booleanos. `NSValue` permite envolver estructuras de uso común como son `NSPoint`, `NSRange`, `NSRect` o `NSSize`.

2.5. Lectura de propiedades uno a muchos

En caso de que una propiedad KVC sea una propiedad uno a muchos, y queramos implementar el acceso a esta propiedad mediante KVC, debemos implementar dos métodos de patrón getter que cumplan con los patrones `countOfKey` y `objectInKeyAtIndex:`, donde `Key` es el nombre de la propiedad KVC. El primer método debe devolver el número de elementos de la propiedad KVC. El segundo recibe un entero con el nombre de la propiedad KVC y devuelve la propiedad que ocupa este índice.

Aunque normalmente se usa un objeto de colección para almacenar una propiedad KVC uno a muchos, también es posible simular esta propiedad. Por ejemplo, el Listado 8.8 muestra la propiedad KVC uno a muchos `vertices`. En realidad los valores de esta propiedad KVC no están almacenados en un array sino que se almacenan en las variables de instancia `desde` y `hasta`, que forman respectivamente el primer y segundo elemento de `vertices`.

```
#import "Punto.h"

@interface Segmento : NSObject {
    Punto* desde;
    Punto* hasta;
}
@property(retain) Punto* desde;
@property(retain) Punto* hasta;
-(id)initWithDesde:(Punto*)paramDesde
              hasta:(Punto*)paramHasta;
-(unsigned int)countOfVertices;
-(Punto*)objectInVerticesAtIndex:(unsigned int)indice;
-(void)getVertices:(Punto**)buffer range:(NSRange)rango;
-(void) dealloc;
@end
```

Listado 8.7: Interfaz de propiedad KVC uno a muchos

```
#import "Segmento.h"

@implementation Segmento
@synthesize desde;
@synthesize hasta;
-(id)initWithDesde:(Punto*)paramDesde
              hasta:(Punto*)paramHasta {
    if (self = [super init]) {
        desde = [paramDesde retain];
        hasta = [paramHasta retain];
    }
    return self;
}
-(unsigned int)countOfVertices {
    return 2u;
}
-(Punto*)objectInVerticesAtIndex:(unsigned int)indice {
    if (indice==0)
        return desde;
    else if (indice==1)
        return hasta;
    else
        return nil;
}
-(void)getVertices:(Punto**)buffer range:(NSRange)rango {
    NSArray* vertices = [self valueForKey:@"vertices"];
    [vertices getObjects:buffer range:rango];
}
-(void) dealloc {
```

```
[desde release];
[hasta release];
[super dealloc];
}
@end
```

Listado 8.8: Implementación de propiedad KVC uno a muchos

Recuerde que la categoría `NSKeyValueCoding` es quién implementa los métodos KVC. Por esta razón, aunque `Segmento` no implementa el método `valueForKey:`, podemos ejecutar este método sobre un objeto de esta clase pasando como argumento `@"vertices"`. La clase `Segmento` no dispone de los métodos de patrón `key`, `getKey` y `setKey:`, con lo que la propiedad `@"vertices"` no se interpreta como un atributo o propiedad KVC uno a uno, pero la clase `Segmento` sí que dispone de los métodos de patrón `countOfKey` y `objectInKeyAtIndex:`, con lo que la propiedad `@"vertices"` se interpreta como una propiedad KVC uno a muchos y el método `valueForKey:`, devuelve un objeto proxy derivado de `NSArray` que permite acceder a los elementos de la propiedad KVC uno a muchos:

```
Punto* p1 = [[Punto alloc] initWithX:3 Y:4];
Punto* p2 = [[Punto alloc] initWithX:8 Y:-2];
Segmento* s = [[Segmento alloc] initWithDesde:p1 hasta:p2];
NSArray* vertices = [s valueForKey:@"vertices"];
Punto* p3 = [vertices objectAtIndex:1]; // p3==p2
```

En caso de que queramos mejorar la eficiencia de los accesos, podemos implementar el método de patrón `getKey:range:`. En el listado anterior también aparece un ejemplo de la implementación de este método. El método debe de guardar en el array de punteros `buffer` los elementos pedidos en el parámetro `rango`.

2.6. Escritura de propiedades uno a muchos

El método `valueForKey:` nos devuelve un array inmutable, con lo que podemos leer los elementos de la propiedad KVC uno a muchos, pero no podemos modificar estos elementos.

Para poder modificar el contenido de una propiedad KVC uno a muchos no debemos acceder a ella con `valueForKey:`, sino con uno de los siguientes métodos:

- (`NSMutableArray*`)
`mutableArrayValueForKey: (NSString*) key`
- (`NSMutableSet*`)
`mutableSetValueForKey: (NSString*) key`

El primero nos devuelve un objeto proxy derivado de `NSMutableArray`, con lo que podemos procesar los elementos de la propiedad KVC uno a muchos como si fueran un array. El segundo nos devuelve un objeto proxy derivado de `NSMutableSet`, con lo que podemos procesar los elementos de la propiedad KVC uno a muchos como si fueran un conjunto.

Los objetos proxy devueltos se limitan a llamar a métodos de patrón del objeto. Estos métodos de patrón son distintos dependiendo de si vamos a acceder a ellos como un array (con `mutableArrayValueForKey:`) o como un conjunto (con `mutableSetValueForKey:`).

En el primer caso debemos de implementar los métodos de patrón `insertObject:inKeyAtIndex:` y `removeObjectFromKeyAtIndex::`. El primero recibe el objeto a insertar y la posición donde insertarlo. El segundo recibe el índice del elemento a eliminar. En caso de que queramos mejorar la eficiencia de los accesos, podemos implementar el método de patrón `replaceObjectInKeyAtIndex:withObject::`.

En el segundo caso debemos de implementar los métodos de patrón `addKeyObject:` y `removeKeyObject::`. Ambos reciben como parámetro el objeto a añadir o eliminar del conjunto.

Lógicamente, también podemos implementar todos los métodos de patrón, en cuyo caso la propiedad KVC uno a muchos se puede procesar como un array o como un conjunto.

3. Métodos de validación

KVC permite que el valor de una propiedad KVC sea validado antes de ser asignado. Sin embargo, un objeto no puede ejecutar esta validación sobre sus propiedades KVC cuando éstas van a ser modificadas, sino que es responsabilidad del usuario del objeto el decidir si quiere validar una propiedad KVC y el lanzar explícitamente este proceso de validación.

3.1. Validar una propiedad

Cuando el usuario de un objeto quiere comprobar si el valor a asignar a una propiedad KVC es correcto, antes de ejecutar el método setter de esta propiedad KVC debe de ejecutar el método KVC:

```
(BOOL)validateValue:(id*)ioValue  
    forKey:(NSString*)key  
    error:(NSError**)outError
```

`ioValue` es el valor a asignar a la propiedad KVC `key`, y también se usa `ioValue` como parámetro de salida para permitir al objeto devolver un valor correcto para la propiedad KVC en caso de que el valor suministrado sea incorrecto. El parámetro `outError` sirve para devolver una explicación de por qué el valor no es válido. En concreto, hay tres posibles respuestas para esta llamada a método:

1. Si el valor de `ioValue` es aceptable el método se limita a devolver `YES`.
2. Si el valor de `ioValue` no es aceptable, pero existe otro valor deducible a partir de `ioValue` que sí que sería aceptable, el método devuelve `YES` y modifica `ioValue` para almacenar el valor compatible.
3. Si el valor de `ioValue` no es aceptable y no se puede crear un valor compatible, el método devuelve `NO` y en `outError` deposita una explicación de por qué el valor no es correcto.

Tenga en cuenta que el método setter del objeto nunca debe de ejecutar la validación de una propiedad KVC, sino que es el usuario del objeto quien debe decidir si quiere ejecutarla. Por ejemplo, en el Listado 8.9 y Listado 8.10 hemos modificado el objeto `Punto` para que permita validar las coordenadas y aceptar sólo valores positivos. Si el método de validación recibe valores negativos, recomienda usar su correspondiente valor positivo. En el apartado 3.2 veremos cómo hace un objeto para validar el valor de cada propiedad KVC. La forma correcta de ejecutarlo sería:

```
Punto* p1 = [Punto new];  
NSNumber* x = [[NSNumber alloc] initWithInt:-3];  
[x autorelease];
```

```
// Cambia x con -3 a otro objeto NSNumber con el valor 3
if ([p1 validateValue:&x forKey:@"x" error:nil] == YES)
    [p1 setValue:x forKey:@"x"];
[p1 release];
```

Obsérvese que es nuestra responsabilidad liberar el objeto `x` (en este caso con `autorelease`), ya que nosotros lo hemos creado. Por el contrario, el objeto devuelto por el método `validateValue:forKey:error:` no debemos de liberarlo, sino que es responsabilidad del método que lo creó el liberarlo (p.e. habiendo hecho un `autorelease` sobre él).

Tenga en cuenta que el método `validateValue:forKey:error:` siempre espera un objeto en `ioValue`. En el ejemplo anterior la propiedad KVC a validar contiene un escalar, con lo que en estos casos debemos de envolver el escalar en un `NSNumber` o `NSValue` según proceda.

3.2. Métodos de patrón para validación

Para cada propiedad KVC que queramos que se pueda validar, debemos de crear un método de patrón con el nombre `validateKey:error:` donde `Key` es la clave de la propiedad KVC a validar. El método `validateValue:forKey:error:` busca un método de patrón con este nombre para ejecutar la validación. Si `validateValue:forKey:error:` no encuentra el método de patrón de una propiedad KVC, por defecto siempre devuelve `YES` y deja inalterado el parámetro `ioValue`.

En el Listado 8.9 y Listado 8.10 hemos implementado el método de patrón `validateX:error:` (`validateY:error:` sería similar) para que garantice que la coordenada `x` del objeto `Punto` siempre sea positiva. Obsérvese que si el método de patrón crea un parámetro de salida, es responsabilidad del método de patrón el liberar esta memoria.

```
#import <Foundation/Foundation.h>

@interface Punto : NSObject {
    NSInteger x;
    NSInteger y;
}
@property(assign) NSInteger x;
@property(assign) NSInteger y;
- (id)initWithX:(NSInteger)x Y:(NSInteger)y;
- (BOOL)validateX:(id*)valor error:(NSError**)error;
@end
```

Listado 8.9: Interfaz de un `Punto` con validación

```
#import "Punto.h"

@implementation Punto
@synthesize x;
@synthesize y;
- (id)initWithX:(NSInteger)paramX Y:(NSInteger)paramY {
    if (self = [super init]) {
        x = paramX;
        y = paramY;
    }
    return self;
}
- (BOOL)validateX:(id*)valor error:(NSError**)error {
    if (*valor==nil) {
        // Este error será capturado al llamar a
        // setValue:forKey: ya que el valor de una clave nunca
        // puede ser nil (sino NSNull) y se ejecutará
        // setNilValueForKey
        // Alternativamente podríamos haber cambiado
        // el valor por 0
        return YES;
    }
    if (![*valor isKindOfClass:[NSNumber class]]) {
        if (*error!=NULL) {
            *error = [NSError errorWithDomain:NSCocoaErrorDomain
                                         code:NSKeyValueValidation
                                         userInfo:nil];
        }
        return NO;
    }
    NSInteger v = [*valor integerValue];
    if (v<0) {
        *valor = [[[NSNumber alloc] initWithInteger:-v]
                  autorelease];
        return YES;
    }
    else
        return YES;
}
@end
```

Listado 8.10: Implementación de un Punto con validación

4. Operadores en caminos de claves

Hemos visto que en KVC métodos como `valueForKeyPath:` y `setValue:forKeyPath:` permiten hacer uso de caminos de claves. KVC también permite el uso de un conjunto definido de operadores en caminos de claves, los cuales están destinados a arrays y conjuntos.

En el apartado 2.5 definimos la propiedad KVC uno a muchos `vertices`, con lo que dado el objeto `s` de tipo `Segmento` podíamos hacer:

```
NSArray* vertices = [s valueForKeyPath:@"vertices"];
```

Los operadores en caminos de claves nos permiten operar sobre las propiedades KVC uno a muchos. Por ejemplo podemos hacer cosas como:

```
Punto* p1 = [[Punto alloc] initWithX:3 Y:5];
Punto* p2 = [[Punto alloc] initWithX:2 Y:1];
Segmento* s = [[Segmento alloc] initWithDesde:p1 hasta:p2];
NSArray* vertices = [s valueForKeyPath:@"vertices"];
NSNumber* n = [s valueForKeyPath:@"vertices.@count"]; // 2
NSNumber* max = [s valueForKeyPath:@"vertices.@max.x"]; // 3
NSNumber* sum = [s valueForKeyPath:@"vertices.@sum.x"]; // 5
NSNumber* med = [s valueForKeyPath:@"vertices.@avg.x"]; // 2.5
[s release];
[p2 release];
[p1 release];
```

5. Describir las propiedades

La clase `NSClassDescription` es una clase abstracta que permite almacenar información KVC sobre una clase. A esta información es a lo que se llama un **descriptor de clase**. Esta clase dispone de tres métodos importantes llamados `attributes`, `toOneRelationshipKeys` y `toManyRelationshipKeys`, los cuales permiten describir respectivamente los atributos, propiedades KVC uno a uno y propiedades KVC uno a muchos de la clase. Una vez creada una instancia de esta clase, podemos registrarla con el método de clase:

```
+ (void)registerClassDescription:(NSClassDescription*)description  
    forClass:(Class)aClass
```

Una vez registrados los descriptores de clase, podemos acceder a ellos mediante el método de clase:

```
+ (NSClassDescription*)classDescriptionForClass:(Class)aClass
```

Aunque esta descripción KVC se podría usar para otros propósitos, en la actualidad sólo se usa para describir las propiedades KVC accesibles desde AppleScript. Como indicamos en el apartado 1, los descriptores de clase permiten describir los objetos modelo, los cuales son accesibles desde AppleScript. La clase `NSScriptClassDescription` es una derivada de `NSClassDescription` que permite describir los objetos modelo para AppleScript.

Tema 9

Key-Value Observing

Sinopsis:

En este tema veremos una forma de notificar cambios en propiedades KVC a otros objetos interesados en conocer cuándo estos cambios se producen. También veremos que el lenguaje Objective-C implementa un mecanismo automático de notificación para estos cambios. Al final del tema se explica que también podemos personalizar estas notificaciones para mejorar el rendimiento de nuestra aplicación.

1. Qué es KVO

Key-Value Observing (KVO) es un mecanismo por el cual unos objetos pueden ser informados de cambios en las propiedades KVC de otros objetos. Las propiedades KVC que pueden ser observadas son tanto atributos como propiedades KVC uno a uno y uno a muchos. Las variables de instancia y propiedades del lenguaje Objective-C no pueden ser observadas si no implementan KVC. En consecuencia KVO es una tecnología que extiende KVC para poder informar de cambios en las propiedades KVC.

Al igual que la categoría `NSKeyValueCoding` añade **métodos KVC** a la clase raíz `NSObject`, la categoría `NSKeyValueObserving` es la que define los **métodos KVO** a la clase raíz `NSObject`.

No debe confundirse KVO con los centros de notificación. Ambos son mecanismos de notificación mediante la implementación del patrón de diseño observador, pero KVO está implementado para todos los objetos compatibles con KVC, mientras que los centros de notificación permiten informar de cambios en propiedades no accesibles por KVC.

Dado que KVO está implementado por la categoría `NSKeyValueObserving` en la clase raíz `NSObject`, suponiendo que una clase es accesible por KVC, con muy poco esfuerzo podemos hacer uso de esta tecnología. En las aplicaciones que siguen el patrón de diseño Modelo-Vista-Controlador, Cocoa Bindings es una tecnología que permite mantener sincronizados fácilmente las propiedades de un objeto vista y su correspondiente objeto modelo. Gracias a KVO un cambio en un objeto modelo se propagaría automáticamente a la vista. Del mismo modo, un cambio en el objeto vista se propaga automáticamente al objeto modelo.

2. Registrar observadores

Es importante diferenciar entre el **objeto observado**, que es el que sus propiedades a observar deben cumplir con KVC y el **objeto observador** que es el que recibe notificaciones cuando el objeto observado cambia. Los métodos KVO los usan tanto el objeto observado como el observador. Por el contrario, sólo es necesario implementar el mecanismo de KVC en el objeto observado.

Si queremos recibir notificaciones KVO cuando las propiedades KVC de un objeto observado cambien necesitamos:

1. Que el objeto observado describa sus propiedades mediante KVC.

2. Usar el método KVO `addObserver:forKeyPath:options:context:` para registrar al objeto observador.
3. Implementar en el objeto observador el método KVO de respuesta `observeValueForKeyPath:ofObject:change:context:`, el cual será ejecutado cuando se cambie alguna propiedad KVC del objeto observado.

2.1. Registrar el objeto observador

Para ser informados de un cambio en una propiedad KVC el objeto observador debe de registrarse en el objeto observado enviándole el mensaje:

```
- (void)addObserver: (NSObject*) anObserver
              forKeyPath: (NSString*) keyPath
                options: (NSKeyValueObservingOptions) options
              context: (void*) context
```

Este método es un método KVO, es decir, esta implementado por la categoría `NSKeyValueObserving`. El parámetro `anObserver` es el objeto observador, `keyPath` es el camino de claves a la propiedad KVC a observar, `options` indica qué eventos observar de acuerdo a la Tabla 8.1 y `context` es un puntero a información que recibirá el objeto observador cuando la propiedad KVC se cambie.

Opción	Descripción
<code>NSKeyValueObservingOptionNew</code>	Indica que el diccionario de cambios debe de incluir el nuevo valor de la propiedad KVC.
<code>NSKeyValueObservingOptionOld</code>	Indica que el diccionario de cambios debe de incluir el antiguo valor de la propiedad KVC.
<code>NSKeyValueObservingOptionInitial</code>	Indica que se debe de enviar una notificación al objeto observador antes de retornar del método de registro. Esta notificación contendrá el valor nuevo pero no el antiguo.
<code>NSKeyValueObservingOptionPrior</code>	Indica que si se activan las notificaciones de valor antiguo y valor nuevo se deben de enviar dos notificaciones del objeto observador (en vez de una): La primera con el valor antiguo y la segunda con el valor nuevo.

Tabla 8.1: Opciones de notificación

Tenga en cuenta que, dado que el método nos pide un `keyPath`, no sólo podemos observar las propiedades KVC de un objeto, sino también las propiedades KVC contenidas en la cadena de claves.

El parámetro `context` puede contener un puntero C o un puntero a objeto Objective-C. Nunca se incrementa la cuenta de referencias al objeto apuntado por `context`, con lo que es responsabilidad del objeto observador el garantizar que este puntero sigue existiendo cuando reciba la notificación.

Conviene tener en cuenta que si la propiedad KVC a observar es una propiedad uno a muchos, representada con un `NSArray` o un `NSSet`. No podemos observar el contenido concreto de un elemento de la colección. En consecuencia, la cadena de claves debe apuntar a la propiedad KVC representada por estos objetos, no a uno de los elementos del contenedor.

2.2. Recibir notificaciones de cambio

Cuando una propiedad KVC de un objeto cambia, los objetos que se hayan registrado como observadores recibirán el mensaje:

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

Este método KVO está definido en la categoría `NSKeyValueObserving` pero está sin implementar. Los objetos observadores deberán implementar este método de respuesta para procesar el evento. En caso contrario se producirá una excepción al no responder el objeto observador a este mensaje.

Entrada	Valor
<code>NSKeyValueChangeKindKey</code>	Indica el tipo de cambio ocurrido
<code>NSKeyValueChangeOldKey</code>	Indica el valor anterior. Sólo existe cuando se solicitó durante el registro del objeto observador.
<code>NSKeyValueChangeNewKey</code>	Indica el valor nuevo. Sólo existe cuando se solicitó durante el registro del objeto observador.
<code>NSKeyValueChangeIndexesKey</code>	Si la propiedad observada es uno a muchos, esta entrada contiene un objeto <code>NSIndexSet</code> con los índices de los elementos alterados.

Tabla 8.2: Entradas en el diccionario de cambios

El parámetro `change` contiene un diccionario con entradas donde se describe el cambio. La Tabla 8.2 describe el contenido de estas entradas.

2.3. Eliminar un objeto observador

Podemos hacer que un objeto deje de ser observador de una propiedad KVC usando el siguiente método. La categoría `NSMutableObserving` implementa la funcionalidad de este método KVO:

```
- (void)removeObserver:(NSObject*)anObserver  
forKeyPath:(NSString*)keyPath
```

Nunca debemos de permitir que la cuenta de referencias de un objeto observador llegue a cero. Antes de que esto ocurra deberemos de haber usado el método anterior para que el objeto deje de ser observador. En caso contrario se podría producir un error en tiempo de ejecución si el objeto observado intenta informar a un observador que hubiera sido eliminado de memoria.

3. Notificaciones automáticas y manuales

Existen varios mecanismos para que un objeto observado informe a sus objetos observadores de cambios en sus propiedades KVO. Estos mecanismos están implementados por la categoría `NSMutableObserving` y los vamos a estudiar en los siguientes apartados.

3.1. Cóctel de punteros `isa`

El **cóctel de punteros `isa`** (swizzling-`isa`) es un cambio introducido en Mac OS X 10.3 para implementar KVO. En el apartado 4 del Tema 4 vimos que un objeto clase es una instancia de la clase `Class` donde se almacena la dispatch table con las operaciones a ejecutar con cada mensaje que recibe un objeto. El cambio introducido en Mac OS X 10.3 consiste en que cuando un objeto observador se registra como observador de un objeto observado, cambia el puntero `isa` del objeto observado para que apunte a un objeto clase intermedio, en vez de al objeto clase real del objeto observado. Este objeto clase intermedio es capaz de detectar llamadas a los métodos de patrón de las propiedades KVC (p.e. `setKey`) y realizar en este momento la notificación.

Como consecuencia de este cambio, cuando registramos un objeto observador en un objeto observado no debemos acceder nunca directamente al puntero `isa` del objeto para obtener su objeto clase (o obtendríamos el objeto clase intermedio), sino que debemos usar el método `class`, el cual sí que nos devuelve su objeto clase correcto.

3.2. Notificaciones automáticas

Las notificaciones automáticas KVO están implementadas por la categoría `NSMutableObserving` con lo que no necesitamos cambiar nada para que las notificaciones automáticas funcionen sobre propiedades KVC. El cóctel de punteros `isa` que se instala en el objeto observado cuando registramos un objeto observador es suficiente para que el mecanismo actúe.

Las llamadas a métodos de patrón que son interceptadas por el cóctel de punteros `isa` son las llamadas al método `setter` `setKey:`. Debido a que por defecto los métodos KVC `setValue:forKey:` y `setValue:forKeyPath:` llaman a este método, las llamadas a los métodos KVC también producen la notificación. Si la propiedad es uno a muchos también se interceptan las llamadas a los métodos de patrón `insertObject:inKeyAtIndex:`, `replaceObjectInKeyAtIndex:,` `removeObjectFromKeyAtIndex:,` `addKeyObject:` y `removeKeyObject:`. Además, KVO está soportado por los objetos devueltos por los métodos `mutableArrayValueForKey:` y `mutableSetValueForKey:` que permitían escribir en propiedades uno a muchos (ver apartado 2.6 del Tema 8).

3.3. Notificaciones manuales

La notificación automática funciona siempre que accedemos a una propiedad a través de sus métodos `getter/setter`. Si para actualizar una propiedad accedemos directamente a su variable de instancia, el mecanismo de notificación de KVO no actúa.

La forma de notificar que una la variable de instancia de una propiedad KVC respectivamente va a ser modificada y ha sido modificada es ejecutar los métodos KVO:

- `(void)willChangeValueForKey: (NSString*) key`
- `(void)didChangeValueForKey: (NSString*) key`

Normalmente el cambio en la propiedad KVC se encierra entre llamadas a estos métodos de la forma:

```
- (void)incrementaSuma {
    [self willChangeValueForKey:@"suma"];
    suma = suma + 10;
    [self didChangeValueForKey:@"suma"];
}
```

Ahora los observadores de la propiedad `@“suma”` sí serán informados de que la propiedad va a cambiar.

En el caso de las propiedades KVC uno a muchos con notificación manual debemos de usar los métodos KVO:

- `(void)willChange:(NSNotificationChange)change
valuesAtIndexes:(NSIndexSet*)indexes
forKey:(NSString*)key`
- `(void)didChange:(NSNotificationChange)change
valuesAtIndexes:(NSIndexSet*)indexes
forKey:(NSString*)key`

Para indicar el tipo de cambio y los índices afectados. Por ejemplo:

```
- (void)removeClientesAtIndexes:(NSIndexSet*)indices {
    [self willChange:NSMutableValueChangeRemoval
    valuesAtIndexes:indices
    forKey:@"clientes"];

    // Eliminar clientes en posiciones dadas en indices

    [self didChange:NSMutableValueChangeRemoval
    valuesAtIndexes:indices
    forKey:@"clientes"];
}
```

3.4. Desactivar la notificación automática

Las notificaciones automáticas tienden a producir gran cantidad de notificaciones. Podemos controlar mejor el número de notificaciones que se lanzan mediante las notificaciones manuales.

El control de notificaciones se produce a nivel de propiedad KVC, de forma que podemos dejar que para algunas propiedades la notificación sea automática y para otras realizar la notificación de forma manual. Para ello podemos redefinir el siguiente método KVO de clase:

```
+ (BOOL)automaticallyNotifiesObserversForKey:(NSString*)key
```

Por defecto este método KVO está implementado por `NSMutableValueObserving` para que devuelva `YES` para todas las propiedades KVC, con lo que por defecto todas las propiedades KVC se controlan de forma automática. Podemos redefinir este método KVO para que algunas propiedades KVC se controlen de forma manual y con el resto de propiedades usar `super` para indicar su comportamiento por defecto de la forma:

```
+ (BOOL)automaticallyNotifiesObserversForKey:(NSString*)key {
    if ([key isEqualToString:@"suma"])
        return NO;
    else
```

```

        return [super automaticallyNotifiesObserversForKey:key];
    }
}

```

Una vez activada la notificación manual para la propiedad KVC `@"suma"`, el cóctel de punteros `isa` deja de actuar para esa propiedad. La forma de indicar que una propiedad KVC con control manual respectivamente va a ser modificada y ha sido modificada es ejecutando los métodos KVO `willChangeValueForKey:` y `didChangeValueForKey:`. Como indicamos en el apartado anterior, el cambio en la propiedad KVC se encierra entre llamadas a estos métodos de la forma:

```

- (void)setSuma:(NSInteger)paramSuma {
    if (paramSuma > suma) {
        [self willChangeValueForKey:@"suma"];
        suma = paramSuma;
        [self didChangeValueForKey:@"suma"];
    }
}

```

Como puede apreciar en este ejemplo, el control manual evita que se lance la notificación cuando no cambia el valor de la propiedad (a pesar de que se ejecuta el método setter).

3.5. Registrar propiedades dependientes

A veces una propiedad KVC depende del valor de otra propiedad KVC. En este caso un cambio en una propiedad KVC debe producir también una notificación en la propiedad dependiente.

Para indicar las propiedades dependientes debemos de redefinir el método KVO:

```
+ (NSSet*)keyPathsForValuesAffectingValueForKey:(NSString*)key
```

Este método se va a llamar una vez por cada propiedad que tenga la clase. En `key` se pasará el nombre de la propiedad por la que se está preguntando. Si la propiedad es dependiente debemos devolver un objeto `NSSet` con los nombres de las propiedades de las que depende. En caso contrario debemos pasar la llamada a la clase base con `super`.

Por ejemplo, el Listado 8.11 y Listado 8.12 muestran una clase `Cliente` donde la propiedad `@"nombreCompleto"` es dependiente de las propiedades KVC `@"nombre"` y `@"apellidos"`.

Obsérvese que `keyPathsForValuesAffectingValueForKey:` es un método de clase. La razón es que normalmente se llama cuando el runtime carga la clase, es decir, antes de instanciar objetos de este tipo.

```
#import <Foundation/Foundation.h>

@interface Cliente : NSObject {
    NSString* nombre;
    NSString* apellidos;
}
@property(retain) NSString* nombre;
@property(retain) NSString* apellidos;
@property(retain,readonly) NSString* nombreCompleto;
+ (NSSet*) keyPathsForValuesAffectingValueForKey: (NSString*) key;
- (id)initWithNombre: (NSString*)paramNombre
    apellidos: (NSString*)paramApellidos;
- (NSString*)nombreCompleto;
@end
```

Listado 8.11: Interfaz de la clase Cliente

```
#import "Cliente.h"

@implementation Cliente
@synthesize nombre;
@synthesize apellidos;
+ (NSSet*)
keyPathsForValuesAffectingValueForKey: (NSString*) key {
    if ([key isEqualToString:@"nombreCompleto"]) {
        return [NSSet setWithObjects:@"nombre",
                           @"apellidos",nil];
    } else {
        return [super
            keyPathsForValuesAffectingValueForKey:key];
    }
}- (id)initWithNombre: (NSString*)paramNombre
    apellidos: (NSString*)paramApellidos {
    if (self = [super init]) {
        nombre = paramNombre;
        apellidos = paramApellidos;
    }
    return self;
}
- (NSString*)nombreCompleto {
    return [NSString stringWithFormat:
           @"%@ %@", nombre,apellidos];
}
@end
```

Listado 8.12: Implementación de la clase Cliente

De esta forma, si un objeto observador se registra para recibir cambios en la propiedad KVC `@"nombreCompleto"`, los cambios que se hagan en `@"nombre"` o en `@"apellidos"` se considerarán cambios también en `@"nombreCompleto"`, y se lanzará la correspondiente notificación al objeto observador. Por ejemplo:

```
Cliente* c = [[Cliente alloc] initWithNombre:@"Fernando"
                                         apellidos:@"Lopez"];
Observador* o = [Observador new];
[c addObserver:o forKeyPath:@"nombreCompleto"
    options:NSKeyValueObservingOptionNew
    | NSKeyValueObservingOptionOld
    context:NULL];
// Esta operación lanza la notificación
c.apellidos = @"Lopez Hernandez";
[c removeObserver:o forKeyPath:@"nombreCompleto"];
[c release];
```

También es posible implementar un método de patrón (en vez de un método KVO) para registrar las propiedades dependientes. Este método de patrón deberá tener la forma `+keyPathsForValuesAffectingKey`. Es decir, en vez de implementar `keyPathsForValuesAffectingValueForKey:`, podríamos haber implementado:

```
+ (NSSet*)keyPathsForValuesAffectingNombreCompleto {
    return [NSSet setWithObjects:@"nombre",@"apellidos",nil];
}
```

Estos métodos de patrón son buscados en la implementación por defecto del método `keyPathsForValuesAffectingValueForKey:`.

Tema 10

Aprovechando toda la potencia del lenguaje

Sinopsis:

En este tema pretendemos resumir el resto de aspectos del lenguaje Objective-C no vistos hasta ahora. Entre estos aspectos están algunas técnicas avanzadas de gestión de memoria, el forwarding, los mensajes remotos, y algunos mecanismos para mejorar el rendimiento del programa.

En este tema también expondremos los estilos de codificación que Apple recomienda a los programadores Cocoa, y hablaremos sobre Objective-C++, una forma de combinar código fuente escrito en C++ y en Objective-C.

1. Directivas del preprocesador y compilador

Las directivas del preprocesador y del compilador son palabras reservadas que dicen al preprocesador o al compilador que realice una determinada acción. Todas las directivas del preprocesador comienzan por `#`, y todas las directivas del compilador comienzan por `@`.

1.1. Directivas del preprocesador

Objective-C añade al lenguaje C la directiva del preprocesador:

```
#import fichero_cabecera
```

Usar `#import` tiene el mismo efecto que usar `#include`, excepto que garantiza que el `fichero_cabecera` sólo se incluye una vez.

Además Objective-C añade el identificador del preprocesador:

```
__OBJC__
```

Este identificador está definido siempre que `gcc` compila un programa Objective-C. Este identificador resulta útil en ficheros de cabecera que puedan ser incluidos tanto por programas C como por programas Objective-C.

1.2. Directivas del compilador

La Tabla 10.1 muestra las directivas de compilador usadas para realizar la declaración de la interfaz e implementación de una clase, categoría o protocolo.

Directiva	Descripción
<code>@interface</code>	Empieza la interfaz de una clase o categoría.
<code>@protocol</code>	Empieza la interfaz de un protocolo formal.
<code>@implementation</code>	Empieza la implementación de una clase o categoría.
<code>@end</code>	Acaba la interfaz o implementación de una clase, categoría o protocolo.

Tabla 10.1: Directivas del compilador para declaración y definición de clases

La Tabla 10.2 muestra las directivas del compilador que sirven para hacer declaraciones adelantadas, es decir, para informar al compilador de que un nombre de clase o protocolo existe. En el apartado 9 del Tema 3 se explicó

cómo se hacían las declaraciones adelantadas de clases, y para qué servían. En el apartado 8.7 del Tema 4 se explicó cómo se usan las declaraciones adelantadas de protocolos.

Directiva	Descripción
<code>@class clase</code>	Indica que <code>clase</code> es un nombre de clase que todavía no se ha definido.
<code>@protocol protocolo</code>	Indica que <code>protocolo</code> es un nombre de protocolo que todavía no se ha definido.

Tabla 10.2: Directivas del compilador para declaraciones adelantadas

Directiva	Descripción
<code>@private</code>	Permite el acceso a las variables de instancia sólo desde dentro de la clase.
<code>@protected</code>	Permite el acceso a las variables de instancia sólo desde dentro de la clase, o desde las derivadas.
<code>@public</code>	Permite el acceso a las variables de instancia desde cualquier ámbito.

Tabla 10.3: Directivas para encapsulación

Directiva	Descripción
<code>@try</code>	Indica un bloque donde se puede producir una excepción.
<code>@catch()</code>	Captura la excepción, si se produce.
<code>@finally</code>	Indica sentencias a ejecutar independientemente de si se produce, o no, una excepción.
<code>@throw</code>	Permite lanzar una excepción.
<code>@synchronized</code>	Permite crear un bloque sincronizado

Tabla 10.4: Directivas del compilador para gestión de excepciones

Directiva	Descripción
<code>@property</code>	Declara un propiedad en la interfaz de una clase, categoría, extensión o protocolo.
<code>@synthesize</code>	Sintetiza los métodos getter y setter de una propiedad en la implementación de una clase, categoría o extensión.
<code>@dynamic</code>	Indica que los métodos getter y setter de la propiedad se obtienen de forma dinámica. Debe usarse esta directiva en la implementación de una clase o categoría.

Tabla 10.5: Directivas del compilador para propiedades

Directiva	Descripción
<code>@encode(tipo)</code>	Expande <code>tipo</code> por la cadena C que usa el runtime

	para representarlo.
@defs(<i>clase</i>)	Expande <i>clase</i> por la secuencia de variables de instancia que la representan en una estructura C.
@compatibility_alias	Permite definir un alias para una clase.
@protocol(<i>protocolo</i>)	Devuelve el objeto protocolo correspondiente al símbolo <i>protocolo</i> .
@selector(<i>método</i>)	Devuelve el selector del método <i>método</i> .
@" <i>cadena</i> "	Devuelve un objeto NSString con el contenido de la cadena C <i>cadena</i> .

Tabla 10.6: Directivas del compilador expandibles

La Tabla 10.3 muestra las directivas del compilador para encapsulación, que fueron ya explicadas en el apartado 5 del Tema 3.

En la Tabla 10.4 se resumen las directivas del compilador para tratamiento de excepciones y bloques sincronizados, tal como se explicó en el apartado 6 y apartado 7 del Tema 5.

En la Tabla 10.5 se resumen las directivas del compilador para propiedades, tal como se explicó en el apartado 1 del Tema 6.

En la Tabla 10.6 se resumen las directivas del compilador expandibles. Las directivas @encode(), @defs() y @compatibility_alias no han aparecido hasta ahora, y merecen una explicación más detallada.

1.2.1. La directiva del compilador @encode()

El compilador puede generar cadenas de caracteres que representan tipos de datos. Estas cadenas son usadas por el runtime al menos en dos escenarios:

- Durante la serialización de objetos (llamada **archivado** en Objective-C), donde se necesita indicar el tipo de las variables archivadas, para luego poder reconstruir el objeto.
- Para representar el tipo de los parámetros y retornos de un método. Muchas veces, además del selector, es necesario conocer que parámetros recibe/devuelve un método.

La directiva @encode() se puede aplicar a cualquier tipo de dato (simple o compuesto) C o Objective-C, y expande por una cadena de caracteres que representan ese tipo. Por ejemplo:

```
const char* tipo = @encode(int); // "i"
```

En la variable tipo obtenemos un puntero a la cadena "i", que es la forma que usa el runtime para representar un int.

La Tabla 10.7 muestra los tipos de datos fundamentales que se pueden representar usando `@encode()`. Observe que las cadenas de caracteres (`char*`) son consideradas un tipo fundamental, debido a que se usan muy frecuentemente.

La Tabla 10.8 muestra la forma de representar tipos compuestos a partir de tipos fundamentales. Para representar un puntero se precede el tipo por "`^`", por ejemplo, para representar un `int*` se usa la cadena "`^i`".

Para representar un array se encierra entre corchetes el número de elementos del array seguido del tipo de los elementos. Por ejemplo un array de 12 elementos de tipo `int*` se representa como "`[12^i]`".

Tenga en cuenta que la directiva del compilador `@encode()`, a diferencia del operador `sizeof()`, sólo puede recibir un tipo, no una variable de ese tipo, con lo que si hacemos:

```
int* A[12];
const char* tipo = @encode(A); // Error de compilacion
int tamano = sizeof(A); // Correcto 48
```

Y la forma correcta de conocer el tipo y tamaño de `A` sería:

```
int* A[12];
const char* tipo = @encode(int* [12]); // Correcto "[12^i]"
int tamano = sizeof(A); // Correcto 48
```

O bien usando el operador `__typeof__`⁴³:

```
int* A[12];
const char* tipo = @encode(__typeof__(A));
int tamano = sizeof(A);
```

Tipo	<code>@encode()</code>	Tipo	<code>@encode()</code>
<code>char</code>	<code>c</code>	<code>unsigned long long</code>	<code>Q</code>
<code>unsigned char</code>	<code>C</code>	<code>float</code>	<code>f</code>
<code>short</code>	<code>s</code>	<code>double</code>	<code>d</code>
<code>unsigned short</code>	<code>S</code>	<code>void</code>	<code>v</code>
<code>int</code>	<code>i</code>	<code>id</code>	<code>@</code>
<code>unsigned int</code>	<code>I</code>	<code>Class</code>	<code>#</code>
<code>long</code>	<code>l</code>	<code>SEL</code>	<code>:</code>
<code>unsigned long</code>	<code>L</code>	<code>char*</code>	<code>*</code>
<code>long long</code>	<code>q</code>	Desconocido	<code>?</code>

Tabla 10.7: Tipos fundamentales de `@encode()`

⁴³ `__typeof__()` es el operador introducido por ISO en C. Las GCC también permiten usar el operador no estándar de C de GNU `typeof()`.

Tipo	@encode()
<code>^tipo</code>	Puntero a variable de tipo <code>tipo</code> .
<code>[tamanotipo]</code>	Array de <code>tamano</code> elementos del tipo <code>tipo</code> .
<code>{nombre=t1t2t3}</code>	Estructura C de nombre <code>nombre</code> y con campos de tipo <code>t1, t2, t3</code> .
<code>(nombre=t1t2t3)</code>	Unión C de nombre <code>nombre</code> y con campos de tipo <code>t1, t2, t3</code> .

Tabla 10.8: Tipos compuestos de @encode()

El nombre y campos de una estructura se ponen entre llaves. Por ejemplo, la siguiente estructura `Cliente` tiene la representación "`{Cliente=i*f}`".

```
struct Cliente {
    int codigo;
    char* nombre;
    float saldo;
};

const char* tipo = @encode(struct Cliente); // "{Cliente=i*f}"
```

Las uniones son similares a las estructuras, sólo que en vez de ponerse entre llaves se ponen entre paréntesis.

En el caso de los modificadores de tipo (que explicaremos en el apartado 4.2), éstos también forman parte de la cadena cuando se está representando parámetros de un método pero, a diferencia de los tipos, no son generados por la directiva `@encode()` (ya que `@encode()` recibe tipos y no funciones), con lo que habría que ponerlos manualmente delante de su tipo. La Tabla 10.9 presenta estos modificadores. En cualquier caso estos modificadores se pueden omitir.

Modificador	@encode
<code>const</code>	<code>r</code>
<code>in</code>	<code>n</code>
<code>out</code>	<code>o</code>
<code>input</code>	<code>N</code>
<code>bycopy</code>	<code>O</code>
<code>byref</code>	<code>R</code>
<code>oneway</code>	<code>V</code>

Tabla 10.9: Modificadores de tipo @encode()

1.2.2. La directiva del compilador @defs()

Un principio fundamental de la programación orientada a objetos es que las variables de instancia son privadas al objeto, y que a la información almacenada en el objeto se accede mediante métodos getter/setter.

A veces el programador puede querer eludir este principio y acceder directamente a los datos almacenados en el objeto. En este caso Objective-C proporciona la directiva del compilador `@defs()` mediante la cual podemos convertir un objeto Objective-C en una estructura C, de forma que la estructura C expone todas las variables de instancia del objeto.

La directiva del compilador `@defs()` debe estar encerrada dentro de una declaración de estructura. Por ejemplo, si en un fichero escribimos:

```
struct PuntoDefs {
    @defs(Punto)
};

Punto* p = [[Punto alloc] init];
struct PuntoDefs* pdefs = (struct PuntoDefs*) p;
pdefs->x = 2;
```

Donde estamos usando el puntero a estructura `pdefs` para acceder a las variables de instancia del objeto `Punto`.

Tenga en cuenta que no podemos ver el resultado de expandir `@defs()` con el comando `cpp` (o con el comando `gcc -E`), ya que estos comandos ejecutan sólo el preprocesador, y `@defs()` es una directiva del compilador.

1.2.3. Directiva del compilador para crear alias

La directiva del compilador `@compatibility_alias` permite definir un nombre alternativo para una clase Objective-C. Su formato es:

```
@compatibility_alias clase_alias clase_existente;
```

Donde `clase_alias` es un nombre de clase que no debe existir, y `clase_existente` es un nombre de clase que tiene que existir. Por ejemplo, podemos crear el alias `Puntito` para la clase `Punto` así:

```
@compatibility_alias Puntito Punto;
Puntito* p = [Punto new];
```

Al ser `Puntito` y `Punto` equivalentes, podemos asignar un objeto de tipo `Punto` a un puntero a tipo `Puntito`, y viceversa.

2. Zonas de memoria

Las **zonas de memoria** son trozos de memoria dinámica que corresponden a un heap privado con un conjunto de páginas, y con su propia lista de bloques de memoria asignados.

Cuando el sistema operativo ejecuta una operación, la asigna una zona por defecto, y la aplicación puede crear luego zonas de memoria adicionales.

El uso de zonas de memoria adicionales tiene sus ventanas y desventajas, con lo que no siempre se recomienda su uso. De hecho, en la mayoría de las aplicaciones usar sólo la zona de memoria por defecto es más eficiente que crear zonas de memoria adicionales.

Debido a que cada zona tiene un heap privado con sus páginas y su lista de bloques de memoria asignados, crear zonas de memoria adicionales aumenta el consumo de memoria. Sin embargo este aumento en el consumo de memoria (y en la complejidad de la aplicación) puede ser ventajoso cuando un conjunto de objetos van a ser usados durante un mismo periodo de tiempo. En este caso, colocar los objetos contiguos en memoria reduce la paginación, y permite aprovechar mejor las cachés del procesador. Si se produce un fallo de página, todos los objetos cargados en la misma página serán cargados en memoria a la vez, lo cual ayuda a reducir la paginación.

Además, en caso de tener activada la gestión de memoria con recolector de basura, sólo podremos utilizar dos zonas: la auto zone y la zona de memoria estándar. En el apartado 2.3 del Tema 6 se explicaron estas zonas. Luego, los siguientes subapartados asumen que estamos usando la gestión de memoria por cuenta de referencias.

2.1. Creación y gestión de zonas de memoria

Las zonas de memoria están representadas por objetos del tipo opaco `NSZone`. La mayoría de las aplicaciones no necesitan crear zonas de memoria, ya que el sistema operativo crea automáticamente una zona de memoria por defecto para los objetos Objective-C de cada aplicación (y otra zona de memoria para `malloc()`). Si no se indica lo contrario, todos los objetos son creados en la zona de memoria por defecto.

Para crear una nueva zona de memoria se usa la función:

```
NSZone* NSCreateZone(NSUInteger startSize  
                      , NSUInteger granularity, BOOL canFree);
```

El parámetro `startSize` indica el tamaño inicial de la zona de memoria, `granularity` indica en qué tamaños hacer crecer la zona, y `canFree` indica

si se puede liberar la memoria asignada. Si `canFree` es 0 estamos indicando que nunca vamos a liberar memoria que reservemos en esta zona, con lo que el runtime puede usar un algoritmo de asignación de memoria dinámica más eficiente. Por ejemplo:

```
NSZone* datos = NSCreateZone(524288, 4096, YES);
```

Crea una zona de memoria de 524288 bytes, la cual crece en bloques de 4 KB. Es recomendable que creamos la zona del tamaño máximo que vayan a tener los datos en ella almacenada, ya que esto garantiza que todos los datos estén en posiciones contiguas de memoria.

Para destruir una zona de memoria disponemos de la función `malloc_destroy_zone()`, pero si mantenemos referencias a objetos de estas zonas, indireccionar estos punteros daría lugar a un fallo de memoria, con lo que esta operación debe realizarse con cuidado. Nunca debemos de liberar la zona de memoria por defecto.

2.2. Reservar y liberar memoria en una zona

Una vez creada una zona, podemos crear objetos en esa zona usando el método de clase de `NSObject`:

```
+ (id)allocWithZone:(NSZone*) zone
```

En caso de estar activado el recolector de basura el parámetro `zone` es ignorado, ya que el objeto siempre se crea en la zona de memoria por defecto (auto zone).

La implementación del método de clase `alloc` se limita a llamar a `allocWithzone:` pasando como parámetro `nil`, lo cual hace que se reserve memoria en la zona de memoria por defecto.

Podemos liberar la memoria de un objeto creado en otra zona exactamente igual que la de los objetos creados en la zona por defecto: Enviando el mensaje `release` o `autorelease` al objeto.

Además de reservar memoria para los objetos Objective-C, podemos reservar memoria para los datos en otras zonas de memoria distintas a la por defecto usando la función:

```
void *NSZoneMalloc(NSZone* zone, NSUInteger size);
```

Y liberar esta memoria con:

```
void NSZoneFree(NSZone* zone, void* pointer);
```

Podemos conocer la zona de memoria por defecto con la función:

```
NSZone* NSDefaultMallocZone(void);
```

Y la zona de memoria que ocupa cualquier objeto con el método de instancia de `NSObject`:

- `(NSZone*) zone`

O bien la zona a la que corresponde un puntero cualquiera con:

```
NSZone* NSZoneFromPointer(void* pointer);
```

3. Forwarding

Cuando el runtime envía un mensaje a un objeto, y encuentra que el objeto no tiene un método capaz de responder a este mensaje, el runtime, antes de producir una excepción, da al objeto receptor una segunda oportunidad de procesar el mensaje. Para ello el runtime envía el mensaje `forward::` al objeto. Este método por defecto está implementado de forma que produce un mensaje de log, y detiene la ejecución del programa. Nosotros podemos sobrescribir este método para absorber el error, enviárselo a un objeto delegado que gestione el mensaje, o gestionar nosotros mismos el mensaje. En este apartado veremos cómo se realiza esta tarea en cada una de las clases raíz.

3.1. Forwarding con Object

La clase `Object` proporciona el método:

- `(id) forward:(SEL) sel : (marg_list) args`

Cuando enviamos un mensaje a un objeto, y el runtime no encuentra un método para responder al mensaje, el runtime ejecuta el método `forward::` sobre este objeto. La implementación por defecto de este método es enviar a `self` el parámetro `sel`, para lo cual llama al método:

- `(id) doesNotRecognize:(SEL) aSelector`

El cual, a su vez, está implementado para pasar a `self` una cadena con una descripción del error. Para ello llama al método:

- `(id) error:(const char*) aString, ...`

Finalmente este método hace un log del error y aborta la ejecución del programa.

Nosotros podemos redefinir el método `forward::` para ignorar el mensaje, o enviárselo a otro objeto delegado. Por ejemplo, para ver si un objeto delegado puede responder a este mensaje, o de lo contrario enviarlo a la clase base (por si ésta hubiera redefinido `forward::`) podemos hacer:

```
- (id)forward:(SEL)sel :(marg_list)args {
    if ([delegado respondsToSelector: sel])
        return [delegado performv:sel :args];
    else
        return [super forward:sel :args];
}
```

3.2. Forwarding con NSObject

3.2.1. El método `forward::`

Cuando el runtime no encuentra en un objeto de la familia `NSObject` el método correspondiente, ejecuta sobre el objeto el método `forward::`, el cual lleva a cabo el siguiente procedimiento⁴⁴:

En primer lugar, el método `forward::` ejecuta sobre `self` el método:

```
- (NSMethodSignature*)
methodSignatureForSelector:(SEL)aSelector
```

Después, el método `forward::` usa el objeto `NSMethodSignature` devuelto para construir un objeto `NSInvocation`. La clase `NSInvocation` es la implementación Objective-C del patrón de diseño `command`. Este patrón de diseño encapsula un mensaje en un objeto con vistas a poder independizar al objeto que hace la petición del objeto que recibe la petición. Este patrón resulta útil en colas de mensajes, operaciones que se puedan deshacer, o para representar mensajes en sistemas de objetos distribuidos. Como detallaremos en el siguiente apartado, el objeto `NSInvocation` contiene tanto información sobre los parámetro formales del método a ejecutar (objeto `NSMethodSignature`), como sus parámetros reales (valores concretos) con los que ejecutarlo.

⁴⁴ En `NSObject` el método `forward::` es privado, con lo que, a diferencia de lo que ocurre con `Object`, no debemos de sobrescribirlo. Recuérdese que en el apartado 5 del Tema 3 explicamos que un método privado es un método que aparece en la implementación de la clase, pero no en la interfaz. En el momento de escribir este documento la documentación oficial de Apple es incompleta respecto al funcionamiento exacto del tratamiento del forwarding. En concreto, la existencia del método `forward::`, y la necesidad de redefinir el método `methodSignatureForSelector::`, tal como se explica en el apartado 3.2.3.

Por último, el método `forward::` pasa el objeto `NSInvocation` al método:

- `(void)forwardInvocation:(NSInvocation*)anInvocation`

Por defecto el método `forwardInvocation:` se limita a llamar al método:

- `(void)doesNotRecognizeSelector:(SEL)aSelector`

El cual imprime un mensaje de log, y lanza una excepción del tipo `NSInvalidArgumentException`⁴⁵ indicando que no se ha encontrado el método.

3.2.2. Parámetros formales y reales

La clase `NSMethodSignature` almacena información de introspección sobre los **parámetros formales**, es decir, el tipo y número de parámetros, así como sobre el retorno de un método.

Para obtener un objeto de este tipo normalmente se llama al método de instancia `methodSignatureForSelector:` de la clase `NSObject`.

Podemos conocer el número de parámetros que recibe un método usando el método de instancia de la clase `NSMethodSignature`:

- `(NSUInteger)numberOfArguments`

Siempre existen al menos dos parámetros, ya que los dos primeros son los parámetros implícitos `self` y `_cmd`.

Para conocer el tipo de cada parámetro se usa el método:

- `(const char*)getArgumentTypeAtIndex:(NSUInteger)index`

El método devuelve una cadena de caracteres que describe el tipo de acuerdo a la nomenclatura que usa el runtime de Objective-C para representar tipos. Esta nomenclatura la vimos en el apartado 1.2.1.

También podemos conocer el tamaño que se consume en la pila para pasar los distintos parámetros usando:

- `(NSUInteger)frameLength`

⁴⁵ Lo que se lanza es un objeto de la clase `NSEException`. `NSInvalidArgumentException` no es una clase sino una constante que se mete dentro del objeto `NSEException`.

Este tamaño depende de la arquitectura hardware para la que estemos compilando el programa.

Para conocer información sobre el retorno de un método se usan los métodos:

- `(NSUInteger)methodReturnLength`
- `(const char*)methodReturnType`

Que devuelven respectivamente el tamaño y tipo de la variable devuelta.

La clase `NSInvocation` almacena información sobre los **parámetros reales**, es decir, valores concretos a asignamos a los parámetros durante una llamada a método.

Para crear un objeto `NSInvocation` se usa el método `factory`:

- + `(NSInvocation*)invocationWithMethodSignature:(NSMethodSignature*)signature`

Antes de poder usar este objeto, debemos de fijar los argumentos con los que ejecutarlo, usando el método:

- `(void)setArgument:(void*)buffer atIndex:(NSInteger)index`

Para fijar el primer argumento resulta más cómodo usar:

- `(void)setSelector:(SEL)selector`

Y para fijar el segundo argumento, también resulta más cómodo usar:

- `(void)setTarget:(id)anObject`

Además el objeto `NSInvocation` tiene el método:

- `(void)invoke`

que permite ejecutar el método, una vez fijados sus parámetros, o el método:

- `(void)invokeWithTarget:(id)anObject`

que primero cambia el target, y luego ejecuta el método.

3.2.3. Redefinir el forwarding

Redefinir el forwarding en `NSObject` es un poco más complicado que en `Object`, debido a que ahora no debemos redefinir `forward::`, sino que de-

bemos de redefinir los métodos `methodSignatureForSelector:` y `forwardInvocation::`.

El método `methodSignatureForSelector:` debe redefinirse para que al pasarnos el selector del método se devuelva un `NSMethodSignature` correspondiente al método no encontrado por el runtime. Suponiendo que tengamos un objeto delegado responsable de capturar las llamadas podemos hacer:

```
- (NSMethodSignature*)
methodSignatureForSelector:(SEL)aSelector {
    NSMethodSignature* s = [[self class]
        instanceMethodSignatureForSelector:aSelector];
    if (!s)
        s = [delegado methodSignatureForSelector: aSelector];
    return s;
}
```

Tenga en cuenta que el método `methodSignatureForSelector:` puede ser llamado por razones distintas al procesamiento de un mensaje de `forward`, con lo que primero debemos de buscar el selector en nuestra propia clase. Para buscar el selector en nuestra clase no podemos volver a llamar al método de instancia `methodSignatureForSelector:`, ya que entraríamos en un bucle infinito, con lo que llamamos al método de clase:

```
+ (NSMethodSignature*)
instanceMethodSignatureForSelector:(SEL)aSelector
```

En caso de no encontrar el selector en el objeto clase, lo buscamos en el objeto delegado.

La información que `methodSignatureForSelector:` devuelve no tiene porque ser veraz, pero es la información que contendrá el objeto `NSInvocation` cuando se ejecute `forwardInvocation::`. Con lo que también podemos hacer una llamada comodín del tipo:

```
- (NSMethodSignature*)
methodSignatureForSelector:(SEL)aSelector {
    NSMethodSignature* s = [[self class]
        instanceMethodSignatureForSelector:aSelector];
    if (!s)
        s = [self methodSignatureForSelector: @selector(self)];
    return s;
}
```

Claramente, los parámetros del método de instancia `self`, del que estamos sacando su selector (`@selector(self)`), no tienen porque coincidir con los de cualquier método, pero lo único que debemos tener en cuenta es que `forwardInvocation:` va a recibir información imprecisa.

Si ahora queremos pasar las llamadas no reconocidas por nuestro objeto a un objeto delegado, podemos redefinir `forwardInvocation:` de la forma:

```
- (void)forwardInvocation:(NSInvocation*)invocation {
    SEL sel = [invocation selector];
    if ([delegado respondsToSelector:sel])
        [invocation invokeWithTarget:delegado];
    else
        [super forwardInvocation:invocation];
}
```

Tenga en cuenta que hay que ejecutar `invokeWithTarget:`, y no `invoke`, porque sino estaríamos ejecutando el método sobre `self`, y no sobre el objeto delegado. Como `forwardInvocation:` también puede haber sido redefinido por la clase base, en caso de no encontrar el método en `delegado`, pasamos la llamada a la clase base.

Y si lo que queremos es que un objeto ignore cualquier mensaje para el que no tenga método, podemos redefinir `forwardInvocation:` para capturar, e ignorar, la llamada de la forma:

```
- (void)forwardInvocation:(NSInvocation*)invocation {
    return;
}
```

3.3. Delegados y herencia

Como ya indicamos en el apartado 8.9 del Tema 4, los objetos delegados, en parte imitan a la herencia porque ambos añaden funcionalidad (nuevos métodos) al objeto.

Un diferencia importante entre ambas formas de reutilización está en que los métodos de introspección, como `respondsToSelector:`, `isMemberOfClass:` o `isKindOfClass:`, sólo buscan en la jerarquía de herencia, no en los delegados.

Normalmente estos métodos de introspección no devuelven información sobre el forwarding, pero siempre se pueden redefinir para que sí que la devuelvan. Tenga en cuenta que redefinir todos los métodos de introspección (incluidos por ejemplo, `methodSignatureForSelector:` y `conformsToProtocol:`) puede resultar una tarea tediosa.

3.4. Posing

El **posing** permite que una clase derivada, llamada **clase impostora**, actúe en lugar de otra clase base, llamada **clase original**. El posing es muy parecido a la categorización porque permite añadir nuevos métodos a una clase. También al igual que en las categorías, la clase impostora puede añadir y sobrescribir nuevos métodos, pero no puede añadir más variables de instancia. Las diferencias que hay entre las categorías y el posing son:

- La clase impostora puede sobrescribir métodos de la clase original, en cuyo caso el nuevo método puede acceder al método de la clase original usando `super`. Por el contrario, en las categorías (véase apartado 7.4 del Tema 4), cuando el método sobrescrito envía mensajes a `super`, estos van directamente a la clase base de la clase, y no al método sobrescrito. De hecho, como motivamos en el apartado 7.4 del Tema 4, no hay forma de enviar mensajes al método sobrescrito de la clase desde el método de la categoría.
- Las clases y las categorías usan namespaces distintos, por el contrario la clase impostora y original usan el mismo namespace, con lo que sus nombres no pueden coincidir.

Para hacer posing, es decir, para que una clase impostora sustituya a otra clase original se utiliza el método de clase:

```
+ (void)poseAsClass:(Class)aClass
```

Este método se ejecuta sobre la clase impostora, y recibe como parámetro la clase original.

Para que el posing tenga efecto, el método `poseAsClass:` tiene que ejecutarse sobre la clase impostora antes de empezarse a usar la clase original.

```
/* PuntoGeometrico.h */
#import "Punto.h"

@interface PuntoGeometrico : Punto {
}
- (double)distancia:(Punto*)p;
@end
```

Listado 10.1: Interfaz de una clase impostora

```
/* PuntoGeometrico.m */
#import "PuntoGeometrico.h"

@implementation PuntoGeometrico
- (double)distancia:(Punto*)p {
    double x1 = x;
    double y1 = y;
    double x2 = p->x;
    double y2 = p->y;
    return sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
}
```

```
}
```

```
@end
```

Listado 10.2: Implementación de una clase impostora

Por ejemplo, el Listado 10.1 y Listado 10.2 muestran un ejemplo de una clase `PuntoGeometrico` que añade el método `distanzia:` a la clase `Punto`. Cuando en el siguiente programa ejecutemos el método `distanzia:` sobre la clase `Punto`, éste será encontrado ya que se está usando `PuntoGeometrico` para resolver las llamadas a métodos.

```
[PuntoGeometrico poseAsClass: [Punto class]];
Punto* p = [Punto new];
Punto* p2 = [Punto new];
[p distanza: p2];
```

Como la clase impostora deriva de la original, todos los métodos de la clase original son heredados por la impostora de su clase original (que a su vez es una de sus clases base), la funcionalidad heredada no se pierde cuando la clase impostora hace posing.

En general se recomienda usar categorías en vez de posing, ya que las categorías son más fáciles de entender y tienen una funcionalidad muy parecida. Sólo en caso de que un método sobrescrito quiera acceder al método de la clase original, usando `super`, tiene sentido usar posing.

4. Mensajes remotos

4.1. Programación distribuida

La **programación distribuida** permite dividir una tarea costosa en tareas que se ejecutan en máquinas distintas. Por ejemplo, una aplicación se puede dividir en un front-end gráfico, y en un back-end de computación. De esta forma el back-end puede estar realizando largas operaciones sin que el front-end se quede bloqueado a la espera de que la operación se realice. Lógicamente la programación distribuida sólo tiene sentido cuando la tarea a realizar es costosa, si no resulta más sencillo hacer todos los cálculos en la misma máquina.

La programación distribuida también permite distribuir un cálculo costoso entre varias máquinas, de forma que cada una de ellas realiza parte del cálculo.

Cuando la aplicación está modelada con programación orientada a objetos, la programación distribuida se basa en el uso de **objetos distribuidos**, es decir, objetos situados en distintas máquinas que pueden enviarse mensajes entre sí.

Cocoa incluye la posibilidad de crear objetos distribuidos, para lo cual el lenguaje Objective-C incluye un conjunto de extensiones que ayudan al envío de **mensajes remotos**, es decir, enviar mensajes a objetos situados en otros procesos en la misma o en distinta máquina.

Para poder enviar mensajes remotos se han propuesto varias extensiones, tanto al lenguaje como al runtime de Objective-C, que vamos a describir ahora⁴⁶. Gracias a que los objetos distribuidos Cocoa se montan encima del lenguaje, los objetos distribuidos van a poder ser vistos como si fueran objetos locales. De hecho, vamos a conseguir que una vez que tengamos un puntero a un objeto remoto, éste sea indistinguible de un puntero a un objeto local.

La Figura 10.1 muestra el proceso general de envío de un mensaje remoto. Para enviar un mensaje remoto, el proceso local necesita primero establecer una conexión con el objeto remoto (el objeto B en nuestro ejemplo). Cuando se establece esta conexión en el proceso local se crea un proxy, que asume la identidad del objeto remoto. Éste es el objeto al que realmente mantenemos un puntero a objeto, y al que el runtime envía mensajes. Cuando el proxy recibe un mensaje, hace forwarding del mensaje hasta el objeto remoto a través de la conexión que tiene con él. El objeto remoto ejecuta el método correspondiente al mensaje y acaba devolviendo una respuesta al proxy, que a su vez retorna a quién lo llamó (el objeto A en nuestro ejemplo).

Las colas aparecen dibujadas para reflejar el hecho de que los mensajes (tanto de ida como de vuelta) pueden no ser atendidos inmediatamente, sino que si un objeto está ocupado ejecutando otras operaciones, el mensaje esperará en la cola a poder ser atendido.

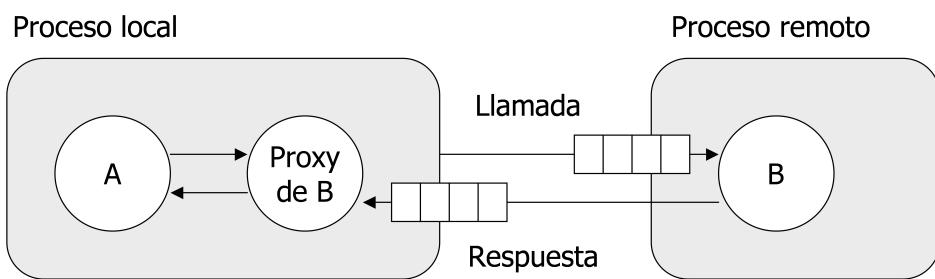


Figura 10.1: Envío de mensajes remotos

Las operaciones que soporta el objeto remoto se definen en un protocolo formal, el cual adopta tanto el proxy como el objeto remoto. Sin embargo la forma de implementar este protocolo es distinta: El proxy lo implementa para

⁴⁶ En este tutorial se introducen los conceptos de programación distribuida relativos al lenguaje. Para completar la comprensión de este tema el lector deberá buscar más información en el tutorial "Programación con Foundation Framework" también publicado en MacProgramadores.

transmitir la petición al objeto remoto, y el objeto remoto lo implementa resolviendo el cálculo correspondiente al método. Además, normalmente en el proceso local no es necesario tener la clase del objeto B, basta con tener el protocolo formal que implementa B.

Para representar el objeto proxy se crea una derivada de la clase `NSProxy`, y para representar la conexión se crea una instancia de `NSConnection`. Ambas clases están documentadas en el Foundation Framework, y no las vamos a estudiar en este tutorial. Para obtener más información sobre estas y otras clases puede dirigirse al tutorial "Programación Cocoa con Foundation Framework" publicado también en MacProgramadores.

4.2. Modificadores de tipo

Para ayudar a optimizar los mensajes remotos, el lenguaje Objective-C proporciona una serie de modificadores de tipo que se resumen en la Tabla 10.10.

Los modificadores de tipo van delante de un tipo C u Objective-C, y modifican algún aspecto del comportamiento de las variables de este tipo.

El modificador `const` se puede usar tanto al declarar variables como parámetros de funciones. Su significado es el mismo que en C. Indica que el valor de la variable (fundamental o puntero) se puede leer, pero no escribir.

Los modificadores `oneway`, `in`, `out`, `inout`, `bycopy` y `byref` están destinados exclusivamente a la creación de objetos remotos, y sólo se pueden usar en los parámetros de los protocolos formales (no se pueden usar en clases ni en categorías). Sin embargo, si se usan en un protocolo formal, la clase o categoría que implementa ese protocolo formal sí que los puede usar. Las siguientes secciones explican cómo usar estos modificadores.

Modificador	Descripción
<code>const</code>	Indica que la variable o parámetro se puede leer, pero no modificar.
<code>oneway</code>	Indica que es un método asíncrono, es decir, que no debemos de esperar respuesta. Usado sólo en objetos remotos.
<code>in</code>	Indica que es un parámetro de entrada que la implementación del método puede leer, pero no escribir. Usado sólo en objetos remotos.
<code>out</code>	Indica que es un parámetro de salida que la implementación del método puede escribir, pero no leer. Usado sólo en objetos remotos.
<code>inout</code>	Indica que es un parámetro de entrada y salida que la implementación del método puede leer y escribir. Usado sólo en objetos remotos.

bycopy	Indica que se pasa una copia del parámetro. Usado sólo en objetos remotos.
byref	Indica que se crea un proxy al parámetro. Usado sólo en objetos remotos.

Tabla 10.10: Modificadores de tipo de Objective-C

4.3. Mensajes síncronos y asíncronos

Como muestra la Figura 10.1, cuando se envía un mensaje a un objeto remoto, el hilo del proceso local tiene que esperar a que el objeto remoto reciba el mensaje, lo procese, y devuelva el resultado. Si el proceso local es una interfaz gráfica de usuario, la aplicación gráfica dejaría de responder durante el tiempo que tarda en procesarse el mensaje. Para evitarlo conviene que el mensaje no se envíe desde el hilo de gestión de eventos de interfaz gráfica de la aplicación, sino que se envíe desde otro hilo secundario.

Decimos que un **mensaje es síncrono** cuando el hilo espera al retorno del mensaje para seguir ejecutándose. En ocasiones podría no interesarnos el retorno de un mensaje remoto, en este caso el hilo podría no esperar al retorno, y seguir ejecutando operaciones. Este tipo de mensajes son los llamados **mensajes asíncronos**.

Por extensión se usan los términos **método síncrono** y **método asíncrono** para referirse a la implementación de las operaciones. Es importante no confundir el término *método sincronizado*, que en el apartado 7 usamos para referirnos a métodos cuya implementación contenía un bloque sincronizado con el término *método síncrono*, que se usa para indicar que el método no retorna hasta que se ha acabado de ejecutar la operación.

Para indicar que un mensaje es asíncrono, se le pone al retorno del método el modificador de tipo `oneway`. El único tipo Objective-C que soporta el modificador `oneway` es el tipo `void`, debido a que es el único caso en el que no hay retorno, y podemos ahorrarnos esperar a que acabe el método. Por ejemplo, el siguiente método está declarado como asíncrono:

```
- (oneway void)guardaDatos;
```

Cuando ejecutemos el método `guardaDatos`, el hilo no esperará a obtener notificación de que la operación se ha completado.

4.4. Paso de punteros

Cuando los parámetros y retorno de los métodos son tipos fundamentales, éstos se pasan por valor desde el proceso local al remoto, o viceversa. Sin

embargo, cuando un parámetro es un puntero, su paso es más complicado que copiar la dirección de memoria del puntero, ya que una dirección de memoria del proceso local no tiene sentido indireccionarla cuando estamos en el proceso remoto.

En el caso de los parámetros de tipo puntero, lo que hace Objective-C es pasar la estructura de datos apuntada por el objeto durante la llamada, y retornar esta estructura de datos (posiblemente modificada) durante el retorno.

Luego, si tenemos una estructura de datos de la forma:

```
struct DatosCliente {  
    int codigo;  
    char nombre[255];  
};
```

e implementamos en el objeto remoto un método como:

```
- (void)setDatosCliente:(struct DatosCliente*)unCliente {  
    codigo = unCliente->codigo;  
    strcpy(nombre,unCliente->nombre);  
}
```

al ser llamado desde el proceso local:

```
struct DatosCliente datos;  
[c setDatosCliente: &datos];
```

El sistema de paso de mensajes remotos lleva a cabo las siguientes operaciones:

1. Copia la estructura de datos apuntada por el puntero desde el proceso local al proceso remoto.
2. Ejecuta el método remoto.
3. Copia la estructura de datos apuntada por el puntero desde el proceso remoto al proceso local.

Observe que al ser éste un método setter, nos podríamos haber ahorrado copiar la estructura de datos desde el proceso remoto al proceso local.

Análogamente, si implementamos un método getter:

```
- (void)getDataCliente:(struct DatosCliente*)unCliente {  
    unCliente->codigo = codigo;  
    strcpy(unCliente->nombre,nombre);  
}
```

De nuevo, el sistema de paso de mensajes remotos lleva a cabo las tres operaciones:

1. Copia la estructura de datos apuntada por el puntero desde el proceso local al proceso remoto.
2. Ejecuta el método remoto.
3. Copia la estructura de datos apuntada por el puntero desde el proceso remoto al proceso local.

Cuando el primer paso ahora no hubiera sido necesario.

La implementación por defecto del sistema de pasos de mensajes es la más segura, pero también es ineficiente. Si conocemos que un método sólo lee, o sólo escribe un parámetro pasado como puntero, podemos poner al parámetro el modificador `in` o `out` respectivamente, para mejorar el rendimiento del sistema de paso de mensajes.

Por ejemplo, para indicar que el método `setDatosCliente:` sólo lee su parámetro, podríamos haberlo declarado como:

```
- (void)setDatosCliente:(in struct DatosCliente*)unCliente
```

Y para indicar que el método `getDatosCliente:` sólo escribe en su parámetro, podríamos haber declarado el método como:

```
- (void)getDatosCliente:(out struct DatosCliente*)unCliente
```

El modificador `inout` es el modificador por defecto para todos los punteros cuando no se indica modificador, a no ser que el parámetro tenga el modificador `const`, en cuyo caso `in` es el modificador por defecto del puntero.

Los modificadores `out` e `inout` sólo tienen sentido en los parámetros de tipo puntero. Si un parámetro es un tipo fundamental, su modificador siempre es `in`. Es decir, sólo se copia desde el proceso local al proceso remoto.

Un caso particular son las cadenas de caracteres C, es decir, los parámetros de tipo `char*`. Aunque el tipo indica que es un puntero, en este caso el parámetro siempre se pasa de tipo `in`, independientemente del modificador que le pongamos.

En caso de querer retornar una cadena por referencia necesitaremos añadir un nivel de indirección más de la forma:

```
- (void)getNombre:(out char**)unNombre {  
    strcpy(*unNombre, nombre);  
}
```

Otro caso particular es el retorno de métodos que devuelven punteros. En este caso el método siempre retorna con el tipo `out`, es decir, sólo se copian los datos apuntados por el puntero desde el proceso remoto al local.

4.5. Paso de objetos

Cuando un método tiene punteros a objetos en sus parámetros o retorno (independientemente de que estén tipificados estática o dinámicamente), por defecto no se copia el contenido del objeto (como pasa en las estructuras), sino que se crea un proxy al objeto en el otro extremo, y luego el otro extremo usa el proxy para acceder al objeto remotamente. Este comportamiento viene dado por el modificador `byref`, que es el modificador por defecto para los objetos cuando no se indica modificador.

De nuevo, crear un proxy por cada parámetro o retorno de tipo puntero a objeto es la solución más general, pero en ocasiones resulta más eficiente pasar una copia del objeto que crear un proxy. En este caso podemos usar el modificador `bycopy`. Por ejemplo:

```
- (double)distancia: (bycopy Punto*) p
```

Ahora no se creará un proxy al parámetro en el proceso remoto, sino que se pasará el contenido del objeto `p`.

En el apartado 4.1 dijimos que normalmente en el proceso local no era necesario tener la clase del objeto remoto, sino que bastaba con tener el protocolo formal que implementaba. En caso de pasar los parámetros con `bycopy` sí que se vuelve necesario que la clase pasada esté en ambos procesos.

También, en caso de usar el modificador `bycopy`, podemos optimizar el envío de mensajes remotos con los modificadores `in` y `out`. Por ejemplo, en el método `distancia:` anterior sólo necesitamos pasar el parámetro `p` desde el proceso local al remoto, con lo que sería más eficiente haberlo declarado como:

```
- (double)distancia: (in bycopy Punto*) p
```

5. Tipos de datos y constantes predefinidas

Objective-C extiende el lenguaje C mediante una serie de tipos de datos y constantes predefinidas que vamos a resumir en este apartado.

5.1. Tipos de datos predefinidos

Objective-C añade al lenguaje C los siguientes tipos de datos predefinidos:

`id`

Este es el tipo de los punteros a objeto dinámicos, los cuales permiten enviar cualquier mensaje a un objeto sin que el compilador compruebe que el método exista en el objeto, sino que es en tiempo de ejecución cuando se determina si el método existe, y en caso de no existir se produce una excepción.

Este tipo es parecido al puntero genérico de C `void*`, pero a diferencia de éste, sólo sirve para apuntar a objetos, y permite ejecutar cualquier método sobre el objeto, mientras que `void*` no permite ejecutar ningún método.

Realmente `id` aparece en los ficheros de cabecera de Objective-C declarado como un `typedef` a un puntero a estructura con un único campo: La variable de instancia `isa`, la cual apunta al objeto clase (véase Listado 4.2 del Tema 4).

Además este es el tipo devuelto por los métodos que no declaran su tipo de retorno.

`Class`

Puntero a un objeto clase. Para obtener el objeto clase usamos el método de instancia `class` de la clase raíz. El tipo `Class` aparece definido en las cabeceras de Objective-C como un `typedef` a una estructura con información de clase tal como muestra el Listado 4.2 del Tema 4.

`Protocol`

Es una clase Objective-C usada para referirse a los objetos protocolo. Los objetos protocolo se obtienen pasando a la directiva del compilador `@protocol()` el nombre del protocolo.

`BOOL`

Tipo lógico cuya variable puede tomar dos valores: `YES` y `NO`. Realmente está declarada en las cabeceras de Objective-C como un `typedef` del tipo `char`.

SEL

Identificador único para un selector. Normalmente implementado como un puntero `const char*` al nombre de método, pero debemos tratarlo como un tipo opaco. Para obtener un selector pasamos a la directiva del compilador `@selector()` el nombre del método.

IMP

Puntero a la implementación de un método. En el apartado 6 veremos cómo se usa este tipo para optimizar los accesos a métodos.

5.2. Constantes predefinidas

Objective-C añade al lenguaje C las siguientes constantes predefinidas.

`nil`

Constante usada para representar un objeto `id` sin inicializar. En las cabeceras de Objective-C está declarada como un identificador del preprocesador con el valor 0.

`Nil`

Constante usada para representar un objeto `Class` sin inicializar. En las cabeceras de Objective-C está declarada como un identificador del preprocesador con el valor 0.

En las cabeceras de Objective-C, tanto `nil` como `Nil` y `NULL` aparecen como identificadores del preprocesador con el valor 0. La diferencia entre estos símbolos es semántica, ya que `nil` se usa para punteros de tipo `id`, `Nil` para punteros de tipo `Class` y `NULL` para el resto de punteros.

`NO/YES`

Valor booleano interpretado como falso y cierto respectivamente para el tipo `BOOL`. En las cabeceras de Objective-C aparecen como un identificador del preprocesador con el valor `(BOOL) 0` y `(BOOL) 1`, respectivamente.

6. Optimización del acceso a métodos

Sabemos que Objective-C, a diferencia de C++ o Java, siempre realiza llamadas dinámicas a métodos, es decir, dado un mensaje (selector) decide el método a ejecutar en tiempo de ejecución. Aunque el aumento en coste de resolver las llamadas a métodos dinámicamente es casi siempre despreciable, en el caso de que un método lo deseemos ejecutar muchas veces (p.e. dentro de un bucle), podría ser recomendable acceder directamente al método a ejecutar.

A continuación vamos a ver cómo obtener la dirección de memoria de un método. Esta operación debe realizarse en tiempo de ejecución, y usa el mismo procedimiento que el runtime para encontrar el método a ejecutar como respuesta a un mensaje. En Objective-C esta técnica es la única forma de evitar que actúe el enlace dinámico en la llamada a un método.

Para realizar esta operación la clase `Object` proporciona los métodos de instancia y de clase:

```
+ (IMP)instanceMethodFor:(SEL)aSelector;
- (IMP)methodFor:(SEL)aSelector;
```

La clase `NSObject` proporciona otros dos métodos parecidos:

```
+ (IMP)instanceMethodForSelector:(SEL)aSelector
- (IMP)methodForSelector:(SEL)aSelector
```

En ambos casos hay que pasar como parámetro el selector del método que deseamos conocer su dirección de memoria. El método de instancia se ejecutará sobre un objeto ya existente, mientras que el método de clase lo podemos ejecutar sin necesidad de haber instanciado ningún objeto.

Estos métodos siempre devuelven una variable de tipo `IMP`, que está declarada como un `typedef` de la forma:

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

Donde los dos primeros parámetros son los parámetros implícitos `self` y `_cmd`, y el resto de parámetros es variable.

Si por ejemplo, queremos obtener un puntero a la implementación del método `distancia:` del Listado 10.1, debemos declarar un puntero a función de la forma:

```
typedef double (*FNDistancia)(id, SEL, Punto*);
```

Ahora podemos obtener un puntero al método:

```
SEL sel = @selector(distancia:);  
FNDistancia pDistancia = (FNDistancia)  
    [PuntoGeometrico instanceMethodForSelector: sel];
```

El casting es necesario para evitar el warning que se produce debido a que los parámetros del `typedef IMP` y del `typedef FNDistancia` no coinciden.

Una vez obtenido el puntero a la función del método, y suponiendo que `pg` es un objeto de tipo `PuntoGeometrico`, y `p` un objeto de tipo `Punto`, podemos ejecutar el método indireccinando el puntero a función de la forma:

```
PuntoGeometrico* pg = ....  
Punto* p = ....  
double d = pDistancia(pg,sel,p);
```

7. Estilos de codificación

Apple ha propuesto una serie de estilos o recomendaciones respecto a la forma de nombrar los símbolos del programa. Estos estilos no son reglas obligatorias para que el programa compile correctamente, pero sí que es recomendable acostumbrarse a seguirlos, ya que el código que generemos será más homogéneo con el que generen los demás programadores. En este apartado vamos a ver cuáles son estos estilos.

En general las reglas de estilo desaconsejan usar abreviaturas para los nombres de los símbolos, se prefiere usar un nombre de símbolo más largo, pero mejor autodocumentado. Las abreviaturas se toleran cuando es una abreviatura bien conocida: `max`, `info`, `msg`, `app`, o cuando es un acrónimo, en cuyo caso se recomienda usar mayúsculas: `TIFF`, `JPG`.

Otra recomendación que no hace el Apple, pero que hacemos nosotros, y que hemos seguido en este tutorial, es que los programadores que escriban su programa en un idioma distinto al inglés, por ejemplo en castellano, usen el castellano para sus propios símbolos, y dejen el inglés para los símbolos de las librerías que usemos. De esta forma es fácil saber si un símbolo es nuestro o de la librería. Por ejemplo, si una clase tiene símbolos en inglés y castellano, sabremos que hemos heredado de una clase de librería a la que hemos añadido nuestros propios métodos.

7.1. Clases, categorías y protocolos formales

Los nombres de las clases, categorías y protocolos empiezan por mayúscula, y si existen varias palabras se pone otra mayúscula por cada palabra. Por ejemplo `Punto`, `PuntoGeometrico`, `CuentaCliente`.

Se recomienda que por cada clase se cree un fichero de cabecera con el mismo nombre de la clase (incluidas mayúsculas y minúsculas), y otro de implementación que también tenga el mismo nombre que la clase. En el caso de las clases privadas, es decir, clases que sólo van a ser usadas desde otra clase (habitualmente para instanciar objetos agregados), se puede introducir la clase privada en los ficheros de cabecera e implementación de la clase que la usa.

La declaración de la interfaz de una categoría se pueden crear, o bien en el mismo fichero donde se declara la interfaz de la clase, o bien en un fichero aparte. En cualquier caso, la declaración de la interfaz de la clase debe ser conocida por el compilador cuando éste encuentra la categoría. En el caso de que la interfaz de la categoría se declare en un fichero aparte, lo recomendable es importar el fichero con la interfaz de la clase antes de declarar la interfaz de la categoría.

Los nombres de las categorías y protocolos formales se guardan en un namespace distinto al de las clases, con lo que una categoría puede tener el mismo nombre que una clase. Aunque no es común con las categorías, sí que a veces se recomienda que un protocolo formal tenga el mismo nombre que la clase. La regla general es:

- Si el protocolo formal va a ser implementado por varias clases se recomienda un nombre distinto.
- Si el protocolo formal agrupa un subconjunto de operaciones de una clase, se recomienda usar el mismo nombre. Por ejemplo, la clase `NSObject` adopta el protocolo formal `NSObject`.

7.2. Prefijos

Tanto en C++ como en Java existen namespaces que ayudan a evitar conflictos de nombres. Por el contrario en Objective-C no existen namespaces, con lo que los conflictos de nombres entre símbolos globales son mucho más frecuentes. Para paliar este efecto Apple recomienda anteponer a los símbolos globales dos o tres letras mayúsculas delante de su nombre. Por ejemplo, la mayoría de las clases de Cocoa empiezan por el prefijo `NS` (de NeXTSTEP). La Tabla 10.11 resume los prefijos usados por Apple en sus librerías de clases Cocoa. En MacProgramadores nos gusta usar el prefijo `MP`.

Prefijo	Cocoa Framework
NS	Foundation framework
NS	Application Kit Framework
AB	Address Book
IB	Interface Builder

Tabla 10.11: Prefijos usados por Apple para Cocoa

Otros símbolos globales, como constantes y funciones, también deberían de usar prefijos. Pero se recomienda no usar prefijos en símbolos locales como puedan ser variables de instancia, métodos, o variables locales.

Una forma alternativa a los prefijos podría ser el uso del castellano para nuestros símbolos.

7.3. Métodos y variables de instancia

Para los métodos y variables de instancia se recomienda usar una minúscula al principio de la primera palabra, y una mayúscula por cada nueva palabra. Por ejemplo `controlador`, `initWithColor:`, `actualizarFormulario`. La excepción son los símbolos que tienen una abreviatura: `ficheroJPG`.

En las variables de instancia, el plural se usa para referirse a una variable de instancia que agrupa varios objetos. Por ejemplo `cuentaClientes`.

Los métodos getter no empiezan por `get`, a no ser que reciban como parámetro un buffer donde depositar los datos. Por ejemplo:

- `(void) getBuffer:(BYTE*) unBuffer`
- `(BYTE*) buffer`

Otra diferencia es que el primero (el que no empieza por `get`) debe devolver un buffer que libera el objeto que lo devuelve, con lo que si el receptor quiere conservar los datos debe hacer una copia. Por el contrario el segundo devuelve los datos en un buffer que crea y destruye el receptor. Los métodos que leen una propiedad booleana no empiezan por `get`, sino por `es` o `esta` (`is` en inglés). Por ejemplo `esVisible`, `estaCerrado`.

Los métodos setter sí que preceden su nombre por `set`. Por ejemplo `setSaldo:`, `setCabeceraMPG:`.

Si el método representa una acción se recomienda empezar el nombre con un verbo en infinitivo: `calcularSaldo`, `cerrarVentana`.

7.4. Funciones, variables globales y constantes

Los nombres de las funciones, variables globales y constantes se escriben igual que los de los métodos, pero con dos excepciones:

- Empiezan por un prefijo.
- La primera letra después del prefijo se pone en mayúsculas.

Por ejemplo, una función podría llamarse `MPSumarCuentas()`, una variable global `MPIncializado` y una constante `MPOpcionImprimir`.

En el caso de los identificadores del preprocesador, igual que en C y C++, se suelen escribir en mayúsculas y separados por guiones bajos: `MAX_CLIENTES`.

7.5. Variables locales

Apple no especifica la forma de nombrar las variables locales, ya que éstas nunca van a ser usadas por otros programadores que quieran llamar a nuestro software. Sin embargo, nosotros le aconsejamos seguir la recomendación Java de usar sólo minúsculas para las variables locales, y separar las palabras por guiones bajos. Por ejemplo `mejor_cliente`, `n_iteraciones`.

8. Objective-C++

Sabemos que Objective-C permite combinar código C y Objective-C. A partir de las GCC 2.95 es posible combinar código C++ con código Objective-C en el mismo fichero. A esta combinación de lenguajes se la llamó **Objective-C++** y permitió acceder desde Objective-C a multitud de librerías de clases escritas en C++. Y viceversa, también es posible acceder desde C++ a librerías de clases escritas en Objective-C.

Los ficheros escritos en Objective-C++ se diferencian por tener la extensión `.mm`, aunque por razones históricas también pueden tener la extensión `.M`. Una vez que el compilador recibe un fichero con esta extensión, sabe que dentro puede encontrar tanto código C, como código C++ y código Objective-C.

Cuando se integran librerías de clases escritas en ambos lenguajes lo más recomendable es intentar reducir el acoplamiento, ya que existen ciertas restricciones a la hora de combinar ambos lenguajes que vamos a describir más abajo.

Es importante tener en cuenta que al crear un fichero Objective-C++ no estamos añadiendo a C++ características de Objective-C, ni viceversa. Por

ejemplo, dentro de un método Objective-C no podemos usar la palabra reservada `this` (sino que debemos usar el receptor especial `self`), y viceversa. De igual forma, no podemos usar la sintaxis de llamada a métodos de C++ (los paréntesis) para llamar a métodos Objective-C, y viceversa (no podemos usar sintaxis de corchetes para llamar a métodos C++).

A continuación se enumeran las formas en que se pueden combinar ambos lenguajes:

- Siempre es posible instanciar objetos y ejecutar métodos de un lenguaje desde métodos del otro lenguaje.
- Un objeto escrito en un lenguaje siempre puede tener variables de instancia puntero a objetos del otro lenguaje.

Las principales restricciones que existen a la hora de combinar ambos lenguajes son:

- Una clase hecha en un lenguaje no puede derivar de otra clase hecha en el otro lenguaje.
- Las excepciones lanzadas en un lenguaje no pueden ser capturadas por el otro lenguaje.
- Objective-C no tiene namespaces, y no puede usar los `namespace` de C++.
- Objective-C no tiene templates, y no puede usar `template` de C++.
- C++ no tiene categorías y no puede usar las categorías de Objective-C.
- No podemos usar el tipo `id` para referirnos a objetos C++.

Cuando estamos dentro de un fichero Objective-C++ estarán definidos tanto el identificador del preprocesador `__cplusplus` como el identificador del preprocesador `__OBJC__`.