

Objective-C

Curso Introductorio

iOS Development Lab

Instructores



Jesús



Diego



Aldo



iOS Development Lab

Objetivo

El alumno aprenderá los conceptos y habilidades necesarias para utilizar Objective-C en complemento con Swift para el desarrollo de aplicaciones en iOS.

Conocimiento previo

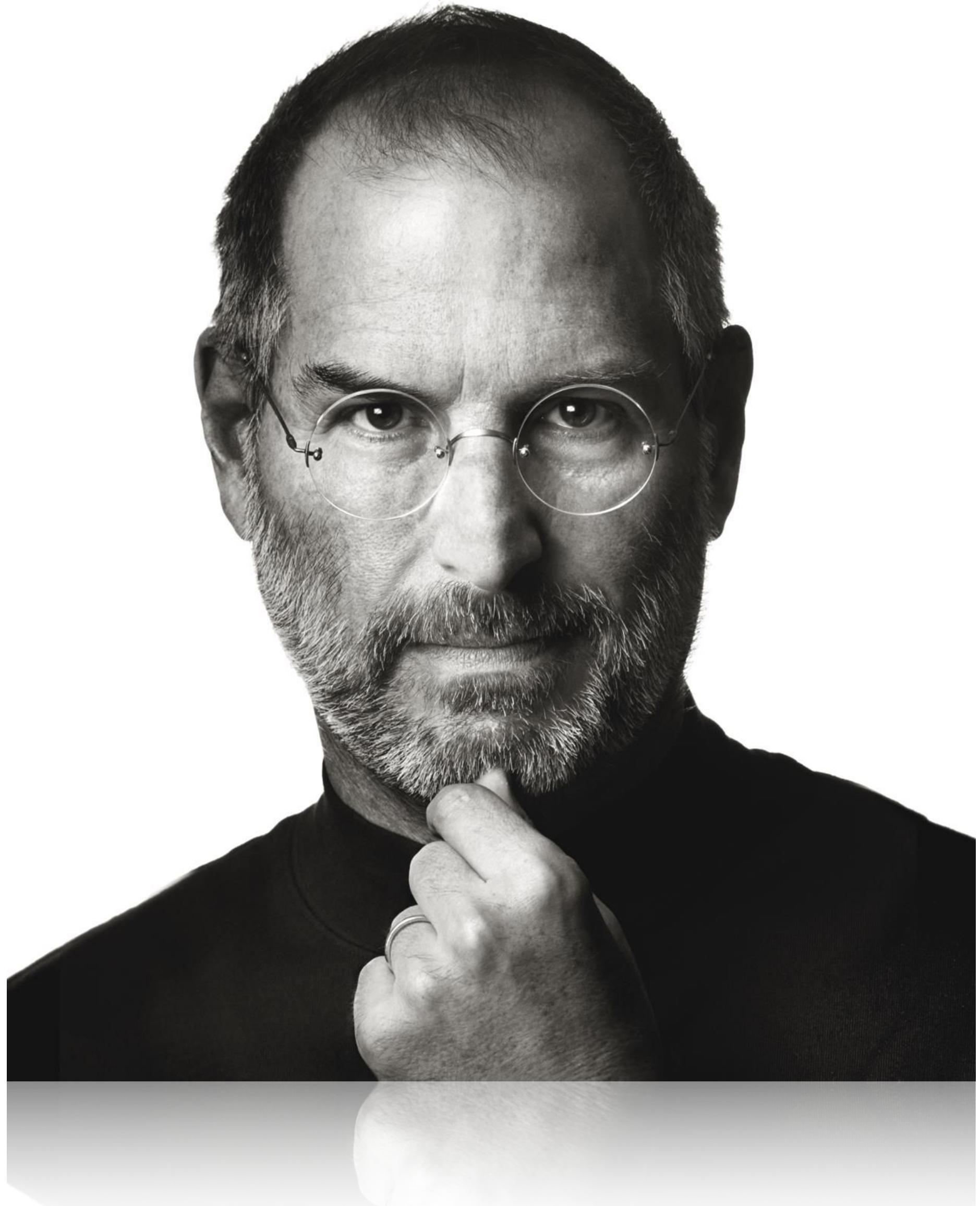
- * Programación con el lenguaje C
- * Programación Orientada a Objetos
- * Motivación

Introducción

iOS Development Lab

Breve Historia

- * Objective-C es un lenguaje de programación orientado a objetos creado como un superconjunto de ANSI C.
- * A finales de los años 80's Objective-C fue elegido como el lenguaje principal de desarrollo por la startup de Steve Jobs, NeXT para su sistema operativo NeXTStep.
- * Cuando Steve Jobs regresó a Apple compró NeXTStep, y conservó el lenguaje Objective-C como lenguaje principal para el desarrollo de aplicaciones.
- * Aún con la aparición de Swift en 2014, la mayoría del sistema iOS de Apple sigue estando escrito en Objective-C.
- * Su origen fue en una pequeña empresa de Connecticut a principios de la década de 1980 llamada StepStone.



Objective C en la actualidad

∞ Meta

Slack

Snapchat

14 cc

7.



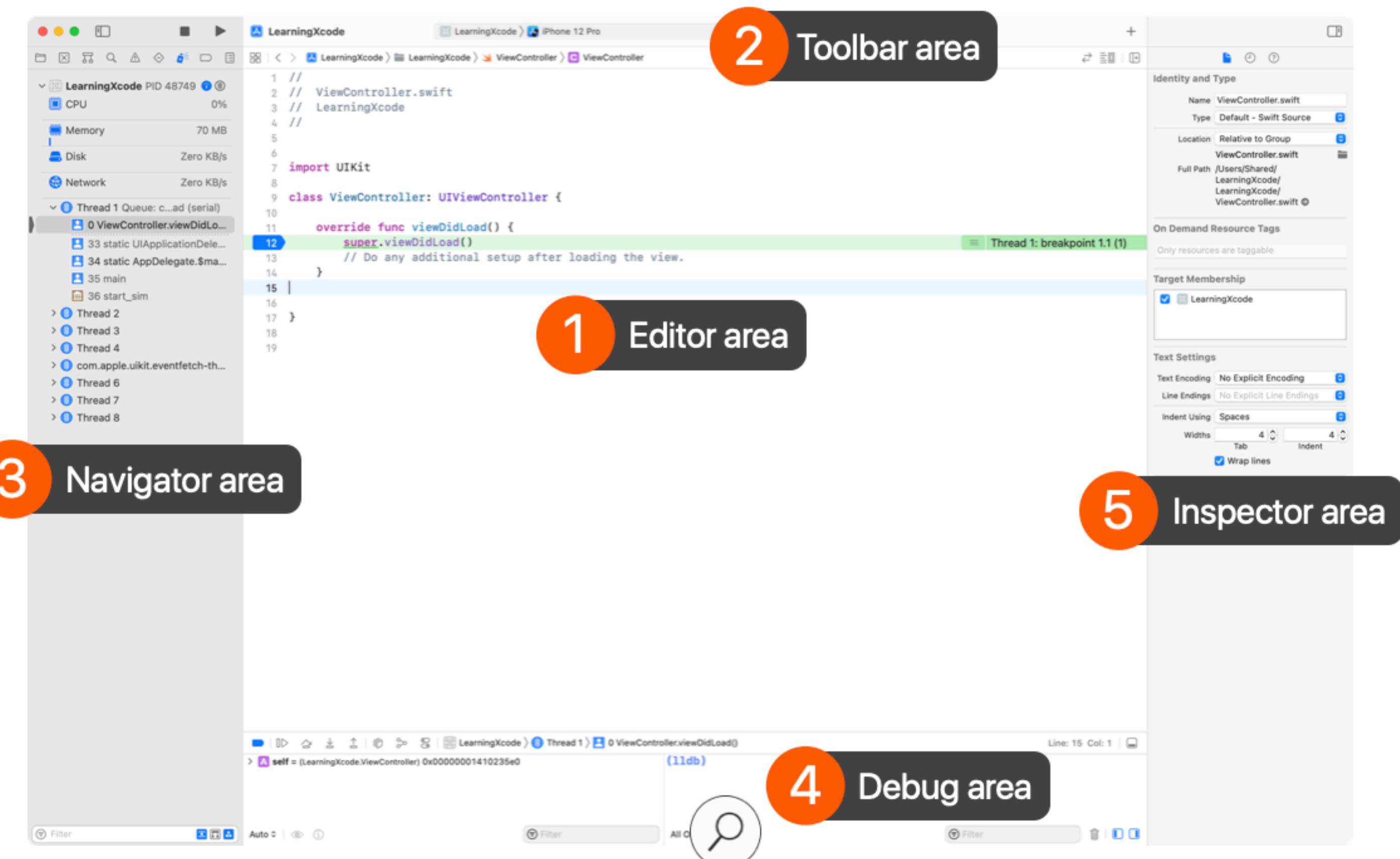
Conociendo Xcode

iOS Development Lab

Conociendo Xcode

Creando un nuevo proyecto de Obj-C

1. Abre Xcode.
2. Selecciona File -> New Project.
3. Selecciona MacOS.
4. Selecciona Command Line Tool.
Presiona Next.
5. Entra el Project Name,
Organization Name and Company
Identifier y presiona Next.
6. Selecciona el location a guardar y
selecciona Create.



Generalidades del lenguaje

iOS Development Lab

Frameworks

Los frameworks son bibliotecas de clases que puedes ocupar en tus aplicaciones.

Las aplicaciones de iOS están escritas en el lenguaje Objective-C (o Swift) utilizando los frameworks de Cocoa Touch, principalmente Foundation y UIKit.

Cocoa Touch: El equivalente de Cocoa para iOS, donde los frameworks están ajustados para la experiencia del usuario móvil basada en el tacto.

Foundation: Son las clases principales para la programación de Objective-C. Ofrecen todos los tipos de datos y servicios fundamentales necesarios para Cocoa y Cocoa Touch.

UIKit: UIKit proporciona muchos de los objetos principales de una aplicación, incluidos aquellos que interactúan con el sistema, ejecutan el bucle de eventos principal de la aplicación y muestran contenido en pantalla.

Tipos de datos primitivos

Objective-C hereda todos los tipos primitivos del lenguaje C y agrega algunos extras.

id: Conocido como el tipo de objeto anónimo o dinámico, puede almacenar una referencia a cualquier tipo de objeto sin necesidad de especificar un símbolo de puntero.

BOOL: Se utiliza para especificar un tipo booleano donde 0 se considera NO (falso) y cualquier valor distinto de cero se considera SÍ (verdadero). Cualquier objeto nulo también se considera NO.

nil: Se utiliza para especificar un puntero de objeto nulo. Cuando las clases se inicializan por primera vez, todas las propiedades de la clase se establecen en 0, lo que significa que apuntan a cero.

Objective-C también tiene otros tipos primitivos, como NSInteger, NSUInteger, CGRect, CGFloat, CGSize, CGPoint, etc.

main.m

Este archivo tiene dos tareas principales:

- * Crear un autorelease pool principal para la aplicación.
- * Invocar el bucle de eventos de la aplicación.



The screenshot shows the Xcode code editor with the main.m file open. The code is displayed in a dark-themed interface. At the top left of the code area, there are three colored dots: red, yellow, and green. The code itself is as follows:

```
1 int main(int argc, char * argv[]) {
2     NSString * appDelegateClassName;
3     @autoreleasepool {
4
5     }
6     return UIApplicationMain(argc, argv, nil, appDelegateClassName);
7 }
```

Autorelease Pools

Son objetos que apoyan al sistema de manejo de memoria. Permiten que el software retrase la liberación de objetos hasta el final de cada ciclo de eventos.

Si se crea un segundo hilo en la aplicación, se tiene que proveer de su propio autorelease pool.

Se crea con la directiva `@autoreleasepool` seguido de llaves.

Las aplicaciones de iOS crean automáticamente un autorelease pool, y realmente no hay que preocuparse por eso.

Clase envolvente

NSNumber

Las clases envolventes son clases especiales que proveen a los tipos primitivos de funciones útiles.

La clase NSNumber es una clase envolvente que almacena a los tipos primitivos como si fueran objetos.

Puede parecer redundante tener una versión orientada a objetos de todos los datos primitivos, pero es necesario ya que si uno quiere almacenar valores numéricos en un NSArray, NSDictionary, o en alguno de las colecciones que existen, necesariamente deben ser objetos, estas colecciones no saben cómo manejar datos primitivos.

Cadenas

NSString

La clase NSString es la clase básica para representar texto.

Al igual que en C, estas cadenas almacenan caracteres, sin embargo, son objetos y no arreglos de bytes.

Es un tipo inmutable, puedes usar las cadenas para construir otras cadenas, pero no puedes editarlas.

La forma más común de crear un string es usando la sintaxis @“alguna cadena”.

Colecciones

Arreglos (NSArray, NSMutableArray)

Hay dos tipos de arreglos en Objective-C: inmutables y mutables. El contenido de una lista inmutable no puede cambiar en tiempo de ejecución. Las listas inmutables se crean como instancia de la clase NSArray. Las listas Mutables se crean mediante la clase NSMutableArray (una subclase de NSArray) y puede ser modificada después de haber sido creado e inicializado.

Colecciones

Sets (NSSet) y Diccionarios (NSDictionary)

NSSet es otra de las colecciones de Objective-C que representa un conjunto no ordenado de elementos distintos. Los NSSet son usados para comprobar si un objeto está en un conjunto, cosa que no puede hacerse con un NSArray eficientemente.

El NSDictionary es una colección no ordenada, que asocia un valor con una clave. Es útil para manejar relaciones de 2 objetos.

Fast Enumeration



donde arrayOfPeople puede ser cualquier objeto que se ajuste al protocolo `NSFastEnumeration`. `NSArray` y `NSSet` enumeran sobre sus objetos, `NSDictionary` enumera sobre claves.

Documentación

iOS Development Lab

Documentación

Ya sea que estés atascado en un error difícil o te estés familiarizando con un código nuevo, tienes acceso a un amplio conjunto de documentación de Xcode que hará avanzar tu desarrollo.

Esta documentación también se encuentra en línea:

[Documentación Oficial Apple](#)

Elementos de apoyo:

[Cheat Sheet](#)

[Objective-C to Swift](#)

[The Big Nerd Ranch Guide](#)

[iOS Development Lab](#)

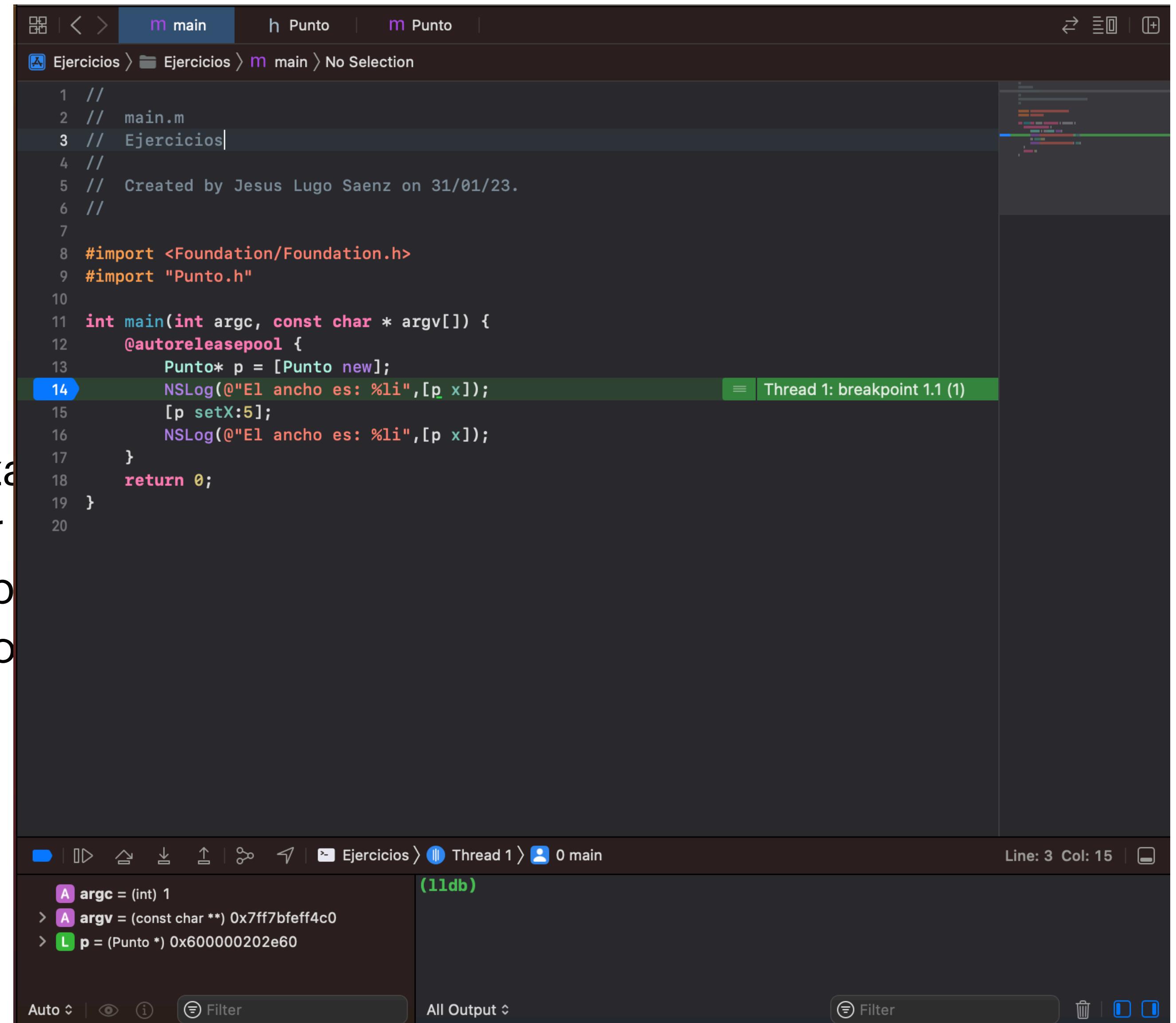


Debugger

iOS Development Lab

Debugger

Para poder utilizar el debugger es necesario poner un breakpoint en el punto especifico.



The screenshot shows the Xcode IDE with a dark theme. The top navigation bar includes tabs for 'main' and 'Punto'. The project browser shows 'Ejercicios > Ejercicios > main > No Selection'. The code editor displays a C-style program:

```
1 //  
2 //  main.m  
3 // Ejercicios|  
4 //  
5 // Created by Jesus Lugo Saenz on 31/01/23.  
6 //  
7  
8 #import <Foundation/Foundation.h>  
9 #import "Punto.h"  
10  
11 int main(int argc, const char * argv[]) {  
12     @autoreleasepool {  
13         Punto* p = [Punto new];  
14         NSLog(@"El ancho es: %li", [p x]);  
15         [p setX:5];  
16         NSLog(@"El ancho es: %li", [p x]);  
17     }  
18     return 0;  
19 }  
20
```

A blue circle marks a breakpoint on line 14. A green status bar at the bottom right indicates 'Thread 1: breakpoint 1.1 (1)'. The bottom panel shows the debugger's variable list and command interface. The variable list includes:

- argc = (int) 1
- argv = (const char **) 0x7ff7bfeff4c0
- p = (Punto *) 0x600000202e60

The command bar includes buttons for step, run, and stop, along with filters for 'Auto' and 'All Output'.

Botones de control

El debugger tiene 4 botones para controlar la depuración del proyecto.

Continua con la ejecución del programa hasta que encuentre otro breakpoint.



Ejecuta la linea seleccionada por el breakpoint y se detiene en la siguiente línea.

Si se hace clic cuando hay una llamada a una función, avanza a la primera línea de la función y luego se detiene la ejecución nuevamente.

Ejecuta todas las líneas restantes en llamada de la función y detiene la ejecución en la línea después de la función.

Clases y Objetos

iOS Development Lab

Interfaz e implementación

Las clases constan de una interfaz y una implementación. La **interfaz** indica la estructura del objeto y la **implementación** contiene la implementación de sus métodos

Definición y creación de clases

.h

```
1 /* Utiliza import para incluir clases de Objective C */
2 #import <Foundation/Foundation.h>
3 // Single line comment
4 /* Multi
5    Line
6    comment
7 */
8 // Interfaz para la clase con nombre ClaseNormal
9 @interface ClaseNormal: NSObject
10 /* NSObject es una superclase de todas las clases en Objective C*/
11 {
12     // Variables de instancia - Instance variables
13 }
14 // Class properties - Propiedades de clase.
15 // Declaraciones de métodos de instancia.
16 @end
```

.m

```
1 // Implementación
2 @implementation ClaseNormal
3 // Definiciones de métodos de instancia
4 @end
5 /* Todas las aplicaciones de Objective C tienen una función 'main'
6 y permiten Command Line Arguments */
7 int main(int argc, const char * argv[])
8 {
9     /* Autoreleasepool es usado en el main para manejo automático de memoria.*/
10    @autoreleasepool {
11        // simple log
12        NSLog(@"Hello, World!");
13        ClaseNormal *class =
14            [[ClaseNormal alloc] init];
15        NSLog(@"%@", [class class]);
16    }
17    return 0;
18 }
```

Interfaz

La declaración de una interfaz va desde la directiva al compilador @interface hasta la directiva al procesador @end.

```
7  /* Punto.h */
8  #import <Foundation/Foundation.h>
9
10 @interface Punto: NSObject {
11     NSInteger x;
12     NSInteger y;
13 }
14
15 - init;
16 + (NSString *) sistema;
17 - (NSInteger) x;
18 - (void) setX: (NSInteger) paramX;
19 - (void) setX: (NSInteger) paramX incrementando: (NSInteger) paramSumar;
20 - (NSInteger) y;
21 - (void) setY: (NSInteger) paramY;
22 - (void) setY: (NSInteger) paramY incrementando: (NSInteger) paramSumar;
23 @end
24
```

Implementación

Lo primero que se suele hacer es importar su interfaz, luego se utilizan las directivas del compilador `@implementation` y `@end` para encerrar la implementación de la clase.

- * Solo si una clase no tiene métodos, podemos omitir su implementación.
- * En la implementación solo se pone el nombre de la clase, ya no se vuelve a poner la clase de la que deriva.

Implementación

```
8  /* Punto.m */
9  #import <Foundation/Foundation.h>
10 #import "Punto.h"
11
12 @implementation Punto
13
14 static NSInteger nPuntos=0;
15
16 -init {
17     if ((self = [super init])) {
18         nPuntos++;
19     }
20     return self;
21 }
22
23 + (NSString *) sistema {
24     return @"Cartesiano";
25 }
26 - (NSInteger) x{
27     return x;
28 }
29 - (void) setX: (NSInteger) paramX {
30     x = paramX;
31 }
32 - (void) setX: (NSInteger) paramX incrementando: (NSInteger) paramSumar {
33     x = paramX + paramSumar;
34 }
35 - (NSInteger) y {
36     return y;
37 }
38 - (void) setY: (NSInteger) paramY {
39     y = paramY;
40 }
41 - (void) setY: (NSInteger) paramY incrementando: (NSInteger) paramSumar {
42     y = paramY + paramSumar;
43 }
44
45 @end
```

Objetos

En Objective-C solo se pueden crear los objetos en memoria dinámica. Lo que significa que en Objective-C solo nos podemos referir a los objetos mediante punteros .

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

Instanciar objetos

El método de clase **alloc** reserva la memoria dinámica del objeto y pone a cero todas las variables de instancia del objeto.

El método de instancia **init** se encarga de inicializar las variables de instancia del objeto.

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

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

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

Destruir objetos

Para destruir un objeto, se establece la variable que lo apunta a nil.

Un puntero que tiene un valor de nil normalmente se usa para representar la ausencia de un objeto.

Si envía un mensaje a una variable que es nil, no pasa nada.



Métodos

iOS Development Lab

Mensajes

Un mensaje siempre está contenido entre corchetes. Un mensaje tiene tres partes:

Receptor: un puntero al objeto al que se le pide que ejecute un método.

Selector: el nombre de método a ser ejecutado.

Argumentos: los valores que se suministraran como parámetros para el método.

Hay que tener clara la diferencia entre un mensaje y un método: un método es un fragmento de código que se puede ejecutar, y un mensaje es el acto de pedirle a una clase u objeto que ejecute un método. El nombre de un mensaje siempre coincide con el nombre del método a ejecutar.

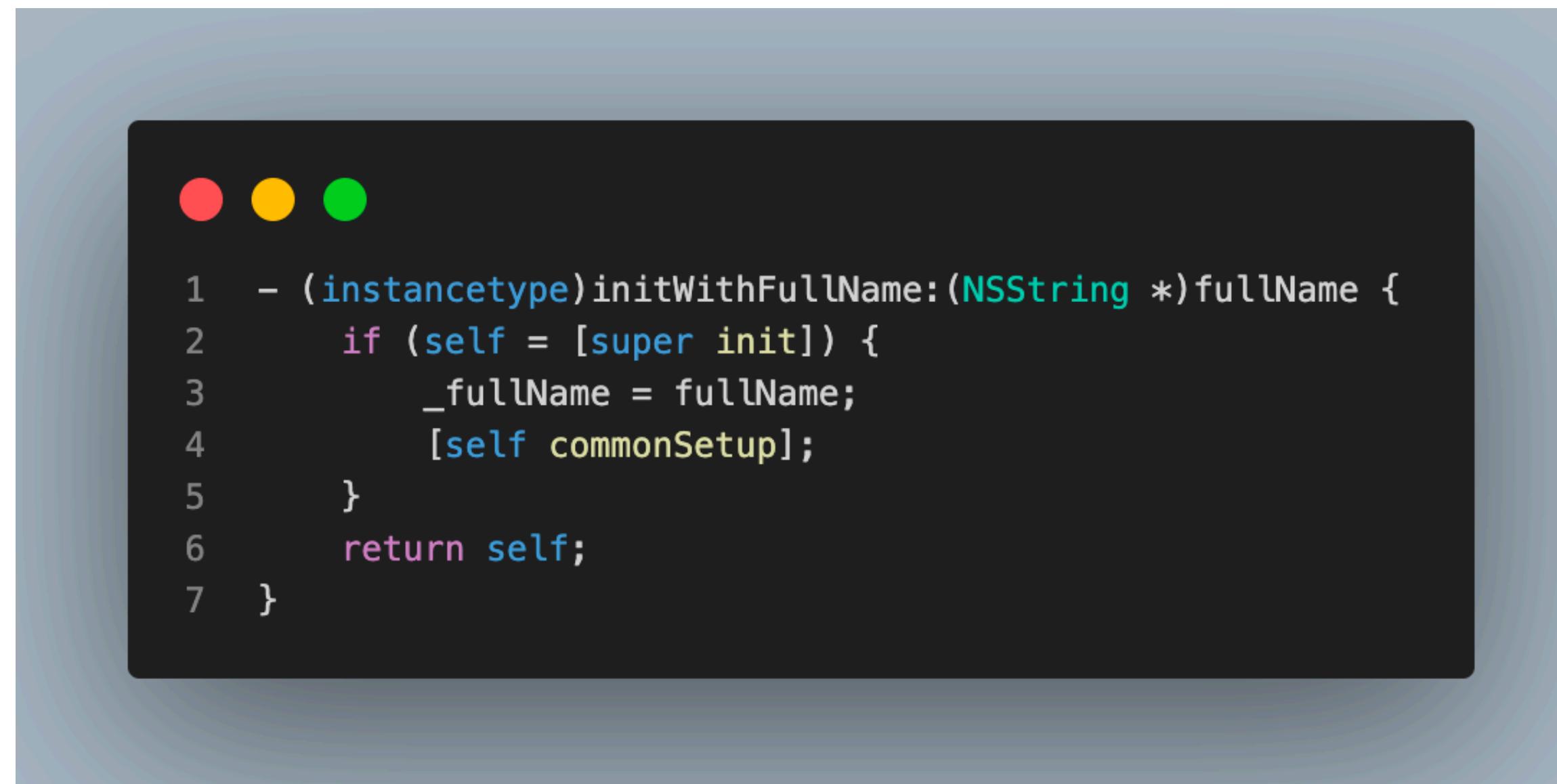
Métodos de clase vs. Métodos de instancia

Los métodos de clase normalmente crean nuevas instancias de la clase o recuperan alguna propiedad global de la clase. Los métodos de clase no operan en una instancia ni tienen acceso a las variables de instancia.

Sintácticamente, los métodos de clase se diferencian de los métodos de instancia por el primer carácter de su declaración. Un método de instancia usa el carácter - justo antes del tipo de retorno, y un método de clase usa el carácter +.

Inicializadores

Un inicializador toma argumentos que el objeto puede usar para inicializarse. A menudo, una clase tiene múltiples inicializadores porque las instancias pueden tener diferentes necesidades de inicialización.



instancetype vs. id

instancetype es una palabra reservada que se utiliza en los métodos inicializadores para indicar que regresa instancias de objetos.

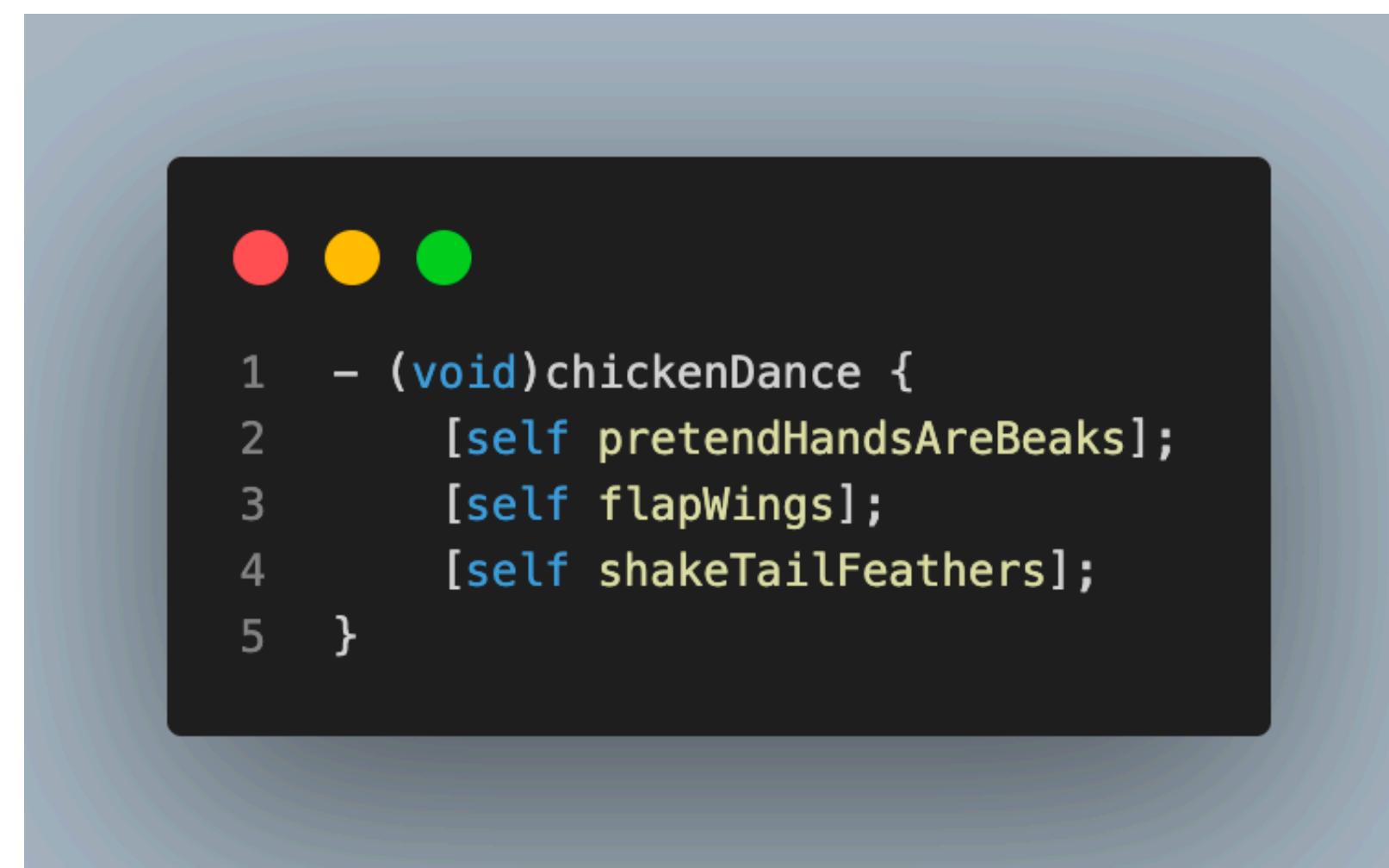
Antes de que se introdujera la palabra reservadainstancetype en Objective-C, los inicializadores devolvían id. Este tipo se define como "un puntero a cualquier objeto". (id se parece mucho a void * en C.).

A diferencia deinstancetype, id se puede usar como algo más que un tipo de retorno. Puedes declarar variables o parámetros de método de tipo id cuando no estés seguro de a qué tipo de objeto terminará apuntando la variable.

Self

Dentro de un método, self es una variable local implícita. No hay necesidad de declararla, y se establece automáticamente para apuntar al objeto al que se envió el mensaje.

Por lo general, self se usa para que un objeto pueda enviarse un mensaje a sí mismo:



Herencia

iOS Development Lab

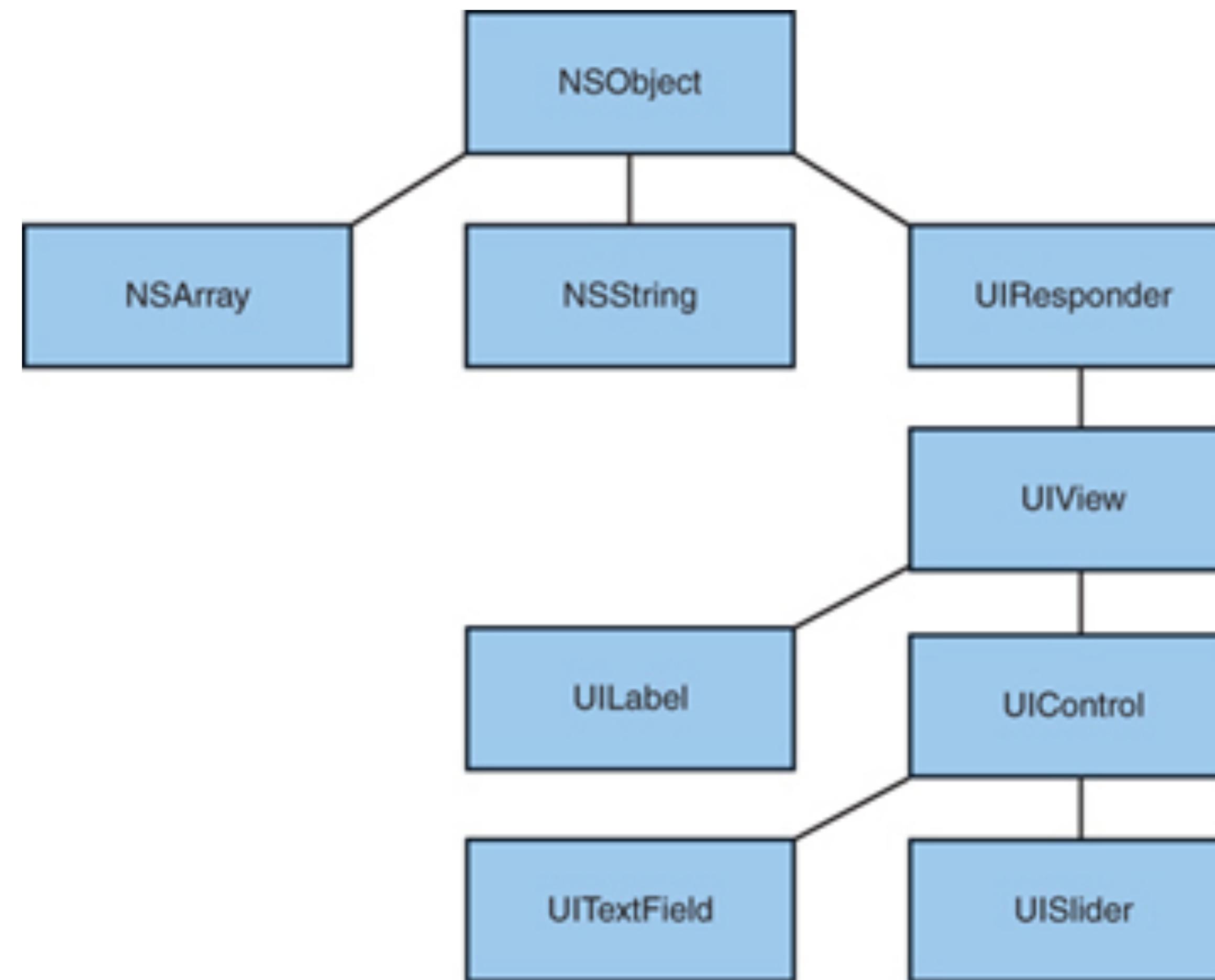
Herencia

Para poder indicar que una clase hereda de otra clase base, el nombre de la clase derivada (clase hija) en la interfaz se precede por dos puntos, y se pone después el nombre de la clase base (clase padre).

```
10 @interface Punto: NSObject {  
11     NSInteger x;  
12     NSInteger y;  
13 }
```

Para ello también debe tenerse en cuenta que la clase base solo se indica en el archivo interfaz, no en el de implementación.

Jerarquía de Clases



La clase raíz

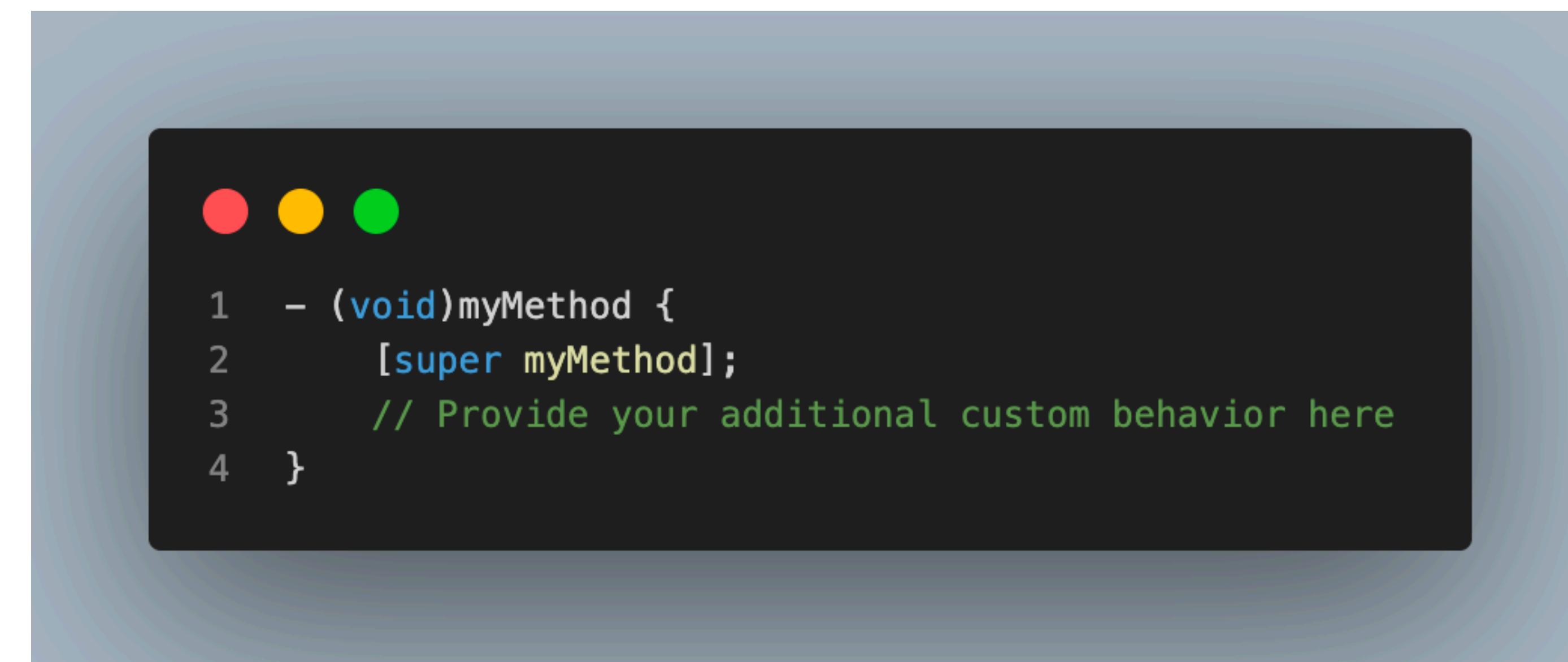
En las librerías de Objective-C las clases derivan, directa o indirectamente, de **NSObject**. Esta clase, llamada **clase raíz**, es la que implementa la funcionalidad del runtime de Objective-C.

Sobreescritura de métodos

Objective-C permite **sobrescribir** métodos en la clase derivada, volviendo a definir un método con el mismo prototipo. Para sobrescribir un método, no es necesario declararlo en la clase derivada (ya que se encuentra declarado en la clase base) basta con declararlo en la implementación.

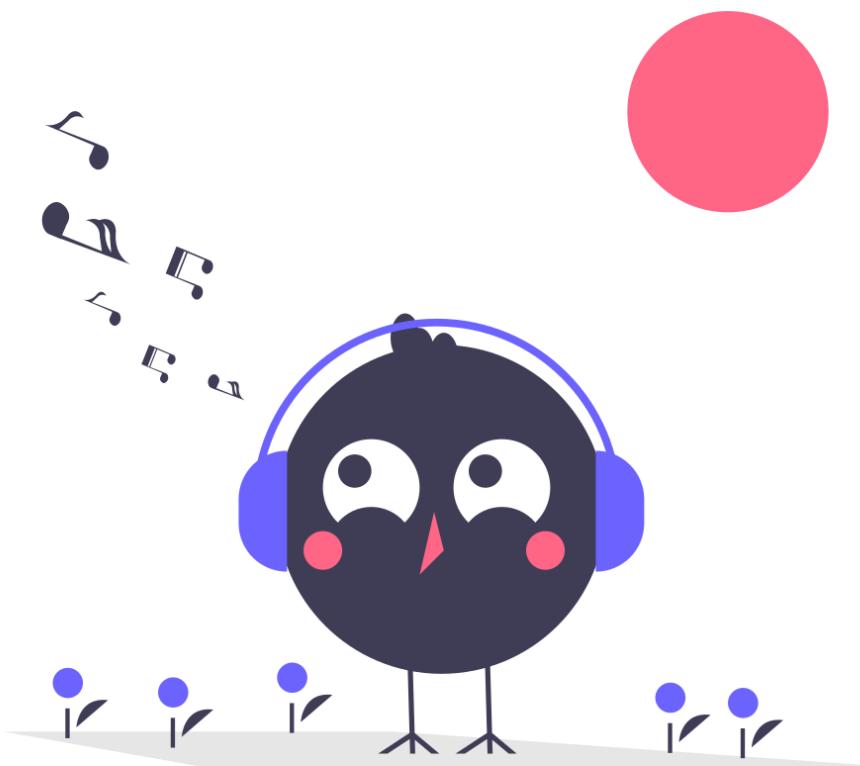
Sobreescritura de métodos

Si estás heredando de otra clase para sobreescribir un método dentro de esa clase, debes ser un poco cauteloso. Si deseas mantener el mismo comportamiento que la superclase, pero solo modificarlo ligeramente, puedes llamar a **super** dentro de la sobreescritura de esta manera:



Ejercicio

Clases y Objetos



Herencia



Métodos



Manejo de Memoria

iOS Development Lab

ARC (Automatic Reference Memory)

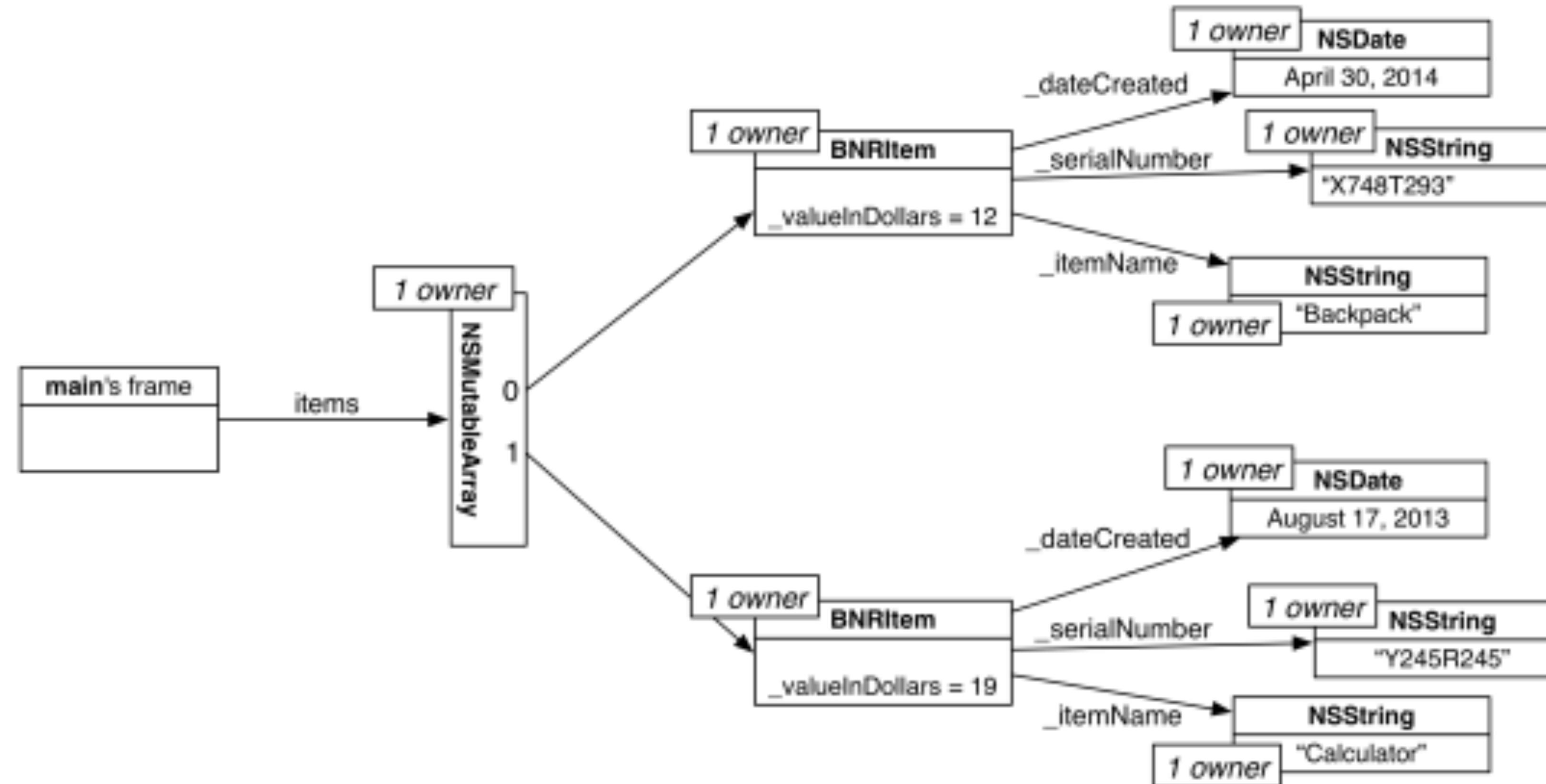
iOS Development Lab

ARC (Automatic Reference Memory)

La administración de memoria de cualquier aplicación la maneja ARC.

ARC es un sistema que nos brinda Objective-C para hacer el manejo de instancias almacenadas en memoria.

Pointer Variables y Object ownership



Punteros y propiedad de los objetos.

La idea de la propiedad de los objetos es útil para determinar cuando un objeto debe ser destruido para que se pueda reutilizar la memoria.

- ★ Un objeto que no tiene propietario será destruido. Un objeto sin propietario no puede mandar mensajes, está aislado y es inútil para la aplicación. Mantenerlo desperdicia memoria muy valiosa. A esto se le llama **fuga de memoria** (*memory leak*).
- ★ Un objeto que con uno o más propietarios no será destruido. Si un objeto es destruido y un objeto o método todavía tiene un puntero hacia él, tiendrás un problema, ya que si se envía un mensaje mediante este apuntador podría bloquear su aplicación. La destrucción de un objeto que todavía se utiliza se le llama desasignación prematura. También se le conoce como puntero colgante o referencia colgante

¿Cómo pierden los objetos a sus propietarios?

Cuando un variable apunta a un objeto se cambia para que apunte a otro objeto.

Cuando una variable que apunta a un objeto se establece como nil.

Cuando el propietario del objeto mismo es destruido.

Cuando un objeto de una colección (como un array) se elimina de esa colección.

Referencias fuertes y débiles.

Cada vez que una variable puntero apunta a un objeto, ese objeto tiene un propietario y permanecerá vivo. Esto se conoce como una **referencia fuerte** (*strong reference*).

Opcionalmente, una variable no puede tomar posesión de un objeto al que apunta. Una variable que no toma la propiedad de un objeto se conoce como una referencia débil (*weak reference*).

Referencias fuertes y débiles.

Una referencia débil es útil para resolver un problema que se llama ***retain cycle***.

Un ***retain cycle*** ocurre cuando dos o más objetos tienen fuertes referencias a entre sí.

Propiedades

iOS Development Lab

Propiedades

Hasta ahora, cada vez que hemos implementado una variable de instancia, se han tenido que implementar los métodos getters y setters.

Las propiedades son una alternativa para escribir métodos de acceso que ahorra mucho teclado y hace que los archivos de clase sean más fáciles de leer.



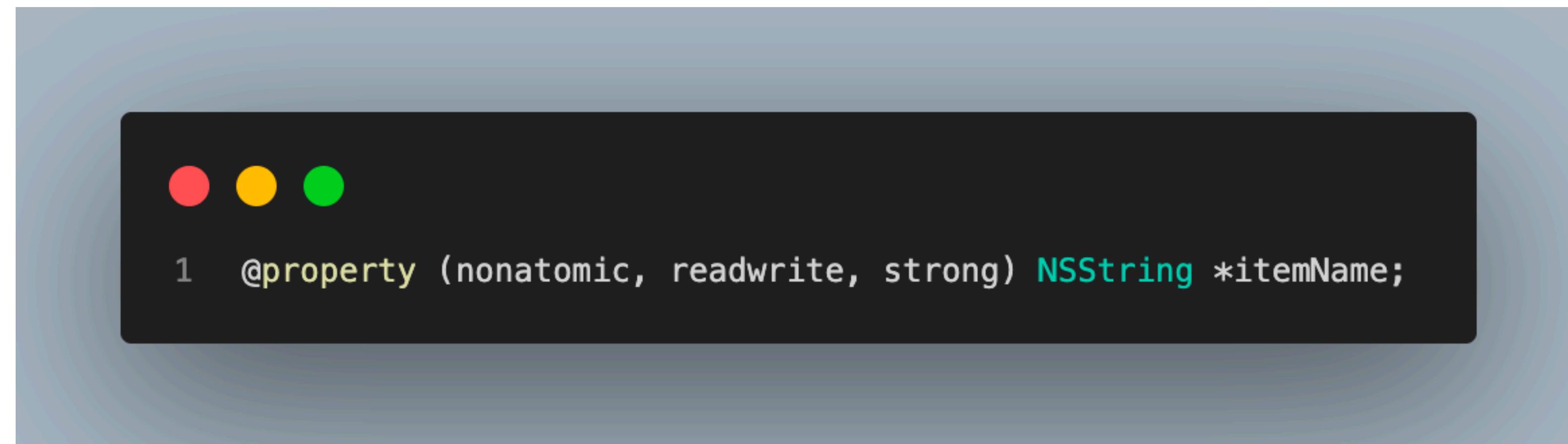
Propiedades

Por defecto, declarar una propiedad proporciona tres cosas: una variable de instancia y dos métodos de acceso (un setter y un getter).

	Without properties	With properties
BNRThing.h	<pre>@interface BNRThing : NSObject { NSString *_name; } - (void)setName:(NSString *)n; - (NSString *)name; @end</pre>	<pre>@interface BNRThing : NSObject @property NSString *name; @end</pre>
BNRThing.m	<pre>@implementation BNRThing - (void)setName:(NSString *)n { _name = n; } - (NSString *)name { return _name; } @end</pre>	<pre>@implementation BNRThing @end</pre>

Atributos de las Propiedades

Una propiedad tiene una serie de atributos que permiten modificar el comportamiento de los métodos de acceso y la variable de instancia que crea. Los atributos se declaran entre paréntesis después de la directiva @property. Aquí hay un ejemplo:



Atributos multihilo

Tiene dos valores: **nonatomic** o **atomic**.

El valor por default es **atomic**, sin embargo, el valor más utilizado es **nonatomic**. Este nos dice que a la hora de acceder a la propiedad, no realizará ningún tipo de bloqueo en caso de que haya más hilos que quieran acceder a él.

Atributos de lectura/escritura

Tiene dos valores: **readwrite** o **readonly**. Estos le dicen al compilador si implementar o no un setter para la propiedad.

Una propiedad **readwrite** implementa tanto un setter como un getter. Esta es la opción por default.

Una propiedad **readonly** sólo implementa un getter.

Atributos de manejo de memoria

Estos atributos son **strong**, **weak**, **copy** y **unsafe_unretained**.

Describen el tipo de referencia que el objeto con la variable de instancia tiene al objeto al que apunta la variable.

unsafe_unretained es el valor por default para las propiedad que no son objetos, es decir son primitivos.

Para los objetos, el default es **strong**.

Atributos de manejo de memoria

copy. Crea una copia inmutable del objeto tras la asignación y normalmente se usa para crear una versión inmutable de un objeto mutable.

weak. Las variables que son weak pueden apuntar a objetos, pero no se convierten en propietarios (o aumentan el retain count en 1). Si el retain count del objeto llega a 0, el objeto se desasignará de la memoria y el puntero weak se establecerá en nil. Es una buena práctica crear todos los delegados y IBOulet como weak, ya que no son de nuestra propiedad.

Atributos de manejo de memoria

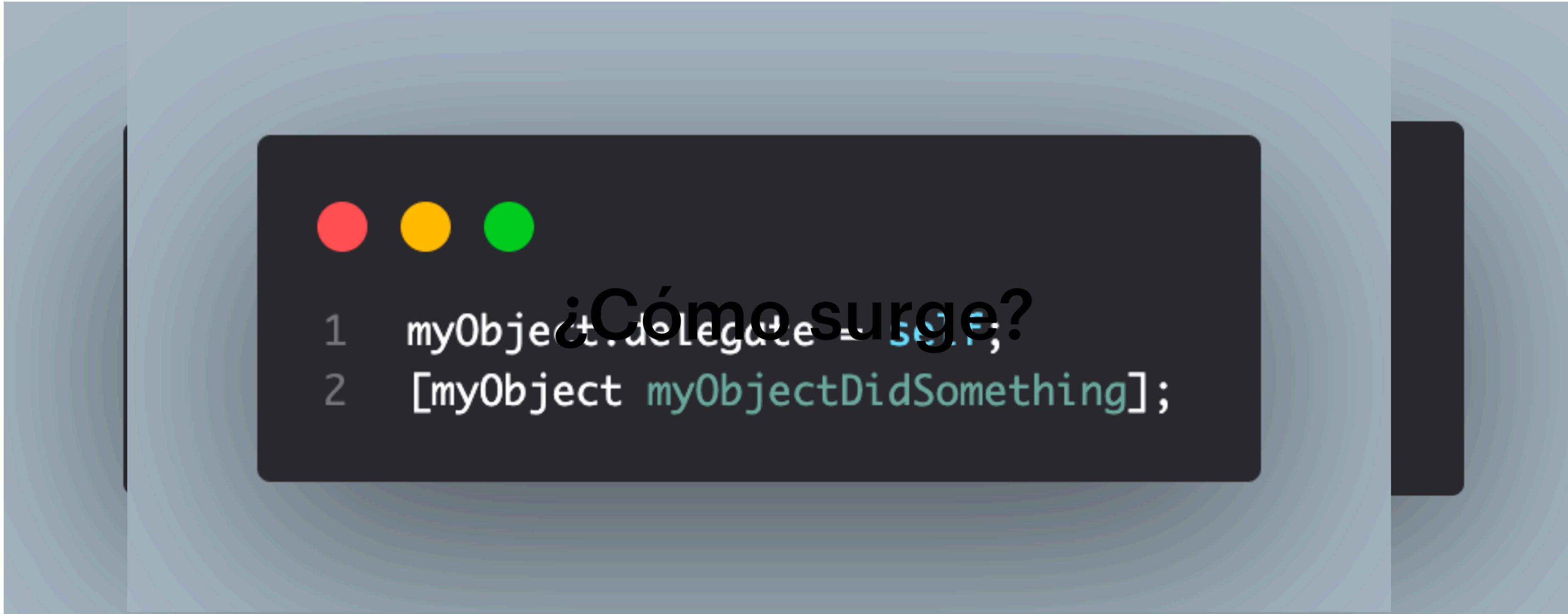
unsafe_unretained. Es similar a una referencia de tipo weak con la diferencia que si el retain count del objeto llega a 0, no establece el puntero a nil.

strong. El setter generado automáticamente retendrá (es decir, incrementará el retain count de) el objeto y lo mantendrá vivo hasta que se libere.

Bloques

iOS Development Lab

Bloques



Bloques

¿Qué es?

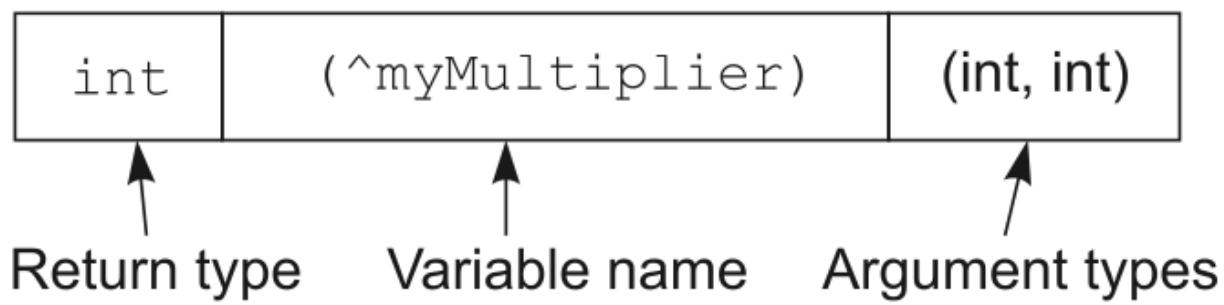
Los bloques son solo eso: bloques de código o funciones anónimas. **GCD (Grand Central Dispatch)** y los bloques son un dúo muy poderoso. Puedes escribir pequeños piezas de código y entregarlos a GCD para ejecutarse en un hilo paralelo sin ningún tipo de los dolor en programas multihilo que normalmente causan. La ejecución paralela nunca ha sido más fácil, y con GCD ya no hay una excusa para no hacerlo.



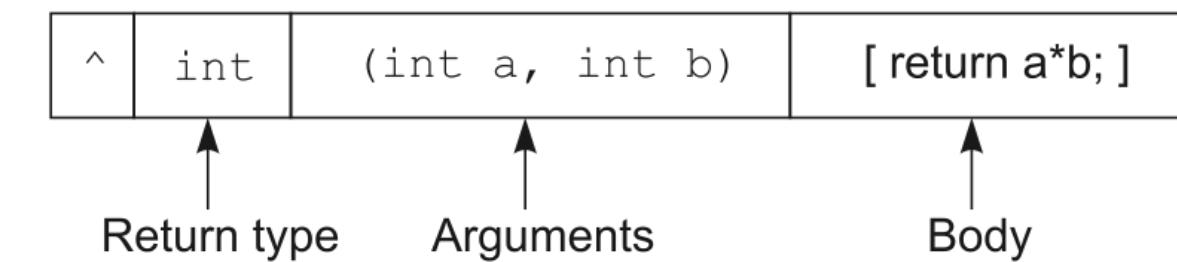
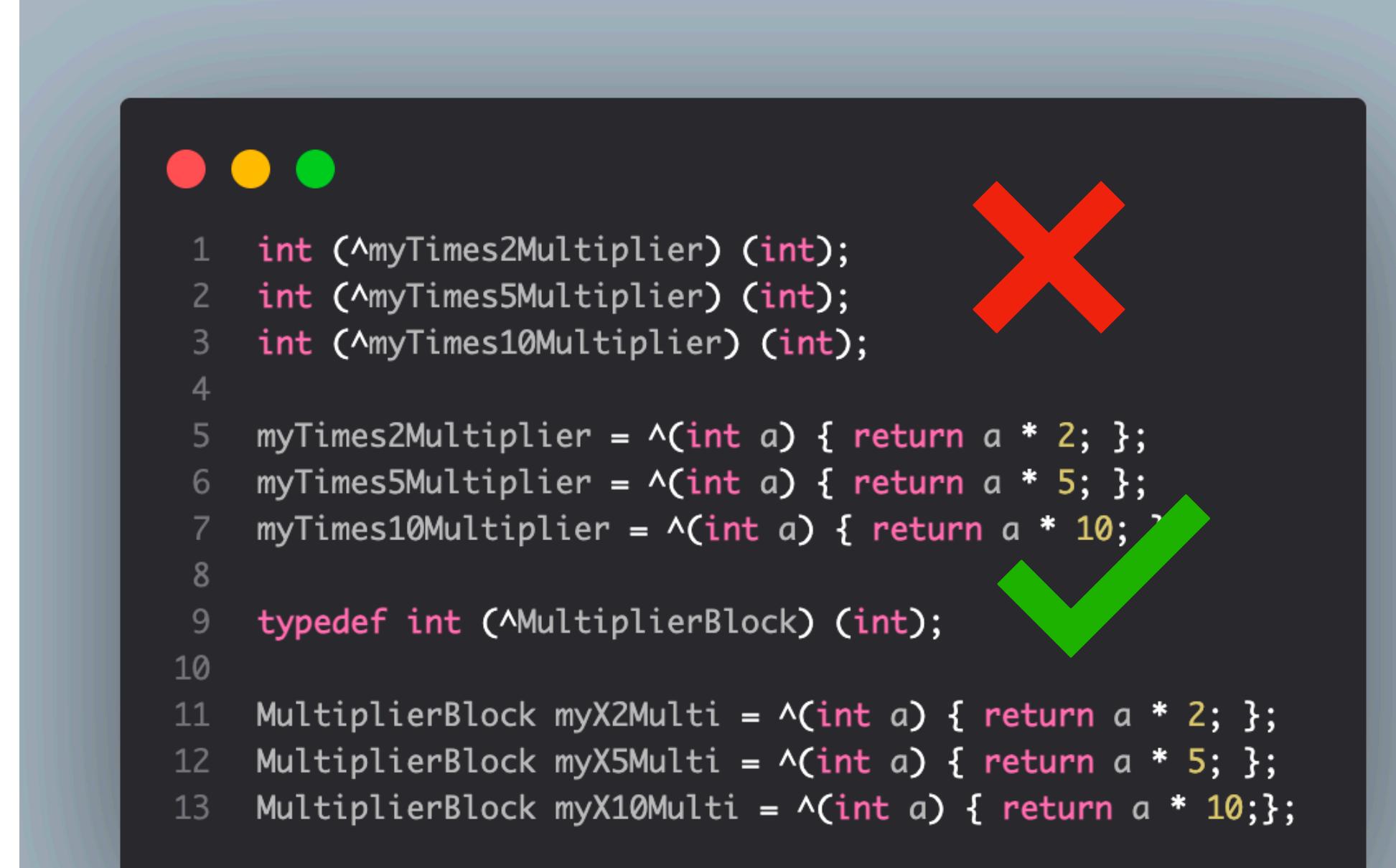
Bloques

Sintaxis

```
● ● ●  
1 int (^myMultiplier)(int, int) = ^int (int a, int b){  
2     return a * b;  
3 };  
4 int result = myMultiplier(7, 8);
```



```
● ● ●  
X  
1 int (^myTimes2Multiplier) (int);  
2 int (^myTimes5Multiplier) (int);  
3 int (^myTimes10Multiplier) (int);  
4  
5 myTimes2Multiplier = ^(int a) { return a * 2; };  
6 myTimes5Multiplier = ^(int a) { return a * 5; };  
7 myTimes10Multiplier = ^(int a) { return a * 10; };  
8  
9 typedef int (^MultiplierBlock) (int);  
10  
11 MultiplierBlock myX2Multi = ^(int a) { return a * 2; };  
12 MultiplierBlock myX5Multi = ^(int a) { return a * 5; };  
13 MultiplierBlock myX10Multi = ^(int a) { return a * 10; };
```



Bloques



```
1 func backward(_ s1: String, _ s2: String) -> Bool {  
2     return s1 > s2  
3 }  
4 var reversedNames = names.sorted(by: backward)  
5 // reversedNames is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]  
6
```

ques



```
1 reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in  
2     return s1 > s2  
3 })
```

Bloques

```
● ● ●  
1 - (void)hitAPI {  
2     [self callAPI:@"https://graph.facebook.com/4" res:^(NSDictionary * _Nullable json, NSError * _Nullable error) {  
3         NSLog(@"res %@", err %@", json, error);  
4     }];  
5 }  
6  
7 - (void)callAPI:(NSString*)url res:(void (^)(NSDictionary * _Nullable json, NSError * _Nullable error))completionHandler {  
8     [self callAPI:url completionHandler:^(NSData * _Nullable data, NSURLResponse * _Nullable response, NSError * _Nullable error) {  
9         if (error) {  
10             completionHandler(nil, error);  
11             return;  
12         }  
13         NSError* error1;  
14         NSDictionary* json = [NSJSONSerialization JSONObjectWithData:data  
15                                         options:kNilOptions  
16                                         error:&error1];  
17         dispatch_async(dispatch_get_main_queue(), ^{  
18             completionHandler(json, error1);  
19         });  
20     }];  
21 }
```

Ejemplo

iOS Development Lab

Categorías

iOS Development Lab

Categorías

Permite a los desarrolladores agregar métodos a una clase que puede no estar allí de forma predeterminada.

En lugar de crear una subclase y agregarle el método, las categorías permiten a un desarrollador **agregar métodos a cualquier clase existente sin subclasificar.**

Esto generalmente se realiza con las clases de estructura de datos comunes en Objective-C (`NSString`, `NSArray`, `NSData`). A menudo es conveniente agregar un método o dos a estas clases que ayudarán con una aplicación que está haciendo.

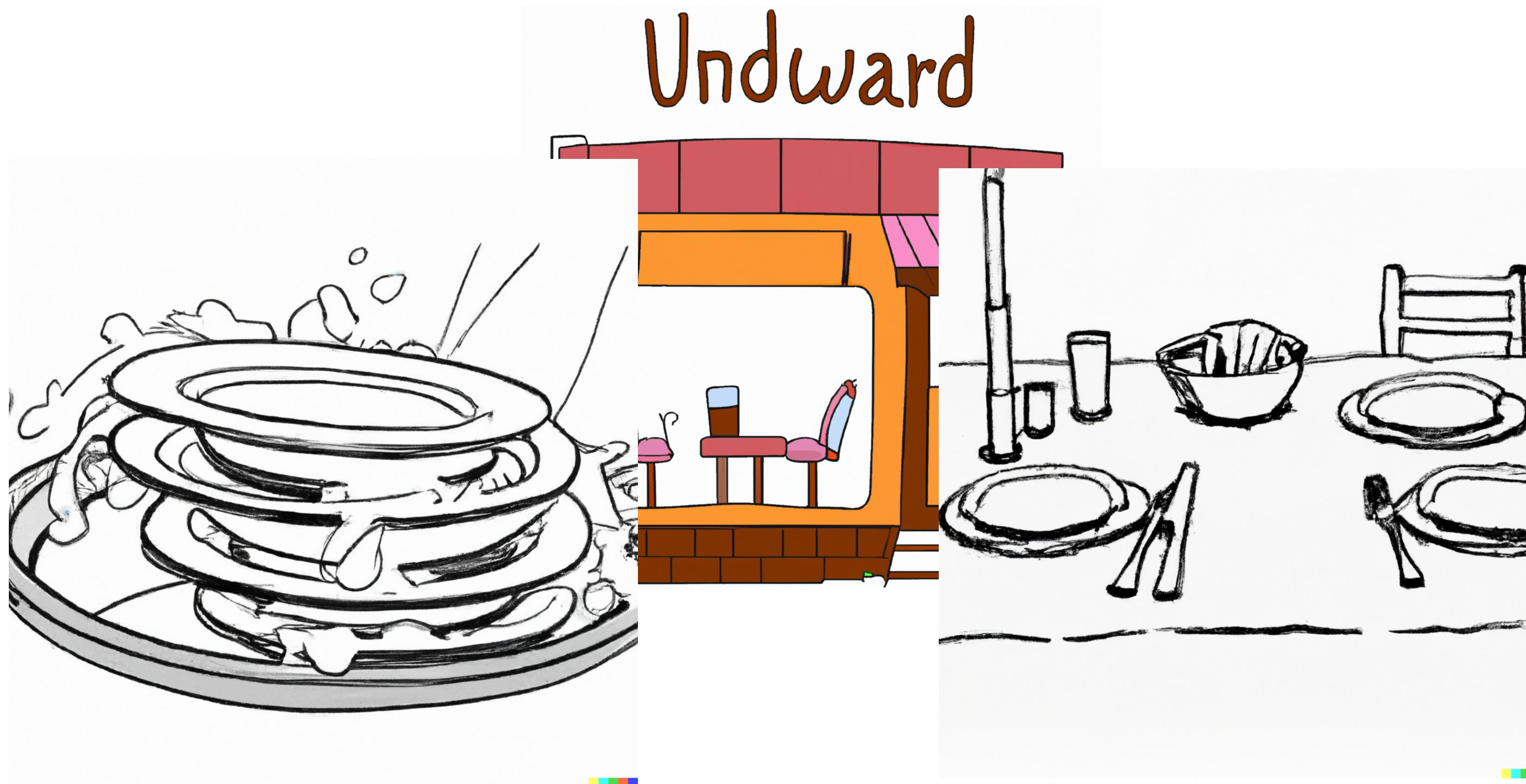
Categorías

Ten en mente que **no deberías** ocupar una categoría para **sobreescibir un método** existente en la clase base. Las categorías no son subclases. Si tratas de hacer esto, el compilador no sabrá qué instrucción seguir.

Ejemplo

iOS Development Lab

Ejemplo



iOS Development Lab

Ejercicio

iOS Development Lab

Delegation

iOS Development Lab

Delegation

La delegación es un patrón de diseño, o una práctica común, que permite que una clase o estructura traspase o delegue algunas de sus responsabilidades a una instancia de otro tipo.

La delegación es básicamente una forma de permitir que una clase reaccione a los cambios realizados en otra clase o de influir en el comportamiento de otra clase mientras se minimiza el acoplamiento entre los dos.

Los tipos que delegan la implementación a otros tipos normalmente lo hacen definiendo un protocolo. El protocolo define las responsabilidades que se pueden delegar y el delegado adopta el protocolo para ejecutar la tarea real.

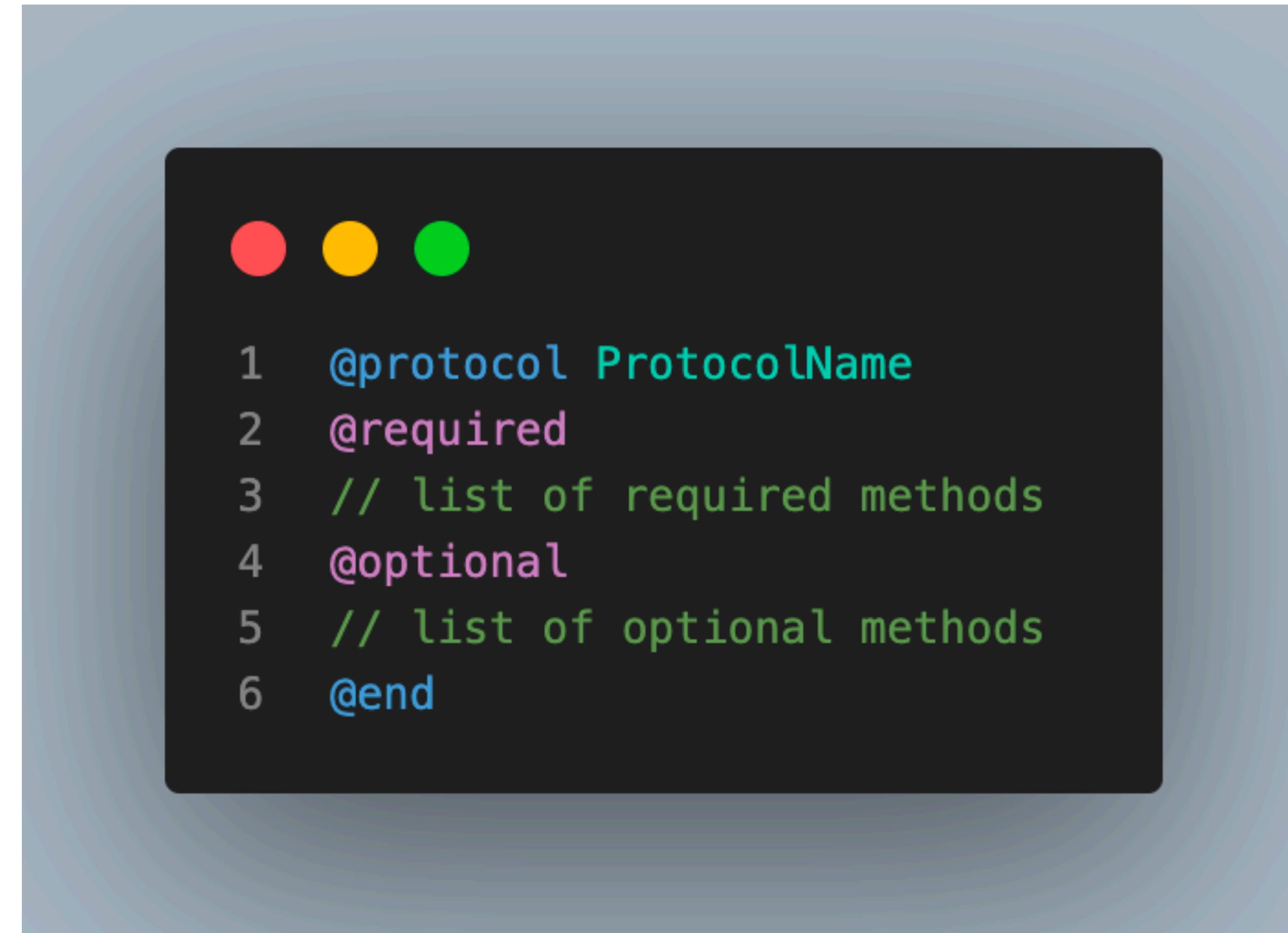
Protocolos

Un protocolo es un conjunto de reglas o procedimientos que definen cómo se hacen las cosas.

En programación, un protocolo define las propiedades o métodos que debe tener un objeto para completar una tarea.

Cuando conformas un protocolo, prometes implementar todos los métodos requeridos por ese protocolo. El compilador verificará que todo esté en orden y no compilará el programa si falta algo.

Creación de un protocolo



Implementación de un protocolo



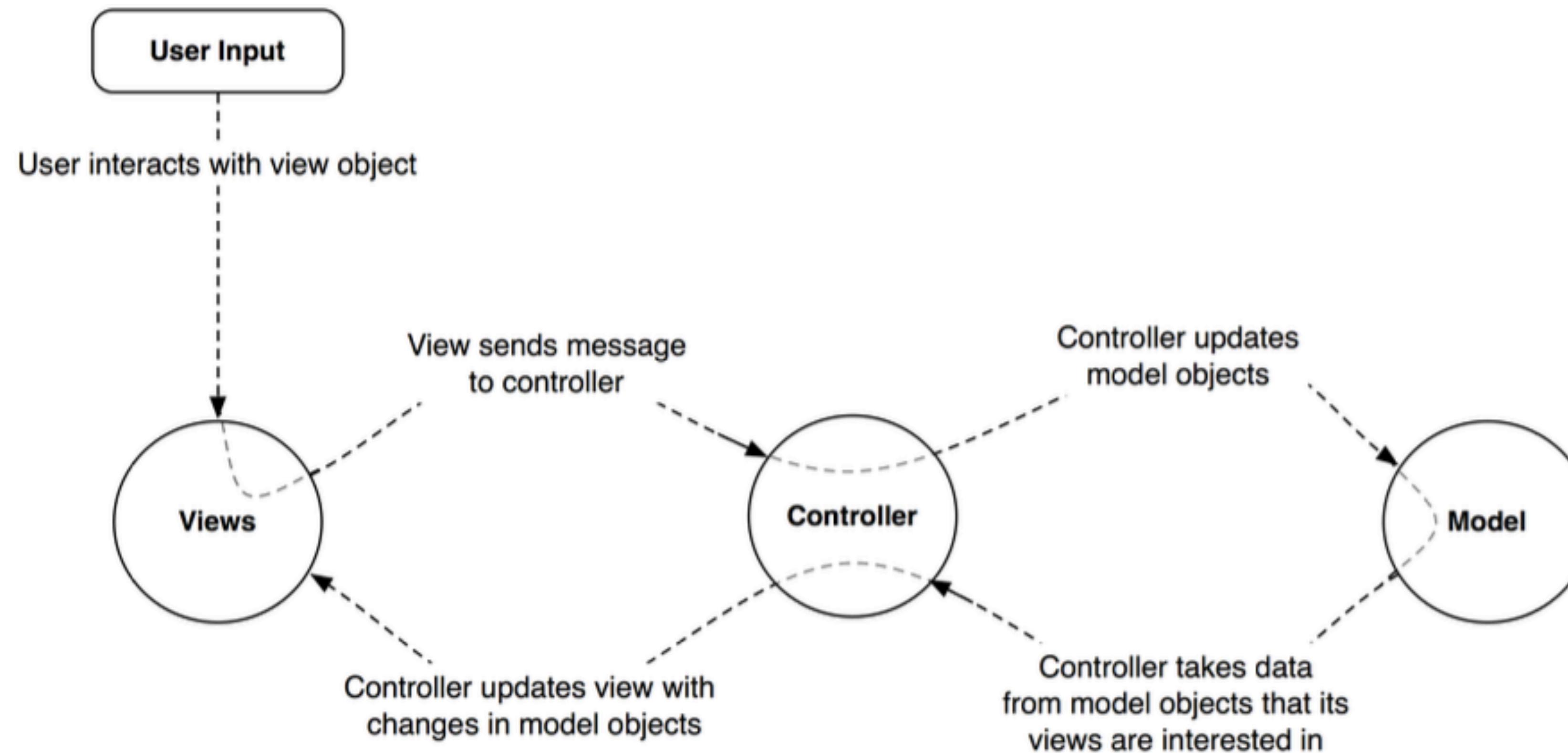
```
1 @interface MyClass : NSObject <MyProtocol1>, ... , <MyProtocol2>
2 ...
3 @end
```

Diseño de Interfaces

iOS Development Lab

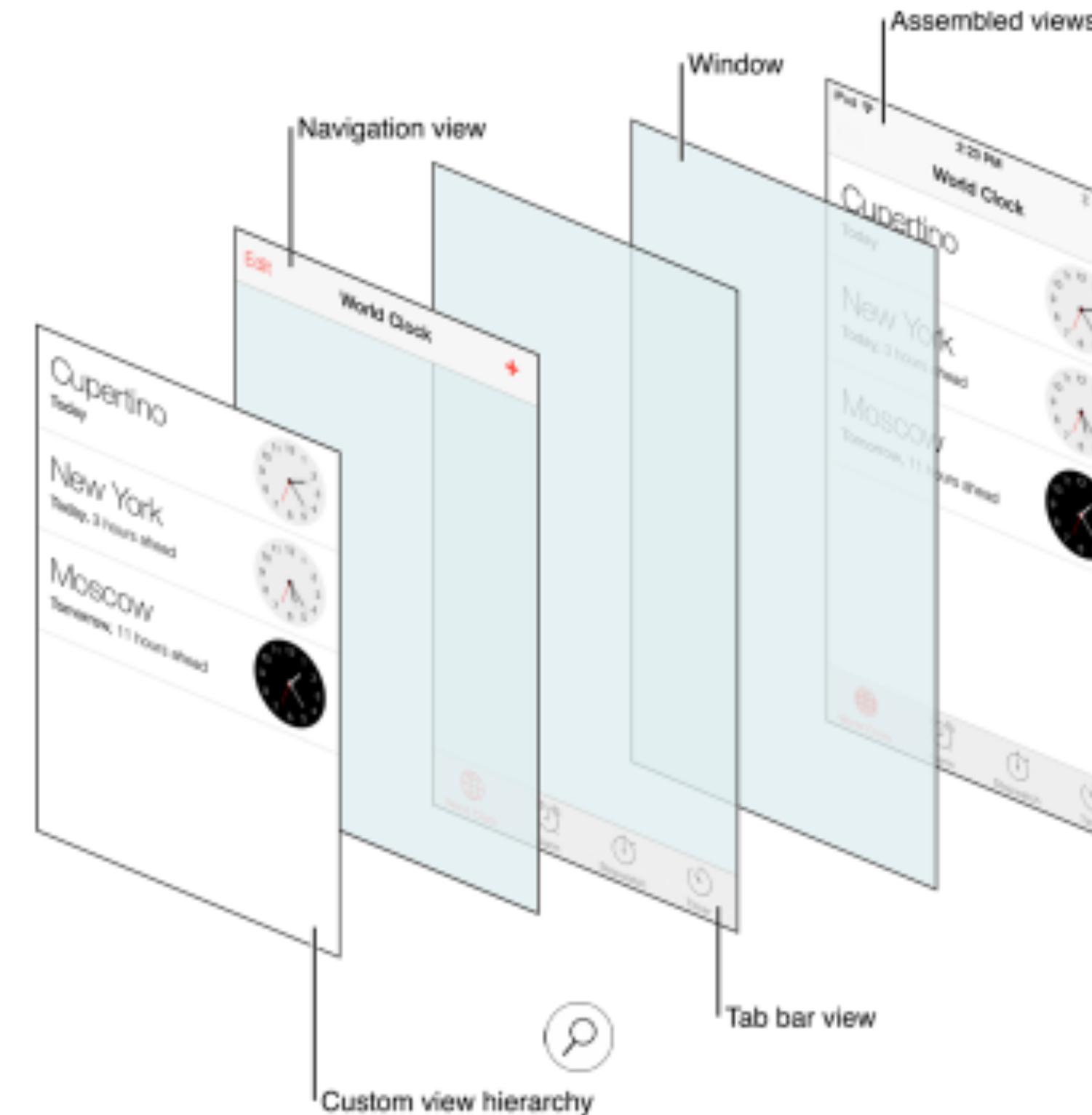
Modelo-Vista-Controlador (MVC)

Es un patrón de diseño utilizado en el desarrollo de iOS. En MVC, cada objeto es un objeto de modelo, un objeto de vista o un objeto de controlador.

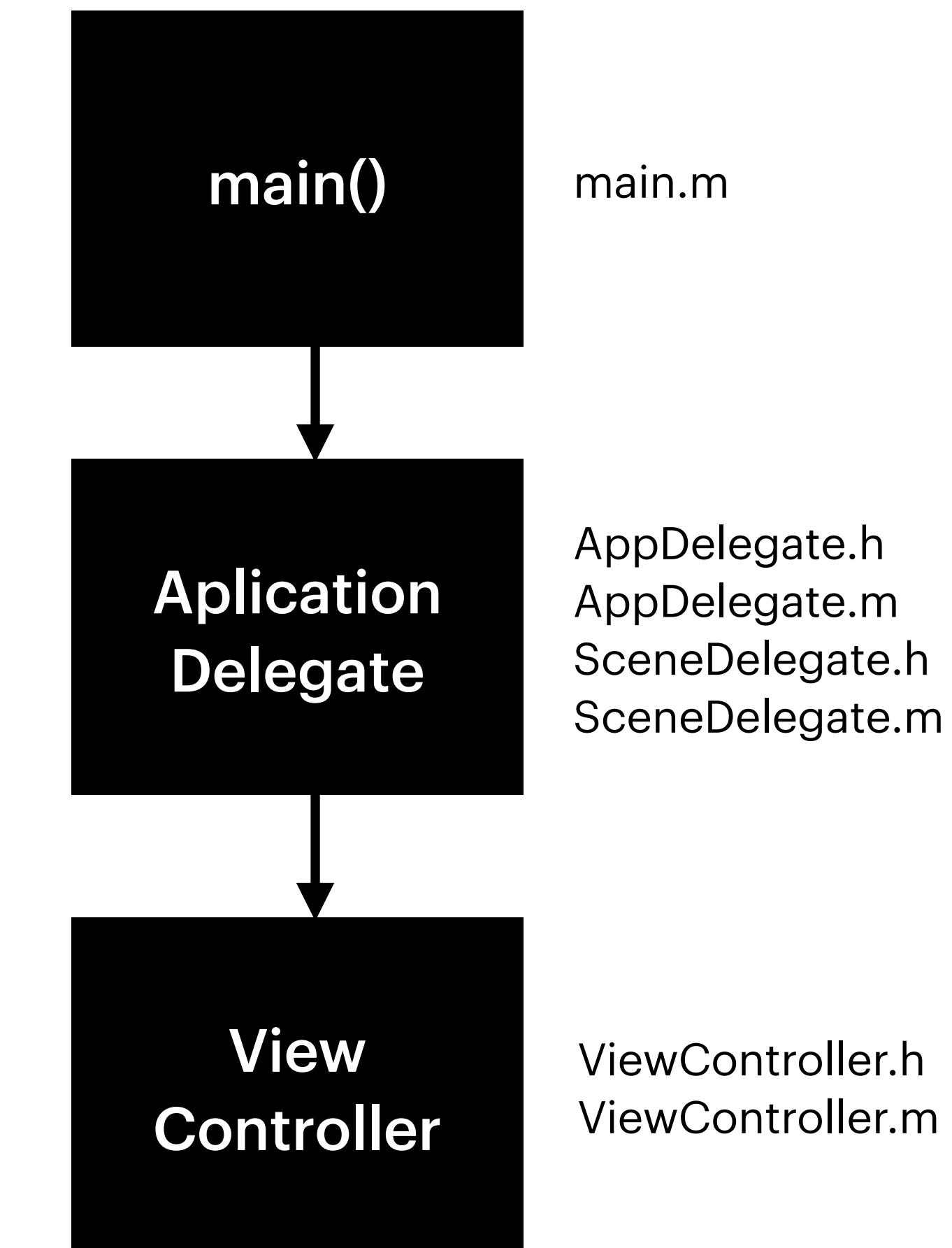


UIKit

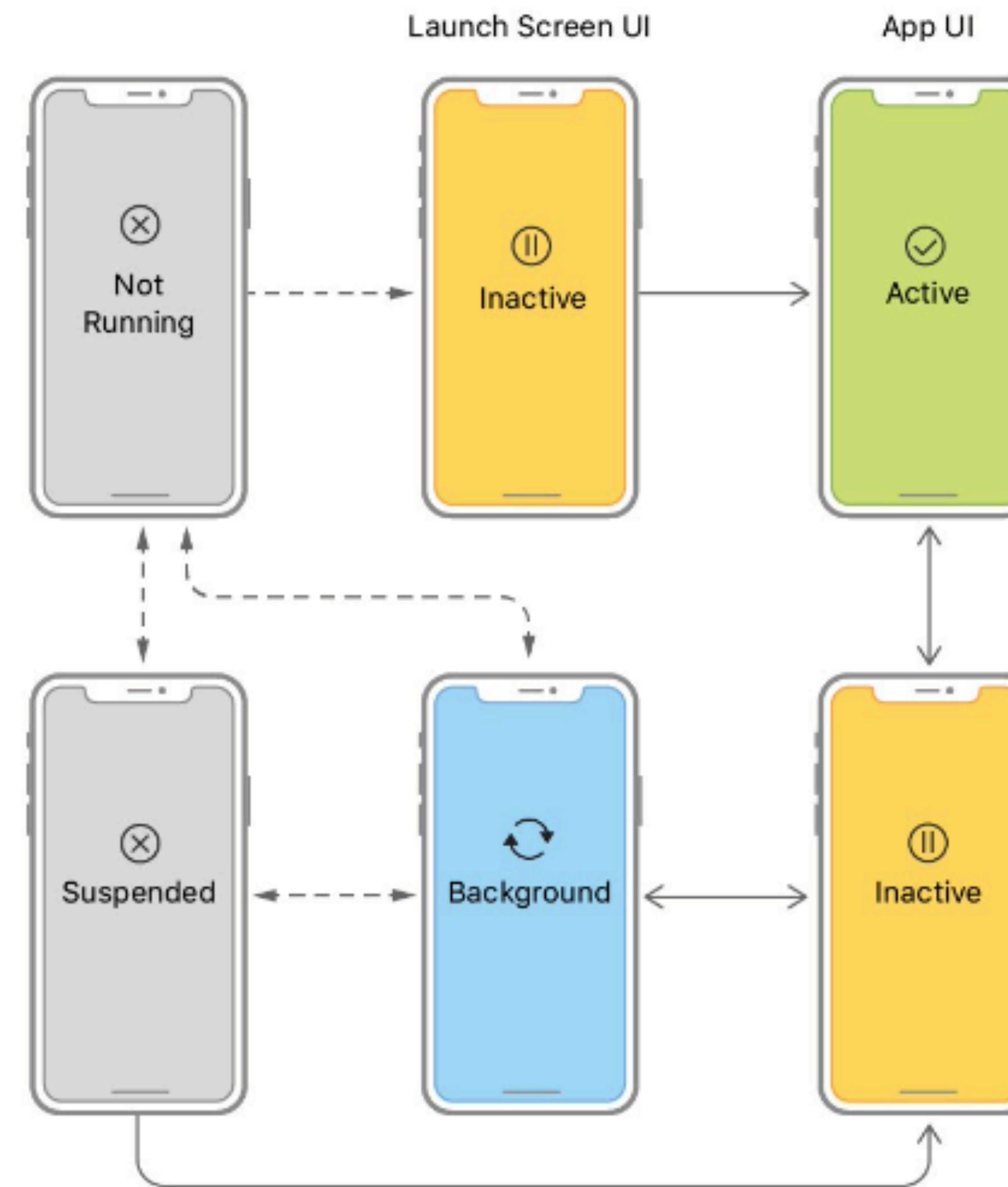
UIKit define cómo se muestra la información al usuario y cómo se responde a las interacciones del usuario y los eventos del sistema. También permite trabajar con animaciones, texto e imágenes.



Estructura básica de las aplicaciones en iOS



Ciclo de vida de las aplicaciones



Función UIApplicationMain

Crea una instancia de una clase llamada UIApplication.

Para cada aplicación, hay una sola instancia de UIApplication.

Este objeto es responsable de mantener el bucle de ejecución. Una vez que se crea el objeto de la aplicación, su bucle de ejecución se convierte esencialmente en un bucle infinito: el hilo de ejecución nunca volverá a main().

Otra cosa que hace la función UIApplicationMain es crear una instancia de la clase que servirá como delegado de UIApplication.

Application Delegate

La clase AppDelegate implementa el protocolo UIApplicationDelegate que define métodos que ayudan al manejo de los eventos en el ciclo de vida de la aplicación. Estos métodos son los siguientes:

Did Finish Launching. Cuando la aplicación haya terminado de iniciarse, se llamará a este primer método.

Configuration For Connecting a Scene Session. Se ejecuta cuando se crea una nueva Scene Session.

Did Discard Scene Sessions. Cuando un usuario descarta una instancia de la aplicación en iPad, se llama a este método con información sobre esa escena

Scene Delegate

Un scene delegate administra una escena en una aplicación. Una escena representa una instancia de la interfaz de la aplicación y normalmente mantiene una ventana en la pantalla del dispositivo. En iPad, las aplicaciones que admiten varias escenas pueden mostrar varias ventanas.

Es responsable de manejar los componentes a nivel de la interfaz de usuario durante el ciclo de vida de su aplicación.

Scene Delegate

Tiene seis funciones importantes:

Will Connect To. Se ejecuta cuando se conecta una escena adicional a la aplicación.

Scene Did Disconnect. Se llama cuando la escena se eliminó de la aplicación. Es el mejor lugar para limpiar y liberar cualquier recurso o para guardar archivos que eran necesarios para la escena

Scene Did Become Active. Se llama para que la escena sepa que pasó del estado inactivo al estado activo.

Scene Will Resign Active. Se llama cuando la escena está a punto de abandonar el estado activo.

Scene Will Enter Foreground. Se ejecuta cuando se pasa del estado de segundo plano al estado activo.

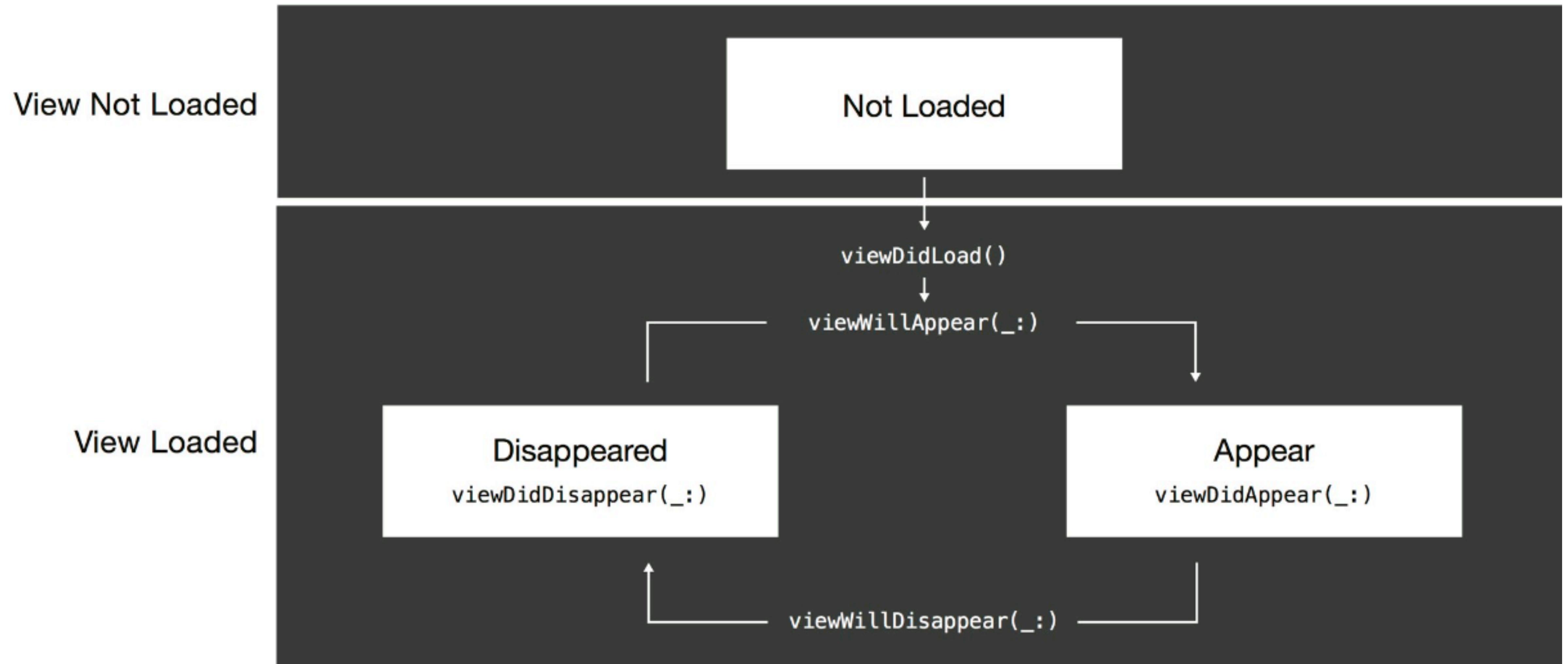
Scene Did Enter Background. Se ejecuta inmediatamente después de Scene Will Resign Active.

View Controller

UIKit define una clase especial que controla una vista, configura vistas secundarias, controla lo que muestran y responde a la interacción del usuario. Esta clase se llama `UIViewController`.

Por lo general, cada pantalla de una aplicación está representada por una escena en un storyboard, y cada escena en un storyboard está asociada con una subclase de `UIViewController`. La subclase `UIViewController` asociada se define en dos archivos `.h` y `.m` que contienen toda la lógica que controla la escena.

Ciclo de vida del View Controller



Interface Builder

Xcode tiene una herramienta integrada llamada Interface Builder que facilita la creación de interfaces de forma visual.

Interface Builder se abre cada vez que selecciona un archivo XIB (.xib) o un archivo de guión gráfico (.storyboard) desde el navegador de proyectos.

A diferencia de un XIB, un storyboard incluye muchas piezas de la interfaz, que definen el diseño de una o varias pantallas, así como la progresión de una pantalla a otra.

Controles

Actions And Outlets

Un outlet es la referencia entre un objeto del Interface Builder al código.

Cuando quieres que el usuario interactúe con un objeto, creas una acción, que es una referencia a una función que se ejecuta cuando sucede esta interacción.

Integración de Swift y Obj-C

iOS Development Lab

Integración de Swift y Obj-C

Bridging Header

iOS Development Lab

Integración de Swift y Obj-C

Bridging Header

Puedes usar los archivos de **Objective-C y Swift juntos** en un solo proyecto, sin importar qué idioma usara el proyecto originalmente. Esto hace que la creación de objetivos de aplicación y marco de lenguaje mixto sea tan sencillo como crear una aplicación o objetivo marco escrito en un solo idioma.

Integración de Swift y Obj-C

¿Sólo ocupamos un bridging header para poder ocupar Objective C y Swift ^{Sí} en un mismo proyecto?

Depende de varios factores, en muchos casos, existe la posibilidad de que las librerías de terceros (Third Party Libraries) estén escritas en Objective C, o que existan nuevas versiones en Swift, aquí debes de tomar en cuenta cuál tiene mayor mantenimiento y es de más estabilidad.

Integración de Swift y Obj-C

- * `import "NombreDelModulo-Swift.h"`
- * La clase que definamos debe heredar de NSObject

Ocupar el Swift Obj-C en el Proyecto de Desarrollo C

- * Importar la librería en el Bridging Header.
- * Ocuparla en Swift.