ETSIT UPM

dit UPM

Desarrollo de Apps para iOS

Introducción a Objective-C

IWEB-LSWC 2013-2014

Santiago Pavón

ver: 2014.02.17

© Santiago Pavón - UPM-DIT

Características Objective-C

- ANSI C añadiendo objetos.
- Sintaxis ampliada con:
  - clases
  - [objeto mensaje]
  - @directivas
  - ...
- Herencia simple.
- Categorías.
- Protocolos.
- Enlazado dinámico.
- Tipado vago o estricto.
- Gestión de Memoria: ARC.
  - Gestión automática de las cuentas de referencias.
- . . .

© Santiago Pavón - UPM-DIT

Extensión de Ficheros

- `fichero.h`
  - cabecera con declaraciones.
  - describe el API público.

- `fichero.m`
  - implementación en C y Objective-C

- `fichero.mm`
  - implementación en C++ y Objective-C

© Santiago Pavón - UPM-DIT

Conocimientos Previos

Nivel: Básico

© Santiago Pavón - UPM-DIT

Conocimientos Previos

- **Lenguaje C**
  - Repasar las transparencias del curso Introducción a C.
- **Programación Orientada a Objetos**
  - Repasar la transparencia del curso de Programación en Java.
    - clases
    - objetos o instancias
    - atributos o variables de instancia o clase
    - métodos de instancia o clase
    - encapsulación
    - composición
    - herencia
    - polimorfismo

© Santiago Pavón - UPM-DIT

Clases y Objetos

Nivel: Básico

© Santiago Pavón - UPM-DIT

Clases y Objetos

- La Clase es la plantilla para crear objetos.
  - Parte pública en ficheros .h
  - Parte privada en ficheros .m
- En las clases crearemos constructores, propiedades, métodos de instancia, métodos de clase, etc.
- Los objetos son instancias de las clases.
  - El estado de un objeto se almacena en sus propiedades.
    - Las propiedades son variables de instancia privadas a las que se accede con métodos getters y setters.
  - El comportamiento del objeto se implementa en sus métodos.
- Las clases también son objetos.

© Santiago Pavón - UPM-DIT

Interface de la Clase **Punto**

Punto.h

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

 @interface Punto : NSObject

 // Metodos de acceso
 - (void) setX:(float)value;
 - (void) setY:(float)value;
 - (float) x;
 - (float) y;

 // Otros metodos
 - (void) print;
 @end
```

© Santiago Pavón - UPM-DIT

Notas

- **`#import <Foundation/Foundation.h>`**
  - Importar la cabecera del framework Foundation.
  - #import evita incluir ficheros ya incluidos.
  - < y > indican fichero de sistema.
  - Nuevo en iOS7: Nueva directiva **`@import`** para importar un framework:

```
@import Foundation;
```

- **`@interface Punto : NSObject .... @end`**
  - El interface de la clase Punto.
  - Entre las directivas @interface y @end.
  - Punto es el nombre de la clase.
  - NSObject es la superclase de Punto.

- **`- (void) setX:(float)value;`**
  - Declaración del método de instancia setX:.
    - Los dos puntos indican presencia de un argumento.
  - void indica que el método no devuelve nada.
    - El tipo de retorno del método entre paréntesis.
  - value es el argumento del método.
  - El tipo del argumento es float.
    - Entre paréntesis.

- **`- (float) x;`**
  - Declaración del método de instancia x.
  - Devuelve un float.
    - Entre paréntesis.
  - No tiene argumentos.
    - No hay dos puntos detrás del nombre del método.

- La declaración de los métodos de instancia empieza con un menos (-).

- En el fichero **Punto.h**.

© Santiago Pavón - UPM-DIT

Implementación Clase **Punto**

Punto.m

```
#import "Punto.h"

@implementation Punto {
    float x;
    float y;
}

- (void) setX:(float)value {x = value;}
- (void) setY:(float)y {self->y = y;}

- (float) x {return x;}
- (float) y {return y;}

- (void) print {
          NSLog(@"Punto = (%f,%f).\n",x,y);
}
@end
```

© Santiago Pavón - UPM-DIT

Notas

- `@implementation Punto ... @end`
    - La implementacion entre las directivas @implementation y @end.

- ```
  { float x;
    float y;
  }
  ```
    - Declaración de las variables de instancia después del nombre de la clase.
    - x e y son variables de instancia de tipo float.
    - A las ivar se puede acceder directamente por su nombre, o como si fueran los miembros de una struct.

- `- (void) setX:(float)value {x = value;}`
    - Añadir un bloque con la implementación de los métodos.

- `- (void) setY:(float)y {self->y = y;}`
    - Los nombres de los argumentos pueden ser distintos a los declarados en la interface.
        - usamos self->y para acceder a la variable de instancia y.
        - usamos y para acceder al argumento y.

- En el fichero **Punto.m**.

© Santiago Pavón - UPM-DIT
  ```

Interface de la Clase **Circulo**

Circulo.h

```
#import "Punto.h"

@interface Circulo : NSObject

- (void) setCenter:(Punto*)c;
- (void) setRadius:(float)r;

- (Punto*) center;
- (float) radius;

- (void) setCenter:(Punto*)c
         andRadius:(float)r;
- (float) area;

//  . . .

@end
```

© Santiago Pavón - UPM-DIT

Notas

- Métodos de acceso setters:
  - `- (void) setCenter:(Punto*)c;`
    - **setCenter:** es un método de instancia que toma como argumento un puntero al nuevo objeto centro.
    - Prestad atención al nombre del método y al uso de los dos puntos para indicar la presencia de un argumento.
      - c es un puntero a un objeto **Punto**.
        - En Objective-C los objetos siempre se manejan con punteros.
  - `- (void) setRadius:(float)r;`
    - **setRadius:** es un método de instancia que toma como argumento el nuevo valor del radio.
    - Prestad atención al nombre del método y al uso de los dos puntos para indicar la presencia de un argumento.
      - r es un float.

- Métodos de acceso getters:
  - `- (Punto*) center;`
    - El método de acceso **center** devuelve un puntero a un objeto **Punto**.
  - `- (float) radius;`
    - El método de acceso **radius** devuelve un float con el valor del radio.

- Otros métodos:
  - `- (void) setCenter:(Punto*)c andRadius:(float)r;`
    - **setCenter:andRadius:** es un método de instancia con dos argumentos.
    - Prestad atención al nombre del método y al uso de los dos puntos para indicar la presencia de argumentos.
      - c es un puntero a un objeto **Punto**.
        - En Objective-C los objetos siempre se manejan con punteros.
      - r es un float.
  - `- (float) area;`
    - El método **area** devuelve el area del circulo.

© Santiago Pavón - UPM-DIT

Notas

- En algunas ocasiones no es necesario importar un fichero .h, sino solamente declarar qué es un determinado identificador.
  - Para indicar que un nombre es una clase podemos usar la directiva **@class**.
  - Para indicar que un nombre es un protocolo podemos usar la directiva **@protocol**.
  - Ejemplo: **@class Punto;**
    - Declara que **Punto** es el nombre de una clase. No proporcionan detalles sobre **Punto**, solo se declara que es el nombre de una clase.

© Santiago Pavón - UPM-DIT

Implementación Clase **Circulo**

Circulo.m

```
#import "Circulo.h"
#import <math.h>

@implementation Circulo {
     Punto* center;
     float radius;
}
- (void)setCenter:(Punto*)c { center = c; }
- (void)setRadius:(float)r { radius = r; }
- (Punto *) center { return center; }
- (float) radius { return radius; }

- (void)setCenter:(Punto*)c
        andRadius:(float)r {
     center = c;
     radius = r;
}
- (float) area { return M_PI * radius * radius; }

// . . .
@end
```

© Santiago Pavón - UPM-DIT

Notas

- La clase **Circulo** tiene dos variables de instancia.
  - **`center`** y **`radius`**

- **`Punto* center;`**
  - **center** es un puntero a un objeto **Punto**.
  - En Objective-C los objetos siempre se manejan con punteros.

- **`float radius;`**
  - el radio del círculo es un **float**.

- Se incluye/importa el fichero de cabecera **`math.h`** para usar el define **`M_PI`**.

© Santiago Pavón - UPM-DIT

Usando Objetos

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

[p setX:1];
[p setY:2.6];

float cx = [p x];

[p print];

Circulo * c = [[Circulo alloc] init];

[c setCenter:p andRadius:5];

float cy = [[c center] y];
```

Entre corchetes para enviar mensajes o invocar métodos

Métodos setter

Método getter

Pido memoria para un nuevo objeto (rellena de ceros)

Enviar el mensaje print al objeto

Inicializo variables del objeto

Mensaje setCenter:andRadius:

Métodos getter

© Santiago Pavón - UPM-DIT

Notas

- **new** es un método de clase para crear un objeto.
  - Pide memoria y la inicializa.
  - El método **new** simplemente llama a los métodos **alloc** e **init**.
    - **alloc** es un método de clase que pide memoria.
    - **init** es un método de instancia que inicializa el objeto.
  - Normalmente se suelen usar las llamadas a **alloc** e **init**.

- La sintaxis para llamar a un método es poner entre corchetes el objeto y el método a invocar.
  - Si el método tiene argumentos, estos se colocan entrelazados con el nombre del método después de cada dos puntos.

- Son sinónimos:
  - invocar un método de un objeto.
  - enviar un mensaje a un objeto.

- Los métodos de acceso se nombran así:
  - **Setter**: el prefijo **set** seguido del nombre de la ivar con las primeras letras de cada palabra en mayúsculas.
  - **Getter**: el mismo nombre que la ivar.
    - Ejemplo: En la clase **Punto** los método de acceso a la ivar x se llaman **setX:** y x.

© Santiago Pavón - UPM-DIT

Llamar Métodos / Mensajes

Sin argumentos.
No hay dos puntos.

```
[objeto mensaje]
```

Un argumento.

```
[objeto mensaje:argumento]
```

```
[objeto mensaje:arg1 yMas:arg2]
```

Dos argumentos.

© Santiago Pavón - UPM-DIT

Métodos de instancia y clase

- Declaración de métodos de instancia:

```
- (float) x;
- (void) print;
- (void) setCenter:(Punto*)c
         andRadius:(float)r;
```

Declaración empieza con -

- Declaración de métodos de clase:

```
+ (instancetype) new;
+ (Punto*) origen;
```

Declaración empiezan con +

© Santiago Pavón - UPM-DIT

Métodos de instancia y clase

- Invocar métodos de instancia:

```
[circulo setCenter:p andRadius:r];
punto = [circulo center];
```

Estos mensajes se envían al objeto apuntado por la variable circulo

- Invocar métodos de clase:

```
punto = [Punto new];
[[punto class] polarMode];
```

El mensaje se envía a la clase Punto

El mensaje se envía a la clase del objeto apuntado por la variable punto.
Puede ser de la clase Punto o una clase derivada.

© Santiago Pavón - UPM-DIT

Notación Punto

- Otra forma de llamar a los métodos de acceso.

```
float r = [c1 radius];
float r = c1.radius;

[c1 setRadius:10];
c1.radius = 10;
```

Idéntico pero con notación punto

Idéntico pero con notación punto

  - Se llama al método getter o setter dependiendo de si la expresión con notación punto está o no a la izquierda de una asignación.
    - A la izquierda de una asignación: El nombre del método setter invocado se forma anteponiendo el prefijo **set** al nombre usado después del punto empezando con mayúscula, y dos puntos. Es decir, **setRadius:**.
    - En otro sitio: El nombre del método getter invocado coincide con el nombre usado después del punto. Es decir, **radius**.

- La notación punto también puede usarse para invocar otros métodos que no sean de acceso.
  - Se siguen las misma reglas para calcular el nombre del método invocado.

© Santiago Pavón - UPM-DIT

¿Dónde declarar las ivar?

- En los ejemplos anteriores las variables de instancia se han declarado en la implementacion de la clase, es decir, después de @implementation en el fichero .m.
- También se pueden declarar:
  - en la interface de la clase, en el fichero .h, después de @interface.
    - Esto es una mala idea. Es mejor declararlas en el .m para que sean privadas y no se vean los detalles de la implementación.
  - en una extensión de clase.
    - Es una interface privada dentro del .m.

© Santiago Pavón - UPM-DIT

- Desde la versión 3 del compilador LLVM (incluida en Xcode 4.2) se permite declarar las variables de instancia:
    - en extensiones de clase:

```
@interface UnaClase () {
    NSString *name;
}
```

    - en la implementación:

```
@implementation UnaClase {
    NSString *name;
}
```

© Santiago Pavón - UPM-DIT

Notas

Esto es antiguo y no se usa:

- Si se declaran las ivar en la interface de la clase, pueden usarse las siguientes directivas para controlar su visibilidad:
  - **@private**
    - Limita el ámbito de la variable de instancia a la clase que la declara.
  - **@protected**
    - Limita el ámbito de la variable de instancia a la clase que la declara y a las clases derivadas.
    - Por defecto son protected.
  - **@public**
    - Todos pueden acceder a la variable de instancia.
- Recomendación: Usar ivars sólo en las implementaciones (.m) para que sean privadas.

```
@interface Ejemplo : NSObject {
   int uno;
@private
   int dos;
   int tres;
@protected
   int cuatro;
@public
   int cinco;
}
. . .
```

Privadas: dos y tres
Protegidas: uno y cuatro
Públicas: cinco

© Santiago Pavón - UPM-DIT

self

- Es una variable.
- Creada implícitamente en todos los objetos.
- Contiene una referencia al propio objeto, al objeto que se le ha enviado el mensaje que se está ejecutando:

```
self.radius = 33;
[self print];
```

- Muy importante: no es lo mismo:

```
self.radius = 33;   // Llamo al método setter
radius = 33;        // Asigno directamente a la ivar
```

- En un método de clase, **self** apunta al objeto clase.
  - Las clases también son objetos. Son instancias de **Class**.

© Santiago Pavón - UPM-DIT

super

- En un **<u>método de instancia</u>**:
  - Apunta a la implementación de la que deriva un objeto.
    - Permite acceder a la implementación de los métodos de la superclase.
- En un **<u>método de clase</u>**:
  - Apunta a la clase de la superclase
    - Permite acceder a la implementación de los métodos de clase de la superclase.

© Santiago Pavón - UPM-DIT

Puntero Nulo: `nil`

- Valor usado para indicar que una referencia puntero a objeto apunta a nada.
  - es un cero.
- Todas las ivars se crean inicializadas a cero
  - por tanto, las ivars de tipo puntero a objeto están inicializadas a **nil** cuando se crean.
- Cuando se envía un mensaje a **nil** no se ejecuta nada.
  - Si el método devuelve un valor, se devuelve cero.
  - Si el método devuelve un struct, el valor devuelto no está definido.

© Santiago Pavón - UPM-DIT

Usando **nil**

```
Punto* punto;
circulo.center = nil;
[circulo setCenter:nil];
if (punto == nil) { algo }
if (punto) { algo }
if (!punto) { algo }
circulo = nil;
punto = [circulo center];
```

La variable creada apunta a **nil**.

Asignar a **nil**.

Asignar a **nil**.

Se ejecuta **algo**

No se ejecuta **algo**

Se ejecuta **algo**

Asignar a **nil**.

Como **circulo** es **nil**, se devuelve **nil**.
Enviar mensajes a **nil** no ejecuta nada.

© Santiago Pavón - UPM-DIT

Comparar Objetos

- Igualdad de punteros con **==** .

```
if (obj1 == obj2) { ... }
```

- Comparar contenidos usando métodos:
  - Comparar el valor de los atributos.

```
if ([obj1 isEqual:obj2]) { ... }
```

© Santiago Pavón - UPM-DIT

El método `-description`

- Devuelve un **NSString** que representa al objeto.
- Método heredado de **NSObject**

```
- (NSString*) description;
```

- Se llama al usar `%@` en una cadena de formato:

```
NSLog(@"Punto = %@",p);
```

© Santiago Pavón - UPM-DIT

Tipo Booleano: BOOL

- Tipo lógico: **BOOL**
- Las valores son dos defines: **YES** (1) y **NO** (0).

```
BOOL nuevo = YES;
if (nuevo) {…}
if (nuevo != NO) {…}
```

© Santiago Pavón - UPM-DIT

El tipo Selector: **SEL**

- Los valores del tipo **SEL** son identificadores de métodos.
  - Es similar a un puntero a función.
- La directiva **@selector** se usa para crear valores literales.

```
SEL accion1 = [boton action];
```

Variable de tipo SEL

Devuelve un valor SEL

Crea un valor de tipo SEL que se refiere al método **centrar:**

```
SEL accion2 = @selector(centrar:);
SEL accion3 = @selector(subir);
```

subir no tiene parámetros

centrar: tiene un parámetro

```
[boton setAction:@selector(go:)];
```

Asigna el selector a enviar cuando pulsa el botón

© Santiago Pavón - UPM-DIT

Herencia

- Objective-C tiene herencia simple.
- Cosas que ya debemos conocer de la POO:
  - Usos de la herencia:
    - organizar jerárquicamente, compartir código, modificar código existente, añadir nuevo código.
  - Términos:
    - superclase, subclase, padre, hijo, reemplazar métodos (override), . . .
  - **super**
    - para llamar a los métodos de la superclase.

© Santiago Pavón - UPM-DIT

La clase **Posicion**

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

@interface Posicion : Punto

 - (void) setAngulo:(float)value;
 - (float) angulo;

@end
```

Extiende la clase **Punto**

Métodos de acceso.

Posicion.h

© Santiago Pavón - UPM-DIT

```
#import "Posicion.h”

@implementation Posicion {
   float angulo;
}

- (void) setAngulo:(float)value {angulo = value;}

- (float) angulo {return angulo;}

- (void) print {
   [super print];
   NSLog(@"Angulo = %f.",self.angulo);
}

@end
```

Posicion.m

© Santiago Pavón - UPM-DIT

Usando **Posicion**

```
Posicion *pos = [[Posicion alloc] init];

pos.x = 1.1;
pos.y = 2.2;
pos.angulo = 3.3;

[pos print];
```

```
2011-01-21 13:33:11.865 Objetive C[3640:a0f] Punto = (1.100000,2.200000).
2011-01-21 13:33:11.866 Objetive C[3640:a0f] Angulo = 3.300000.
```

© Santiago Pavón - UPM-DIT

Tipado Estático

- Los objetos se crean en el heap, y se manejan con punteros.
- Hay dos formas de declarar el tipo de un puntero:
  - **Tipado estático**:

```
Punto* p2;
```

La variable **p2** apunta a un objeto de la clase **Punto**

  - **Sin tipado**, usando **id**:

```
id p1;
```

La variable **p1** puede apuntar a un objeto de cualquier clase. En tiempo de ejecución se resolverá cual es la clase del objeto apuntado en ese momento.

    - Cuidado: **id** ya lleva un asterisco internamente.
- En tiempo de ejecución no hay diferencia entre las dos formas de declarar el tipo de un puntero.
  - El compilador genera exactamente el mismo código.
- Pero en la primera forma, el compilador comprueba los tipos e indica en el IDE donde hay errores de tipado.

© Santiago Pavón - UPM-DIT

Enlazado Dinámico

- El enlazado de métodos es siempre dinámico.
  - En tiempo de ejecución se decide que código hay que ejecutar al enviar un mensaje a un objeto.
  - Primero se mira cual es la clase real del objeto que recibe el mensaje.
    - Esa clase (o sus superclases) deben implementar el método.
  - Y entonces se ejecuta el método.
    - Si la clase (y las superclases) no implementan el método, entonces la aplicación fallará.

© Santiago Pavón - UPM-DIT

```
// Dog y Bird extienden a Animal.

Animal* a;
Dog* d = [[Dog alloc] init];
Bird* b = [[Bird alloc] init];

a = d;     // OK. a y d apuntan a un perro.
[a eat];   // Ejecuto comer del perro.
[d eat];   // Ejecuto comer del perro.
[a bark]; // Error del compilador: Los animales no ladran.
[d bark]; // Ejecuto ladra del perro.

a = b;     // OK. a y b apuntan a un pajaro.
[a eat];   // Ejecuto comer del pajaro.
[b eat];   // Ejecuto comer del pajaro.
[a bark]; // Error del compilador: Los animales no ladran.
[b bark]; // Error del compilador: Los pajaros no ladran.

d = (Dog*)a; // Casting problematico:
             // Creia que a apuntaba a un perro, pero no es asi.
             // Voy a tener problemas mas adelante.
[d eat];     // Esto es perfecto para el compilador. Los perros comen.
             // Pero se ejecutara el metodo comer del pajaro.
[d bark];    // Esto es perfecto para el compilador. Los perros comen.
             // Fallo al ejecutar: el objeto apuntado por d no ladra.
```

© Santiago Pavón - UPM-DIT

```
id i;

i = b;     // OK. i puede apuntar a cualquier clase de objeto.

[i eat];   // OK para el compilador: sabe que el metodo comer existe en algun sitio.
           // Ejecuto el metodo comer de la clase pajaro.

[i bark];  // OK para el compilador: sabe que el metodo comer existe en algun sitio.
           // Fallo al ejecutar: i apunta a un pajaro, y los pajaros no ladran.

[i sleep]; // Error del compilador: El compilador nunca ha visto el metodo dormir.
           //                       El compilador sabe que esto va a fallar.


b = (Bird*) i;   // Este casting no creara problemas.
                 // Estoy seguro de que i apunta a un pajaro.
                 // El compilador no se queja. Se fia de nosotros.


d = (Dog*) i;   // Creia que i apuntaba a un perro.
                // Pero estoy equivocado, era un pajaro.
                // El compilador no se queja. Se fia de nosotros.
                // Tendre problemas al pedirle al objeto apuntado por d que ladre.

[d bark]; // Fallo al ejecutar: el objeto apuntado por d no tiene el metodo ladrar.
```

© Santiago Pavón - UPM-DIT

Instrospección

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

Class laclase = [p class];

NSString * cnombre = [p className];

BOOL b1 = [p isKindOfClass:[Punto class]];

BOOL b2 = [p isMemberOfClass:[Punto class]];
```

¿es una instancia de esa clase o de una superclase?

¿es una instancia de esa clase? (sin mirar herencia)

© Santiago Pavón - UPM-DIT

Consultar si el objeto responde a ese selector

```
[punto respondsToSelector:accion1];

[punto performSelector:accion3];
[punto performSelector:accion2
              withObject:arg];
```

Enviar (ejecutar) el selector guardado en accion3 al objeto punto

El selector enviado tiene un argumento

Enviar el selector a los objetos de un NSArray

```
NSArray *a = ???;
[a makeObjectsPerformSelector:accion3];
[a makeObjectsPerformSelector:accion2
                     withObject:arg];
```

Selector con un argumento

© Santiago Pavón - UPM-DIT

Crear Objetos

- Un objeto nuevo se crea con los métodos **+alloc** y **-init**.
- **+alloc** es un método de clase de **NSObject**.
  - obtiene la memoria necesaria (en el heap).
    - inicializa ivars con ceros/nil.
- **-init** es un método de instancia heredado de **NSObject**.
  - principalmente usado para inicializar atributos.
    - y otras tareas de inicialización que necesitemos hacer.
  - pueden existir varios métodos de inicialización.
    - sus nombres deben comenzar con el prefijo init.
    - pueden tener parámetros.

```
s = [[NSString alloc] initWithFormat:@"%f/%f",x,y];
```

- **+new** = **+alloc** e **-init**
  - no suele usarse **new**, suele usarse la pareja **alloc** + **init**

```
Circulo *c = [[Circulo alloc] init];
```

© Santiago Pavón - UPM-DIT

¿Cómo debe ser el método **init**?

El tipo devuelto es **instancetype**

Inicialización del padre

[super init] podría devolver **nil** o un puntero de otra clase. Lo asignamos a **self**.

Apple recomienda no usar métodos getter aquí. Recomienda asignar valores directamente a las ivars. La razón es por si usamos algo que aun no se ha inicializado.

```
@implementation Circulo

- (instancetype) init {
    if (self = [super init]) {
        _radius = 1;
        _center = [[Punto alloc] init];
    }
    return self;
}

@end
```

Inicialización del objeto creado

Devolvemos **self**

- **instancetype** indica al compilador que el valor devuelto por el método es una instancia de la clase a la que pertenece el método.

© Santiago Pavón - UPM-DIT

Múltiples métodos **init**

```
- (instancetype) init {
    return [self initWithRadius:1
                  andWithCenter:[[Punto alloc] init]];
}
```

Llamamos al inicializador **designado** usando self.

Devolvemos **self**, que es el valor devuelto por el inicializador designado.

```
- (instancetype) initWithRadius:(float)r {
    return [self initWithRadius:r
                  andWithCenter:[[Punto alloc] init]];
}

- (instancetype) initWithRadius:(float)r
                  andWithCenter:(Punto*)c {
      if (self = [super init]) {
         _radius = r;
         _center = c;
      }
      return self;
 }
```

Este es el **inicializador designado**. Suele ser el método **init** con más parámetros. Sigue el patrón de código visto anteriormente. Los demás métodos **init** se apoyan en él.

Llamamos a **init** o al inicializador **initXXX designado** en la clase padre.

© Santiago Pavón - UPM-DIT

Múltiples métodos **init**

```
- (instancetype) initWithColor:(Color*)color {
      self = [self init];
      self.color = color;
      return self;
 }
```

Inicializamos con alguno de los métodos init que ya tenemos, seguramente usaremos el inicializador designado.

Devolvemos el valor devuelto por el inicializador usado.

Este inicializador inicializa otros elementos. Pero nótese que si **self** es **nil**, esta sentencia no hace nada.

© Santiago Pavón - UPM-DIT

Múltiples métodos **init**

Creando objetos con los distintos inicializadores existentes.

```
Circulo *c2 = [[Circulo alloc] initWithRadius:20
                                andWithCenter:p];

Circulo *c3 = [[Circulo alloc] initWithRadius:30];

Circulo *c4 = [[Circulo alloc] init];

Circulo *c5 = [[Circulo alloc] initWithColor:azul];
```

© Santiago Pavón - UPM-DIT

Otras Formas de Crear Objetos

- Algunas clases tienen métodos de clase (factorias) para crear nuevos objetos:

```
s = [NSString  stringWithFormat:@"%f/%f",x,y];
```

- Crear un objeto nuevo como una copia de uno ya existente.
  - Se usan métodos cuyo nombre empieza por **copy** o **mutableCopy**.
  - Se estudiarán cuando se vean los protocolos **NSCopying** y **NSMutableCopying**.

© Santiago Pavón - UPM-DIT

El Tipo de retorno: `instancetype`

- Introducido en Xcode 5.
- Es un nuevo tipo .
- Pero SOLO puede ser usado como tipo de retorno de un método.
- Se usa para indicarle al compilador que el valor devuelto por el método es una instancia de la clase a la que pertenece el método.

© Santiago Pavón - UPM-DIT

- Con el compilador de Xcode 4 y anteriores, el siguiente código parece correcto y no se muestra ningún aviso/error en el IDE:

```
NSDictionary *d = [[NSString alloc] init];
```

  - Este código parecía correcto porque el método **init** devolvía **id**.
    - No se muestra ninguna queja advirtiendo de la incompatibilidad entre las clases **NSDictionary** y **NSString**
  - Con Xcode 5 ese método devuelve **instancetype**, es decir, indica que devuelve una instancia de **NSString**.
    - El compilador de Xcode 5 detecta el problema y marca el error en el IDE.
- Con Xcode 5 muchos métodos se han cambiado para que devuelvan **instancetype** y así detectar estos fallos en tiempo de compilación.
- Recomendación: hay que usar este tipo en nuestro código desde ya mismo.

© Santiago Pavón - UPM-DIT

Gestión de Memoria

© Santiago Pavón - UPM-DIT

Cuenta de Retenciones

- Para saber cuando debe liberarse la memoria de un objeto Objective-C, se usa un contador de retenciones en el objeto.
- **MRC - Manual Reference Counting**
  - El programador escribe el código adicional para gestionar la memoria de los objetos Objective-C que usa en su programa.
- **ARC - Automatic Reference Counting**
  - Es el compilador el que añade automáticamente el código necesario para gestionar la memoria.
    - Soportado desde iOS 5, compilador LLVM 3.0
  - En iOS no existe recolección automática de basura.
- **Desde Xcode 5 se usa ARC por defecto.**

© Santiago Pavón - UPM-DIT

Automatic Reference Counting

- Con ARC el compilador añade automáticamente las sentencias necesarias (*llamadas a* ***retain***, ***release*** *y* ***autorelease***) para gestionar la memoria.
  - Y el código se ejecuta más eficientemente.
- Aplica sólo para objetos Objective-C.
  - No aplica a malloc/free, clases CoreFoundation (CF), clases CoreGraphics (CG), file open/close, etc.
- ARC sigue los convenios de nombres de Cocoa.
  - Esto permite interoperar con código MRC.

© Santiago Pavón - UPM-DIT

ARC: Tipos de Referencias

- Existen varios tipos de referencias:
  - Referencias **strong**:
    - Mantienen con vida a los objetos que apuntan.
    - La memoria reservada en el heap para alojar a un objeto solo se libera cuando no quedan más referencias strong apuntando al objeto.
      - es decir, no queda ninguna variable local o global, ninguna ivar, ningún parámetro, etc. de tipo apuntado al objeto.
  - Referencias **weak**:
    - No mantienen con vida a los objetos que apuntan.
    - Cuando no quedan referencias strong, la memoria del objeto se libera, y se asigna nil a la referencia weak.
- Otros tipos de referencias que podemos tener son:
  - **unsafe_unretained**
  - **autoreleasing**
- Por supuesto, ARC genera el código adecuado según el tipo de referencia declarado.

© Santiago Pavón - UPM-DIT

Referencias Strong

- Se usa el calificador **__strong**.

```
__strong NSString *name;
```

  - Es la opción por defecto cuando no usamos ningún calificador.

- En la declaración, las variables (parámetros, propiedades, etc.) se inicializan a **nil**.

- Los objetos solo se mantienen en memoria mientras exista alguna referencia strong que los apunte.
  - Cuando no queda ninguna referencia strong, se libera la memoria del objeto.
    - Las referencias a un objeto desaparecen cuando:
      - se asigna un nuevo valor a la variable,
      - o termina el ámbito de la variable.

- Podemos estar tranquilos porque nunca tendremos una referencia apuntado a una zona de memoria que sea basura, o que no nos pertenezca.

© Santiago Pavón - UPM-DIT

Referencias Weak

- Se usa el calificador **__weak**.

```
__weak NSString *name;
```

- En la declaración, las variables (parámetros, propiedades, etc.) se inicializan a nil.

- Las referencias weak **no** retienen la memoria de los objetos apuntados.
  - Cuando no quedan referencias strong apuntando a un objeto:
    - Se libera la memoria del objeto.
    - Se asigna **nil** a las referencia weak que apuntaban al objeto variable.
  - Las variables weak nunca apuntarán a una zona de memoria que ya ha sido liberada, ya que antes se las asigna **nil**.

© Santiago Pavón - UPM-DIT

- Las referencias weak son útiles para evitar bucles de referencias strong.
  - Hay que evitar tener dos objetos padre-hijo reteniéndose mutuamente.

- Insisto, las referencias weak no retienen los valores.

```
__weak id x = [NSObject new];
```

- Tras ejecutar esta sentencia, `x` es `nil`.
  - El objeto se crea, pero nadie lo retiene.
  - Luego se libera la memoria del objeto.
  - y se asigna `nil` a `x`.

© Santiago Pavón - UPM-DIT

- Cuidado porque una referencia weak puede convertirse en nil en cualquier momento.
  - Un thread puede provocar que se destruya el objeto referenciado mientras otro thread lo está usando.
  - El siguiente código no es correcto:

```
__weak MyClass * weakVar = .... ;
if (weakVar) [self method:weakVar.prop];
```

  - Deberíamos apuntar el objeto con una referencia strong y usar esta referencia. Nos aseguramos así de que el objeto está vivo al ejecutar el código.

```
__weak MyClass * weakVar = .... ;
MyClass *strongVar = weakVar;
if (strongVar) [self method:strongVar.prop];
```

(Recordar esto para cuando veamos los bloques)

© Santiago Pavón - UPM-DIT

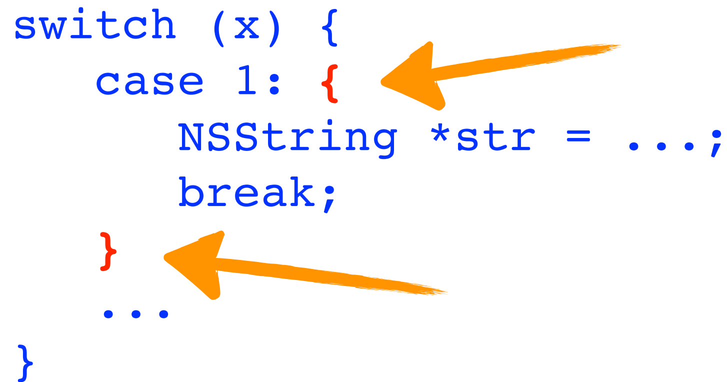
ARC: Reglas de Obligado Cumplimiento

- No podemos ni invocar, ni implementar los métodos: `retain`, `retainCount`, `release`, `autorelease`
- No podemos invocar `dealloc`.
- En los métodos `init`, hay que asignar el resultado de `[super init]` a `self`.
- Las struct/union de C no pueden contener referencias a objetos Objective-C.
- Deben anotarse los casting id <--> void*
- No puede usarse NSAutoreleasePool

© Santiago Pavón - UPM-DIT

- Toll-Free Bridging puede requerir el uso de modificadores.

- Si en el `case` de una sentencia `switch` se declaran variables que apuntan a objetos, debe encerrarse entre llaves las sentencias del case para que el compilador pueda saber cual es el ámbito de las variables declaradas.



```
switch (x) {
    case 1: {
        NSString *str = ...;
        break;
    }
    ...
}
```

- Hay más. Ver *Transitioning to ARC Release Notes*.

© Santiago Pavón - UPM-DIT

Propiedades

© Santiago Pavón - UPM-DIT

Propiedades

- Existen directivas para:
  - declarar e implementar **automáticamente** los **métodos de acceso** a las variables de instancia.
  - y crear **automáticamente** las **variables de instancia** que almacenan los valores.
- El objetivo es escribir menos código
  - podemos escribir nuestra propia versión de los métodos de acceso si los generados automáticamente no nos sirven.

© Santiago Pavón - UPM-DIT

strong: El objeto Punto no se destruye mientras yo lo esté apuntando

Circulo.h

```
@interface Circulo : NSObject {

@property (nonatomic,strong) Punto* center;
@property (nonatomic) float radius;

@end
```

@property: Declarar las propiedades **center** y **radius**

Métodos creados automáticamente: **center**, **setCenter:**, **radius** y **setRadius:**

nonatomic: Acceso a la propiedad no es thread-safe

La ivar creada se llama **_center**

Circulo.m

```
@implementation Circulo

@synthesize center = _center;
@synthesize radius = _radius;

@end
```

@synthesize: Implementar los métodos de acceso a las propiedades, y crear las ivars que almacenan su valor

La ivar creada se llama **_radius**

Actualmente no es necesario escribir las directivas **@synthesize**. Se autogeneran sólas.

© Santiago Pavón - UPM-DIT

@property

- La directiva **@property**

```
@property (nonatomic,strong) Punto* center;
```

- Declara el nombre y el tipo de una propiedad.
  - En el ejemplo, center es un Punto.
- Indica qué métodos de acceso deben implementarse.
  - solo el getter,
  - o los dos, el getter y el setter.
- Indica cómo deben llamarse los métodos de acceso.
- Indica cómo deben implementarse los métodos de acceso.
  - Thread-safe, strong, weak, ...

© Santiago Pavón - UPM-DIT

Opciones de `@property`

- Si se usan, deben meterse entre paréntesis después de @property:

```
@property (nonatomic,strong) Punto* center;
```

- Opción para no proteger acceso desde threads (no thread-safe):

  `nonatomic`

- Opciones de gestión de memoria:

  `copy, strong, weak, assign`

- Opciones para indicar que métodos deben crearse:

  `readwrite, readonly`

- Opciones para cambiar los nombres de los métodos de acceso:

  `getter=nombre, setter=nombre`

© Santiago Pavón - UPM-DIT

Gestión de Memoria

- Opciones para indicar el tipo de referencia en las propiedades:

```
@property (strong) NSString *name;
    // La ivar es strong.
    // No se destruye el objeto mientras existan referencias
    // strong apuntándole.
    // Nota: con ARC “retain” es sinónimo de strong.

@property (weak) NSString *name;
    // La ivar es weak.
    // Si el objeto se destruye, me asignan automáticamente un nil.

@property (copy) NSString *name;
    // Al acceder a la propiedad nos devuelven una copia
    // del valor guardado, y la relación es strong.

@property (assign) float radius;
    // La propiedad se guarda en la ivar realizando una asignación.
```

- **@synthesize** genera ivars usando el tipo de referencia indicado.

© Santiago Pavón - UPM-DIT

@synthesize @dynamic

- **@synthesize**
  - Directiva para implementar los métodos de acceso y declarar la variable de instancia automáticamente.
    - Si ya existe alguno de los métodos, entonces no los crea.
    - Si ya existe, entonces no la crea.

  - **Si no se usa ninguna directiva, el compilador añade esta directiva automáticamente:**

```
@synthesize nombre = _nombre;
```

    - **nombre** es el nombre de la propiedad y la ivar creada se llama **_nombre**.

- **@dynamic**
  - Directiva para no implementar los métodos de acceso, ni crear la variable de instancia.
    - Se proporcionarán de alguna otra forma.

© Santiago Pavón - UPM-DIT

Almacenamiento de la Propiedad

- Las propiedades típicamente suelen ir asociadas a una ivar donde se almacena el valor de la propiedad.
- Pero hay otras posibilidades:
  - El valor de la propiedad se obtiene de varias ivars.
    - Ver el siguiente ejemplo.
  - No existe ivar, y el valor lo genera otro método.
  - ...

© Santiago Pavón - UPM-DIT

```
@interface Circulo : NSObject
@property Punto* center;
@property (readonly) float radius;
@end


@implementation Circulo {
    float xCenter;
    float yCenter;
}

- (float) radius {
    return 5;
}

- (void) setCenter:(Punto*)c {
    xCenter = c.x;
    yCenter = c.y;
}

- (Punto *) center {

    Punto *c = [[Punto alloc] init];
    c.x = xCenter;
    c.y = yCenter;

    return c;
}
```

No existen las ivars **_center** ni **_radius**

ivars privadas donde almaceno el centro

Es readonly. Siempre devuelve 5.

Saco del parámetro c los valores a guardar en las ivars

Construyo un objeto Punto con las coordenadas del centro

© Santiago Pavón - UPM-DIT

Nombres de las ivars

- Si la propiedad **nombre** se sintetiza así:

```
@synthesize nombre;
```

  - entonces la ivar creada se llama **nombre**.

- Si la propiedad **nombre** se sintetiza así:

```
@synthesize nombre = nombre2;
```

  - entonces la ivar creada se llama **nombre2**.

  - Usos:
    - Si ya tenemos una ivar y queremos usarla como respaldo de una propiedad que tiene otro nombre.
    - Evitar equivocaciones usando la ivar cuando lo que se desea es llamar a los métodos de acceso. (ver siguiente transparencia)

© Santiago Pavón - UPM-DIT

No confundir ivars y propiedades

- No es lo mismo:

```
self.center = c;  // Llamo al metodo setter.
center = c;       // Asigno directamente a la ivar.
```

- Un despiste/equivocación provocará terribles problemas en el programa.
- Podemos minimizar riesgos si usamos nombres diferentes para las propiedades y las ivars asociadas. (**MUY RECOMENDABLE**)

```
@synthesize center = _center;
```

  - Si se nos olvida escribir **self.** delante del nombre de la propiedad **center**, aparece un error indicando que **center** es una variable que no existe.
  - Para usar directamente la ivar, escribiremos **_center**.
- NOTA: Ya hemos visto que las directivas **@synthesize** generadas automáticamente usan nombres distintos para la propiedad y la ivar.

© Santiago Pavón - UPM-DIT

Propiedades Privadas

- Las propiedades públicas:
  - declaradas en el fichero .h.
- Las propiedades privadas:
  - declaradas en una extensión de clase dentro del fichero .m

    - color es una propiedad privada.

```
@interface Circulo ()
@property int color;
@end

@implementation Circulo
@end
```

Extensión de clase.
Hay unos paréntesis vacíos

Extendemos la clase Circulo añadiendo la propiedad privada **color**.

Circulo.m

© Santiago Pavón - UPM-DIT

Lectura Pública y Escritura Privada

- Para crear una propiedad con acceso público de lectura
  - Usaremos la opción **readonly** en la interface.
- y con acceso privado para escritura
  - En la implementación, usando una extensión de clase redefinimos la misma propiedad pero usando la opción **readwrite**.

```
@interface Circulo : NSObject
@property (readonly) int color;
@end
```

Circulo.h

```
@interface Circulo ()
@property (readwrite) int color;
@end

@implementation Circulo
@end
```

Circulo.m

© Santiago Pavón - UPM-DIT

¿Propiedades privadas o ivars?

- En la implementación de una clase:
  - ¿Qué es mejor para almacenar el estado?
    - Usar propiedades privadas.
    - Usar variables de instancia directamente.

- Respuesta: Hay gustos para todo.
  - Es más popular usar propiedades.

© Santiago Pavón - UPM-DIT

Casos especiales en los que hay que añadir la directiva **@synthesize** manualmente

- Si la propiedad es **readonly** y se implementa manualmente su método getter
  - No se autogenera la directiva @synthesize
    - Aparecerá un error de compilación porque no se ha creado la ivar.
- Si la propiedad es **readwrite** y se implementan manualmente los métodos getter y setter.
  - Caso idéntico al anterior
- Si la propiedad se ha declarado en un **protocolo**.

© Santiago Pavón - UPM-DIT

Framework: Foundation

© Santiago Pavón - UPM-DIT

Framework Foundation

- Proporciona:
  - Clases
    - colecciones
    - valores
  - Notificaciones
  - Sistema de ficheros
  - Threads
  - Comunicaciones
  - etc.

© Santiago Pavón - UPM-DIT

La Clase **NSObject**

- Clase base en la jerarquía de herencia.
- Implementa:
  - muchos métodos:

    -isEqual -description …
  - gestión de memoria:

    +alloc -init -retain -release …

© Santiago Pavón - UPM-DIT

La clase **NSString**

- Strings internacionalizados usando Unicode
  - soportar cualquier lenguaje.
- No es un **char*** de C.
- Usar **@""** para crear literales:

```
NSString *s1 = @"Hola Mundo";
```

- Son inmutables.
  - No puede alterarse su contenido.
  - Hay que crear un objeto nuevo con el nuevo contenido.

```
NSString *s2 = [s1 uppercaseString];
```

Devuelve otro objeto

© Santiago Pavón - UPM-DIT

Métodos de **NSString**

```
-(NSString*) stringByAppendingString:(NSString*)string;
-(NSString*) capitalizedString;
-(BOOL) hasPrefix:(NSString*)string;
-(int) intValue;
-(double) doubleValue;
-(unsigned int) length;
-(NSComparisonResult) compare:(NSString*)string;
-(BOOL) isEqualToString:(NSString*)string

+(id) stringWithFormat:(NSString*)format, ...
```

etc ... (consultar documentación)

© Santiago Pavón - UPM-DIT

La Clase **NSMutableString**

- Subclase de **NSString**.
  - Versión mutable.
    - Permite cambiar contenido de los objetos.

- Métodos:

```
+ (id) string;

- (void)appendString:(NSString*)string;

- (void)appendFormat:(NSString*)format, …;
```

etc...

© Santiago Pavón - UPM-DIT

NSArray y NSMutableArray

- NSArray
  - Lista ordenadas de objetos Objective-C.
    - No puede contener tipos primitivos de C.
    - No puede contener nil. Usar **NSNull**.
  - Inmutable.
  - Usar `@[,,,]` para crear literales:

nil indica fin de argumentos

```
NSArray *a = [NSArray arrayWithObjects:@"uno",@"dos",nil];
NSArray *a = @[@"uno",@"dos"];
```

  - Usar el operador indexación `[]` para acceder a los valores:

```
[a objectAtIndex:1]
a[1]
```

- NSMutableArray es la versión mutable.
  - Deriva de **NSArray**.

© Santiago Pavón - UPM-DIT

```
NSArray *a = @[@"uno",@"dos",@"tres"];

NSLog(@"Tamaño = %u", [a count]);
NSLog(@"Segundo = %@“,a[1]);
NSLog(@"Ultimo = %@", [a lastObject]);

NSMutableArray *ma = [a mutableCopy];

[ma addObject:@"cuatro"];

NSArray *a2 = [ma copy];
```

© Santiago Pavón - UPM-DIT

NSDictionary y NSMutableDictionary

- NSDictionary
  - Es inmutable.
  - Es una tabla de Hash.
    - Almacena objetos asociados a una clave.
      - La clave también es un objeto.
    - No pueden existir dos valores con la misma clave.
    - Las claves se comparan con **-isEqual:**.
  - Ni la clave ni el valor pueden ser **nil** (usar **NSNull**), ni tipos primitivos de C.
  - Usar `@{key:value, key:value, key:value}` para crear literales:

```
NSDictionary *d = [NSDictionary dictionaryWithObjectsAndKeys:
                                 @"one",@"uno", @"two",@"dos", nil];
NSDictionary *d = @{@"uno":@"one", @"dos":@"two"};
```

nil indica fin de argumentos

  - Usar el operador indexación `[]` para acceder a los valores:

```
[d objectForKey:@"dos"];
d[@"dos"]
```

- NSMutableDictionary es la versión mutable.

© Santiago Pavón - UPM-DIT

```
// Clave: palabra en español
// Valor: traducción al inglés
NSDictionary *d = @{@"uno":@"one",
                    @"dos":@"two"};

NSString *esp = @"dos"
NSString *ing = d[esp];

NSLog(@"%@ en inglés es %@\n", esp, ing);
```

© Santiago Pavón - UPM-DIT

NS[Mutable][Ordered]Set

- Colección de objetos no repetidos.
- **Mutable**: Versión mutable.
- **Ordered**: Versión con valores ordenados.
  - buscar elementos más rápidamente.

```
NSMutableSet *ms = [NSMutableSet setWithCapacity:5];
[ms addObject:@"uno"];
[ms addObject:@"dos"];
[ms removeObject:@"dos"];

NSLog(@"Tamaño = %u", [ms count]);
NSLog(@"Uno cualquiera = %@", [ms anyObject]);
```

© Santiago Pavón - UPM-DIT

Más Clases de Foundation

- **NSNumber**
  - Envoltorio de tipos primitivos numéricos, con o sin signo (short, int, float, double, long long int, BOOL, ...).

```
NSNumber *n = [NSNumber numberWithFloat:0.5];
float f = [n floatValue];
```

  - Literales para **NSNumber**.

```
NSNumber *n = @123;  // @5.5 @'a' @33U @7L @YES @NO @false
```

- **NSValue**
  - Envoltorio de cualquier tipo de dato (números, punteros, structs, objetos, etc...).
  - Existen varias categorías que añaden métodos extra a la clase.

```
NSValue *v = [NSValue valueWithCGRect:CGRectMake(0,0,10,10)];
CGRect r = [v CGRectValue];
```

- **NSNull**
  - Objeto singleton.
  - Envoltorio para crear un objeto que represente nil.

```
[NSNull null];
```

- **NSData, NSMutableData**
  - Buffer de bytes.

© Santiago Pavón - UPM-DIT

- **`NSDate, NSCalendar, NSDateFormatter, NSDateComponents`**
  - Fechas y horas.
- **`NSRegularExpression`**
  - Expresiones regulares.
- **`NSOperation, NSOperationQueue, ...`**
  - Operaciones concurrentes.
- **`NSCache`**
  - Colección de pares clave-valor similar a un NSDictionary.
  - Puede liberarse la memoria de los objetos que contiene si hay escasez de memoria.
- **`NSUserDefaults`**
  - Preferencias de usuario.
- **`NSProgress`**
  - Informes de progreso.
- . . .

© Santiago Pavón - UPM-DIT

Recorridos

- Recorrido clásico:

```
for (int i=0 ; i<[a count] ; i++)
    NSLog(@"Pos %i = %@",i,a[i]);
```

- Recorrido con un enumerador:

```
NSEnumerator *en = [a objectEnumerator];
NSString * sa;
while (sa = [en nextObject])
    NSLog(@"valor = %@",sa);
```

© Santiago Pavón - UPM-DIT

- Enumeración rápida:

```
NSArray* arr = ???;
for (NSString *s in arr)
    NSLog(@"valor = %@",s);


NSDictionary *dic = ???;
for (id key in dic) {
   NSLog(@"key=%@ valor=%@",key, dic[key]);
}
```

© Santiago Pavón - UPM-DIT

- Enumeración usando un bloque:

```
NSArray* arr = ???;
[arr enumerateObjectsUsingBlock:
    ^(id obj, NSUInteger idx, BOOL *stop) {
        NSLog(@"valor en %d = %@", idx, obj);
    }];


NSDictionary *dic = ???;
[dic enumerateKeysAndObjectsUsingBlock:
    ^(id key, id obj, BOOL *stop) {
        NSLog(@"key=%@ valor=%@",key,obj);
    }];
```

© Santiago Pavón - UPM-DIT

Literales

- En Xcode 4.5 (LLVM 4.0, clang 3.1) se han introducido nuevos literales para Objective-C
  - Son mejoras del compilador y pueden usarse en iOS 5 y anteriores.
  - Se pueden incorporar a un proyecto antiguo ejecutando:
    - **Menú Edit > Refactor > Convert To Modern Objective-C**
  - Detalles en:
    - **http://clang.llvm.org/docs/ObjectiveCLiterals.html**

- Literales para **NSNumber**.

```
NSNumber *n = @123;  // @5.5 @'a' @33U @7L @YES @NO @false
```

- Literales para **NSArray** y **NSDictionary**.

```
NSArray *a = @[a, b, c];
NSDictionary *d = @{ key1:value1, key2:value2 };
```

© Santiago Pavón - UPM-DIT

- Literales con **Boxed Expressions**:
  - Sintaxis: `@(`*`expresion`*`)`
    - La expresión debe ser un valor escalar (número, enumerado, BOOL) o un puntero a un string de C (terminado en \0 y codificado en UTF-8).
  - Ejemplos:

```
NSNumber *piMedios = [NSNumber numberWithDouble:(M_PI/
2)];
NSNumber *piMedios = @(M_PI/2);

typedef enum { Red, Green, Blue } Color;
NSNumber *miColor = [NSNumber numberWithInt:
((int)Green)];
NSNumber *miColor = @(Green);

NSString *path =
      [NSString stringWithUTF8String:(getenv("PATH"))];
NSString *path = @(getenv("PATH"));
```

© Santiago Pavón - UPM-DIT

- Operador de indexación `[]` para arrays, diccionarios o cualquier clase
    - **Indexación al estilo array** - El índice en un valor entero

```
NSMutableArray *array = ...;
id obj1 = array[5];
array[7] = obj2;
```

- Se implementa llamando a los métodos:

```
- (id) objectAtIndexedSubscript:(NSUInteger)idx;

- (void) setObject:(id)obj
         atIndexedSubscript:(NSUInteger)idx;
```

- **Indexación al estilo diccionario** - El índice es un objeto Objective-C

```
NSMutableDictionary *dictionary = ...;
id obj1 = dictionary[@"name"];
dictionary[@"name"] = obj2;
```

- Se implementa llamando a los métodos:

```
- (id) objectForKeyedSubscript:(id)key;

- (void) setObject:(id)obj
         forKeyedSubscript:(id<NSCopying>)key;
```

© Santiago Pavón - UPM-DIT

Listas de Propiedades

- Estructura de datos formada sólo por:
  - **`NSArray`**, **`NSDictionary`**, **`NSNumber`**, **`NSString`**, **`NSDate`** o **`NSData`**.
    - o por subclases de estos.
- Existen muchos métodos para trabajar con listas de propiedades.
  - Ejemplo: Los objetos **`NSArray`** y **`NSDictionary`** pueden guardarse y cargarse de ficheros **.plist** con una sentencia cuando son listas de propiedades:

```
[miArray writeToFile:@"datos.plist" atomically:YES];
```

© Santiago Pavón - UPM-DIT

Categorías

© Santiago Pavón - UPM-DIT

Categorías

- Definir nuevos métodos en clases ya existentes sin tener que crear subclases.
    - Si hay conflictos entre los nombres usados en la clase original y los añadidos en la categoría:
        - gana la categoría.
            - Se pierde el método original para siempre.
                - Redefinir los métodos de una clase en una categoría se considera una mala práctica de programación. Es mejor crear una subclase.
    - Si un método se define en más de una categoría, el lenguaje no especifica cual se usará.
    - No es obligatorio implementar todos los métodos definidos en una categoría.

- No pueden añadirse variables de instancia.

© Santiago Pavón - UPM-DIT

Categoría llamada: **Punto+Metrica**

```
@interface Punto (Metrica)
-(float)distancia:(Punto*)p;
@end
```

Punto+Metrica.h

```
@implementation Punto (Metrica)
-(float)distancia:(Punto*)p {
    return sqrt(pow(x-p.x,2) +
                pow(y-p.y,2));
}
@end
```

Punto+Metrica.m

© Santiago Pavón - UPM-DIT

Usando **Punto+Metrica**

```
#import "Punto+Metrica.h"

Punto *punt1 = [[Punto alloc] init];
punt1.x = 1;
punt1.y = 1;
Punto *punt2 = [[Punto alloc] init];
punt2.x = 4;
punt2.y = 5;

NSLog(@"Distancia = %0.2f\n",
    [punt1 distancia:punt2]);
```

© Santiago Pavón - UPM-DIT

Usos de Categorías

- Extender una clase con más métodos.
- Organizar clases grandes:
  - agrupando métodos relacionados en categorías.
  - dividir la implementación de una clase en varios ficheros.
- Crear propiedades y métodos privados.
- Declarar de forma privada los protocolos a implementar.
- Crear referencias forward a métodos privados.
  - es decir, declarar el método antes de usarlo.
    - Desde Xcode 4.3 no es necesario crear declaraciones forward para los métodos Objective-C, pero sigue siendo necesario para las funciones C.
- Añadir protocolos informales a un objeto.

© Santiago Pavón - UPM-DIT

Extensiones de Clases

- Las extensiones de clase son como categorías, pero con poderes especiales:
  - en ellas pueden definirse propiedades y variables de instancia.
  - también pueden redefinirse las propiedades existentes.
    - convertir en readwrite una propiedad declarada como readonly en la interface.
- Se declaran igual que una categoría pero:
  - Sin poner un nombre entre los paréntesis.
  - Sólo pueden declararse dentro del mismo fichero .m donde se implementa la clase.
    - No se pueden implementar en otra categoría.
  - Los métodos que definen deben implementarse en el bloque @implementation de la clase que extienden.

© Santiago Pavón - UPM-DIT

En una extensión de clase se ponen los paréntesis vacíos

```
@interface Punto () {
    int _border;
}
@property (strong) base;
-(void)limpiar;
@end
```

Declaración de una ivar

Declaración de una propiedad

Actualización: en las últimas versiones no son necesarias las declaraciones forward.

```
@implementation Punto
    -(void)limpiar {
     x = y = 0;
    }
@end
```

Implementado dentro del bloque **@implementation** de la clase, no en el de una categoría.

© Santiago Pavón - UPM-DIT

Protocolos

© Santiago Pavón - UPM-DIT

Protocolo

- Un protocolo es una declaración de los métodos que alguien debe implementar.
  - El protocolo sólo los declara, no los implementa.
    - Parecido a una interface de java.
  - Los implementan las clases que adopten el protocolo.
    - Ejemplo: protocolos para definir delegados y data sources.
  - Los protocolos se definen en los ficheros de cabeceras .h
    - Los ficheros que adoptan (implementan) el protocolo incluirán los ficheros de cabeceras.
- Las clases adoptan el protocolo:
  - nombrándolo en su declaración @interface entre **<** y **>**.
  - e implementando los métodos del protocolo.
- Pueden adoptarse varios protocolos.
  - nombrarlos separados por comas.

© Santiago Pavón - UPM-DIT

- Usar **@optional** y **@required** para indicar que métodos deben implementarse forzosamente y cuales son opcionales.
  - los requeridos son de obligada implementación.
  - los opcionales sólo si se desea.
  - Por defecto, son required.
- Consultar si se ha adoptado un protocolo:

```
[obj conformsToProtocol:@protocol(nombre)]
```

© Santiago Pavón - UPM-DIT

Declarar un Protocolo

```
@protocol Escalable

-(void) aumentar:(float)factor;
-(void) reducir:(float)factor;

@optional

-(void) duplicar;

@required

@property (nonatomic,assign) float escala;

@end
```

© Santiago Pavón - UPM-DIT

Adoptar un Protocolo

- En la interface de la clase:
  - Declaramos públicamente que la clase adopta el protocolo:

```
@interface Figura : NSObject <Escalable>
  . . .
@end
```

- En una extensión de clase:
  - La adopción del protocolo es un asunto privado de la clase, y no se declara públicamente a otros:

```
@interface Figura () <Escalable>
  . . .
@end
```

- En la implementación de Figura se implementarán los elementos obligatorios:

```
@synthesize escale = _escala;
-(void) aumentar:(float)factor {. . .}
-(void) reducir:(float)factor {. . .}
```

- La implementación de los elementos @optional es opcional:

```
-(void) duplicar {. . .}
```

© Santiago Pavón - UPM-DIT

id<*protocolo*>

- Podemos declarar variables que sean de cualquier tipo de clase (es decir de tipo **id**) pero que sean conformes a un protocolo(s):

```
@property (strong) id<Ayudante> ayudante;
id<Ayudante> otro_ayudante;
-(void)setAyudante:(id<Ayudante>)ayudante;
```

- Sirve para que el compilador compruebe tipos y muestre warnings de aviso.
  - No tiene ningún efecto en tiempo de ejecución.
- También es una forma de documentar el código.

© Santiago Pavón - UPM-DIT

Ejemplo: ¿Quién hace las tareas para el jefe?

```
@protocol Ayudante
-(void)hacerTarea:(Tarea)t;
@end
```

Apunta a cualquier objeto que sea conforme con el protocolo Ayudante. El compilador se quejará si el objeto apuntado no es conforme con Ayudante.

```
@interface Jefe : NSObject
@property (strong) id<Ayudante> ayudante;
@end
```

```
@interface Empleado:NSObject <Ayudante>
@end

@implementation Empleado
-(void)hacerTarea:(Tarea)t { . . . }
@end
```

© Santiago Pavón - UPM-DIT

Protocolos Formales e Informales

- **Protocolo Informal:**
  - Consiste en crear una categoría sobre NSObject.
    - Cualquier objeto responderá a los métodos definidos en esa categoría.
- **Protocolo Formal:**
  - Es un protocolo normal.

© Santiago Pavón - UPM-DIT

Protocolo **`NSCopying`**

© Santiago Pavón - UPM-DIT

Crear una clase conforme a **NSCopying**

- Si la clase base NO implementa **NSCopying**:

```
@interface Hijo : Padre <NSCopying>
   ...
@end

@implementation Hijo
-(id) copyWithZone:(NSZone*)zone {
   id copia = [[[self class] allocWithZone:zone] init???];
   if (copia) {
      // Añadir aquí las sentencias para copiar
      // los atributos definidos en la clase Padre
   }
   return copia;
}
@end
```

© Santiago Pavón - UPM-DIT

Notas

El método **copy** llama internamente a **copyWithZone**.

zone es la región de memoria de la que tomaremos la memoria necesaria.

Sobre la sentencia:

```
copia = [[[self class] allocWithZone:zone] init???];
```

- No se pone el nombre de la clase (**Hijo**), sino que se pone **[self class]** para usar la clase real del objeto que estamos copiando.
  - Supongamos que creamos una clase derivada de **Hijo**, llamada **Nieto**. Si ejecutamos el método **copy** sobre una instancia de **Nieto**, se ejecutaría la versión del método **copyWithZone** heredada de **Hijo**, que ejecutaría el método **allocWithZone** de la clase **Hijo**, en vez de ejecutar el método **allocWithZone** de la clase real **Nieto**.

Los atributos de la superclase **Padre** se asignan en **init???**.

Luego se copian/retienen/asignan los atributos nuevos creados en la clase **Hijo**.

© Santiago Pavón - UPM-DIT

Crear una clase conforme a **NSCopying**

- Si la clase base **YA** implementa **NSCopying**:

```
@interface Hijo : Padre
   ...
@end

@implementation Hijo
-(id) copyWithZone:(NSZone*)zone {
   id copia = [super copyWithZone:zone];
   if (copia) {
      // Añadir aquí las sentencias para copiar
      // los atributos definidos en la clase Padre
   }
   return copia;
}
@end
```

© Santiago Pavón - UPM-DIT

Otros Casos

- No es mi superclase la que implementa **NSCopying**, sino una clase más arriba en la jerarquía de herencia.
  - Hay que inicializar más atributos.
- Pensad en el tipo de copia que se quiere hacer:
  - en profundidad, en superficie, o mezcla de ambas.
- Si se quiere copiar una clase no mutable, que sólo contiene objetos no mutables:
  - no copiar. Retener el original.

© Santiago Pavón - UPM-DIT

El protocolo **NSMutableCopying**

- Este protocolo declara un método para hacer una copia mutable de un objeto.

```
- (id)mutableCopyWithZone:(NSZone*)zone
```

- Este protocolo lo llama **mutableCopy**.

© Santiago Pavón - UPM-DIT

Protocolo **`NSCoding`**

© Santiago Pavón - UPM-DIT

Protocolo **`NSCoding`**

- Si una clase es conforme a este protocolo, sus objetos pueden serializarse.

```
@protocol NSCoding
    - (void)encodeWithCoder:(NSCoder*)aCoder;
    - (id)initWithCoder:(NSCoder*)aDecoder;
@end
```

- **NSCoder** es una clase abstracta.
  - Define métodos para convertir objetos en **NSData**, e inversa.
  - Algunas subclases:
    - **NSArchiver**, **NSUnarchiver**, **`NSKeyedArchiver`**, **`NSKeyedUnarchiver`**, **NSPortCoder**

© Santiago Pavón - UPM-DIT

Método para Codificar

```
- (void)encodeWithCoder:(NSCoder*)encoder {

    // Si nuestra superclase es conforme a NSCoding,
    // codificamos los campos de la superclase.
    [super encodeWithCoder:encoder];

    // Codificar los campos de la clase:
    [encoder encodeObject:self.nombre
forKey:@"nombre"];
    [encoder encodeObject:self.coche
forKey:@"vehiculo"];
    [encoder encodeInt:self.edad       forKey:@"edad"];
    [encoder encodeFloat:self.otro     forKey:@"mas"];
}
```

© Santiago Pavón - UPM-DIT

Método para Decodificar

- Si derivamos de **NSObject** o de una superclase que no es conforme a **NSCoding**:

```
- (id)initWithCoder:(NSCoder*)decoder {

    if (self = [super init]) {
        self.nombre = [decoder
decodeObjectForKey:@"nombre"];
        self.coche  = [decoder
decodeObjectForKey:@"vehiculo"];
        self.edad   = [decoder decodeIntForKey:@"edad"];
        self.otro   = [decoder decodeFloatForKey:@"mas"];
    }
    return self;
}
```

© Santiago Pavón - UPM-DIT

- Si derivamos de una superclase que es conforme a **NSCoding**:

```
- (id)initWithCoder:(NSCoder*)decoder {

      if (self = [super initWithCoder:decoder]) {
          self.nombre = [decoder
decodeObjectForKey:@"nombre"];
          self.coche  = [decoder
decodeObjectForKey:@"vehiculo"];
          self.edad   = [decoder
decodeIntForKey:@"edad"];
          self.otro   = [decoder
decodeFloatForKey:@"mas"];
      }
      return self;
}
```

© Santiago Pavón - UPM-DIT

Cómo se usa: Codificar

```
// Crear un codificador que guarda los datos en data:
NSMutableData *data =[[NSMutableData] alloc] init];
NSKeyedArchiver *archiver =
   [[NSKeyedArchiver alloc]
       initForWritingWithMutableData:data];

// Codificar varios valores asociandolos a una clave:
[archiver encodeObject:objeto1 forKey:@"clave1"];
[archiver encodeObject:objeto2 forKey:@"clave2"];
[archiver encodeObject:objeto3 forKey:@"clave3"];

// Ya he terminado de codificar:
[archiver finishEncoding];

// Salvar (el buffer NSData) en un fichero:
BOOL ok = [data writeToFile:@"abc" atomically:YES];
```

© Santiago Pavón - UPM-DIT

Cómo se usa: Decodificar

```
// Leer los datos de un fichero:
NSData *data = [[NSData alloc]
                      initWithContentOfFile:@"abc"];

// Crer un decodificador que leer de data:
NSKeyedUnarchiver *unarchiver =
           [[NSKeyedUnarchiver alloc]
                  initForReadingWithData:data];

// Decodificar los valores asociados a cada clave:
self.objeto1 = [unarchiver decodeObjectForKey:@"clave1"];
self.objeto2 = [unarchiver decodeObjectForKey:@"clave2"];
self.objeto3 = [unarchiver decodeObjectForKey:@"clave3"];

// Ya he terminado de decodificar:
[unarchiver finishDecoding];
```

© Santiago Pavón - UPM-DIT

Serializar una Jerarquía de Objetos

- Guardar una jerarquía de objetos en un fichero:
  - Se hace una copia en profundidad

```
UnaClase *obj  = ????
NSString *path = @"miejemplo.save";
BOOL res = [NSKeyedArchiver archiveRootObject:obj
                                       toFile:path];
```

- Recuperar la jerarquía de objetos desde el fichero:

```
UnaClase *obj  = nil;
NSString *path = @"miejemplo.save";
obj = [NSKeyedUnarchiver unarchiveObjectWithFile:path];
```

- Nota: la clase **UnaClase** debe ser conforme a **NSCoding**.

© Santiago Pavón - UPM-DIT

Excepciones

© Santiago Pavón - UPM-DIT

Excepciones

```
@try {

    [excepcion raise];

}
@catch (NSException *e) {

   @throw;
}
@catch (NSString *s) {

}
@finally {

}
```

© Santiago Pavón - UPM-DIT

Blocks

© Santiago Pavón - UPM-DIT

¿Qué es un Block?

- Un bloque es un dato cuyo valor son sentencias.
  - Parametrizable con una lista de argumentos.
  - Puede devolver un valor tras su ejecución.
- Se maneja como si fuera un objeto Objective-C.
  - Por ejemplo, puede guardarse en un array o diccionario.
- Permite crear programas más claros y compactos.
- Un Block puede ser:
  - el argumento de métodos y funciones.
  - el valor devuelto por métodos y funciones.
  - asignable a una variable, que se usará como una función.

© Santiago Pavón - UPM-DIT

Crear un Objeto Block

^ es la marca sintáctica que indica que esto es un block

Sentencias del block entre llaves.

```
^int (int v1,int v2){ return v1+v2; }
```

Tipo de retorno

Usar return para valor devuelto.

Lista de argumentos (con su tipo) del block entre paréntesis. Puede no existir.

© Santiago Pavón - UPM-DIT

- Si el tipo de retorno es void o el compilador puede inferirlo, entonces podemos omitirlo:

```
^(int v1,int v2){return v1+v2;}
```

- Si la lista de argumentos es void, podemos omitirla:

```
^int {return 666;}
```

- Las dos cosas:

```
^{return 666;}
```

© Santiago Pavón - UPM-DIT

Uso con Variables

La variable **suma** es del tipo block, toma dos argumentos **int**, y devuelve un **int**.

Asignamos a suma un valor

```
int (^suma)(int,int) =
              ^(int v1, int v2){return v1+v2;};

int res = suma(100,200);
```

suma se usa como si fuera una función

© Santiago Pavón - UPM-DIT

Uso como Argumento

Este método existe en la clase NSDictionary.
Toma como parámetro un Block.

```
[diccionario enumerateKeysAndObjectsUsingBlock:
    ^(id key, id value, BOOL *stop) {
        NSLog(@“%@ => %@”, key, value);
    }
];
```

El método ejecuta este block para todos los pares key-value del diccionario

Asignaremos **YES** a **stop** para detener el recorrido

© Santiago Pavón - UPM-DIT

Uso con **typedef**

Defino un tipo llamado **suma_t**

```
typedef int (^suma_t)(int v1, int v2);

suma_t suma = ^(int v1, int v2) {
                    return v1+v2;
              };

int res = suma(100,200);
```

© Santiago Pavón - UPM-DIT

No ejecutar bloques **nil**

- No está permitido invocar la ejecución de un bloque que sea nil.
  - Comprobar antes que no sea nil.

```
if (suma != nil) {
    int res = suma(100,200);
    ...
}
```

© Santiago Pavón - UPM-DIT

Ámbito de Ejecución

- Las sentencias del block se ejecutan en el ámbito donde se creó el block.
  - pueden acceder a las variables, parámetros, ivars, funciones, etc. del ámbito donde se creó.

  - **Esto permite ahorrar mucho código.**
    - No tengo que pasar el entorno donde se creó el bloque, al punto donde se va a ejecutar el bloque.

© Santiago Pavón - UPM-DIT

Variables Locales: se copia el valor

Se copia el valor de **factor**, es decir, se copia un 3.

factor es **READ-ONLY** dentro del block. No se permite modificar su valor dentro del block.

```
int factor = 3;

int (^suma2)(int) =
    ^(int v){return v + factor;};
```

```
factor = 5;
```

Cambiar el valor de **factor** aquí, ya no afecta al block.
En el block se copio un 3.

```
int res = suma2(100);  // res es 103, no es 105
```

© Santiago Pavón - UPM-DIT

Compartir Variables Locales con __block

- Para compartir una variable local debe etiquetarse con **__block**.
  - No se copia el valor.
  - Desde el block se puede leer y cambiar el valor de la variable local.

```
__block int factor = 3;

int (^suma3)(int) =
    ^(int v) {
        factor++;    // incrementa factor
        return v+factor;
    };

factor = 10;

suma3(100);      // 111
```

© Santiago Pavón - UPM-DIT

Ejemplo

```
__block id res = nil;

NSString *name = @"James";

[array enumerateObjectsUsingBlock:
   ^(id object, NSUInteger index, BOOL *stop) {
         if ([[object name] isEqual:name]) {
             res = object;
             *stop = YES;
         }
    }];

if (res != nil) {
   NSLog(@"%@ is %@",name,res);
}
```

Podemos cambiar el valor de **res** desde dentro del block

Cambiamos **res**

© Santiago Pavón - UPM-DIT

Objetos Referenciados en un block

- Los blocks retienen automáticamente los objetos que referencian.
  - También cuando el block usa una variable de instancia del objeto .
- Peligro:
  - Pueden crearse bucles de retenciones que provocarán perdidas de memoria.

© Santiago Pavón - UPM-DIT

Ejemplo de Bucle de Retenciones

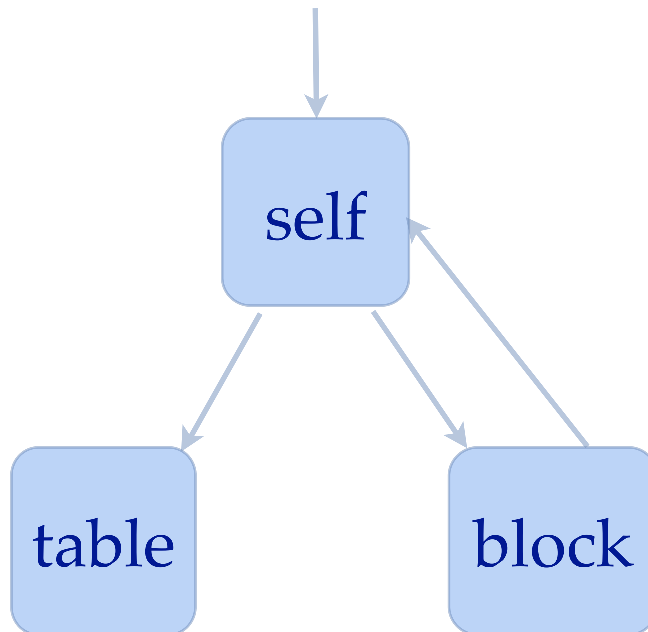
- Si un objeto retiene un block
  - por ejemplo guardándolo en una ivar.
- y si ese block retiene al objeto
  - Por ejemplo referenciando el objeto o una ivar del objeto
- Entonces hemos creado un bucle de retenciones.
- Al liberar el objeto tendremos una perdida de memoria (leak).

© Santiago Pavón - UPM-DIT

```
self.table = ...;
self.block = ^{[self.table reload]};
```

El bloque se retiene al guardarlo en la propiedad self.block

Como dentro del bloque se referencia a self, entonces el bloque retiene a self.

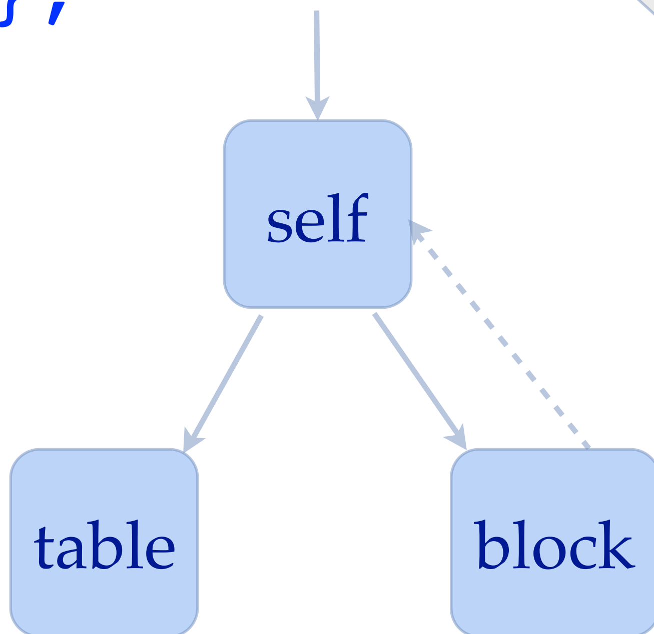


Romperemos el bucle haciendo weak la referencia a self

© Santiago Pavón - UPM-DIT

```
__weak LaClase *weakSelf = self;

self.block = ^{
      [weakSelf.table reload];
};
```

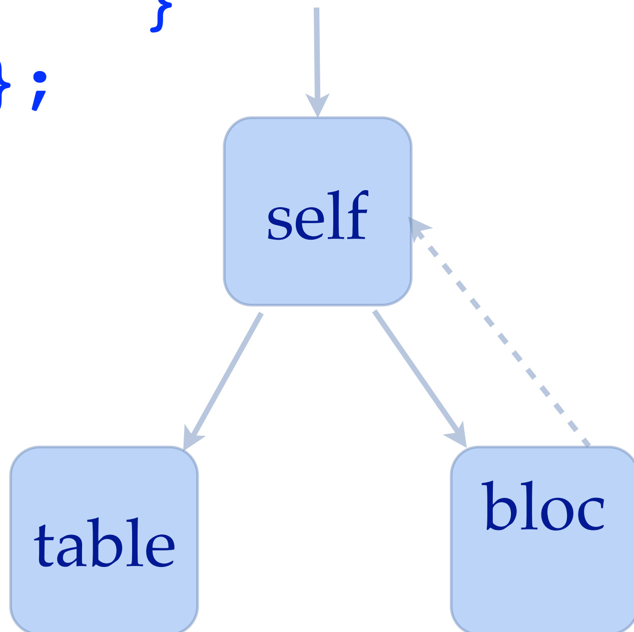


El bucle ya no referencia a self, luego el bloque NO retiene a self.
Ya no hay bucle de retenciones.

© Santiago Pavón - UPM-DIT

```
__weak LaClase *weakSelf = self;

self.block = ^{
    LaClase *strongSelf = weakSelf;
    if (strongSelf) {
        [strongSelf.table reload];
    }
};
```



Versión Thread-safe:

Para evitar que otro thread libere self mientras lo uso en el block, lo retengo con strongSelf.

© Santiago Pavón - UPM-DIT

Usos

- Completion handlers.
  - Muy útil para sustituir a algunos delegados.
- Notification handlers.
- Error handlers.
- Enumeraciones.
- Animaciones y transiciones en Views.
- Ordenación.
- Multithreading (con El API de Grand Central Dispatch).

© Santiago Pavón - UPM-DIT

© Santiago Pavón - UPM-DIT