
ETSIT UPM

dit UPM

# Desarrollo de Apps para iOS

# Objective-C - Gestión de Memoria

IWEB-LSWC 2013-2014

Santiago Pavón

ver: 2013.06.25 p1

# Cuenta de Retenciones

- Se usan contadores de retenciones para saber cuando debe liberarse la memoria de los objetos Objective-C.

- **MRC - Manual Reference Counting**
  - Nosotros gestionamos la memoria de los objetos Objective-C
- **ARC - Automatic Reference Counting**
  - El compilador añade automáticamente el código necesario para gestionar la memoria.
    - Incluido en iOS 5, compilador LLVM 3.0

# Recolección Automática de Basura

- No en iOS.
- No existe un GC molestando en cualquier momento porque se ha puesto a hacer limpieza.

# Liberar Memoria

- La memoria obtenida con **`alloc`**
  - debe liberarse con **`dealloc`**.
    - no debemos llamar a **`dealloc`** directamente.
    - se invoca indirectamente desde otros métodos.
      - excepción: llamada a **`[super dealloc]`** desde **`dealloc`**.
- El propietario de un objeto es el responsable de liberarlo.
  - pero no siempre está claro quien es el propietario.
- Cocoa define un protocolo de gestión de memoria.
  - basado en un contador de retenciones.

- Usando ARC la liberación se hace automáticamente.
- Con MRC la gestión de memoria la debemos programar nosotros.

# MCR

# Cuenta de Retenciones

- Todo objeto tiene un contador de retenciones (referencias).
  - heredado de `NSObject`
- `+alloc` y `–copy`
  - crean objetos con el contador a 1.
- `-retain`
  - incrementa el contador en 1.
- `-release`
  - decrementa el contador en 1.
  - Si el contador llega a 0, se llama automáticamente a `dealloc`.
- `-retainCount`
  - devuelve el valor del contador.

# Ejemplo de Uso

1. Creamos un objeto con `alloc`

```
Circle *c4 = [[Circle alloc] init];
```

   - El contador de retenciones vale 1.

2. Usamos `c4` en varios sitios
   - A `c4` le enviarán mensajes `retain` y `release`.
     - Estarán balanceados
3. Cuando hayamos terminado con `c4`:

```
[c4 release];
```

   - Si el contador llega a cero, `dealloc` es llamado automáticamente.

- Si `c4` llamó a `release`, no enviarle más mensajes.
  - Conveniente asignarle `nil` para evitar problemas.

```
c4 = nil;
```

# Reglas de Gestión de Memoria de Cocoa

- Cuando nosotros creamos un objeto con **`new`**, **`alloc`**, **`copy`** o **`mutableCopy`** su contador de retenciones es 1.
  - Nosotros debemos llamar a **`release`** o a **`autorelease`** cuando hayamos terminado de usarlo.
- Si no creamos el objeto, sino que obtenemos el objeto de otra forma:
  - supondremos que su contador es 1, y que su creador le ha enviado un mensaje **`autorelease`**. No haremos nada más.
  - Sólamente en el caso de que queramos guardarnos ese objeto, llamaremos a **`retain`**, y a **`release`** al terminar con él.
- Si enviamos **`retain`** a un objeto, en algún momento deberemos llamar a **`release`** o a **`autorelease`**.

# Variables de Instancia

- Al asignar objetos a variables de instancia:
  - debemos retener o copiar el objeto

```
-(void) setCenter:(Point*)c {
   if (center != c) {
      [center release];
      center = [c retain];
   }
}
-(void) setCenter: (Point*)c {
   if (center != c) {
      [center release];
      center = [c copy];
   }
}
```

# Redefinir `dealloc`

- Debemos redefinir dealloc en nuestros objetos:
  - primero hay que liberar la memoria apuntada por las variables de instancia.
  - y después liberamos la memoria ocupada por el propio objeto

```
- (void) dealloc {
    [center release];
    [super dealloc];
}
```

# Resumen de las Reglas de Gestión de Memoria

- Al obtener un objeto, hay que preocuparse de:
  - cómo obtuvimos el objeto
  - cuanto tiempo vamos a guardarlo.

| Objeto obtenido por | no lo guardamos | queremos guardarlo |
|---|---|---|
| new, alloc o copy | al terminar:<br>release<br>autorelease | release en dealloc |
| otra forma | nada | retain al obtenerlo, y release en dealloc. |

# Propiedades

```
@property (retain) Point* center;
```

- El método de acceso hace retain del objeto pasado como parámetro.

```
@property (copy) Point* center;
```

- El método de acceso hace una copia del objeto pasado como parámetro.

```
@property (assign) float radius;
```

- El método de acceso hace una asignación del valor pasado como parámetro.
- caso por defecto

# Autorelease Pool

- Pool de objetos a los que se enviará un mensaje release en un futuro.
- Para añadir un objeto a este pool, usamos un método heredado de NSObject

```
- (id) autorelease;
```

- Pool es una instancia de **`NSAutoreleasePool`**
  - Plantillas de Xcode lo crean en main.
  - UIKits tras manejar cada evento crea un nuevo pool.

# Ejemplo de autorelease

```
- (NSString*) description {

    NSString * des =
        [[NSString alloc]
                initWithFormat:@"Circle(%f,%f,%f)",
                                center.x,center.y,radius];

        [des release];    // HORROR: acabamos de liberar
                          // la memoria del objeto.

        return des; // apunta a la memoria liberada.
}
```

# Ejemplo de autorelease

```
- (NSString*) description {

    NSString * des =
        [[NSString alloc]
                initWithFormat:@"Circle(%f,%f,%f)",
                                center.x,center.y,radius];

        [des autorelease];


        return des;
}
```

# Bucles de Referencias

- Hay que evitar crear bucles de referencias:
  - obj1 retiene a obj2
  - obj2 retiene a obj1

- Ejemplo: los nodos de un árbol apuntan a los hijos y al padre.
  - No retener el padre.
  - Cuidado: Al hacer dealloc de un nodo, poner a nil el atributo padre de todos sus hijos.
    - para que no quede colgando, apuntando a basura.

- Estrategias:
  - que un objeto no retenga al objeto que lo posee.
  - que el objeto que va a vivir menos no retenga al que va a vivir más tiempo.

# El Círculo con MRC

- El atributo **`centro`** debe retenerse:
  - Indicar **`retain`** al declarar la propiedad.
  - initXXX retiene el objeto guardado en centro.

```
[self setCenter:c];
```

```
self.center = c;
```

- Redefinir **`dealloc`** para enviar el mensaje **`release`** al atributo **`centro`**.

# Circle.h

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

@class Point;

@interface Circle : NSObject

@property (retain) Point* center;
@property float radius;

- (id) initWithRadius:(float)r
         andWithCenter:(Point*)c
@end
```

# Circle.m

```
#import "Circle.h"
#import "Point.h"

@implementation Circle
@synthesize center, radius;

-(id) initWithRadius:(float)r
      andWithCenter:(Point*)c {
        if (self = [super init]) {
          radius = r;
          self.center = c;
        }
        return self;
}
-(void)dealloc {
    [center release];
    [super dealloc];
}
@end
```

# ARC

# Automatic Reference Counting

- Introducido en Xcode 4.2
  - Funciona en iOS4 y posteriores.
    - en iOS4 no soporta referencias weak.
    - en iOS 5 se soportan todas las características.
- Es recomendable que migremos a ARC .
  - La gestión de memoria de los objetos es más fácil para el programador 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.
- El compilador nos añade las llamadas a retain, release y autorelease.
- Se siguen los convenios de nombres de Cocoa.
  - Esto permite interoperar con código MRC.

# Nueva forma de pensar

- Con ARC nos olvidamos de las llamadas a retain, release y autorelease.
  - Lo añade el compilador por nosotros.
- Ahora los punteros implican la propiedad de los objetos.
  - Los punteros mantienen con vida a los objetos.
  - Un objeto se destruye cuando no queda ningún puntero que lo apunte.

# Ejemplo

```
@interface Queue : NSObject

- (void) push:(id)data;

- (id) pop;

@end
```

# Ejemplo - ARC

```
@implementation Queue {
    NSMutableArray *ma;
}

- (id) init {
    if (self = [super init])
      ma = [NSMutableArray array];
    return self;
}

- (void) push:(id)data {
    [ma insertObject:data atIndex:0];
}

- (id) pop {
    id data = [ma lastObject];
    [ma removeLastObject];
    return data;
}
@end
```

# Ejemplo - MRC

```
@implementation Queue {
    NSMutableArray *ma;
}
- (id) init {
    if (self = [super init])
      ma = [[NSMutableArray array] retain];
    return self;
}
- (void) push:(id)data {
    [ma insertObject:data atIndex:0];
}
- (id) pop {
    id data = [[[ma lastObject] retain] autorelease];
    [ma removeLastObject];
    return data;
}
- (void) dealloc {
    [ma release];
    [super dealloc];
}
@end
```

# Referencias

- Por defecto, cuando una variable apunta a un objeto, se hace propietaria de él (lo retiene), pero hay otras posibilidades.
  - Esto aplica en general a todo tipo de variables: locales, globales, ivars, parámetros, etc.
- Los tipos de referencias que podemos tener son:
  - **strong**
  - **weak**
  - **unsafe_unretained**
  - **autoreleasing**
- ARC maneja estos tipos de relaciones para saber que código debe autogenerar para gestionar la memoria.

# Referencias Strong

```
__strong NSString *name;
```

- Opción por defecto cuando no usamos ningún calificador.

- Las variables se inicializan a nil.
- Obtiene la propiedad (retain) del objeto referenciado.
- Se pierde la propiedad (release) sobre el objeto apuntado cuando:
  - se asigna un nuevo valor a la variable
  - o termina el ámbito de la variable.

- Nunca contienen un puntero colgando (apuntando a basura).

# Referencias Weak

```
__weak NSString *name;
```

- Las variables se inicializan a nil.
- Las variables **no** retienen al objeto apuntado.
- Si se libera la memoria del objeto apuntado, se asigna nil a la variable.

- Estas variables nunca apuntarán a una zona de memoria que ha sido liberada, ya que antes se las asigna nil.

- Nota: útil para evitar bucles de referencias strong.
  - 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`.

- 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)*

# Referencias Unsafe_unretained

```
__unsafe_unretained NSString *name;
```

- Las variables **no** se inicializan a nil.
- Las variables **no** retienen al objeto apuntado.
- Si se libera la memoria del objeto apuntado, **no** se asigna nil a la variable.
  - Cuidado con los punteros que quedan colgando.
    - Conviene ponerlos a nil.

      (Es decir, son las asignaciones de toda la vida.)
- Se usan para evitar bucles de retenciones, en campos de struct y union.

# Referencia Autoreleasing

- El calificador usado es `__autoreleasing`.
- Es el calificador por defecto usado en los parámetros de vuelta (puntero a puntero a objeto) de los métodos.
  - Los valores devueltos en el parámetro han sido autoreleased.
  - No somos los propietarios de los objetos que nos devuelven en estos parámetros.
  - Al ser el valor por defecto no hace falta escribirlo.
- Los convenios de Cocoa no permiten transferir la propiedad de los objetos usando parámetros.
  - Podemos usar otro calificador para indicar otro comportamiento.
- Ejemplo típico: Devolver objeto de error.

```
NSError *error = nil;
BOOL ok = [self taskWithError(&error)];
if (! ok) {
   NSLog(@"ERROR = %@",[error localizedDescription]);
}
```

```
- (BOOL) taskWithError:(__autoreleasing NSError**)err {
   ...
   *err = [[NSError alloc] init...];
   ...  // El compilador añade un autorelease
}

// Nota: no hace falta ponerlo, es el caso por defecto.
```

# 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

- 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*.

# dealloc

- ARC crea automáticamente el método `dealloc` para liberar (release) las variables de instancia, y además añade la llamada a `[super dealloc]`.
- Nosotros sólo lo implementaremos si hay que liberar recursos que no sean objetos Objective-C.
  - cerrar un fichero, hacer un free, desregistrar una notificación, ...
  - Si únicamente hay que liberar ivars, entonces no lo escribiremos.
- Lo que antes era:

```
- (void)dealloc {
    [ivar release];
    free(buffer);
    [super dealloc];
}
```

- Ahora es simplemente::

```
- (void)dealloc {
    free(buffer);
}
```

Con ARC también está prohibido llamar a **[super dealloc]**

# Convenio de Nombres

- Los métodos que comienzan con la palabras **`alloc`**, **`new`**, **`init`**, **`copy`**, **`mutableCopy`** transfieren la propiedad del objeto.
  - Nota: La segunda palabra comienza con mayúsculas si es que existe (notación Camello).
- No tenemos que preocuparnos de los convenios de nombres mientras no mezclemos ficheros compilados con MRC y ARC.
- Pero es muy recomendable respetarlos ya que la vida da muchas vueltas.
  - Xcode muestra warnings cuando no se respeta el convenio de nombres de cocoa.

# Saltarse el Convenio de Nombres

- Si por alguna oscura razón tenemos métodos que no respetan el convenio de nombres, y además mezclamos ARC y MRC
  - Ejemplo: un método que se llama **`duplica`** en vez de llamarse **`copyAlgo`**.
    - Me obligan a usar ese nombre, es ajo y agua.
  - Deberemos anotar los métodos para indicar si los objetos que devuelven han sido retenidos o no.

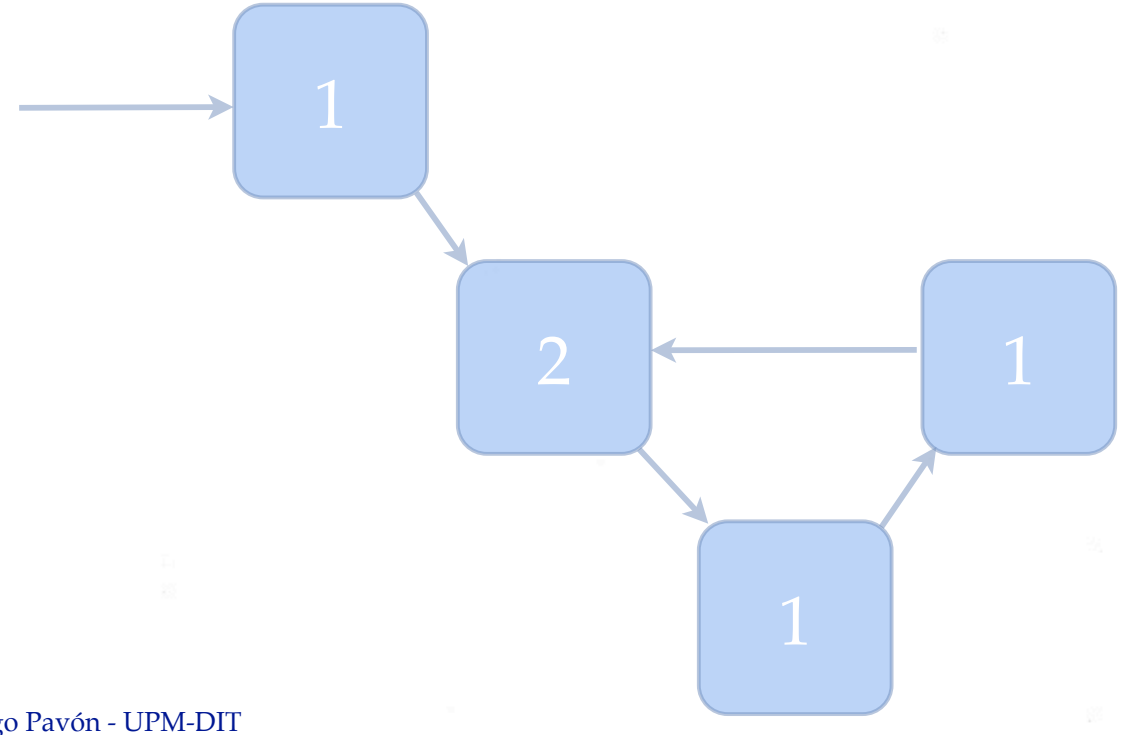
```
NS_RETURNS_NOT_RETAINED
NS_RETURNS_RETAINED
```

  - Así ARC sabe cuando debe retener y liberar.

```
- (id) duplica NS_RETURNS_RETAINED;
```

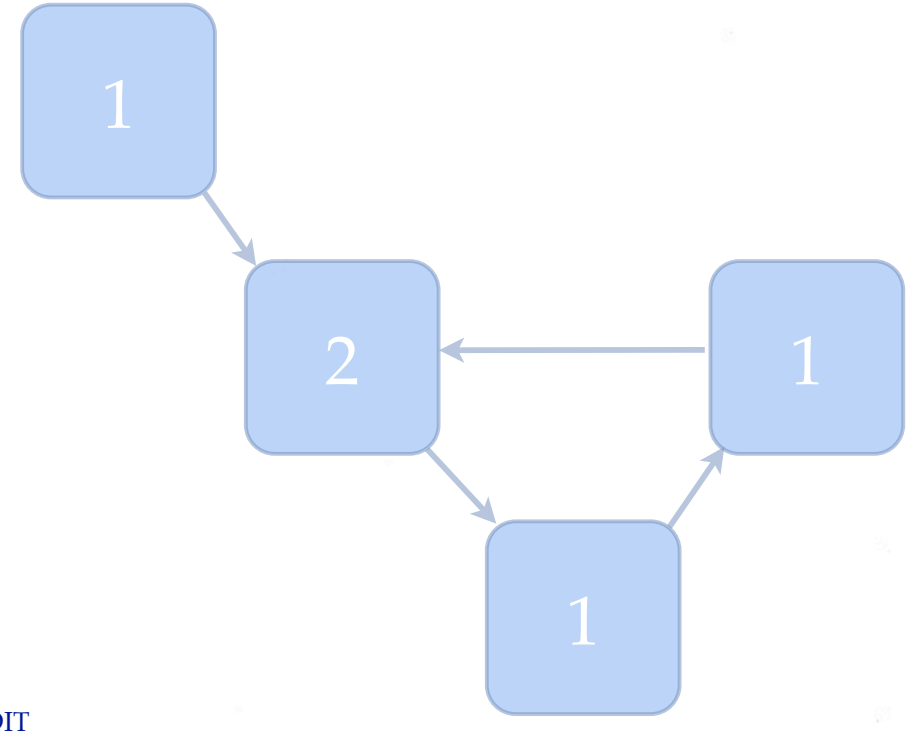
# Evitar Bucles de Referencias Strong

- Los bucles de referencias strong producen perdidas de memoria (leaks).
  - Hay que evitarlos.



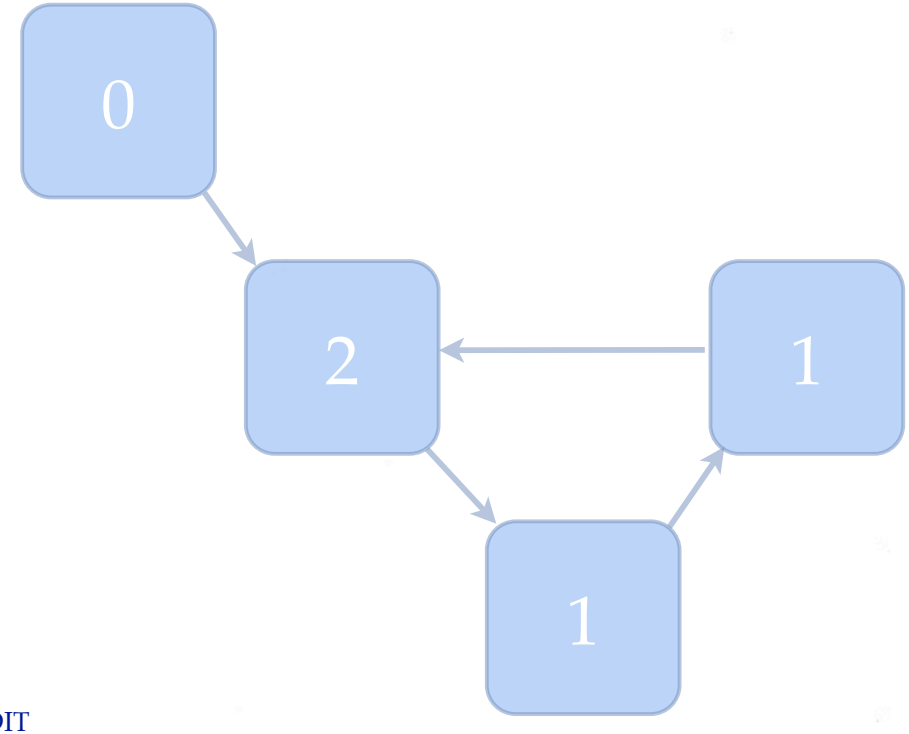
# Evitar Bucles de Referencias Strong

- Los bucles de referencias strong producen perdidas de memoria (leaks).
  - Hay que evitarlos.



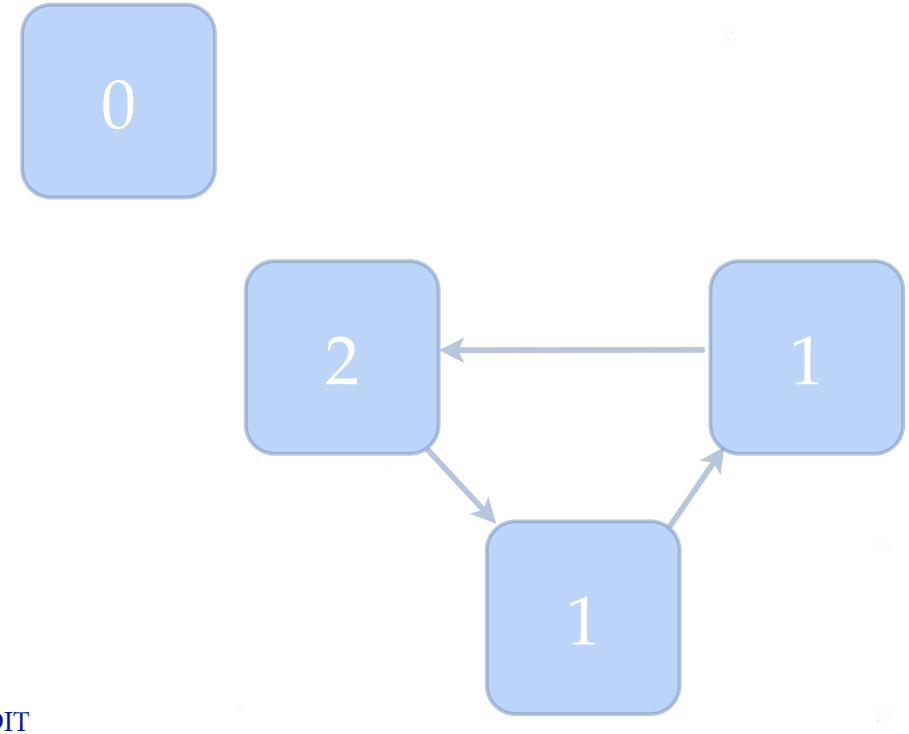
# Evitar Bucles de Referencias Strong

- Los bucles de referencias strong producen perdidas de memoria (leaks).
  - Hay que evitarlos.



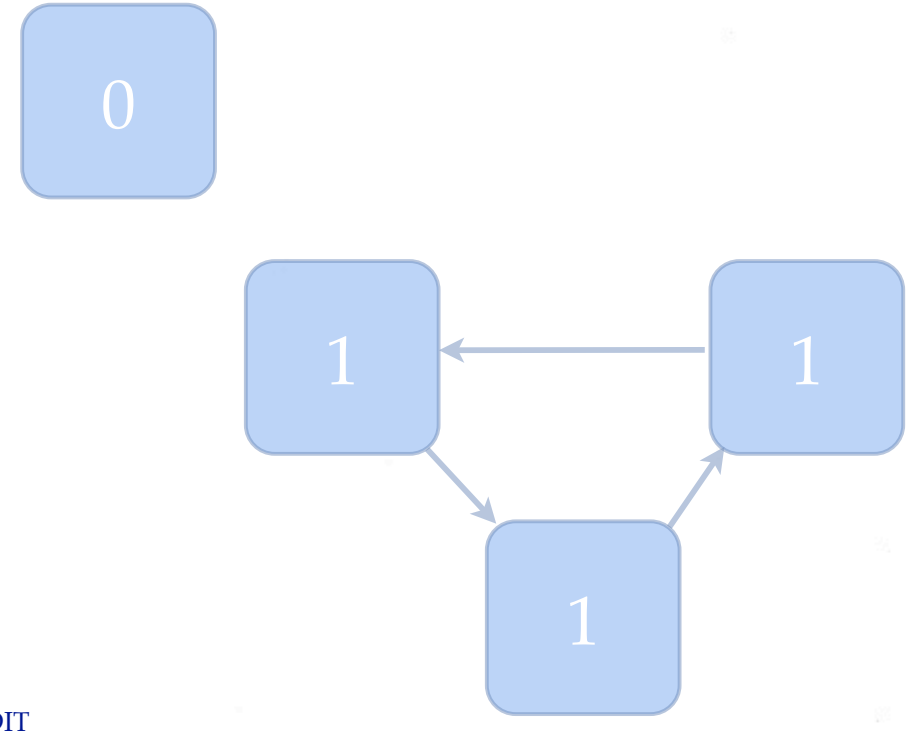
# Evitar Bucles de Referencias Strong

- Los bucles de referencias strong producen perdidas de memoria (leaks).
  - Hay que evitarlos.



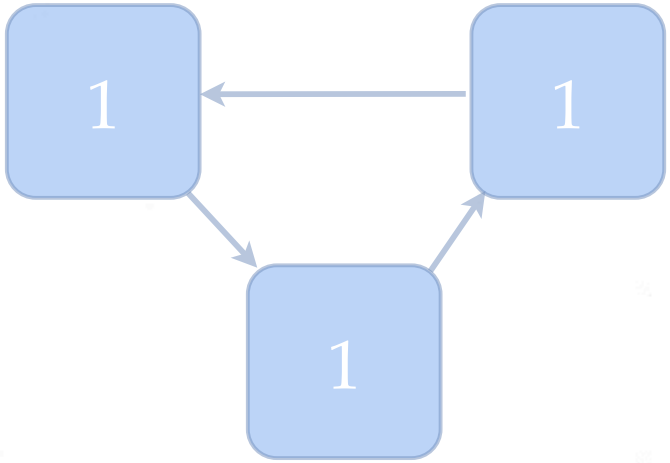
# Evitar Bucles de Referencias Strong

- Los bucles de referencias strong producen perdidas de memoria (leaks).
  - Hay que evitarlos.



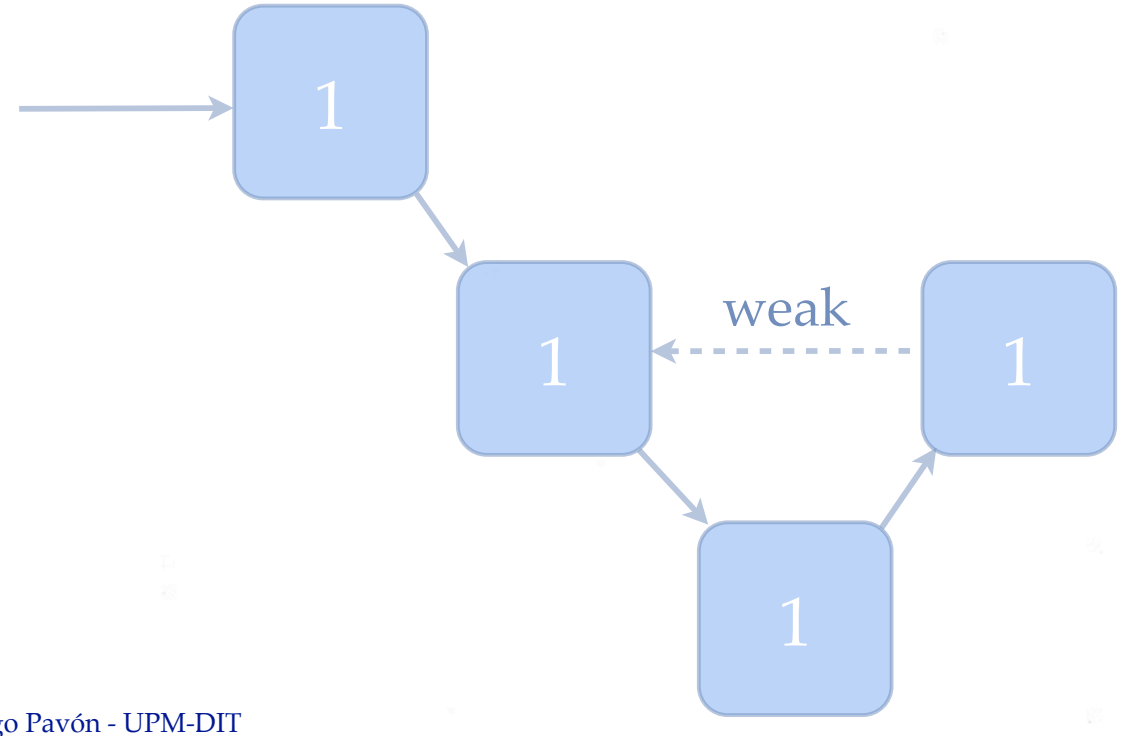
# Evitar Bucles de Referencias Strong

- Los bucles de referencias strong producen perdidas de memoria (leaks).
  - Hay que evitarlos.



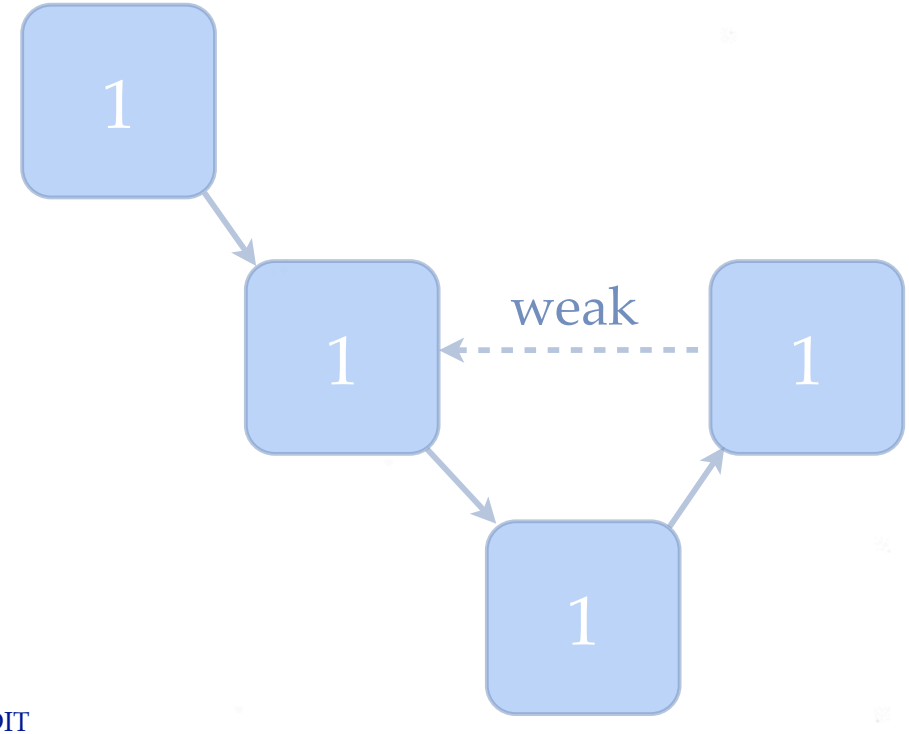
# Evitar Bucles de Referencias Strong

- Solución: Crear referencias weak
  - No retienen el objeto referenciado.
  - Se asignan a nil cuando se libera la memoria del objeto referenciado.



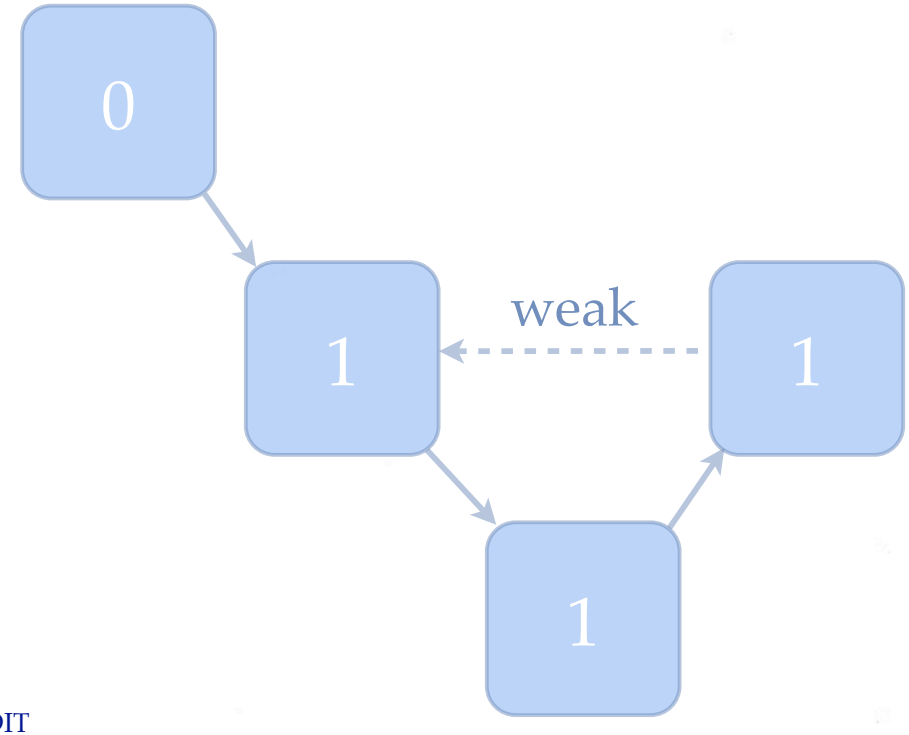
# Evitar Bucles de Referencias Strong

- Solución: Crear referencias weak
  - No retienen el objeto referenciado.
  - Se asignan a nil cuando se libera la memoria del objeto referenciado.



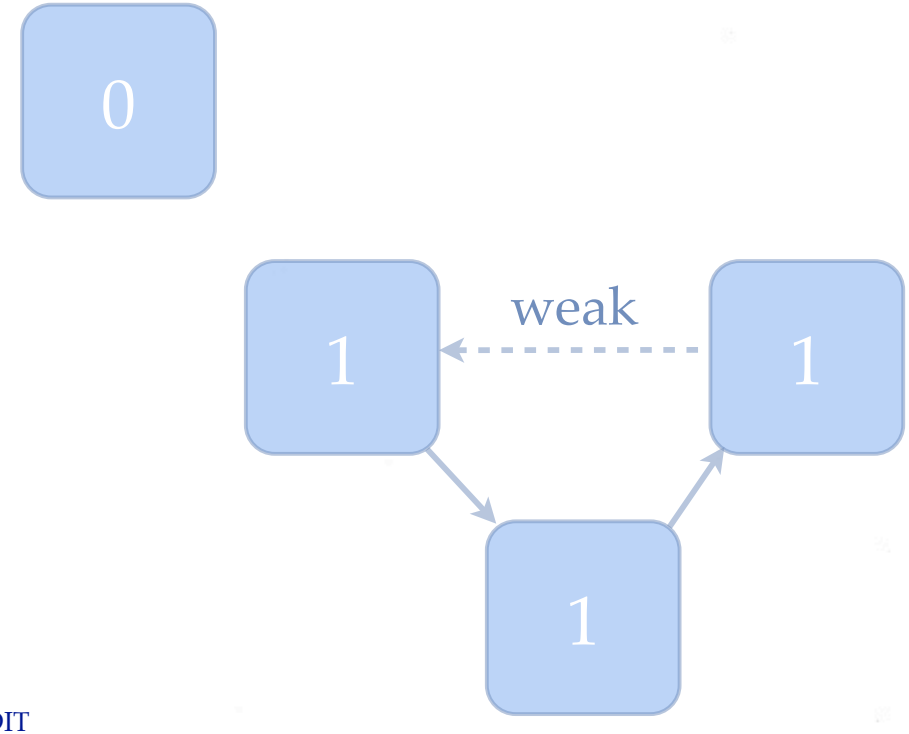
# Evitar Bucles de Referencias Strong

- Solución: Crear referencias weak
  - No retienen el objeto referenciado.
  - Se asignan a nil cuando se libera la memoria del objeto referenciado.



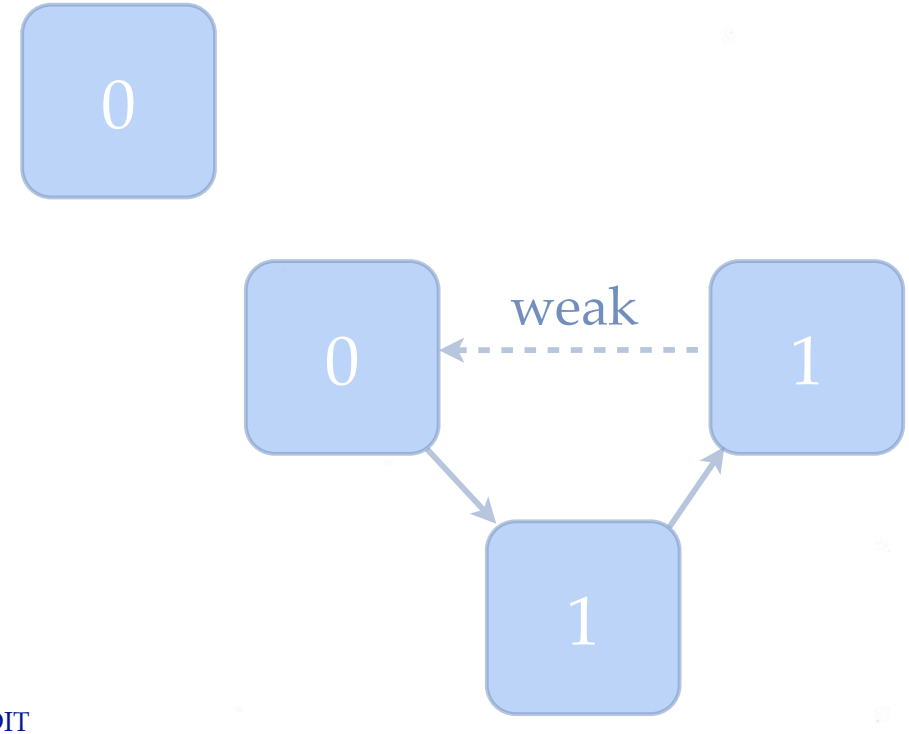
# Evitar Bucles de Referencias Strong

- Solución: Crear referencias weak
  - No retienen el objeto referenciado.
  - Se asignan a nil cuando se libera la memoria del objeto referenciado.



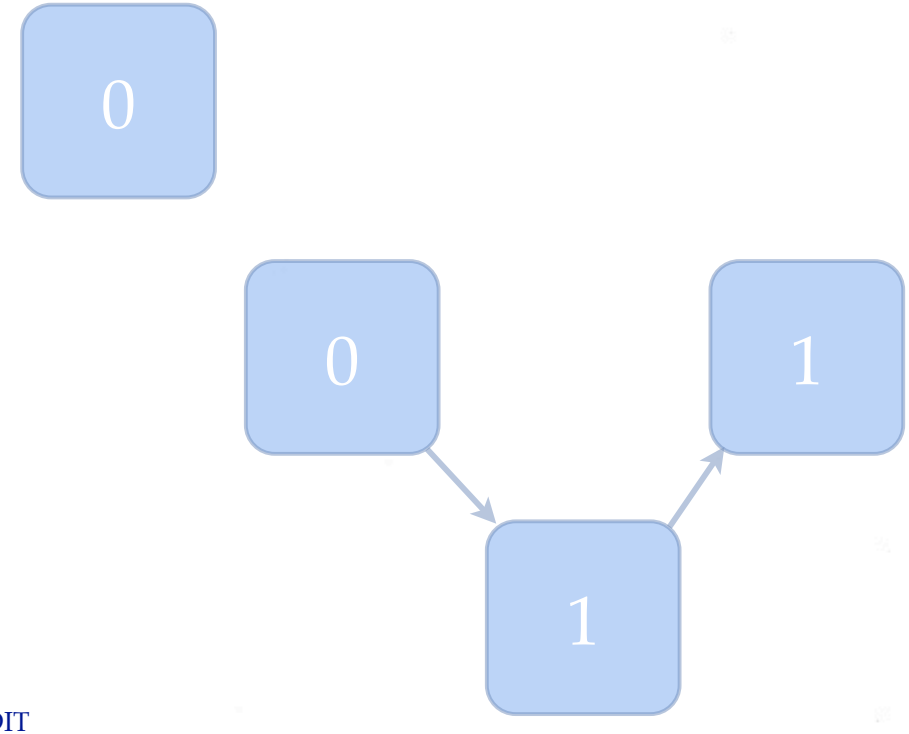
# Evitar Bucles de Referencias Strong

- Solución: Crear referencias weak
  - No retienen el objeto referenciado.
  - Se asignan a nil cuando se libera la memoria del objeto referenciado.



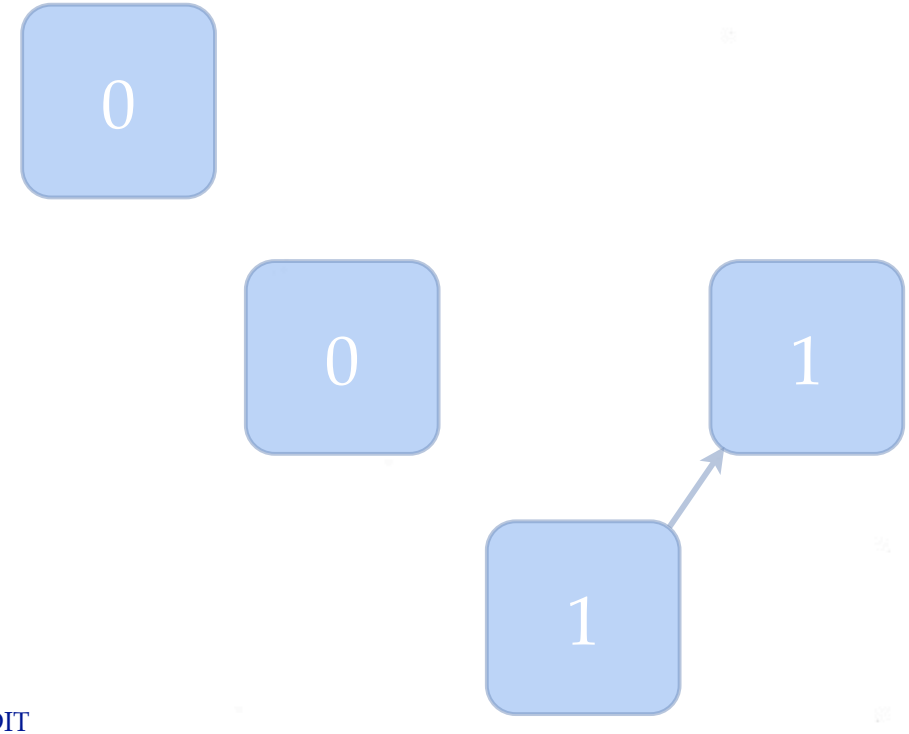
# Evitar Bucles de Referencias Strong

- Solución: Crear referencias weak
  - No retienen el objeto referenciado.
  - Se asignan a nil cuando se libera la memoria del objeto referenciado.



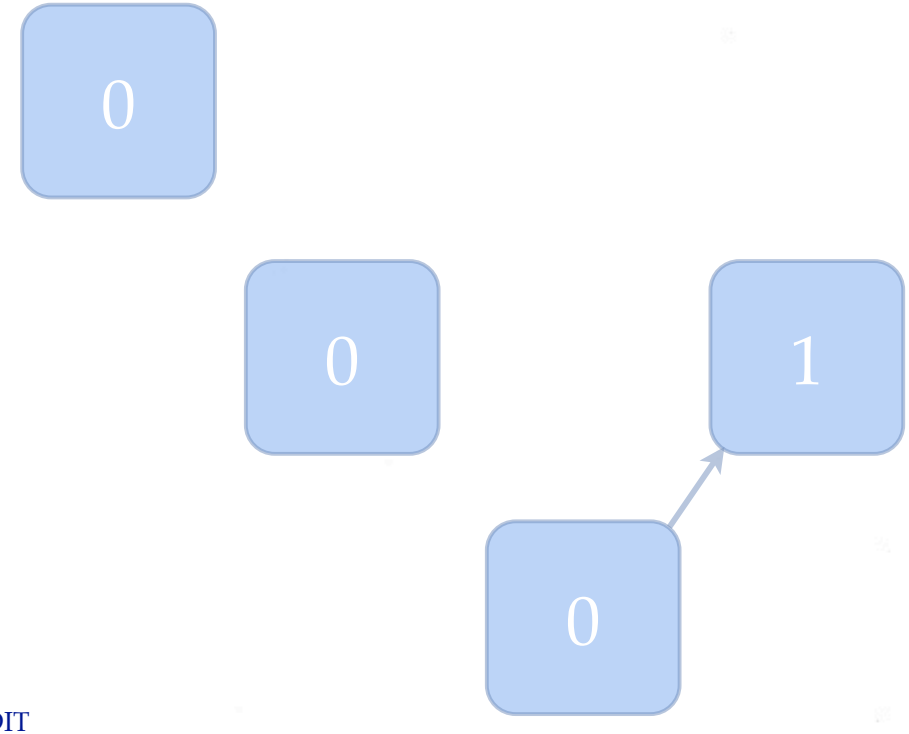
# Evitar Bucles de Referencias Strong

- Solución: Crear referencias weak
  - No retienen el objeto referenciado.
  - Se asignan a nil cuando se libera la memoria del objeto referenciado.



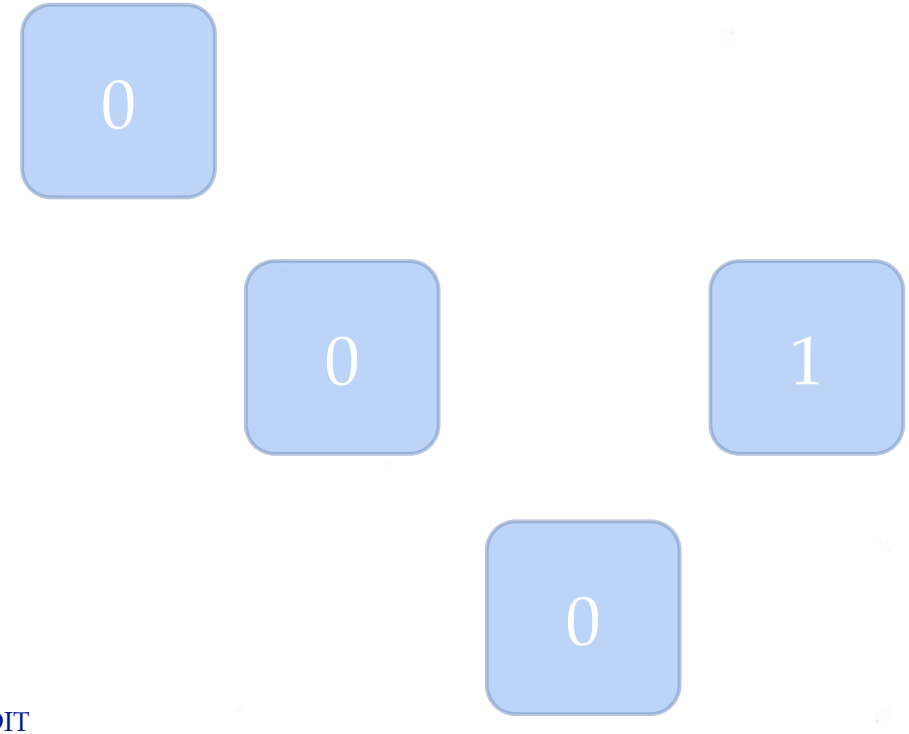
# Evitar Bucles de Referencias Strong

- Solución: Crear referencias weak
  - No retienen el objeto referenciado.
  - Se asignan a nil cuando se libera la memoria del objeto referenciado.



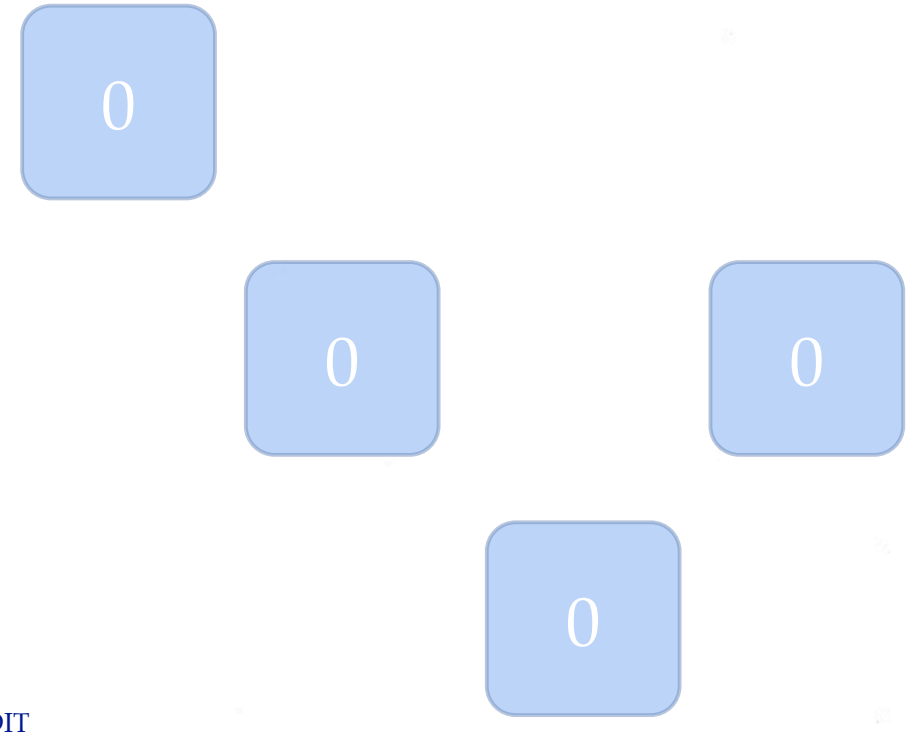
# Evitar Bucles de Referencias Strong

- Solución: Crear referencias weak
  - No retienen el objeto referenciado.
  - Se asignan a nil cuando se libera la memoria del objeto referenciado.



# Evitar Bucles de Referencias Strong

- Solución: Crear referencias weak
  - No retienen el objeto referenciado.
  - Se asignan a nil cuando se libera la memoria del objeto referenciado.



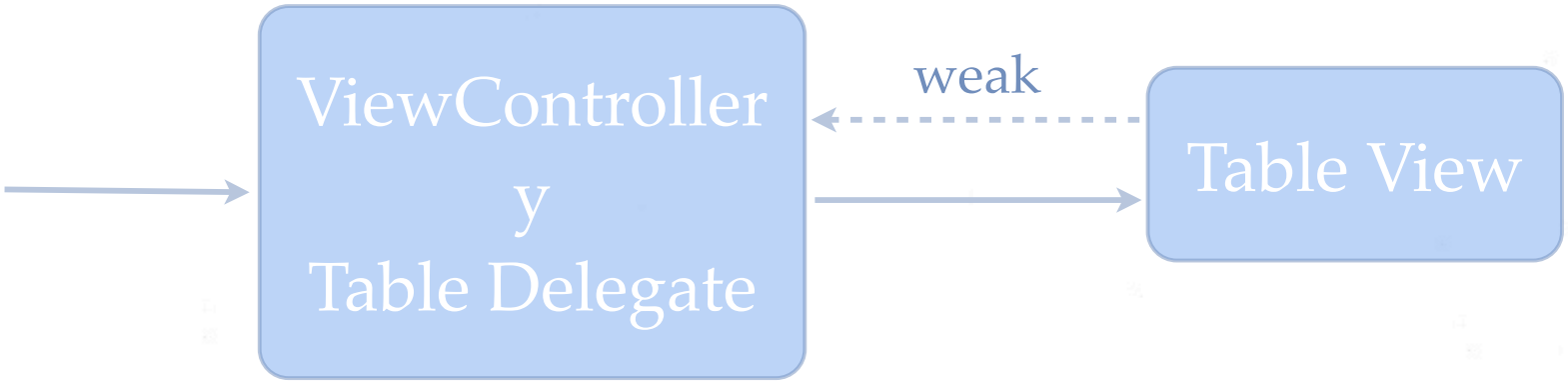
# Evitar Bucles de Referencias Strong

- Solución: Crear referencias weak
  - No retienen el objeto referenciado.
  - Se asignan a nil cuando se libera la memoria del objeto referenciado.

# Evitar Bucles de Referencias Strong

```
__weak ExampleClass *example;
```

- Ejemplo típico:



# Evitar Bucles de Referencias Strong

- Otras soluciones (poco recomendables):
  - Asignar explícitamente nil a las ivars.
  - Usar referencias unsafe_unretained.

# Punteros a objeto en un struct C

- Están prohibidas las referencias a objetos Objective-C como campos de un struct/union.
  - Razón: El compilador necesita conocer cuando se crean, destruyen o se cambian las referencias para poder gestionar las inicializaciones, retenciones y liberación de la memoria.
- Podemos tener referencias calificadas como **`__unsafe_unretained`**.
  - Son simples asignaciones sin ninguna gestión.

```
struct Person {
    __unsafe_unretained NSString *name;
    int age;
}
```

- Recomendación: No usar structs, mejor usar clases.

# CF Casting: `id⇔void*`

- Toll-Free Bridging
  - Compatibilidad entre objetos que siguen las convenciones CF (void*) y objetos objective-C (id).
  - Ejemplo: los objetos Objective-C tipo NSString*, y los objetos Core Foundation de tipo CFStringRef con compatibles, y pueden usarse unos en lugar de los otros.
- El ARC necesita saber quien es el responsable de gestionar la memoria de los objetos cuando un tipo de objetos se usa en lugar del otro.
  - ARC gestiona la memoria de los objetos Objective-C, pero no la de los objetos CF.
- Los castings entre objetos Toll-Free deben anotarse con las siguientes palabras claves para informar a ARC sobre si debe gestionar la liberación de memoria o no.

```
__bridge
__bridge_retained
__bridge_transfer
```

- El objeto apuntado por nSString es un objeto Objective-C que lo está gestionando ARC, y ARC es la responsable de liberar su memoria cuando sea oportuno.
  - Ese objeto lo usaremos como un CFStringRef, pero no tenemos que liberar su memoria llamando a CFRelease.

```
CFStringRef x = (__bridge CFStringRef)nSString;
```

- El objeto apuntado por nSString es un objeto Objective-C que lo está gestionando ARC, pero queremos liberar a ARC de la responsabilidad de liberar su memoria.
  - Ese objeto lo usaremos como un CFStringRef, y en algún momento llamaremos CFRelease para liberar su memoria.

```
CFStringRef x = (__bridge_retained CFStringRef)nSString;
```

- El objeto apuntado por cFStringRef es un objeto que lo creó CF, y ARC no es la responsable de liberar su memoria.
  - Ese objeto lo usaremos como un NSString, y al final tenemos que liberar su memoria llamando a CFRelease.

```
NSString *x = (__bridge NSString*)cFStringRef;
```

- El objeto apuntado por cFStringRef es un objeto que lo creó CF, pero queremos que ARC sea la responsable de liberar su memoria.
  - Ese objeto lo usaremos como un NSString, y cuando sea oportuno ARC liberará su memoria.

```
NSString *x = (__bridge_transfer NSString*)cFStringRef;
```

# Autorelease Pool

- Los objetos `NSAutoreleasePool` son especiales, no pueden retenerse.
  - No podemos usarlos directamente con ARC.
- Se ha creado una directiva para usarlos: **`@autoreleasepool`**

```
@autoreleasepool {
    for (int i=0 ; i<MAX_POS ; i++){
        // sentencias
    }
}
```

# Activar ARC

- Por defecto, los proyectos nuevos desde Xcode 4.2 usan ARC.
- En el IDE, en la pestaña ***Build Settings*** del target del proyecto existe una opción que indica si se usa ARC o no.

```
Objective-C Automatic Reference Counting = Yes|No
```

- Podemos cambiar el comportamiento por defecto para ficheros individuales usando opciones del compilador:

**`-fobjc-arc`**, **`-fno-objc-arc`**

  - Añadir estas opciones a los ficheros afectados en:

*Target del proyecto > Build Phases > Compile Sources*

# Migrar a ARC

- Tenemos que compilar con LLVM 3.0
- Podemos cambiar el código a mano.
- O automáticamente ejecutando:

  *Edit > Refactor > Convert to Objective-C ARC...*

  - Se generaran errores de compilación.
    - arreglarlos y reintentar

# Transformar el código para ARC

- Borrar llamadas e implementaciones de `retain`, `release`, `autorelease`.
- Sustituir `NSAutoreleasePool` por `@autoreleasepool`.
- Sustituir `@property(assign)` por `@property(weak)`.
- ...

# Versiones iOS

- ARC funciona en iOS5.
- En iOS4 no se soportan referencias weak.
  - Hay que usar unsafe_unretained.
  - La herramienta de conversión a ARC lo hace automáticamente.
  - Si queremos desplegar en iOS4:
    - Seleccionar en *Build Settings > iOS Deployment Target* la versión donde desplegaremos.

# Documentación

- Portal de desarrollo iOS:
  - Transitioning to ARC Release Notes.
  - Foros `http://devforums.apple.com`
- Presentaciones WWDC 2011

# Propiedades

# Opciones de @property

- Opciones de gestión de memoria con ARC:
  **`strong`**, **`weak`**, **`unsafe_unretained`**, **`copy`**, **`assign`**
  - Nota: con ARC **`retain`** es sinónimo de strong.

- Opciones de gestión de memoria con MRC:
  **`retain`**, **`copy`**, **`assign`**

# MRC - Opciones

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

```
@property (retain) NSString *name;
    // Indicar que se retenga el objeto.
    // (aumentar su cuenta de retenciones).

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

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

# ARC - Opciones

- 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 apuntandole.
    // Nota: con ARC el uso de retain es sinonimo de strong.

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

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

@property (unsafe_unretained) NSString *name;
    // La ivar es unsafe_unretained.
    // Como weak, pero sin asignar nil si se destruye el objeto.
    // Para crear apps compatibles con versiones anteriores de iOS.

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

# Blocks

# Objetos Referenciados en un block

- Los blocks retienen automáticamente los objetos que referencian.
- Si el block usa una variable de instancia, se retiene self.
- Se invoca release sobre los objetos retenidos cuando se destruye el block, o cuando salimos del ámbito en el que se creó el block.

- Respecto de las variables a objetos etiquetadas con __block
  - Con MRC, si el block usa una variable de tipo objeto etiquetada con __block **no** se retiene el objeto apuntado.
    - Muy usado para evitar bucles de retenciones.
  - Con ARC **si** se retiene. Todas las variables por defecto son strong aunque estén etiquetadas con __block
    - Para reproducir el comportamiento de MRC usar una de estas opciones:

```
__block __unsafe_unretained NSString *s;
__block __weak NSString *s;
```

# Gestión de Memoria

- Los block son objetos Objective-C.
  - Pueden usarse como cualquier objeto.
- Inicialmente se crean en el stack.
- Si el block se va a usar cuando el ámbito donde se creó ya ha sido destruido, entonces hay que copiarlo antes en el heap.
  - Hacerlo al guardarlos en variables de instancia, en variables globales, al devolverlos con return, al asignarlos indirectamente.
  - Para copiarlos en el heap debe llamarse a **`copy`**.
    - Un requisito: los literales block deben copiarse, no pueden retenerse.
      - En la práctica es mejor copiar siempre los blocks en vez de retenerlos.
        - Sólo se mantiene una copia en el heap.
  - Para liberarlos debe llamarse a **`release`** o a **`autorelease`**.
- Cuando se copia un block en el heap, las variables locales marcadas con **`__block`** también se mueven al heap.

- Con MRC nosotros debemos escribir las llamadas a copy y (auto)release.

```
return [[^{...} copy] autorelease];
```

- Con ARC se gestionan las copias y liberaciones por nosotros.

```
return ^{...};
```

```
(Hay casos en los que tenemos que llamar a copy. Lo veremos después)
```

- Los únicos mensajes que acepta un block son **`copy`**, **`retain`**, **`release`** y **`autorelease`**.
  - Existen también funciones C para copiarlos y liberarlos:
    - **`Block_copy()`**, **`Block_release()`**

# Usar copy con ARC

- Los bloques deben copiarse en el heap cuando es posible que se usen después de haber destruido el contexto en el que fueron creados.
- Apple dice:

  Blocks "just work" when you pass blocks up the stack in ARC mode, such as in a return. You don't have to call Block Copy any more. You still need to use `[^{} copy]` when passing "down" the stack into `arrayWithObjects:` and other methods that do a retain.

- Aclaración sobre el significado de stack up y stack down:
  - En la mayoría de los sistemas, iOS incluido, el stack crece hacia direcciones de memoria menores, es decir, los elementos recién añadidos al stack ocupan direcciones de memoria inferiores a las de los elementos ya existentes en él. Entonces, cuando se añade algo al stack, el valor de puntero que marca el límite del stack decrece; y cuando se eliminan datos del stack el valor del puntero aumenta.
    - Por tanto:
      - stack up significa que el stack ha disminuido porque se han eliminado datos, es decir, se ha borrado un contexto del stack..
      - stack down significa que el stack ha aumentado porque se han metido nuevos datos en él, es decir, se ha añadido un nuevo contexto en el stack.

- Lo que Apple dice significa:
  - Si se pasa un bloque en una situación en la que se elimina un contexto del stack, el compilador sabe que ese bloque puede residir en el contexto que se va a destruir, y por esto siempre añade una llamada a copy para copiarlo en el heap. No tenemos que preocuparnos.
    - Ejemplo: cuando un método termina devolviendo un bloque, el compilador siempre añade una llamada a copy para que el bloque devuelto esté en el heap. Así se evita devolver un bloque alojado en la zona del stack que se va a destruir.
  - Cuando pasamos un bloque a un nuevo contexto y en este contexto se va a retener el bloque, es necesario que nosotros llamemos explícitamente a copy. De no hacerlo, existe la posibilidad de que el bloque se llame después de haber destruido el contexto en el que se creó. La aplicación se moriría.
    - Ejemplo: cuando pasamos un bloque como parámetro a un método no se realiza un copy del bloque. Si el método va a retener el bloque, deberemos hacer un copy del bloque explícitamente. Un caso típico en los que hay que usar copy es al guardar un bloques en un array o diccionario.
    - Nota: este fallo puede que no se detecte hasta que compilemos sin depuración.

```
typedef int (^suma_t)(int);

- (suma_t) ambito {
    __block int k = 33;
    suma_t s = ^(int v){return v+k;};
    return [[ s copy] autorelease];
}

suma_t suma = [self ambito];

int res = suma(100);

NSLog(@"res = %d",res);
```

Uso __block para que se cree un NSStackBlock.
En el heap se copia un NSMallocBlock.
Sin __block se crea un NSGlobalBlock.

Con ARC no hace falta ni el copy, ni el autorelease.

Con MRC copio el block en el heap porque este ámbito no existe cuando se ejecuta el block.

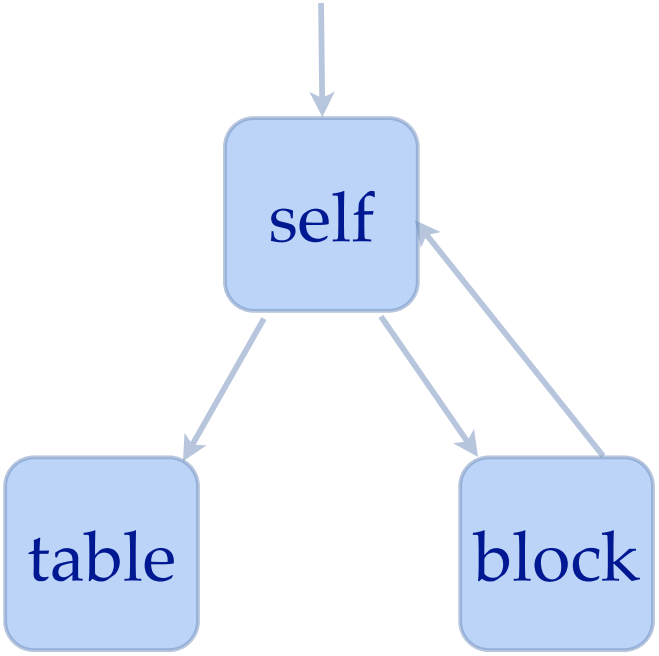
# Blocks - Bucles Pasando por self

- 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).

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

El bloque se retien al asignarse a una propiedad de self.

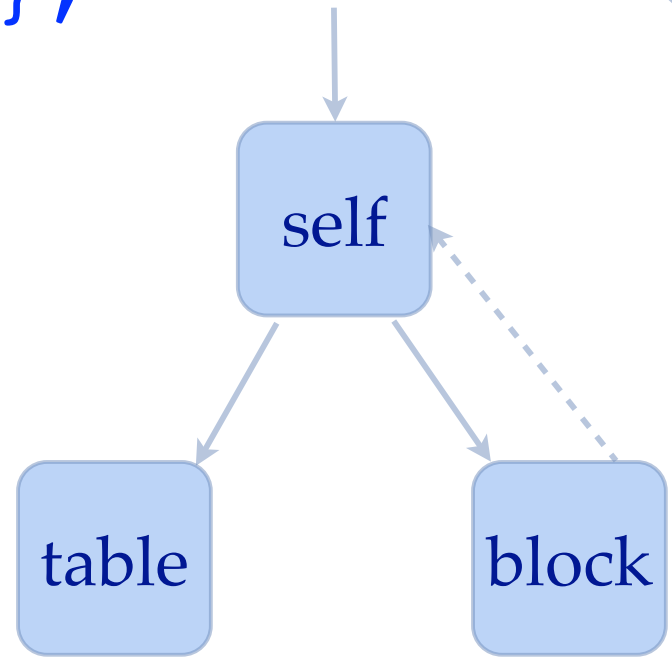
Como dentro del bloque se referencia a self, entonces el bloque retiene a self.




Romperemos el bucle haciendo weak la referencia a self

```
__weak LaClase *weakSelf = self;

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

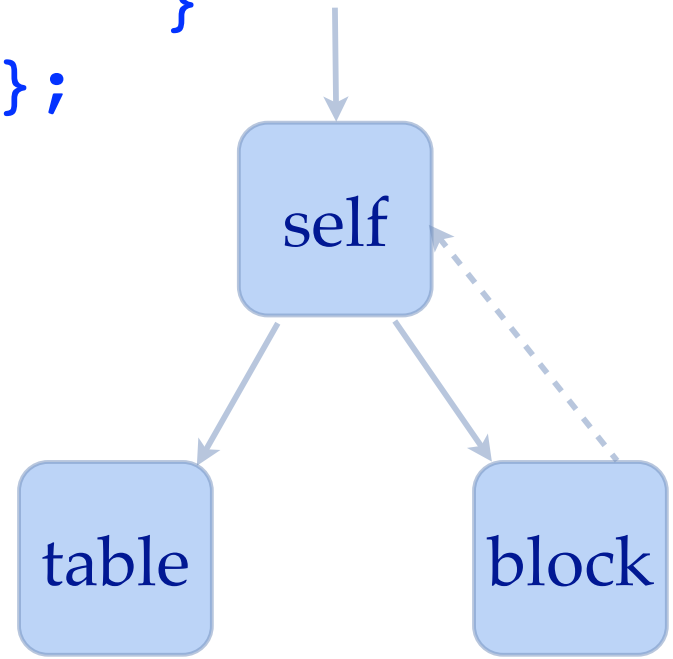



El bucle ya no referencia a self, luego el bloque NO retiene a self.

**Ya no hay bucle de retenciones.**

```
__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.