應用程式記憶體管理

Reference Count

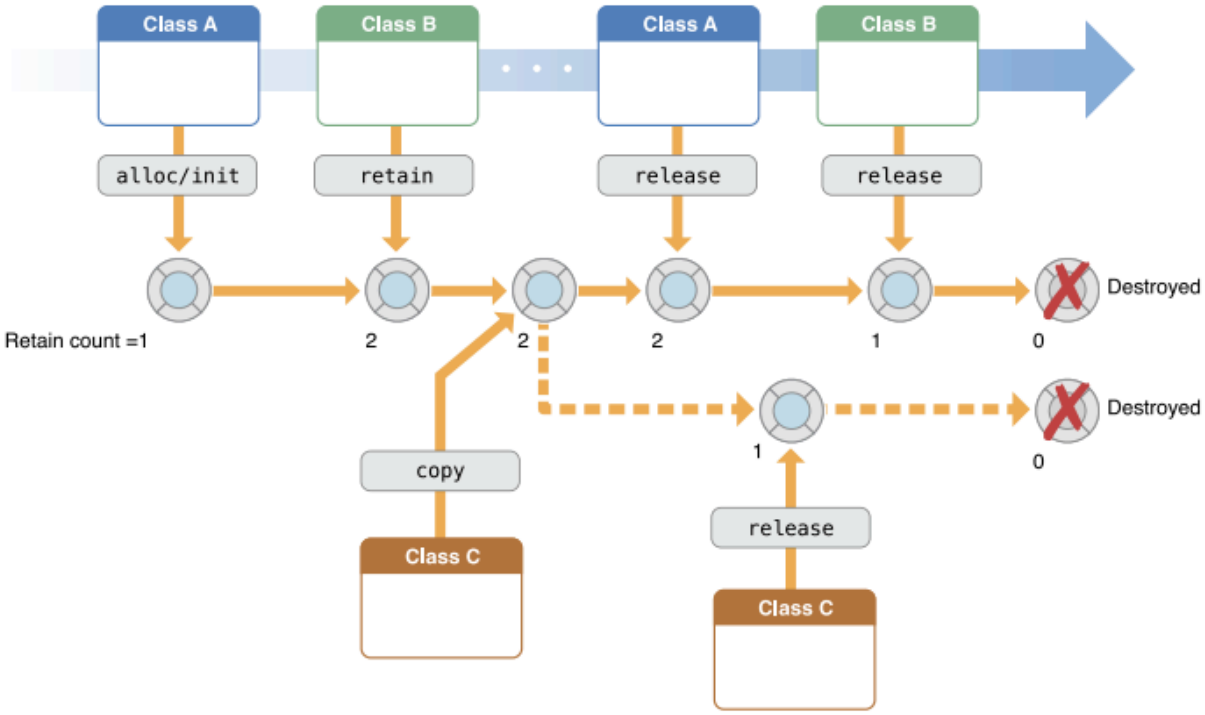
所謂的reference count就是該物件被參考到的記數次數，當物件剛產生（在Heap配置記憶體）reference count=1，如果不再使用則將reference count-1。當一個物件的reference count=0的時候，物件的dealloc method就會被呼叫到，釋放物件所佔有的記憶體。

NSObject Protocol和NSObject有關reference counting的幾個method

NSObject Protocol
- retain: 把物件的reference count+1
- release: 把物件的reference count-1
- autorelease: 在最近的autorelease pool block之後，會把該物件的reference count-1
- retainCount: 以前是用來傳回目前的reference count，但在ARC就被廢棄，而且即使不使用ARC這個數字也不見得準確。比如說NSString的記憶體管理方式就和一般物件不一樣。

NSObject
- copy: 複製一份物件到新的記憶體，reference count由1開始計算，而且不會變動原本物件的reference count

與Java和.NET不同的是在iOS和未來的Mac OS X作業系統不會幫忙應用程式回收沒有在使用的物件(heap裡面)，因此應用程式開發人員必須肩負記憶體管理的責任。Objective-C使用物件的reference count來判斷這個物件是否還有在使用中。
以下圖為例，Class A透過alloc/init產生一個物件k，此時這個物件的retain count為1，如果在Class B使用Class A產生的物件k時，呼叫物件k的retain method，此時物件k的retain count為2。此時如果Class C也要使用物件k，但是Class C使用物件k的copy則會複製一份占有一樣大空間的物件k'而此物件的reference count=1，而且此時Class C對物件k'任何資料的異動都不會影響到Class A和Class B使用中的物件k。
當Class A不需要在使用物件k時，呼叫物件k的release，此時物件k的retain count會減1，接著Class B也呼叫物件k的release，物件k的retain count再減1，此時就會傳遞dealloc給物件k (物件k的dealloc method會被呼叫到)。
Class C不需要使用物件k'時，也呼叫k'的release，此時k'的dealloc也會被呼叫到。



如果沒有正確的使用這個機制來管理物件在heap記憶體的使用狀況，通常就會發生memory leak或者使用到遺失指標（Bad Access）的錯誤，Bad Access還好處理在測試時就會發現。如果是memory leak則會慢慢把記憶體消耗殆盡。

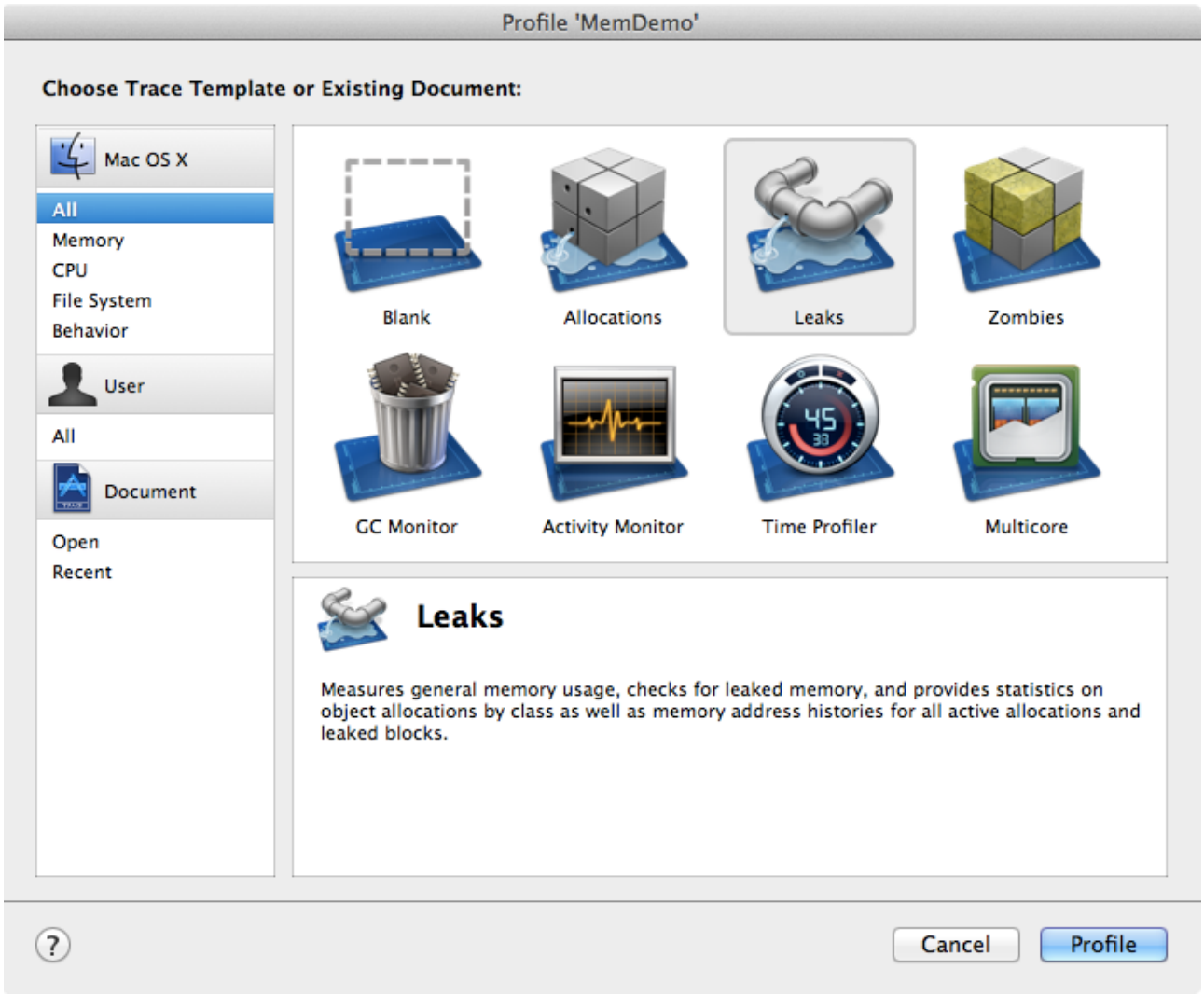
Objective-C記憶體管理的方式
1. Manual Retain-Release (MRR)
由程式設計師自己掌控物件的reference counting。

2. Automatic Reference Count (ARC)
其實這個只是讓程式設計師設定物件使用方法和使用程式設計寫法，在compile成machine code的時候換成MRR的寫法。所以，基本上和MRR沒有兩樣。如果設定錯誤一樣會有問題。

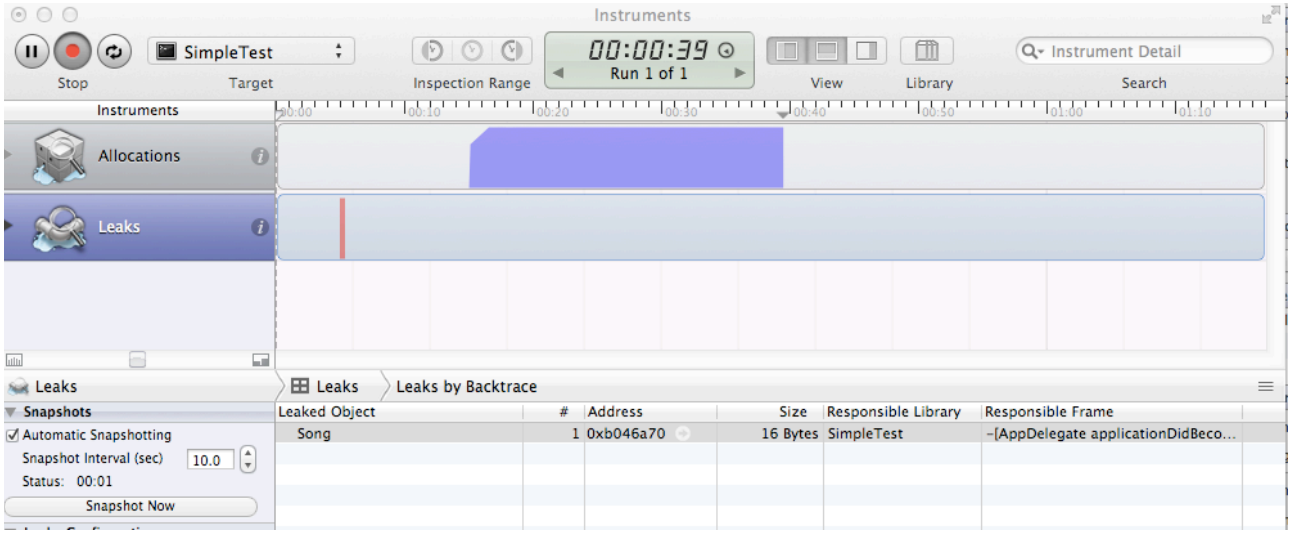
區域變數
先看一下有問題的寫法，下面這個test1() function會產生一個Song的物件，然後使用Song的物件幹一些無聊事，接著就結束test1()

```
int test1() {
    // 產生物件
    Song *s1 = [[Song alloc] init];

    [s1 setName: @"It's my life"];

    [s1 play];

    return 0;
}
```

透過Xcode裡的Product -> Analyze分析程式碼可以發現這個function的問題會有警告，另外也可以透過Product -> Profile開啟Instrument， 會看到下面這張圖的畫面，選擇Leaks分析memory leak



接著可以看到Instrument分析記憶體使用狀況，聽說如果有付保護費，這個工具也可以看測試裝置上iOS應用程式執行的記憶體狀況。
執行上面那個簡單的iOS application就可以在Leaks那個選項看到有一個Song的物件飄在記憶體裡面～

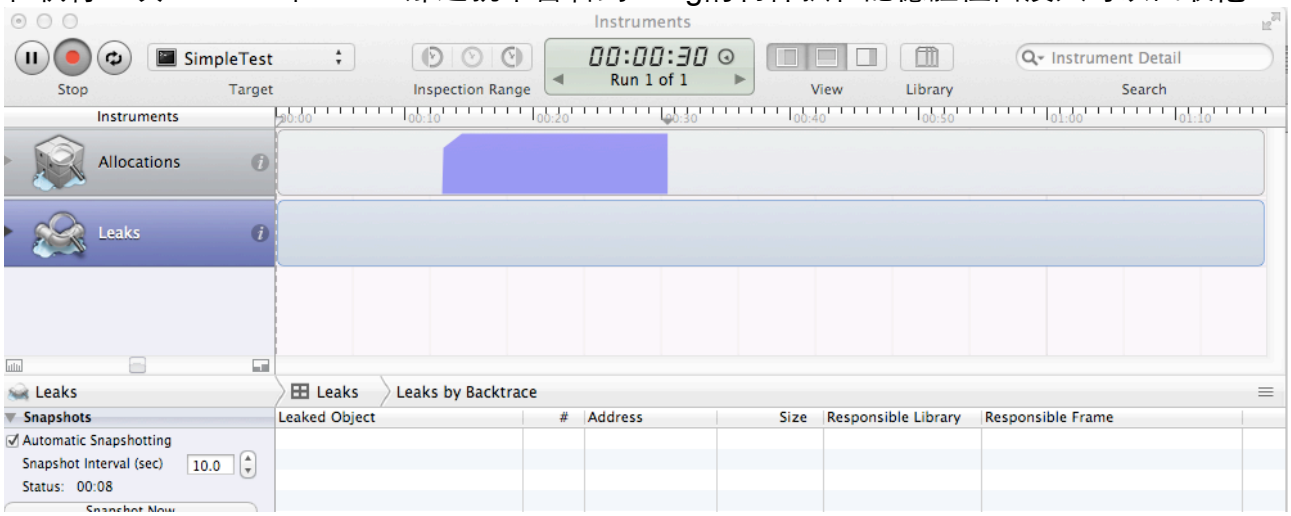


接著我們把test1() function的寫法改成下列這樣

```
int test1() {
    // 產生物件
    Song *s1 = [[Song alloc] init];
```

```
    [s1 setName: @"It's my life"];

    [s1 play];

    // 釋放記憶體
    [s1 release];
    return 0;
}
```

在執行一次Profile，在Leaks那邊就不會看到Song的物件飄在記憶體裡面沒人可以回收他。



因此，要記得**自己透過alloc/init建立的物件，自己就有回收的責任**。在local variable使用狀況下，最好是**透過alloc/init建立物件的同時也補上release的訊息傳遞**。

PS. 如果用Command Line Project測試的話，可能不會看到memory leak，因為GC在Mac OS X據說是最近才拿掉，所以不確定是不是GC功能還在，因此instrument看不到記憶體還飄在heap。

類別裡的物件型別的attribute

假設一個狀況Song類別有一個attribute型別是Singer類別，此時Song類別宣告如下

```
@interface Song : NSObject {
    Singer* _singer;
}

@property (copy) NSString* name;
@property (assign) CGFloat length;

-(void)setSinger:(Singer *) singer;
-(Singer *)singer;

-(void)play;

@end
```

定義的程式碼就只看setter和getter的程式碼也很單純

```
-(void)setSinger:(Singer *)singer {
    _singer = singer;
}

-(Singer *)singer {
    return _singer;
}
```

這樣很像都寫完了，那我們就來使用Song類別看看

```
int testError() {
    Singer *singer = [[Singer alloc] init];
    Song *song = [[Song alloc] init];

    [song setSinger: singer];

    [singer release];
    // 如果只是一般的設定物件到song物件裡，在此時有可能singer
    // 物件已經被release掉，所以下面這行有可能會出錯。
    NSLog( @"Singer: %@", [song singer] );

    [song release];
    return 0;
}
```

如果是這樣的使用狀況執行時就會看到Bad Access的錯誤。於是我們修改使用Song類別的寫法。我們知道retain可以把物件的reference counting+1，因此程式如下

```
int testFixErrorWithError() {
    Singer *singer = [[Singer alloc] init];
    Song *song = [[Song alloc] init];

    [singer retain];
    [song setSinger: singer];

    [singer release];

    NSLog( @"Singer: %@", [song singer] );

    [song release];
    return 0;
}
```

改成這樣寫程式可以執行完畢，但是注意看輸入的Log

```
2013-06-07 15:33:41.421 MemDemo[1281:303] Singer: <Singer: 0x1001082e0>
2013-06-07 15:33:41.424 MemDemo[1281:303] Song <Song: 0x10010a7c0> is
released ...
2013-06-07 15:33:41.424 MemDemo[1281:303] Exit main
```

Singer物件被做出來，但是程式執行完畢dealloc都沒被呼叫到，就表示這個物件飄在記憶體裡面。那我們就在Song物件的dealloc加上一行release singer物件。

```
- (void)dealloc
{
    NSLog( @"Song %@ is released ...", self );
    // Song dealloc時也要釋放singer物件記憶體
    [_singer release];
    [super dealloc];
}
```

再跑一次，終於結果看到Singer物件被release了

```
2013-06-07 15:36:27.081 MemDemo[1293:303] Singer: <Singer: 0x1001082e0>
2013-06-07 15:36:27.084 MemDemo[1293:303] Song <Song: 0x10010a7c0> is
released ...
2013-06-07 15:36:27.085 MemDemo[1293:303] <Singer: 0x1001082e0> is
released ...
2013-06-07 15:36:27.086 MemDemo[1293:303] Exit main
```

因此，快快樂樂寫了下一個程式

```
int testFixErrorWithError2() {
    Singer *singer = [[Singer alloc] init];
```

```
    Singer *singer2 = [[Singer alloc] init];
    Song *song = [[Song alloc] init];

    NSLog( @"Singer1: %@", singer );
    NSLog( @"Singer2: %@", singer2 );

    [singer retain];
    [song setSinger: singer];

    [singer release];

    [singer2 retain];
    [song setSinger: singer2];
    [singer2 release];

    //NSLog( @"Singer: %@", [song singer] );


    [song release];
    return 0;
}
```

這樣寫程式也可以順利執行結束，但是console看到的log卻很令人吐血

```
2013-06-07 15:41:08.650 MemDemo[1325:303] Singer1: <Singer: 0x1001082e0>
2013-06-07 15:41:08.653 MemDemo[1325:303] Singer2: <Singer: 0x100109fd0>
2013-06-07 15:41:08.653 MemDemo[1325:303] Song <Song: 0x10010a7d0> is
released ...
2013-06-07 15:41:08.654 MemDemo[1325:303] <Singer: 0x100109fd0> is
released ...
2013-06-07 15:41:08.654 MemDemo[1325:303] Exit main
```

天殺的singer1...沒有被釋放記憶體....，原因是我們設定新的singer物件的時候，前一個singer物件沒有呼叫release，所以前一個物件的reference count會多1。所以在設定新的Singer物件之前，先取得原本在Song裡面的Singer物件呼叫release method。

```
int testFixError() {
    Singer *singer = [[Singer alloc] init];
    Singer *singer2 = [[Singer alloc] init];
    Song *song = [[Song alloc] init];

    NSLog( @"Singer1: %@", singer );
    NSLog( @"Singer2: %@", singer2 );

    [singer retain];
    [song setSinger: singer];

    [singer release];

    [song.singer release];
    [singer2 retain];
    [song setSinger: singer2];
    [singer2 release];

    [song release];
    return 0;
}
```

Log看起來一切都正常了

```
2013-06-07 15:44:52.583 MemDemo[1341:303] Singer1: <Singer: 0x1001082e0>
2013-06-07 15:44:52.586 MemDemo[1341:303] Singer2: <Singer: 0x100109fd0>
2013-06-07 15:44:52.586 MemDemo[1341:303] <Singer: 0x1001082e0> is
released ...
2013-06-07 15:44:52.587 MemDemo[1341:303] Song <Song: 0x10010a7d0> is
released ...
2013-06-07 15:44:52.587 MemDemo[1341:303] <Singer: 0x100109fd0> is
released ...
2013-06-07 15:44:52.588 MemDemo[1341:303] Exit main
```

如果每個物件都要這樣寫，那大家遲早會瘋掉，所以我們就把release和retain的動作搬到setter method裡面。

```
-(void)setSinger:(Singer *)singer {
    [_singer release];
    _singer = singer;
    [_singer retain];
}
```

同時也要記得Song dealloc的時候也要釋放自己拿到手上的物件的使用空間

```
- (void)dealloc
{
    NSLog( @"Song %@ is released ...", self );
    // Song dealloc時也要釋放singer物件記憶體
    [_singer release];
    [super dealloc];
}
```

因此，一般狀況比較好的寫法就是**在setter的時候先呼叫原本attribute的release，設定之後再呼叫一次新的物件的retain。同時也要在dealloc把物件型別的attribute release。**

[補充] 如果剛建立的Song物件裡的_singer是null value在Objective-C是nil，對nil send任何message（不習慣send message就想成呼叫任何方法）都會被吃掉，不會像Java or C#...會發生Null Pointer Exception之類的錯誤。因此在這個狀況使用setter不會出錯。

Function回傳值為物件

用Singer來做範例，先寫一個簡單的Singer class，然後寫一個first class function回傳一個Singer物件。在這個例子裡面override Singer class 的dealloc method寫出一個log觀察被呼叫的時間點。

因為物件是我們寫的function透過alloc/init產生，但是因為要回傳給使用者，所以不能使用local variable那招使用release，因此在這裡要改用autorelease。

Listing. Singer.h

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

@interface Singer : NSObject

@end
```

Listing. Singer.m

```
#import "Singer.h"

@implementation Singer

- (void)dealloc
{
    NSLog( @"%@ is released ...", self );
    [super dealloc];
```

```
}

@end
```

Listing. createSinger function

```
Singer* createSinger() {
    Singer* newSinger = [[[Singer alloc] init] autorelease];

    NSLog( @"createSinger function: Logging Singer %@", newSinger );

    return newSinger;
}
```

Listing. main function

```
int main(int argc, const char * argv[])
{

    @autoreleasepool {

        // insert code here...
        Singer* singer = createSinger();

        NSLog( @"main function: Logging Singer %@", singer );
    }
    NSLog( @"Exit main" );
    return 0;
}
```

執行結果
就我們所知，autorelease會在最近一個autorelease pool block結束時，才把物件的reference count-1。所以如果執行結果在main function那個log之後，而Exit main那個訊息之前，我們預期Singer物件的dealloc會被呼叫到。

```
2013-06-07 13:05:06.094 MemDemo[1025:303] createSinger function: Logging Singer <Singer: 0x100108300>
2013-06-07 13:05:06.096 MemDemo[1025:303] main function: Logging Singer <Singer: 0x100108300>
2013-06-07 13:05:06.097 MemDemo[1025:303] <Singer: 0x100108300> is released ...
2013-06-07 13:05:06.097 MemDemo[1025:303] Exit main
```

補充說明在NSLog裡面使用%@把物件印出去會呼叫該物件的description method，預設的是<Class Name: Address>。所以在上面例子可以看到createSinger function產生的物件和main拿到的物件都是佔用同一個記憶體位置。

因此如果我們必須寫一個function回傳物件時，記得在alloc/init時加上autorelease。如這樣

```
Singer* newSinger = [[[Singer alloc] init] autorelease];
```

autorelease method也會return self，所以不用擔心newSinger會是nil。

Effective Objective-C 2.0跟記憶體有關有下列幾個item
- Item 29: Understand Reference Counting
- Item 30: Use ARC Make Reference Counting Easier
- Item 31: Release References and Clean Up Observation State Only in dealloc

- Item 32: Beware of Memory Management with Exception-Safe Code
- Item 33: Use Weak Reference to Avoid Retain Cycle
- Item 34: Use Autorelease Pool Blocks to Reduce High-Memory Waterline
- Item 35: Use Zombies to Help Debug Memory-Management Problems
- Item 36: Avoid Using retainCount