Translations: English | Chinese

大綱

開始吧

下載這篇教學

- 所有這篇初學者指南的原始碼都可以由 objc.tar.gz 下載。這篇教學中的許多範例都是由 Steve Kochan 在 Programming in Objective-C. 一書中撰寫。如果你想得到更多詳細資訊及範例，請直接參考該書。這個網站上登載的所有範例皆經過他的允許，所以請勿複製轉載。

設定環境

- Linux/FreeBSD: 安裝 GNUStep
  - 為了編譯 GNUstep 應用程式，必須先執行位於 /usr/GNUstep/System/Makefiles/GNUstep.sh 的 GNUstep.sh 這個檔案。這個路徑取決於你的系統環境，有些是在 /usr, some /usr/lib，有些是 /usr/local。

如果你的 shell 是以 csh/tcsh 為基礎的 shell，則應該改用 GNUStep.csh。建議把這個指令放在 .bashrc 或 .cshrc 中。

- Mac OS X: 安裝 XCode
- Windows NT 5.X: 安裝 cygwin 或 mingw，然後安裝 GNUStep

前言

- 這篇教學假設你已經有一些基本的 C 語言知識，包括 C 資料型別、什麼是函式、什麼是回傳值、關於指標的知識以及基本的 C 語言記憶體管理。如果您沒有這些背景知識，我非常建議你讀一讀 K&R 的書：The C Programming Language（譯注：台灣出版書名為 C 程式語言第二版）這是 C 語言的設計者所寫的書。
- Objective-C，是 C 的衍生語言，繼承了所有 C 語言的特性。是有一些例外，但是它們不是繼承於 C 的語言特性本身。
- nil：在 C/C++ 你或許曾使用過 NULL，而在 Objective-C 中則是 nil。不同之處是你可以傳遞訊息給 nil（例如 [nil message];），這是完全合法的，然而你卻不能對 NULL 如法炮製。
- BOOL：C 沒有正式的布林型別，而在 Objective-C 中也不是「真的」有。它是包含在 Foundation classes（基本類別庫）中（即 import NSObject.h；nil 也是包括在這個標頭檔內）。BOOL 在 Objective-C 中有兩種型態：YES 或 NO，而不是 TRUE 或 FALSE。
- #import vs #include：就如同你在 hello world 範例中看到的，我們使用了 #import。#import 由 gcc 編譯器支援。我並不建議使用 #include，#import 基本上跟 .h 檔頭尾的 #ifndef #define #endif 相同。許多程式員們都同意，使用這些東西這是十分愚蠢的。無論如何，使用 #import 就對了。這樣不但可以避免麻煩，而且萬一有一天 gcc 把它拿掉了，將會有足夠的 Objective-C 程式員可以堅持保留它或是將它放回來。偷偷告訴你，Apple 在它們官方的程式碼中也使用了 #import。所以萬一有一天這種事真的發生，不難預料 Apple 將會提供一個支援 #import 的 gcc 分支版本。
- 在 Objective-C 中， method 及 message 這兩個字是可以互換的。不過 messages 擁有特別的特性，一個 message 可以動態的轉送給另一個物件。在 Objective-C 中，呼叫物件上的一個訊息並不一定表示物件真的會實作這個訊息，而是物件知道如何以某種方式去實作它，或是轉送給知道如何實作的物件。

編譯 hello world

- hello.m

```
#import <stdio.h>

int main( int argc, const char *argv[] ) {
    printf( "hello world\n" );
    return 0;
}
```

- 輸出

```
hello world
```

- 在 Objective-C 中使用 #import 代替 #include
- Objective-C 的預設副檔名是 .m

創建 classes

@interface

- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一書中的範例，並經過允許而刊載。
- Fraction.h

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

@interface Fraction: NSObject {
    int numerator;
    int denominator;
}

-(void) print;
-(void) setNumerator: (int) d;
-(void) setDenominator: (int) d;
-(int) numerator;
-(int) denominator;
@end
```

- NSObject：NeXTStep Object 的縮寫。因為它已經改名為 OpenStep，所以這在今天已經不是那麼有意義了。
- 繼承（inheritance）以 Class: Parent 表示，就像上面的 Fraction: NSObject。
- 夾在 @interface Class: Parent { .... } 中的稱為 instance variables。
- 沒有設定存取權限（protected, public, private）時，預設的存取權限為 protected。設定權限的方式將在稍後說明。
- Instance methods 跟在成員變數（即 instance variables）後。格式為：scope (returnType) methodName: (parameter1Type) parameter1Name;
  - scope 有class 或 instance 兩種。instance methods 以 **-** 開頭，class level methods 以 **+** 開頭。
- Interface 以一個 @end 作為結束。

@implementation

- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一書中的範例，並經過允許而刊載。
- Fraction.m

```
#import "Fraction.h"
#import <stdio.h>

@implementation Fraction
-(void) print {
    printf( "%i/%i", numerator, denominator );
}

-(void) setNumerator: (int) n {
    numerator = n;
}

-(void) setDenominator: (int) d {
    denominator = d;
}

-(int) denominator {
    return denominator;
}

-(int) numerator {
    return numerator;
}
@end
```

- Implementation 以 @implementation ClassName 開始，以 @end 結束。

- Implement 定義好的 methods 的方式，跟在 interface 中宣告時很近似。

◦ 把它們湊在一起

- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing 一書中的範例，並經過允許而刊載。
- main.m

```
#import <stdio.h>
#import "Fraction.h"

int main( int argc, const char *argv[] ) {
    // create a new instance
    Fraction *frac = [[Fraction alloc] init];

    // set the values
    [frac setNumerator: 1];
    [frac setDenominator: 3];

    // print it
    printf( "The fraction is: " );
    [frac print];
    printf( "\n" );

    // free memory
    [frac release];

    return 0;
}
```

- output

```
The fraction is: 1/3
```

- Fraction *frac = [[Fraction alloc] init];
  - 這行程式碼中有很多重要的東西。
  - 在 Objective-C 中呼叫 methods 的方法是 [object method]，就像 C++ 的 object->method()。
  - Objective-C 沒有 value 型別。所以沒有像 C++ 的 Fraction frac; frac.print(); 這類的東西。在 Objective-C 中完全使用指標來處理物件。
  - 這行程式碼實際上做了兩件事： [Fraction alloc] 呼叫了 Fraction class 的 alloc method。這就像 malloc 記憶體，這個動作也做了一樣的事情。
  - [object init] 是一個建構子（constructor）呼叫，負責初始化物件中的所有變數。它呼叫了 [Fraction alloc] 傳回的 instance 上的 init method。這個動作非常普遍，所以通常以一行程式完成：Object *var = [[Object alloc] init];
- [frac setNumerator: 1] 非常簡單。它呼叫了 frac 上的 setNumerator method 並傳入 1 為參數。
- 如同每個 C 的變體，Objective-C 也有一個用以釋放記憶體的方式： release。它繼承自 NSObject，這個 method 在之後會有詳盡的解說。

• 詳細說明...

◦ 多重參數

- 目前為止我還沒展示如何傳遞多個參數。這個語法乍看之下不是很直覺，不過它卻是來自一個十分受歡迎的 Smalltalk 版本。
- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing一書中的

範例，並經過允許而刊載。

- Fraction.h

```
...
-(void) setNumerator: (int) n andDenominator: (int) d;
...
```

- Fraction.m

```
...
-(void) setNumerator: (int) n andDenominator: (int) d {
    numerator = n;
    denominator = d;
}
...
```

- main.m

```
#import <stdio.h>
#import "Fraction.h"

int main( int argc, const char *argv[] ) {
    // create a new instance
    Fraction *frac = [[Fraction alloc] init];
    Fraction *frac2 = [[Fraction alloc] init];

    // set the values
    [frac setNumerator: 1];
    [frac setDenominator: 3];

    // combined set
    [frac2 setNumerator: 1 andDenominator: 5];

    // print it
    printf( "The fraction is: " );
    [frac print];
    printf( "\n" );

    // print it
    printf( "Fraction 2 is: " );
    [frac2 print];
    printf( "\n" );

    // free memory
    [frac release];
    [frac2 release];

    return 0;
}
```

- output

```
The fraction is: 1/3
Fraction 2 is: 1/5
```

- 這個 method 實際上叫做 setNumerator:andDenominator:
- 加入其他參數的方法就跟加入第二個時一樣，即 method:label1:label2:label3: ，而呼叫的方法是 [obj method: param1 label1: param2 label2: param3 label3: param4]
- Labels 是非必要的，所以可以有一個像這樣的 method： method:::，簡單的省略 label 名稱，但以 : 區隔參數。並不建議這樣使用。

◦ 建構子（Constructors）

- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing一書中的範例，並經過允許而刊載。
- Fraction.h

```
...
-(Fraction*) initWithNumerator: (int) n denominator: (int) d;
...
```

- Fraction.m

```
...
-(Fraction*) initWithNumerator: (int) n denominator: (int) d {
    self = [super init];

    if ( self ) {
        [self setNumerator: n andDenominator: d];
    }

    return self;
}
...
```

- main.m

```
#import <stdio.h>
#import "Fraction.h"

int main( int argc, const char *argv[] ) {
    // create a new instance
    Fraction *frac = [[Fraction alloc] init];
    Fraction *frac2 = [[Fraction alloc] init];
    Fraction *frac3 = [[Fraction alloc] initWithNumerator: 3 denominator: 10];

    // set the values
    [frac setNumerator: 1];
    [frac setDenominator: 3];

    // combined set
    [frac2 setNumerator: 1 andDenominator: 5];

    // print it
    printf( "The fraction is: " );
    [frac print];
    printf( "\n" );

    printf( "Fraction 2 is: " );
    [frac2 print];
    printf( "\n" );

    printf( "Fraction 3 is: " );
    [frac3 print];
    printf( "\n" );

    // free memory
    [frac release];
    [frac2 release];
    [frac3 release];

    return 0;
}
```

- output

```
The fraction is: 1/3
Fraction 2 is: 1/5
```

```
Fraction 3 is: 3/10
```

- @interface 裡的宣告就如同正常的函式。
- @implementation 使用了一個新的關鍵字：super
  - 如同 Java，Objective-C 只有一個 parent class（父類別）。
  - 使用 [super init] 來存取 Super constructor，這個動作需要適當的繼承設計。
  - 你將這個動作回傳的 instance 指派給另一新個關鍵字：self。Self 很像 C++ 與 Java 的 this 指標。
- if (self) 跟 (self != nil) 一樣，是為了確定 super constructor 成功傳回了一個新物件。nil 是 Objective-C 用來表達 C/C++ 中 NULL 的方式，可以引入 NSObject 來取得。
- 當你初始化變數以後，你用傳回 self 的方式來傳回自己的位址。
- 預設的建構子是 -(id) init。
- 技術上來說，Objective-C 中的建構子就是一個 "init" method，而不像 C++ 與 Java 有特殊的結構。

存取權限

- 預設的權限是 @protected
- Java 實作的方式是在 methods 與變數前面加上 public/private/protected 修飾語，而 Objective-C 的作法則更像 C++ 對於 instance variable（譯注：C++ 術語一般稱為 data members）的方式。
- Access.h

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

@interface Access: NSObject {
@public
    int publicVar;
@private
    int privateVar;
    int privateVar2;
@protected
    int protectedVar;
}
@end
```

- Access.m

```
#import "Access.h"

@implementation Access
@end
```

- main.m

```
#import "Access.h"
#import <stdio.h>

int main( int argc, const char *argv[] ) {
    Access *a = [[Access alloc] init];

    // works
    a->publicVar = 5;
    printf( "public var: %i\n", a->publicVar );

    // doesn't compile
    //a->privateVar = 10;
    //printf( "private var: %i\n", a->privateVar );

    [a release];
```

```
    return 0;
}
```

- output

```
public var: 5
```

- 如同你所看到的，就像 C++ 中 private: [list of vars] public: [list of vars] 的格式，它只是改成了@private, @protected, 等等。

Class level access

- 當你想計算一個物件被 instance 幾次時，通常有 class level variables 以及 class level functions 是件方便的事。
- ClassA.h

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

static int count;

@interface ClassA: NSObject
+(int) initCount;
+(void) initialize;
@end
```

- ClassA.m

```
#import "ClassA.h"

@implementation ClassA
-(id) init {
    self = [super init];
    count++;
    return self;
}

+(int) initCount {
    return count;
}

+(void) initialize {
    count = 0;
}
@end
```

- main.m

```
#import "ClassA.h"
#import <stdio.h>

int main( int argc, const char *argv[] ) {
    ClassA *c1 = [[ClassA alloc] init];
    ClassA *c2 = [[ClassA alloc] init];

    // print count
    printf( "ClassA count: %i\n", [ClassA initCount] );

    ClassA *c3 = [[ClassA alloc] init];

    // print count again
    printf( "ClassA count: %i\n", [ClassA initCount] );

    [c1 release];
```

```
    [c2 release];
    [c3 release];

    return 0;
}
```

- output

```
ClassA count: 2
ClassA count: 3
```

- static int count = 0; 這是 class variable 宣告的方式。其實這種變數擺在這裡並不理想，比較好的解法是像 Java 實作 static class variables 的方法。然而，它確實能用。
- +(int) initCount; 這是回傳 count 值的實際 method。請注意這細微的差別！這裡在 type 前面不用減號 - 而改用加號 +。加號 + 表示這是一個 class level function。（譯注：許多文件中，class level functions 被稱為 class functions 或 class method）
- 存取這個變數跟存取一般成員變數沒有兩樣，就像 ClassA 中的 count++ 用法。
- +(void) initialize method is 在 Objective-C 開始執行你的程式時被呼叫，而且它也被每個 class 呼叫。這是初始化像我們的 count 這類 class level variables 的好地方。

◦ 異常情況（Exceptions）

- 注意：異常處理只有 Mac OS X 10.3 以上才支援。
- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing一書中的範例，並經過允許而刊載。
- CupWarningException.h

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

@interface CupWarningException: NSException
@end
```

- CupWarningException.m

```
#import "CupWarningException.h"

@implementation CupWarningException
@end
```

- CupOverflowException.h

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

@interface CupOverflowException: NSException
@end
```

- CupOverflowException.m

```
#import "CupOverflowException.h"

@implementation CupOverflowException
@end
```

- Cup.h

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

```
@interface Cup: NSObject {
    int level;
}

-(int) level;
-(void) setLevel: (int) l;
-(void) fill;
-(void) empty;
-(void) print;
@end
```

- Cup.m

```
#import "Cup.h"
#import "CupOverflowException.h"
#import "CupWarningException.h"
#import <Foundation/NSException.h>
#import <Foundation/NSString.h>

@implementation Cup
-(id) init {
    self = [super init];

    if ( self ) {
        [self setLevel: 0];
    }

    return self;
}

-(int) level {
    return level;
}

-(void) setLevel: (int) l {
    level = l;

    if ( level > 100 ) {
        // throw overflow
        NSException *e = [CupOverflowException
            exceptionWithName: @"CupOverflowException"
            reason: @"The level is above 100"
            userInfo: nil];
        @throw e;
    } else if ( level >= 50 ) {
        // throw warning
        NSException *e = [CupWarningException
            exceptionWithName: @"CupWarningException"
            reason: @"The level is above or at 50"
            userInfo: nil];
        @throw e;
    } else if ( level < 0 ) {
        // throw exception
        NSException *e = [NSException
            exceptionWithName: @"CupUnderflowException"
            reason: @"The level is below 0"
            userInfo: nil];
        @throw e;
    }
}

-(void) fill {
    [self setLevel: level + 10];
}

-(void) empty {
```

```
    [self setLevel: level - 10];
}

-(void) print {
    printf( "Cup level is: %i\n", level );
}
@end
```

- main.m

```
#import "Cup.h"
#import "CupOverflowException.h"
#import "CupWarningException.h"
#import <Foundation/NSString.h>
#import <Foundation/NSException.h>
#import <Foundation/NSAutoreleasePool.h>
#import <stdio.h>

int main( int argc, const char *argv[] ) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    Cup *cup = [[Cup alloc] init];
    int i;

    // this will work
    for ( i = 0; i < 4; i++ ) {
        [cup fill];
        [cup print];
    }

    // this will throw exceptions
    for ( i = 0; i < 7; i++ ) {
        @try {
            [cup fill];
        } @catch ( CupWarningException *e ) {
            printf( "%s: ", [[e name] cString] );
        } @catch ( CupOverflowException *e ) {
            printf( "%s: ", [[e name] cString] );
        } @finally {
            [cup print];
        }
    }

    // throw a generic exception
    @try {
        [cup setLevel: -1];
    } @catch ( NSException *e ) {
        printf( "%s: %s\n", [[e name] cString], [[e reason] cString] );
    }

    // free memory
    [cup release];
    [pool release];
}
```

- output

```
Cup level is: 10
Cup level is: 20
Cup level is: 30
Cup level is: 40
CupWarningException: Cup level is: 50
CupWarningException: Cup level is: 60
CupWarningException: Cup level is: 70
CupWarningException: Cup level is: 80
CupWarningException: Cup level is: 90
CupWarningException: Cup level is: 100
CupOverflowException: Cup level is: 110
```

```
CupUnderflowException: The level is below 0
```

- NSAutoreleasePool 是一個記憶體管理類別。現在先別管它是幹嘛的。
- Exceptions（異常情況）的丟出不需要擴充（extend）NSException 物件，你可簡單的用 id 來代表它： @catch (id e) { ... }
- 還有一個 finally 區塊，它的行為就像 Java 的異常處理方式，finally 區塊的內容保證會被呼叫。
- Cup.m 裡的 @"CupOverflowException" 是一個 NSString 常數物件。在 Objective-C 中，@ 符號通常用來代表這是語言的衍生部分。C 語言形式的字串（C string）就像 C/C++ 一樣是 "String constant" 的形式，型別為 char *。

繼承、多型（Inheritance, Polymorphism）以及其他物件導向功能

id 型別

- Objective-C 有種叫做 id 的型別，它的運作有時候像是 void*，不過它卻嚴格規定只能用在物件。Objective-C 與 Java 跟 C++ 不一樣，你在呼叫一個物件的 method 時，並不需要知道這個物件的型別。當然這個 method 一定要存在，這稱為 Objective-C 的訊息傳遞。
- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing一書中的範例，並經過允許而刊載。
- Fraction.h

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

@interface Fraction: NSObject {
    int numerator;
    int denominator;
}

-(Fraction*) initWithNumerator: (int) n denominator: (int) d;
-(void) print;
-(void) setNumerator: (int) d;
-(void) setDenominator: (int) d;
-(void) setNumerator: (int) n andDenominator: (int) d;
-(int) numerator;
-(int) denominator;
@end
```

- Fraction.m

```
#import "Fraction.h"
#import <stdio.h>

@implementation Fraction
-(Fraction*) initWithNumerator: (int) n denominator: (int) d {
    self = [super init];

    if ( self ) {
        [self setNumerator: n andDenominator: d];
    }

    return self;
}

-(void) print {
    printf( "%i / %i", numerator, denominator );
}
```

```
-(void) setNumerator: (int) n {
    numerator = n;
}

-(void) setDenominator: (int) d {
    denominator = d;
}

-(void) setNumerator: (int) n andDenominator: (int) d {
    numerator = n;
    denominator = d;
}

-(int) denominator {
    return denominator;
}

-(int) numerator {
    return numerator;
}
@end
```

- Complex.h

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

@interface Complex: NSObject {
    double real;
    double imaginary;
}

-(Complex*) initWithReal: (double) r andImaginary: (double) i;
-(void) setReal: (double) r;
-(void) setImaginary: (double) i;
-(void) setReal: (double) r andImaginary: (double) i;
-(double) real;
-(double) imaginary;
-(void) print;

@end
```

- Complex.m

```
#import "Complex.h"
#import <stdio.h>

@implementation Complex
-(Complex*) initWithReal: (double) r andImaginary: (double) i {
    self = [super init];

    if ( self ) {
        [self setReal: r andImaginary: i];
    }

    return self;
}

-(void) setReal: (double) r {
    real = r;
}

-(void) setImaginary: (double) i {
    imaginary = i;
}

-(void) setReal: (double) r andImaginary: (double) i {
```

```
    real = r;
    imaginary = i;
}

-(double) real {
    return real;
}

-(double) imaginary {
    return imaginary;
}

-(void) print {
    printf( "%_f + %_fi", real, imaginary );
}

@end
```

- main.m

```
#import <stdio.h>
#import "Fraction.h"
#import "Complex.h"

int main( int argc, const char *argv[] ) {
    // create a new instance
    Fraction *frac = [[Fraction alloc] initWithNumerator: 1 denominator: 10];
    Complex *comp = [[Complex alloc] initWithReal: 10 andImaginary: 15];
    id number;

    // print fraction
    number = frac;
    printf( "The fraction is: " );
    [number print];
    printf( "\n" );

    // print complex
    number = comp;
    printf( "The complex number is: " );
    [number print];
    printf( "\n" );

    // free memory
    [frac release];
    [comp release];

    return 0;
}
```

- output

```
The fraction is: 1 / 10
The complex number is: 10.000000 + 15.000000i
```

- 這種動態連結有顯而易見的好處。你不需要知道你呼叫 method 的那個東西是什麼型別，如果這個物件對這個訊息有反應，那就會喚起這個 method。這也不會牽涉到一堆繁瑣的轉型動作，比如在 Java 裡呼叫一個整數物件的 .intValue() 就得先轉型，然後才能呼叫這個 method。

◦ 繼承（Inheritance）

- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing一書中的範例，並經過允許而刊載。
- Rectangle.h

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

@interface Rectangle: NSObject {
    int width;
    int height;
}

-(Rectangle*) initWithWidth: (int) w height: (int) h;
-(void) setWidth: (int) w;
-(void) setHeight: (int) h;
-(void) setWidth: (int) w height: (int) h;
-(int) width;
-(int) height;
-(void) print;
@end
```

- Rectangle.m

```
#import "Rectangle.h"
#import <stdio.h>

@implementation Rectangle
-(Rectangle*) initWithWidth: (int) w height: (int) h {
    self = [super init];

    if ( self ) {
        [self setWidth: w height: h];
    }

    return self;
}

-(void) setWidth: (int) w {
    width = w;
}

-(void) setHeight: (int) h {
    height = h;
}

-(void) setWidth: (int) w height: (int) h {
    width = w;
    height = h;
}

-(int) width {
    return width;
}

-(int) height {
    return  height;
}

-(void) print {
    printf( "width = %i, height = %i", width, height );
}
@end
```

- Square.h

```
#import "Rectangle.h"

@interface Square: Rectangle
-(Square*) initWithSize: (int) s;
-(void) setSize: (int) s;
```

```
-(int) size;
@end
```

- Square.m

```
#import "Square.h"

@implementation Square
-(Square*) initWithSize: (int) s {
    self = [super init];

    if ( self ) {
        [self setSize: s];
    }

    return self;
}

-(void) setSize: (int) s {
    width = s;
    height = s;
}

-(int) size {
    return width;
}

-(void) setWidth: (int) w {
    [self setSize: w];
}

-(void) setHeight: (int) h {
    [self setSize: h];
}
@end
```

- main.m

```
#import "Square.h"
#import "Rectangle.h"
#import <stdio.h>

int main( int argc, const char *argv[] ) {
    Rectangle *rec = [[Rectangle alloc] initWithWidth: 10 height: 20];
    Square *sq = [[Square alloc] initWithSize: 15];

    // print em
    printf( "Rectangle: " );
    [rec print];
    printf( "\n" );

    printf( "Square: " );
    [sq print];
    printf( "\n" );

    // update square
    [sq setWidth: 20];
    printf( "Square after change: " );
    [sq print];
    printf( "\n" );

    // free memory
    [rec release];
    [sq release];

    return 0;
}
```

- output

```
Rectangle: width = 10, height = 20
Square: width = 15, height = 15
Square after change: width = 20, height = 20
```

- 繼承在 Objective-C 裡比較像 Java。當你擴充你的 super class（所以只能有一個 parent），你想自訂這個 super class 的 method，只要簡單的在你的 child class implementation 裡放上新的實作內容即可。而不需要 C++ 裡呆呆的 virtual table。
- 這裡還有一個值得玩味的地方，如果你企圖像這樣去呼叫 rectangle 的 constructor： Square *sq = [[Square alloc] initWithWidth: 10 height: 15]，會發生什麼事？答案是會產生一個編譯器錯誤。因為 rectangle constructor 回傳的型別是 Rectangle*，而不是 Square*，所以這行不通。在某種情況下如果你真想這樣用，使用 id 型別會是很好的選擇。如果你想使用 parent 的 constructor，只要把 Rectangle* 回傳型別改成 id 即可。

◦ 動態識別（Dynamic types）

- 這裡有一些用於 Objective-C 動態識別的 methods（說明部分採中英並列，因為我覺得英文比較傳神，中文怎麼譯都怪）：

-(BOOL) isKindOfClass: classObj	is object a descendent or member of classObj 此物件是否是 classObj 的子孫或一員
-(BOOL) isMemberOfClass: classObj	is object a member of classObj 此物件是否是 classObj 的一員
-(BOOL) respondsToSelector: selector	does the object have a method named specifiec by the selector 此物件是否有叫做 selector 的 method
+(BOOL) instancesRespondToSelector: selector	does an object created by this class have the ability to respond to the specified selector 此物件是否是由有能力回應指定 selector 的物件所產生
-(id) performSelector: selector	invoke the specified selector on the object 喚起此物件的指定 selector

- 所有繼承自 NSObject 都有一個可回傳一個 class 物件的 class method。這非常近似於 Java 的 getClass() method。這個 class 物件被使用於前述的 methods 中。
- Selectors 在 Objective-C 用以表示訊息。下一個範例會秀出建立 selector 的語法。
- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing一書中的範例，並經過允許而刊載。
- main.m

```
#import "Square.h"
#import "Rectangle.h"
#import <stdio.h>

int main( int argc, const char *argv[] ) {
    Rectangle *rec = [[Rectangle alloc] initWithWidth: 10 height: 20];
    Square *sq = [[Square alloc] initWithSize: 15];

    // isMemberOfClass

    // true
    if ( [sq isMemberOfClass: [Square class]] == YES ) {
        printf( "square is a member of square class\n" );
    }

    // false
    if ( [sq isMemberOfClass: [Rectangle class]] == YES ) {
```

```
        printf( "square is a member of rectangle class\n" );
    }

    // false
    if ( [sq isMemberOfClass: [NSObject class]] == YES ) {
        printf( "square is a member of object class\n" );
    }

    // isKindOfClass

    // true
    if ( [sq isKindOfClass: [Square class]] == YES ) {
        printf( "square is a kind of square class\n" );
    }

    // true
    if ( [sq isKindOfClass: [Rectangle class]] == YES ) {
        printf( "square is a kind of rectangle class\n" );
    }

    // true
    if ( [sq isKindOfClass: [NSObject class]] == YES ) {
        printf( "square is a kind of object class\n" );
    }

    // respondsToSelector

    // true
    if ( [sq respondsToSelector: @selector( setSize: )] == YES ) {
        printf( "square responds to setSize: method\n" );
    }

    // false
    if ( [sq respondsToSelector: @selector( nonExistant )] == YES ) {
        printf( "square responds to nonExistant method\n" );
    }

    // true
    if ( [Square respondsToSelector: @selector( alloc )] == YES ) {
        printf( "square class responds to alloc method\n" );
    }

    // instancesRespondToSelector

    // false
    if ( [Rectangle instancesRespondToSelector: @selector( setSize: )] == YES ) {
        printf( "rectangle instance responds to setSize: method\n" );
    }

    // true
    if ( [Square instancesRespondToSelector: @selector( setSize: )] == YES ) {
        printf( "square instance responds to setSize: method\n" );
    }

    // free memory
    [rec release];
    [sq release];

    return 0;
}
```

- output

```
square is a member of square class
square is a kind of square class
square is a kind of rectangle class
square is a kind of object class
```

```
square responds to setSize: method
square class responds to alloc method
square instance responds to setSize: method
```

Categories

- 當你想要為某個 class 新增 methods，你通常會擴充（extend，即繼承）它。然而這不一定是個完美解法，特別是你想要重寫一個 class 的某個功能，但你卻沒有原始碼時。Categories 允許你在現有的 class 加入新功能，但不需要擴充它。Ruby 語言也有類似的功能。
- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing一書中的範例，並經過允許而刊載。
- FractionMath.h

```
#import "Fraction.h"

@interface Fraction (Math)
-(Fraction*) add: (Fraction*) f;
-(Fraction*) mul: (Fraction*) f;
-(Fraction*) div: (Fraction*) f;
-(Fraction*) sub: (Fraction*) f;
@end
```

- FractionMath.m

```
#import "FractionMath.h"

@implementation Fraction (Math)
-(Fraction*) add: (Fraction*) f {
    return [[Fraction alloc] initWithNumerator: numerator * [f denominator] +
                                                denominator * [f numerator]
                             denominator: denominator * [f denominator]];
}

-(Fraction*) mul: (Fraction*) f {
    return [[Fraction alloc] initWithNumerator: numerator * [f numerator]
                             denominator: denominator * [f denominator]];

}

-(Fraction*) div: (Fraction*) f {
    return [[Fraction alloc] initWithNumerator: numerator * [f denominator]
                             denominator: denominator * [f numerator]];
}

-(Fraction*) sub: (Fraction*) f {
    return [[Fraction alloc] initWithNumerator: numerator * [f denominator] -
                                                denominator * [f numerator]
                             denominator: denominator * [f denominator]];
}
@end
```

- main.m

```
#import <stdio.h>
#import "Fraction.h"
#import "FractionMath.h"

int main( int argc, const char *argv[] ) {
    // create a new instance
    Fraction *frac1 = [[Fraction alloc] initWithNumerator: 1 denominator: 3];
    Fraction *frac2 = [[Fraction alloc] initWithNumerator: 2 denominator: 5];
    Fraction *frac3 = [frac1 mul: frac2];
```

```
    // print it
    [frac1 print];
    printf( " * " );
    [frac2 print];
    printf( " = " );
    [frac3 print];
    printf( "\n" );

    // free memory
    [frac1 release];
    [frac2 release];
    [frac3 release];

    return 0;
}
```

- output

```
1/3 * 2/5 = 2/15
```

- 重點是 @implementation 跟 @interface 這兩行：@interface Fraction (Math) 以及 @implementation Fraction (Math).
- （同一個 class）只能有一個同名的 category，其他的 categories 得加上不同的、獨一無二的名字。
- Categories 在建立 private methods 時十分有用。因為 Objective-C 並沒有像 Java 這種 private/protected/public methods 的概念，所以必須要使用 categories 來達成這種功能。作法是把 private method 從你的 class header (.h) 檔案移到 implementation (.m) 檔案。以下是此種作法一個簡短的範例。
- MyClass.h

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

@interface MyClass: NSObject
-(void) publicMethod;
@end
```

- MyClass.m

```
#import "MyClass.h"
#import <stdio.h>

@implementation MyClass
-(void) publicMethod {
    printf( "public method\n" );
}
@end

// private methods
@interface MyClass (Private)
-(void) privateMethod;
@end

@implementation MyClass (Private)
-(void) privateMethod {
    printf( "private method\n" );
}
@end
```

- main.m

```
#import "MyClass.h"

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

```
    MyClass *obj = [[MyClass alloc] init];

    // this compiles
    [obj publicMethod];

    // this throws errors when compiling
    //[obj privateMethod];

    // free memory
    [obj release];

    return 0;
}
```

- output

```
public method
```

◦ Posing

- Posing 有點像 categories，但是不太一樣。它允許你擴充一個 class，並且全面性地的扮演（pose）這個 super class。例如：你有一個擴充 NSArray 的 NSArrayChild 物件。如果你讓 NSArrayChild 扮演 NSArray，則在你的程式碼中所有的 NSArray 都會自動被替代為 NSArrayChild。
- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing一書中的範例，並經過允許而刊載。
- FractionB.h

```
#import "Fraction.h"

@interface FractionB: Fraction
-(void) print;
@end
```

- FractionB.m

```
#import "FractionB.h"
#import <stdio.h>

@implementation FractionB
-(void) print {
    printf( "(%i/%i)", numerator, denominator );
}
@end
```

- main.m

```
#import <stdio.h>
#import "Fraction.h"
#import "FractionB.h"

int main( int argc, const char *argv[] ) {
    Fraction *frac = [[Fraction alloc] initWithNumerator: 3 denominator: 10];

    // print it
    printf( "The fraction is: " );
    [frac print];
    printf( "\n" );

    // make FractionB pose as Fraction
    [FractionB poseAsClass: [Fraction class]];
```

```
    Fraction *frac2 = [[Fraction alloc] initWithNumerator: 3 denominator: 10];

    // print it
    printf( "The fraction is: " );
    [frac2 print];
    printf( "\n" );

    // free memory
    [frac release];
    [frac2 release];

    return 0;
}
```

- output

```
The fraction is: 3/10
The fraction is: (3/10)
```

- 這個程式的輸出中，第一個 fraction 會輸出 3/10，而第二個會輸出 (3/10)。這是 FractionB 中實作的方式。
- poseAsClass 這個 method 是 NSObject 的一部份，它允許 subclass 扮演 superclass。

◦ Protocols

- Objective-C 裡的 Protocol 與 Java 的 interface 或是 C++ 的 purely virtual class 相同。
- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing一書中的範例，並經過允許而刊載。
- Printing.h

```
@protocol Printing
-(void) print;
@end
```

- Fraction.h

```
#import <Foundation/NSObject.h>
#import "Printing.h"

@interface Fraction: NSObject <Printing, NSCopying> {
    int numerator;
    int denominator;
}

-(Fraction*) initWithNumerator: (int) n denominator: (int) d;
-(void) setNumerator: (int) d;
-(void) setDenominator: (int) d;
-(void) setNumerator: (int) n andDenominator: (int) d;
-(int) numerator;
-(int) denominator;
@end
```

- Fraction.m

```
#import "Fraction.h"
#import <stdio.h>

@implementation Fraction
-(Fraction*) initWithNumerator: (int) n denominator: (int) d {
    self = [super init];
```

```
    if ( self ) {
        [self setNumerator: n andDenominator: d];
    }

    return self;
}

-(void) print {
    printf( "%i/%i", numerator, denominator );
}

-(void) setNumerator: (int) n {
    numerator = n;
}

-(void) setDenominator: (int) d {
    denominator = d;
}

-(void) setNumerator: (int) n andDenominator: (int) d {
    numerator = n;
    denominator = d;
}

-(int) denominator {
    return denominator;
}

-(int) numerator {
    return numerator;
}

-(Fraction*) copyWithZone: (NSZone*) zone {
    return [[Fraction allocWithZone: zone] initWithNumerator: numerator
                                                  denominator: denominator];
}
@end
```

- Complex.h

```
#import <Foundation/NSObject.h>
#import "Printing.h"

@interface Complex: NSObject <Printing> {
    double real;
    double imaginary;
}

-(Complex*) initWithReal: (double) r andImaginary: (double) i;
-(void) setReal: (double) r;
-(void) setImaginary: (double) i;
-(void) setReal: (double) r andImaginary: (double) i;
-(double) real;
-(double) imaginary;
@end
```

- Complex.m

```
#import "Complex.h"
#import <stdio.h>

@implementation Complex
-(Complex*) initWithReal: (double) r andImaginary: (double) i {
    self = [super init];

    if ( self ) {
```

```
        [self setReal: r andImaginary: i];
    }

    return self;
}

-(void) setReal: (double) r {
    real = r;
}

-(void) setImaginary: (double) i {
    imaginary = i;
}

-(void) setReal: (double) r andImaginary: (double) i {
    real = r;
    imaginary = i;
}

-(double) real {
    return real;
}

-(double) imaginary {
    return imaginary;
}

-(void) print {
    printf( "%_f + %_fi", real, imaginary );
}
@end
```

- main.m

```
#import <stdio.h>
#import "Fraction.h"
#import "Complex.h"

int main( int argc, const char *argv[] ) {
    // create a new instance
    Fraction *frac = [[Fraction alloc] initWithNumerator: 3 denominator: 10];
    Complex *comp = [[Complex alloc] initWithReal: 5 andImaginary: 15];
    id <Printing> printable;
    id <NSCopying, Printing> copyPrintable;

    // print it
    printable = frac;
    printf( "The fraction is: " );
    [printable print];
    printf( "\n" );

    // print complex
    printable = comp;
    printf( "The complex number is: " );
    [printable print];
    printf( "\n" );

    // this compiles because Fraction comforms to both Printing and NSCopyable
    copyPrintable = frac;

    // this doesn't compile because Complex only conforms to Printing
    //copyPrintable = comp;

    // test conformance

    // true
    if ( [frac conformsToProtocol: @protocol( NSCopying )] == YES ) {
```

```
        printf( "Fraction conforms to NSCopying\n" );
    }

    // false
    if ( [comp conformsToProtocol: @protocol( NSCopying )] == YES ) {
        printf( "Complex conforms to NSCopying\n" );
    }

    // free memory
    [frac release];
    [comp release];

    return 0;
}
```

- output

```
The fraction is: 3/10
The complex number is: 5.000000 + 15.000000i
Fraction conforms to NSCopying
```

- protocol 的宣告十分簡單，基本上就是 @protocol ProtocolName (methods you must implement) @end。
- 要遵從（conform）某個 protocol，將要遵從的 protocols 放在 <> 裡面，並以逗點分隔。如：@interface SomeClass <Protocol1, Protocol2, Protocol3>
- protocol 要求實作的 methods 不需要放在 header 檔裡面的 methods 列表中。如你所見，Complex.h 檔案裡沒有 -(void) print 的宣告，卻還是要實作它，因為它（Complex class）遵從了這個 protocol。
- Objective-C 的介面系統有一個獨一無二的觀念是如何指定一個型別。比起 C++ 或 Java 的指定方式，如：Printing *someVar = (Printing *) frac; 你可以使用 id 型別加上 protocol：id <Printing> var = frac;。這讓你可以動態地指定一個要求多個 protocol 的型別，卻從頭到尾只用了一個變數。如：<Printing, NSCopying> var = frac;
- 就像使用@selector 來測試物件的繼承關係，你可以使用 @protocol 來測試物件是否遵從介面。如果物件遵從這個介面，[object conformsToProtocol: @protocol(SomeProtocol)] 會回傳一個 YES 型態的 BOOL 物件。同樣地，對 class 而言也能如法炮製 [SomeClass conformsToProtocol: @protocol(SomeProtocol)]。

• 記憶體管理

- 到目前為止我都刻意避開 Objective-C 的記憶體管理議題。你可以呼叫物件上的 dealloc，但是若物件裡包含其他物件的指標的話，要怎麼辦呢？要釋放那些物件所佔據的記憶體也是一個必須關注的問題。當你使用 Foundation framework 建立 classes 時，它如何管理記憶體？這些稍後我們都會解釋。
  - 注意：之前所有的範例都有正確的記憶體管理，以免你混淆。

◦ Retain and Release（保留與釋放）

- Retain 以及 release 是兩個繼承自 NSObject 的物件都會有的 methods。每個物件都有一個內部計數器，可以用來追蹤物件的 reference 個數。如果物件有 3 個 reference 時，不需要 dealloc 自己。但是如果計數器值到達 0 時，物件就得 dealloc 自己。[object retain] 會將計數器值加 1（值從 1 開始），[object release] 則將計數器值減 1。如果呼叫 [object release] 導致計數器到達 0，就會自動 dealloc。
- Fraction.m

```
...
-(void) dealloc {
```

```
    printf( "Deallocing fraction\n" );
    [super dealloc];
}
...
```

- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing一書中的範例，並經過允許而刊載。
- main.m

```
#import "Fraction.h"
#import <stdio.h>

int main( int argc, const char *argv[] ) {
    Fraction *frac1 = [[Fraction alloc] init];
    Fraction *frac2 = [[Fraction alloc] init];

    // print current counts
    printf( "Fraction 1 retain count: %i\n", [frac1 retainCount] );
    printf( "Fraction 2 retain count: %i\n", [frac2 retainCount] );

    // increment them
    [frac1 retain]; // 2
    [frac1 retain]; // 3
    [frac2 retain]; // 2

    // print current counts
    printf( "Fraction 1 retain count: %i\n", [frac1 retainCount] );
    printf( "Fraction 2 retain count: %i\n", [frac2 retainCount] );

    // decrement
    [frac1 release]; // 2
    [frac2 release]; // 1

    // print current counts
    printf( "Fraction 1 retain count: %i\n", [frac1 retainCount] );
    printf( "Fraction 2 retain count: %i\n", [frac2 retainCount] );

    // release them until they dealloc themselves
    [frac1 release]; // 1
    [frac1 release]; // 0
    [frac2 release]; // 0
}
```

- output

```
Fraction 1 retain count: 1
Fraction 2 retain count: 1
Fraction 1 retain count: 3
Fraction 2 retain count: 2
Fraction 1 retain count: 2
Fraction 2 retain count: 1
Deallocing fraction
Deallocing fraction
```

- Retain call 增加計數器值，而 release call 減少它。你可以呼叫 [obj retainCount] 來取得計數器的 int 值。 當當 retainCount 到達 0，兩個物件都會 dealloc 自己，所以可以看到印出了兩個 "Deallocing fraction"。

Dealloc

- 當你的物件包含其他物件時，就得在 dealloc 自己時釋放它們。Objective-C 的一個優點是你可以傳遞訊息給 nil，所以不需要經過一堆防錯測試來釋放一個物件。

- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing一書中的範例，並經過允許而刊載。
- AddressCard.h

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

@interface AddressCard: NSObject {
    NSString *first;
    NSString *last;
    NSString *email;
}

-(AddressCard*) initWithFirst: (NSString*) f
                last: (NSString*) l
                email: (NSString*) e;
-(NSString*) first;
-(NSString*) last;
-(NSString*) email;
-(void) setFirst: (NSString*) f;
-(void) setLast: (NSString*) l;
-(void) setEmail: (NSString*) e;
-(void) setFirst: (NSString*) f
        last: (NSString*) l
        email: (NSString*) e;
-(void) setFirst: (NSString*) f last: (NSString*) l;
-(void) print;
@end
```

- AddressCard.m

```
#import "AddressCard.h"
#import <stdio.h>

@implementation AddressCard
-(AddressCard*) initWithFirst: (NSString*) f
                last: (NSString*) l
                email: (NSString*) e {
    self = [super init];

    if ( self ) {
        [self setFirst: f last: l email: e];
    }

    return self;
}

-(NSString*) first {
    return first;
}

-(NSString*) last {
    return last;
}

-(NSString*) email {
    return email;
}

-(void) setFirst: (NSString*) f {
    [f retain];
    [first release];
    first = f;
}

-(void) setLast: (NSString*) l {
```

```
    [l retain];
    [last release];
    last = l;
}

-(void) setEmail: (NSString*) e {
    [e retain];
    [email release];
    email = e;
}

-(void) setFirst: (NSString*) f
        last: (NSString*) l
        email: (NSString*) e {
    [self setFirst: f];
    [self setLast: l];
    [self setEmail: e];
}

-(void) setFirst: (NSString*) f last: (NSString*) l {
    [self setFirst: f];
    [self setLast: l];
}

-(void) print {
    printf( "%s %s <%s>", [first cString],
                                [last cString],
                                [email cString] );
}

-(void) dealloc {
    [first release];
    [last release];
    [email release];

    [super dealloc];
}
@end
```

- main.m

```
#import "AddressCard.h"
#import <Foundation/NSString.h>
#import <stdio.h>

int main( int argc, const char *argv[] ) {
    NSString *first =[[NSString alloc] initWithCString: "Tom"];
    NSString *last = [[NSString alloc] initWithCString: "Jones"];
    NSString *email = [[NSString alloc] initWithCString: "tom@jones.com"];
    AddressCard *tom = [[AddressCard alloc] initWithFirst: first
                                            last: last
                                            email: email];

    // we're done with the strings, so we must dealloc them
    [first release];
    [last release];
    [email release];

    // print to show the retain count
    printf( "Retain count: %i\n", [[tom first] retainCount] );
    [tom print];
    printf( "\n" );

    // free memory
    [tom release];

    return 0;
```

```
}
```

- output

```
Retain count: 1
Tom Jones <tom@jones.com>
```

- 如 AddressCard.m，這個範例不僅展示如何撰寫一個 dealloc method，也展示了如何 dealloc 成員變數。
- 每個 set method 裡的三個動作的順序非常重要。假設你把自己當參數傳給一個自己的 method（有點怪，不過確實可能發生）。若你先 release，「然後」才 retain，你會把自己給解構（destruct，相對於建構）！ 這就是為什麼應該要 1) retain 2) release 3) 設值 的原因。
- 通常我們不會用 C 形式字串來初始化一個變數，因為它不支援 unicode。下一個 NSAutoreleasePool 的例子會用展示正確使用並初始化字串的方式。
- 這只是處理成員變數記憶體管理的一種方式，另一種方式是在你的 set methods 裡面建立一份拷貝。

Autorelease Pool

- 當你想用 NSString 或其他 Foundation framework classes 來做更多程式設計工作時，你需要一個更有彈性的系統，也就是使用 Autorelease pools。
- 當開發 Mac Cocoa 應用程式時，autorelease pool 會自動地幫你設定好。
- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing一書中的範例，並經過允許而刊載。
- main.m

```
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>
#import <stdio.h>

int main( int argc, const char *argv[] ) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSString *str1 = @"constant string";
    NSString *str2 = [NSString stringWithString: @"string managed by the pool"];
    NSString *str3 = [[NSString alloc] initWithString: @"self managed string"];

    // print the strings
    printf( "%s retain count: %x\n", [str1 cString], [str1 retainCount] );
    printf( "%s retain count: %x\n", [str2 cString], [str2 retainCount] );
    printf( "%s retain count: %x\n", [str3 cString], [str3 retainCount] );

    // free memory
    [str3 release];

    // free pool
    [pool release];
    return 0;
}
```

- output

```
constant string retain count: ffffffff
string managed by the pool retain count: 1
self managed string retain count: 1
```

- 如果你執行這個程式，你會發現幾件事：第一件事，str1 的 retainCount 為 ffffffff。
- 另一件事，雖然我只有 release str3，整個程式卻還是處於完美的記憶體管理下，原因是第一個常數字串已經自動被加到 autorelease pool 裡了。還有一件事，字串

是由 stringWithString 產生的。這個 method 會產生一個 NSString class 型別的字串，並自動加進 autorelease pool。

- 千萬記得，要有良好的記憶體管理，像 [NSString stringWithString: @"String"] 這種 method 使用了 autorelease pool，而 alloc method 如 [[NSString alloc] initWithString: @"String"] 則沒有使用 auto release pool。
- 在 Objective-C 有兩種管理記憶體的方法， 1) retain and release or 2) retain and release/autorelease。
- 對於每個 retain，一定要對應一個 release 「或」一個 autorelease。
- 下一個範例會展示我說的這點。
- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing一書中的範例，並經過允許而刊載。
- Fraction.h

```
...
+(Fraction*) fractionWithNumerator: (int) n denominator: (int) d;
...
```

- Fraction.m

```
...
+(Fraction*) fractionWithNumerator: (int) n denominator: (int) d {
    Fraction *ret = [[Fraction alloc] initWithNumerator: n denominator: d];
    [ret autorelease];

    return ret;
}
...
```

- main.m

```
#import <Foundation/NSAutoreleasePool.h>
#import "Fraction.h"
#import <stdio.h>

int main( int argc, const char *argv[] ) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    Fraction *frac1 = [Fraction fractionWithNumerator: 2 denominator: 5];
    Fraction *frac2 = [Fraction fractionWithNumerator: 1 denominator: 3];

    // print frac 1
    printf( "Fraction 1: " );
    [frac1 print];
    printf( "\n" );

    // print frac 2
    printf( "Fraction 2: " );
    [frac2 print];
    printf( "\n" );

    // this causes a segmentation fault
    //[frac1 release];

    // release the pool and all objects in it
    [pool release];
    return 0;
}
```

- output

```
Fraction 1: 2/5
Fraction 2: 1/3
```

- 在這個例子裡，此 method 是一個 class level method。在物件建立後，在它上面

呼叫 了 autorelease。在 main method 裡面，我從未在此物件上呼叫 release。
- 這樣行得通的原因是：對任何 retain 而言，一定要呼叫一個 release 或 autorelease。物件的 retainCount 從 1 起跳，然後我在上面呼叫 1 次 autorelease，表示 1 - 1 = 0。當 autorelease pool 被釋放時，它會計算所有物件上的 autorelease 呼叫次數，並且呼叫相同次數的 [obj release]。
- 如同註解所說，不把那一行註解掉會造成分段錯誤（segment fault）。因為物件上已經呼叫過 autorelease，若再呼叫 release，在釋放 autorelease pool 時會試圖呼叫一個 nil 物件上的 dealloc，但這是不允許的。最後的算式會變為：1 (creation) - 1 (release) - 1 (autorelease) = -1
- 管理大量暫時物件時，autorelease pool 可以被動態地產生。你需要做的只是建立一個 pool，執行一堆會建立大量動態物件的程式碼，然後釋放這個 pool。你可能會感到好奇，這表示可能同時有超過一個 autorelease pool 存在。

• Foundation framework classes

- Foundation framework 地位如同 C++ 的 Standard Template Library。不過 Objective-C 是真正的動態識別語言（dynamic types），所以不需要像 C++ 那樣肥得可怕的樣版（templates）。這個 framework 包含了物件組、網路、執行緒，還有更多好東西。

◦ NSArray

- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing一書中的範例，並經過允許而刊載。
- main.m

```
#import <Foundation/NSArray.h>
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSEnumerator.h>
#import <stdio.h>

void print( NSArray *array ) {
    NSEnumerator *enumerator = [array objectEnumerator];
    id obj;

    while ( obj = [enumerator nextObject] ) {
        printf( "%s\n", [[obj description] cString] );
    }
}

int main( int argc, const char *argv[] ) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSArray *arr = [[NSArray alloc] initWithObjects:
                    @"Me", @"Myself", @"I", nil];
    NSMutableArray *mutable = [[NSMutableArray alloc] init];

    // enumerate over items
    printf( "----static array\n" );
    print( arr );

    // add stuff
    [mutable addObject: @"One"];
    [mutable addObject: @"Two"];
    [mutable addObjectsFromArray: arr];
    [mutable addObject: @"Three"];

    // print em
    printf( "----mutable array\n" );
    print( mutable );

    // sort then print
```

```
    printf( "----sorted mutable array\n" );
    [mutable sortUsingSelector: @selector( caseInsensitiveCompare: )];
    print( mutable );

    // free memory
    [arr release];
    [mutable release];
    [pool release];

    return 0;
}
```

- output

```
----static array
Me
Myself
I
----mutable array
One
Two
Me
Myself
I
Three
----sorted mutable array
I
Me
Myself
One
Three
Two
```

- 陣列有兩種（通常是 Foundation classes 中最資料導向的部分），NSArray 跟 NSMutableArray，顧名思義，mutable（善變的）表示可以被改變，而 NSArray 則不行。這表示你可以製造一個 NSArray 但卻不能改變它的長度。
- 你可以用 Obj, Obj, Obj, ..., nil 為參數呼叫建構子來初始化一個陣列，其中 nil 表示結尾符號。
- 排序（sorting）展示如何用 selector 來排序一個物件，這個 selector 告訴陣列用 NSString 的忽略大小寫順序來排序。如果你的物件有好幾個排序方法，你可以使用這個 selector 來選擇你想用的方法。
- 在 print method 裡，我使用了 description method。它就像 Java 的 toString，會回傳物件的 NSString 表示法。
- NSEnumerator 很像 Java 的列舉系統。while (obj = [array objectEnumerator]) 行得通的理由是 objectEnumerator 會回傳最後一個物件的 nil。在 C 裡 nil 通常代表 0，也就是 false。改用 ((obj = [array objectEnumerator]) != nil) 也許更好。

◦ NSDictionary

- 基於 "Programming in Objective-C," Copyright © 2004 by Sams Publishing一書中的範例，並經過允許而刊載。
- main.m

```
#import <Foundation/NSString.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSDictionary.h>
#import <Foundation/NSEnumerator.h>
#import <Foundation/Foundation.h<
#import <stdio.h>

void print( NSDictionary *map ) {
    NSEnumerator *enumerator = [map keyEnumerator];
```

```
    id key;

    while ( key = [enumerator nextObject] ) {
        printf( "%s => %s\n",
                [[key description] cString],
                [[[map objectForKey: key] description] cString] );
    }
}

int main( int argc, const char *argv[] ) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSDictionary *dictionary = [[NSDictionary alloc] initWithObjectsAndKeys:
        @"one", [NSNumber numberWithInt: 1],
        @"two", [NSNumber numberWithInt: 2],
        @"three", [NSNumber numberWithInt: 3],
        nil];
    NSMutableDictionary *mutable = [[NSMutableDictionary alloc] init];

    // print dictionary
    printf( "----static dictionary\n" );
    print( dictionary );

    // add objects
    [mutable setObject: @"Tom" forKey: @"tom@jones.com"];
    [mutable setObject: @"Bob" forKey: @"bob@dole.com" ];

    // print mutable dictionary
    printf( "----mutable dictionary\n" );
    print( mutable );

    // free memory
    [dictionary release];
    [mutable release];
    [pool release];

    return 0;
}
```

- output

```
----static dictionary
1 => one
2 => two
3 => three
----mutable dictionary
bob@dole.com => Bob
tom@jones.com => Tom
```

優點與缺點

優點

- Cateogies
- Posing
- 動態識別
- 指標計算
- 彈性訊息傳遞
- 不是一個過度複雜的 C 衍生語言
- 可透過 Objective-C++ 與 C++ 結合

缺點

- 不支援命名空間
- 不支援運算子多載（雖然這常常被視為一個優點，不過**正確地**使用運算子多載可以降低程式碼複雜度）
- 語言裡仍然有些討厭的東西，不過不比 C++ 多。

• 更多資訊

- Object-Oriented Programming and the Objective-C Language
- GNUstep mini tutorials
- Programming in Objective-C
- Learning Cocoa with Objective-C
- Cocoa Programming for Mac OS X

Last modified: April 13, 2004.
中文翻譯：William Shih (xamous)，January 7, 2005