**SUMMARY DOCUMENT**

**TRƯỚC KHI SÀI 1 CON TRỎ BẤT KỲ BAO H CŨNG PHẢI KHAI BÁO TRƯỚC, RỒI MỚI SÀI.**

**CHAPTER 2**

* Code begin

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

{

NSAutoreleasePool \* pool = [[NSAutoreleasePool alloc] init];

NSLog (**@**”Programming is fun!”); // Xuất 1 chuỗi trên giao diện console dùng NSLog(@” ”)

[pool drain];

return 0;

}

*Objective-C, lowercase and uppercase letters are distinct.*

Displaying value of variables

#import <Foundation/Foundation.h>

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

{

NSAutoreleasePool \* pool = [[NSAutoreleasePool alloc] init];

int sum;

sum = 50 + 25;

NSLog (**@**”The sum of 50 and 25 is **%i** ”, sum); // Xuất 1 biến kiểu int dùng %i hoặc %d

[pool drain];

return 0;

}

**CHAPTER 3**

* Classes-Objects-Methods

1. Object

- An object is a thing. Think about object-oriented programming as a thing and something you want to do to that thing. This is in contrast to a programming language such as C, known as a procedural programming language. In C, you typically think about what you want to do first and then you worry about the objects, almost the opposite of object orientation.

- Objects are unique representations from a class, and each object contains some information (data) that is typically private to that object.

1. Instances and Methods

*Theory*

- A unique occurrence of a class is an instance, and the actions that are performed on the instance are called methods.

- The methods provide the means of accessing and changing that data.

- @*interface* section describes the class, its data components, and its methods.

- @implementation section contains the actual code that implements these methods.

- “+ methods” are class methods - that is, methods which do not have access to an instances properties. Used for methods like alloc or helper methods for the class that do not require access to instance variables.

- “- methods” are instance methods - relate to a single instance of an object. Usually used for most methods on a class.

- Instance method can always directly access its instance variables.

- A class method can’t access instance variables because it’s dealing only with the class itself, not with any instances of the class.

*Multiple Parameters*

@interface Fraction: NSObject

{

…

-(void) setNumerator: (int) n andSetDenominator: (int) d andResult: (int) e;

…

}

@implementation Fraction

{

…

-(void) setNumerator: (int) n andSetDenominator: (int) d andResult: (int) e

{

numerator = n;

denominator = d;

denominator = e;

}

…

}

*Forming names in Objective-C*

- Must begin with a letter or under-score (\_), and they can be followed by any combination of letters (upper- or lowercase), underscores, or the digits 0–9.

- On the other hand, the following names are not valid for the stated reasons:

1. sum$value $— is not a valid character.
2. piece flag— Embedded spaces are not permitted.
3. 3Spencer— Names can’t start with a number.
4. int— This is a reserved word.

*Note:*

- Naming a class, start it with a capital letter.

- Instance variables, objects, and method names, on the other hand, typically begin with lowercase letters.

1. AddressBook — This could be a class name.
2. currentEntry — This could be an object.
3. current\_entry — Some programmers use underscores as word separators.
4. addNewEntry — This could be a method name.

*Coding*

- The Objective-C programming language has the following particular syntax for applying methods to classes and instances:

[ ClassOrInstance method ];

*Ex:* - yourCar = [Car new]; //get a new car

* Now, yourCar can be used to refer to your instance of the car

- [yourCar drive];

- currentMileage = [suesCar currentOdometer] // shows an instance method that returns information

Demo code to use define the object alloc init and use:  *Page 32 Chapter 3: Classes Object Methods*

#import <Foundation/Foundation.h>

*@interface Fraction: NSObject*

{

int numerator;

int denominator;

}

-(void) print;

-(void) setNumerator: (int) n;

-(void) setDenominator: (int) d;

@end

*@implementation Fraction*

-(void) print

{

NSLog(@”%i/%i”, numerator,denominator);

}

-(void) setNumerator: (int) n

{

numerator = n;

}

-(void) setDenominator: (int) d

{

denominator = d;

}

@end

int main(int argc, char \*argv [])

{

…

//Create an instance of a Fraction

Fraction \*fraction;

fraction= [[Fraction alloc]init];

// or fraction = [Fraction new];

//Set fraction 1/3

[fraction setNumerator: 1];

[fraction setDenominator: 3];

//Display the fraction

[fraction print];

//Release the point

[fraction release];

…

}

*- Return Value*

@interface Fraction: NSObject

{

}

-(int) getNumerator;

-(int) getDenominator;

@end

.................................................................

@implementation Fraction

-(int) getNumerator

{

return numerator;

}

-(int) getDenominator

{

return denominator;

}

@end

**CHAPTER 7**

* More on Classes

Theory

*- Synthesized Accessor Methods*:As of Objective-C 2.0, you can have your setter and getter methods automatically generated for you, following these steps:

* The first step is to use the @property directive in your interface section to identify your properties. Ex: @property int numerator, denominator;
* The second step is to use the @synthesize directive in the implementation section. Ex: @synthesize numerator, denominator;

- *Accessing Properties Using the Dot Operator*: To get the value of the numerator stored in myFraction, you could write this: [myFraction numerator] but now you can also write the following equivalent expression using the dot operator: *myfraction.numerator*.

* The general format here is: **instance.property**.
* *Methods Without Argument Names*:No name is given for the second argument to the method here.

This method is named set::, and the two colons mean the method takes two arguments, even though they’re not all named. Ex -(int) set: (int) n: (int) d;

*- Operations on Fractions*: You’ll name the method add:, and you’ll have it take a fraction as an argument: -(void) add: **(Fraction \*) f**; *OR* **–(Fraction\*)** add: **(Fraction\*) f**;

- *The* *static Keyword*: You can have a local variable retain its value through multiple invocations of a method by placing the keyword static in front of the variable’s declaration: static int hitCount = 0; *they are initialized only once* *when program execution begins* and retain their values through successive method calls. The static variable can be accessed only from within the method which has this static variable. You can move the declaration of the variable outside any method decla- ration (typically near the beginning of your implementation file) to make it accessible to any methods.

- *The self**Keyword*: like “this” key word in C/C++.

- *Allocating and Returning Objects from Methods*: Control to release the pointer which is in method but can not release in the method because must be return the pointer to the method. Page 150 (Cool !!!)

Coding

*- Synthesized Accessor Methods*

@interface Fraction: NSObject

{

int numerator;

int denominator;

}

@property int numerator, denominator;

..................................................................................

@implementation Fraction

@synthesize numerator, denominator;

*- Accessing Properties Using the Dot Operator*

int main (…)

{

myfraction.numerator = 3;

//[myFraction numerator];

}

- *Methods Without Argument Names*

@interface Fraction:NSObject

{

…

}

-(int) set : (int) n : (int) d;

………………………………………………..

@implementation Fraction

-(int) set : (int) n : (int) d

{

…

}

int main (…)

{

…

Fraction \*fraction = [[Fraction alloc]init];

[fraction set :1 :3];

…

}

*- Operations on Fractions*

Page 157

* *Static Keyword*

-(void) showPage

{

static int pageCount = 0; // *it is initialized only once* *when program execution begins*

++pageCount;

}

#import “Printer.h”

static int pageCount;

@implementation Printer

…

@end

* Self Keyword

Page 160

* *Allocating and Returning Objects from Methods*

Page 161

**EXTRA: KỸ THUẬT RELEASE CHO CÁC CON TRỎ SAU KHI SỬ DỤNG XONG**

Theory

-release

* Release một con trỏ thực chất là giải phóng vùng giá trị mà con trỏ trỏ tới. Nên khi release một con trỏ ta quan tâm đến con trỏ đang trỏ tới vùng giá trị nào.

- retain (Page 416)

* Khi gán 1 con trỏ với 1 con trỏ khác (tức 2 con trỏ trỏ vào cùng 1 vùng nhớ!) thì 2 con trỏ này sẽ cùng chỉ đến 1 vùng giá trị (tương tự C/C++) do đó bug sẽ xuất hiện nếu con trỏ thứ nhất được release nhưng con trỏ thứ 2 vẫn trỏ tới vùng giá mà con trỏ thứ nhất vừa được release. (Khi một con trỏ được release thực chất là xóa vùng giá trị của con trỏ trỏ tới). Do đó để tránh tình trạng này xảy ra ta sẽ dùng referece couting để kiểm soát việc release một vùng nhớ mà các con trỏ cùng trỏ tới. Khi ta gán một con trỏ này với con trỏ khác (tức 2 con trỏ trỏ vào cùng 1 vùng nhớ!) thì cần tăng biến đếm reference counting lên 1 để khi ta release con trỏ thứ nhất sẽ không ảnh hưởng con trỏ thứ 2 vì mỗi lần release thì vùng giá trị sẽ giảm xuống 1 mà ta vừa tăng lên 1 nên sẽ không ảnh hưởng.
* Để tăng reference counting ta sẽ sử dụng lệnh retain. Ex [firstPoint retain];
* ??? Tăng reference counting thực chất là tăng cho vùng giá trị được trỏ tới do đó:

[firstPoint retain] == [second retain]

* Dùng hàm trả về đối tượng con trỏ để release
* Khi ta khai báo và cấp phát một đối tượng con trỏ trong hàm và sử dụng đối tượng đó nhưng trong hàm đó ta chưa muốn release đối tượng đó hoặc có lý do nào đó ta không release đối tượng mà ta đã cấp phát trước đó được thì ta sẽ release thông qua hàm mà đối tượng đó được trả về hoặc ta có thể dùng chế độ autorelease.
* Dùng đối tượng được gán cho hàm để release
* Khi ta khai báo và cấp phát một đối tượng con trỏ trong hàm và return đối tượng cho hàm thì ta không thể release đối tượng đó trong hàm. Do đó ta sẽ khai báo (chưa cấp phát) đối tượng mới trong hàm main để gán đối tượng này với hàm (sau khi gán đối tượng này tự động được cấp phát), như vậy khi đối tượng này dùng xong ta sẽ relase đối tượng này cũng chính là release đối tượng trong hàm được khai báo và cấp phát vì cả 2 cùng trỏ tới 1 vùng nhớ.
* Release vùng giá trị cũ của con trỏ trong class
* Khi ta sử dụng 1 biến con trỏ trong 1 class và gán nó với 1 con trỏ khác trong phương thức của lớp đó thì trước khi sử dụng nó ta nên release vùng giá trị cũ mà nó đã nắm giữ trước khi sử dụng. Nguyên nhân là làm sạch vùng giá trị trước đó mà con trỏ nắm giữ nhằm bảo đảm không lỗi bộ nhớ.
* Sử dụng autorelease
* Autorelease là kỹ thuật được sử dụng nhằm mục đích cho con trỏ tự động release khi pool release. Khi ta gọi hàm [faction autorelease] thì reference counting sẽ vẫn chưa giảm (tức chưa release) mà chỉ đánh dấu là đối tượng này sẽ được release, và thực sự release khi pool release.
* Sử dụng phương thức deadlloc
* Khi viết phương thức dealloc thực chất là ta đang override phương thức dealloc của lớp NSObject mà ta đang kế thừa. Việc override phương thức này nhằm mục đích giải phóng lần cuối tất cả các đối tượng trong lớp hiện tại khi mà vùng nhớ được release. (Page 427) Sau khi system tự động gọi phương thức dealloc của lớp hiện tại nó sẽ tiếp tục gọi phương thức dealloc của lớp parent tức là NSObject thông qua lệnh [super dealloc] để hoàn thành việc release.
* Sử dụng garbage collection
* Kỹ thuật này không xuất hiện trong lập trình Iphone. Nó là kỹ thuật được dùng để cho hệ thống tự động kiểm soát các đối tượng và release tự động các đối tượng. (Page 431)
* Reference counting
* When an object is created, its reference count is set to 1. Each time you need to ensure that the object be kept around, you increment its reference count by 1 by sending it a retain message, like so: [myFraction retain];
* When you no longer need an object, you decrement its reference count by 1 by sending it a release message, like this: [myFraction release];

***Note:*** Khi sử dụng NSArray hay NSMutableArray thì các object được add vào sẽ tự động tăng reference counting lên 1 và giảm xuống 1 khi remove khỏi array.

Coding

* retain
* Wrong Code

firstPoint = [[XYPoint alloc]init];

secondPoint = [[XYPoint alloc]init];

[firstPoint setX: 1 andY: 2];

secondPoint = firstPoint; //Cùng trỏ tới một vùng nhớ

[firstPoint releasel]; // Giải phóng con trỏ firstPoint

NSLog(@"Tọa độ %i %i", secondPoint.x, secondPoint.y); // *Không xuất được vì vùng nhớ đã xóa bởi firstPoint*

* Right Code

firstPoint = [[XYPoint alloc]init];

secondPoint = [[XYPoint alloc]init];

[firstPoint setX: 1 andY: 2];

secondPoint = firstPoint; //Cùng trỏ tới một vùng nhớ

[firstPoint retain]; // Tăng reference counting

[firstPoint releasel]; // Giải phóng con trỏ firstPoint

NSLog(@"Tọa độ %i %i", secondPoint.x, secondPoint.y); // Xuất thành công vì vùng nhớ chưa được xóa hết do tăng reference

* Dùng hàm trả về đối tượng con trỏ để release

-(XYPoint\*) Original

{

return original;

}

-(void) setOriginal: (XYPoint\*) point

{

//Improve

original = [[XYPoint alloc]init]; //Chưa release dc sau khi thoát khỏi hàm

[original setX : point.x andY: point.y];

}

int main(…)

{

…

[myRectangle setOriginal: myPoint];

…

[[myRectangle Original] release]; //Sau khi sài xong thì release được đối tượng

…

}

* Dùng đối tượng được gán cho hàm để release

-(Fraction\*) sumFractions: (Fraction\*) fraction

{

Fraction \* result;

result = [[Fraction alloc]init];

result.numerator = numerator \* fraction.denominator + denominator \* fraction.numerator;

result.denominator = denominator \* fraction.denominator;

return result; // **result chưa được release**

}

int main (…)

{

…

Fraction \* sum;

sum = [fractiona sumFractions: fractionb];

[sum release]; // **result mới được thật sự release**

…

}

* Release giá trị cũ của con trỏ trong class

-(void)setNewString: (NSString \*) newString

{

[str release]; **// Xóa vùng giá trị cũ trước khi khởi gán**

str = [newString retain]; **//++Reference Counting: Tránh trường hợp xảy ra khi autorelase hoặc chuỗi trong hàm main bị release sẽ bị lỗi vùng nhớ.**

NSLog(@"%i",[str retainCount]);

}

* Sử dụng autorelease

int main (…)

{

StringClass \*newString;

newString = [[StringClass alloc] init];

[newString autorelease]; **//Đánh dấu đối tượng sẽ được release sau này nên hiện tại reference couting chưa giảm.**

…

[newString setNewString:chuoi]; // Object vẫn dùng được sau khi autorelease

}

* Sử dụng phương thức deadlloc

//Auto calls

-(void) dealloc

{

NSLog(@"StringClass release");

[str release]; **//release instance variable: str lần cuối.**

[super dealloc]; **//gọi phương thức release của lớp parent trường hợp này là lớp NSObject.**

}

**CHAPTER 8**

* Inheritance

Theory

Coding

**CHAPTER 15**

Theroy

- Remove Object (Page 370)

* You can get more sophisticated with your approach to equal objects by using the removeObject: method and then writing your own isEqual: method for testing whether two objects are equal. If you use removeObject:, the system automatically invokes the isEqual: method for each element in the array, giving it the two elements to compare. In this case, because your address book contains AddressCard objects as its elements, you would have to add an isEqual: method to that class (you would be overrid- ing the method that the class inherits from NSObject).The method could then decide for itself how to determine equality. It would make sense to compare the two corresponding names and emails. If both were equal, you could return YES from the method; otherwise, you could return NO.Your method might look like this:

Coding

-(BOOL) isEqual (AddressCard \*) theCard

{

if ([name isEqualToString: theCard.name] == YES && [email isEqualToString: theCard.email] == YES)

return YES;

else

return NO;

}

***EXTENAL CHAPTER***

* *Property*

*Theory*

* ***nonatomic,*** which tells the compiler not to worry about multithreading, ***and retain,*** which tells the compiler to retain the *passed-in variable* (???) before setting the instance variable.
* In other situations, you might want to use the ***“assign”*** property attribute instead of retain, which tells the compiler NOT to retain the passed-in variable. Or perhaps the “copy” property attribute, which makes a ***copy*** of the passed-in variable before setting.
* You start with the ***@synthesize*** keyword, then give the name of the property, then (if it has a different name) tell it the instance variable that should back the property.
* You use the ***@dynamic*** statement to tell the compiler to suppress a warning if it can’t find an implementation of accessor methods specified by an @property declaration.

Coding

- Interface

@property (nonatomic, retain) [NSArray](http://developer.apple.com/documentation/Cocoa/Reference/Foundation/Classes/NSArray_Class/) \* sushiTypes;

@property (nonatomic, retain) [NSString](http://developer.apple.com/documentation/Cocoa/Reference/Foundation/Classes/NSString_Class/) \* lastSushiSelected;

- Implementation

@synthesize sushiTypes = \_sushiTypes;

@synthesize lastSushiSelected = \_lastSushiSelect;

- Dynamic

|  |
| --- |
| @implementation MyClass |
| @dynamic title; |
| @end |

Link:

<http://www.raywenderlich.com/2712/using-properties-in-objective-c-tutorial>

Protocal

[http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Chapters/ocProtocols.html#//apple\_ref/doc/uid/TP30001163-CH15-SW1](http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjectiveC/Chapters/ocProtocols.html)

Tip: I’ll list the rules out first, and then we’ll have a discussion of each one.

1. Always make a property for every instance variable.
2. If it’s a class, mark it with the retain attribute. Otherwise, mark it with the assign attribute.
3. Whenever creating a variable, use the alloc/init/autorelease idiom.
4. Whenever setting a variable, always use “self.xxx = yyy” (in other words, use the property).
5. For each of your instance variables, call “self.xxx = nil” in dealloc. If it’s an outlet or something you created in viewDidLoad, do the same in viewDidUnload.

***Explain:***

1. For rule #1: By making an instance variable for each property, you can let the compiler write the memory management code for you. The drawback is it doesn’t keep the private data of your class well encapsulated so can lead to more connected code if you are not careful.
2. For rule #2: By retaining variables whenever they are set, you can make sure that you can access your instance variables at any time.
3. For rule #3: When you create a variable, you want to use the alloc/init/autorelease idiom (like you can see creating the NSArray with the sushiTypes earlier). This way, the memory will be freed up for you automatically later on. If you need to keep the variable around long-term, you should assign it to a property, put it in an array, or the like to increment the reference count.
4. For rule #4: By using the self.xxx syntax (i.e. using the property) whenever you set a variable, you’ll go through the properties and hence make sure to release the old variable/retain the new variable. Note some programmers worry about using getters/setters/properties in initializers and dealloc, but I don’t think it’s something to worry about if you wrote & understand the code.
5. For rule #5: Calling “self.xxx = nil” will go through your properties and decrement the reference count. Don’t forget about viewDidUnload!

* *Protocal*

* Categories and Extensions

<https://developer.apple.com/library/mac/#documentation/General/Conceptual/DevPedia-CocoaCore/Category.html>

* ***Document Online***

<http://www.otierney.net/objective-c.html#multiple>