Objective-C

Presented By
Abdelrahman Sayed

Lecture Three (final)

Agenda

- Knowing The Class.
- Categories.
- Protocols.
- Memory Management.
- Foundation Framework.

[Knowing The Class]

Asking Questions About Classes

- As you start working with variables that can contain objects from different classes, you might need to ask questions such as the following:
  - Is this object a rectangle?
  - Does this object support a print method?
  - Is this object a member of the Graphics class or one of its descendants?

Asking Questions About Classes Cont.

Method	Question or Action
`-(BOOL) isKindOfClass: class-object`	Is the object a member of *class-object* or a descendant?
`-(BOOL) isMemberOfClass: class-object`	Is the object a member of *class-object*?
`-(BOOL) respondsToSelector: selector`	Can the object respond to the method specified by *selector*?
`+(BOOL) instancesRespondToSelector: selector`	Can instances of the specified class respond to *selector*?
`+(BOOL)isSubclassOfClass: class-object`	Is the object a subclass of the specified class?

Example

```
[myObj isMemberOfClass:[MyClass class]];
```

```
if([mySquare isKindOfClass : [Square class]] == YES)
```

```
if([Square isSubclassOfClass : [Rectangle class]] == YES)
```

```
if([mySquare  respondsToSelector:@selector(alloc)] == YES)
```

[Categories]

Categories

- Sometimes, we need to extend the behavior of existing classes.
- In such situations, it doesn't make sense to add this behavior to the class interface (.h file).
- It might not make sense to subclass the existing class, because you may want your extended behavior available not only to the original class but also any subclasses of that class.

Categories Definition

- Extending your class's behavior without having subclasses or adding the behavior to the original class
- Adding methods to an existing class, to extend the behavior, is done through using a Category

Category Interface

```
@interface ClassName   (CategoryName)
    // additional methods go here
@end
```

- Example:

```
@interface ClassA   (classAMultiply)
-(int) mul :(int) x :(int) y;
@end
```

Category Implementation

```
#import "ClassName+CategoryName.h"

@implementation ClassName(CategoryName)

    // implementation of additional methods goes here

@end
```

- Example

```
#import "ClassA+ClassAMultiply.h"

@implementation ClassA(ClassAMultiply)

-(int) mul :(int) x :(int) y{

    // your implementation goes here
}

@end
```

Category Properties

- Adding methods to a class while using the same class name.
- Adding methods to classes without having their source code.
- Splitting the implementation of a complex class across multiple source code files.
- Once you've declared a category and implemented the methods, you can use those methods from any instance of the class

Category Properties Cont.

- Those methods could be used with any subclass of this class.
- No (variables – properties) are to be added, only the existing ones are to be used.
- You do NOT have to implement all the categories.
- It is useful for ongoing programming.

Category Method Name Clashes

- Be very careful about method names.
- If the name of a method declared in a category is the same as a method in:
  - The original class
  - Another category on the same class
  - Another category on the superclass

The behavior is undefined as to which method implementation is used at runtime.

[Category Demo]

[Protocols]

What Is A Protocol ?

- A protocol is a list of methods that is shared among classes.
- Methods listed in the protocol do not have corresponding implementations; they're meant to be implemented by someone else (like you!).
- A protocol list a set of methods, some of which you can optionally implement, and others that you are required to implement.
- If you decide to implement all of the required methods for a particular protocol, you are said to conform to or adopt this protocol.
- You are allowed to define a protocol where all methods are optional, one where all are required, or both

Protocols Properties

- Defining a protocol in a class means adding one or more unimplemented methods.
- The group of unimplemented methods is called a protocol.
- Each class may (conform – adopt) more than a protocol.

Defining A Protocol

In the interface section

```
@protocol ProtocolName

@required
   // define methods that should be implemented with the protocol

@optional
   // define methods that may be implemented or not

@end
```

Note: Methods that are defined without adding (required, optional) are required by default

Conform / Adopt

- If a class implements the unimplemented methods of a protocol, this is called adopting or conforming a protocol.

- To conform a protocol in a class

```
@interface MyClass : Parent <ProtocolName>
```

```
@interface MyClass : Parent <ProtocolName, P2>
```

Extending A Protocol

- You can define a protocol that conforms another protocol.

```
@protocol ProtocolName <OldProtocol>
```

- In this case any class that conforms to ProtocolName protocol has to implement also all required methods in the oldProtocol.

[Protocol Demo]

[Memory Management]

Memory Management

- There are three basic memory management models that are supported for Objective-C developers:
  1. Automatic garbage collection
  2. Manual reference counting and the auto release pool
  3. Automatic Reference Counting (ARC)

Automatic Garbage Collection

- With garbage collection, the system automatically determines which objects are no longer referenced, Then the garbage collector automatically frees them during the program's execution.

- The iOS runtime environment doesn't support garbage collection, so you don't have the option to use it when developing programs for that platform.

- You can only use it when developing Mac OS X applications.

Automatic Garbage Collection contd

- Beginning May 1, 2015, new Mac apps and app updates submitted to the Mac App Store may no longer use garbage collection, which was deprecated in OS X Mountain Lion.

Manual Reference Counting

- If you are going to create applications without the use of either ARC or garbage collection.
- If you have to support code that you can’t migrate to run with ARC.
- Then you need to know how to manage memory using manual reference counts.

Reference Counting

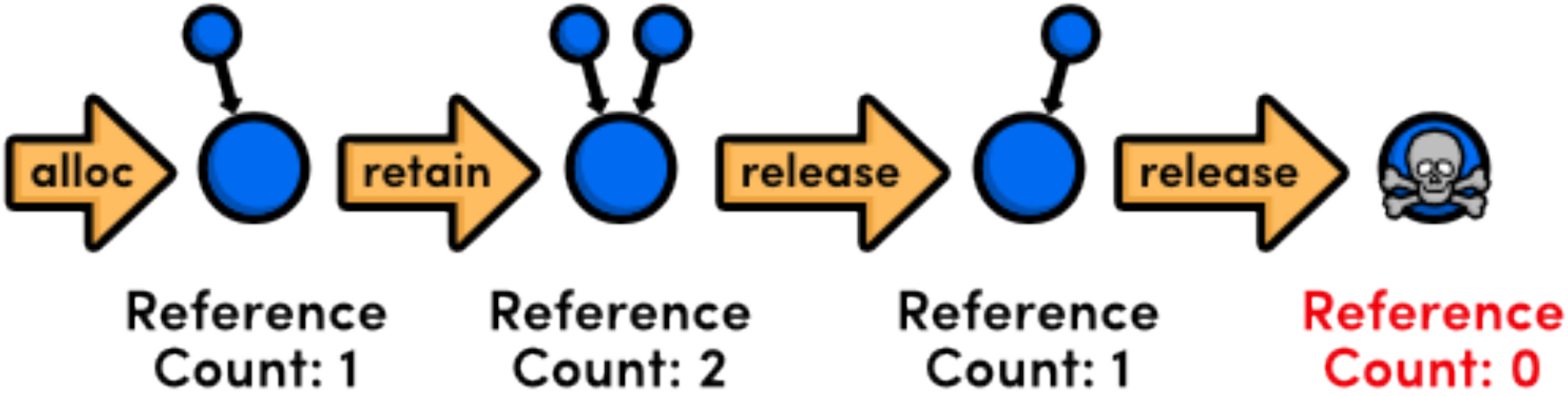
- Each object has a reference count.

```
JETSHelloWorld *obj = [JETSHelloWorld new]; // Ref count became 1
```

- Reference count increments and decrements
  - Increment using retain method
  - Decrement using release method

- When reference count is zero for an object, it Is automatically *de allocated*.

Reference Counting Cont.



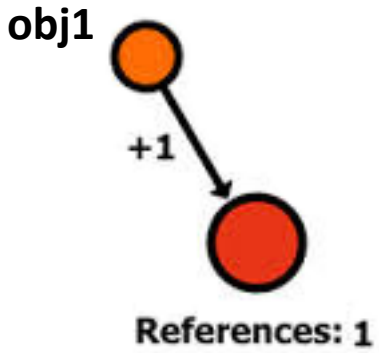
Reference Counting Cont.

- Some referencing operations increments the reference count, others don't.
  - Adding an object to a collection (Array, Dictionary), automatically increments its reference count.
  - Direct assignment does not affect reference count (problem).

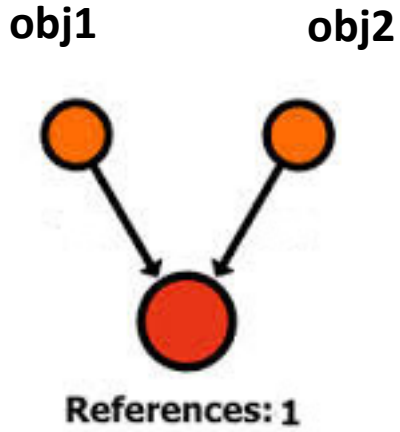
  You can fix the above problem by manually calling retain after a direct assignment operation.

Example

```
JETSClassA *obj1 = [JETSClassA new];
```

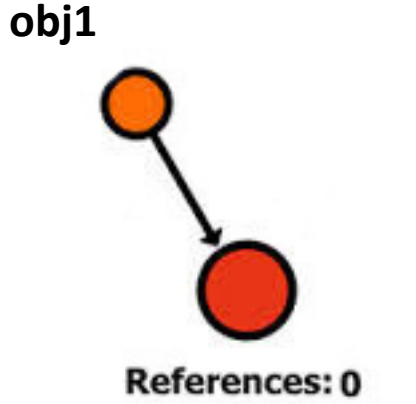


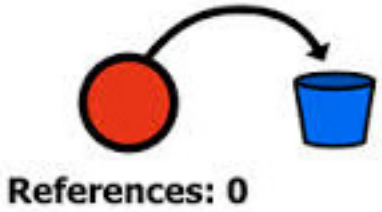

```
JETSClassA *obj2 = obj1;
```



Example contd.

```
[obj2 release];
```





Memory Allocation

- The following methods does not increase the reference count of an existing object but they create a new object
  - alloc
  - copy
  - new

Memory Releasing

- release
  - Decrements the reference counter.
- dealloc
  - We override it but NEVER call it, it is called automatically.
- autorelease
  - Marks the object as autoreleased.

Autorelease

- Autorelease means send release message later.
- Release message is sent when autorelease pool is drained.
- Autorelease pool is released when the current code finishes execution.
- When an object's reference count drops to zero, the runtime calls the dealloc method of the object's class just before it destroys the object.

Example

```
ClassA *myObj = [[[ClassA alloc] init] autorelease];
```

- or like this:

```
return [myObj autorelease];
```

[Foundation Framework]

Foundation Framework

- The Foundation framework includes:
  - The root object class
  - Classes representing basic data types such as strings and byte arrays
  - Collection classes for storing other objects
  - Classes representing system information such as date

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

Mutable vs. Immutable

- Mutable: Any mutable variable is changeable.
- Immutable: Immutable variables are unchangeable.

NSNumber

- Object wrapper around primitive types like int, float, double, etc.

- Methods:

```
-(double) doubleValue

-(int) intValue

-(BOOL) isEqualToNumber :(NSNumber*) aNumber
```

NSString

- Used throughout iOS instead of C language's char * type.
- Compiler will create an NSString for you using @"foo" notation.
- An NSString instance can not be modified! They are immutable.

NSString

- Methods:

```
-(id) initWithString :(NSString*) aString

-(NSString*) stringByAppendingString :(NSString*) aString

-(NSString*) lowercaseString

-(NSString*) uppercaseString
```

NSString Cont.

- **Creating and Initializing Strings:**
  - **- initWithString :**
  - **- initWithFormat :**
  - **+ stringWithFormat :**
  - **+ stringWithString :**
- **Getting a String's Length:**
  - length (Property)

NSString Cont.

- **Getting Characters and Bytes:**
  - - characterAtIndex :
  - - getCharacters : range :
- **Dividing Strings:**
  - - componentsSeparatedByString :
  - - substringFromIndex :
  - - substringWithRange :
  - - substringToIndex :

NSString Cont

- **Replacing Substrings:**
  - - stringByReplacingOccurrencesOfString : withString :
  - - stringByReplacingOccurrencesOfString :

    withString : options : range :
  - - stringByReplacingCharactersInRange : withString :

NSString Cont

- **Identifying and Comparing Strings:**
  - - caseInsensitiveCompare :
  - - compare :
  - - compare : options : range :
  - - hasPrefix :
  - - hasSuffix :
  - - isEqualToString :

NSMutableString

- Mutable version of NSString.
- Can do some of the things NSString can do without creating a new one (i.e. in-place changes).
- Methods

```
-(void) appendingString :(NSString*) aString

-(void) deleteCharactersInRange :(NSRange) aRange
```

NSArray

- Ordered collection of objects
- Immutable
- Methods:

```
-(int) count
-(id) objectAtIndex :(int) index
-(id) lastObject
-(BOOL) containsObject :(id) anObject
```

NSMutableArray

- Mutable version of NSArray
- Methods:

```
-(void) addObject :(id) anObject

-(void) insertObject :(id) anObject atIndex :(int) index

-(void) removeObjectAtIndex :(int) index

-(void) removelastObject
```

NSDictionary

- Look up objects using a key to get a value.
- Immutable hash table.
- Methods:

```
-(int) count

-(id) objectForKey :(id) key

-(NSArray *) allKeys

-(NSArray *) allValues
```

NSMutableDictionary

- Mutable version of NSDictionary

- Methods:

```
-(void) setObject :(id) anObject forKey :(id) key

-(void) removeObjectForKey :(id) key

-(void) removeAllObjects

-(void) addEntriesFromDictionary :(NSDictionary*) otherDictionary
```

[Lab Exercise]

1.Knwoing The Class

- Create ClassA , ClassB , MyClass.
- ClassB is subclass of ClassA.
- ClassA has method methodA.
- ClassB has method methodB.
- MyClass has method myMethod.
- Create objA, objB and myObj and Check the following:
  - objA, objB, myObj is kind / is member of ClassA, ClassB, MyClass
  - obA, obB, myOb will respond to methodA, methodB and myMethod

2.Calculator

- Update your calculator program to Add squaring feature using (Category)

3.Protocol

- Create MyProtocol protocol which has two methods
  - calcArea (required)
  - printShapeName (optional)
- Create class Rectangle and class Triangle which conform to MyProtocol protocol and implement your method.

4.IP Cutter

- Create an application that accepts a **well formed** IP address in the form of a string and cuts it into separate parts based on the dot delimiter (3 ways).

- *The user enter IP like this: 163.121.12.30*

- The output should be:

  163

  121

  12

  30

Thanks