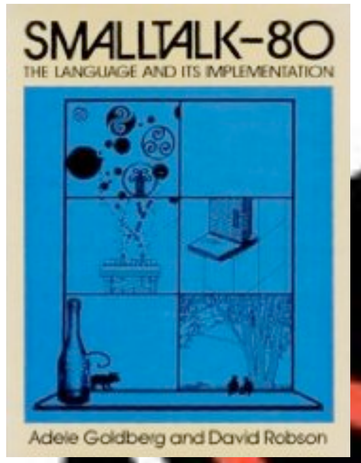
MODERN OBJECTIVE-C

MODERN OBJECTIVE-C

SUBTITLE

- Programming language
- A library of objects
- Runtime environment
  - Defers many operations from compile time to runtime
  - Executes the compiled code

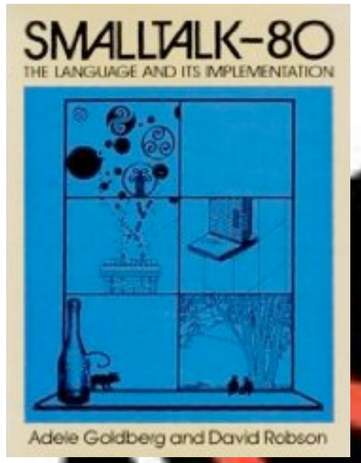


Simpler, safer though automation

MODERN OBJECTIVE-C

SUBTITLE

- Popularity by TIOBE Programming Community Index
  - 2007 - 45th
  - 2011 - 6th
  - 2012-16 - 4th



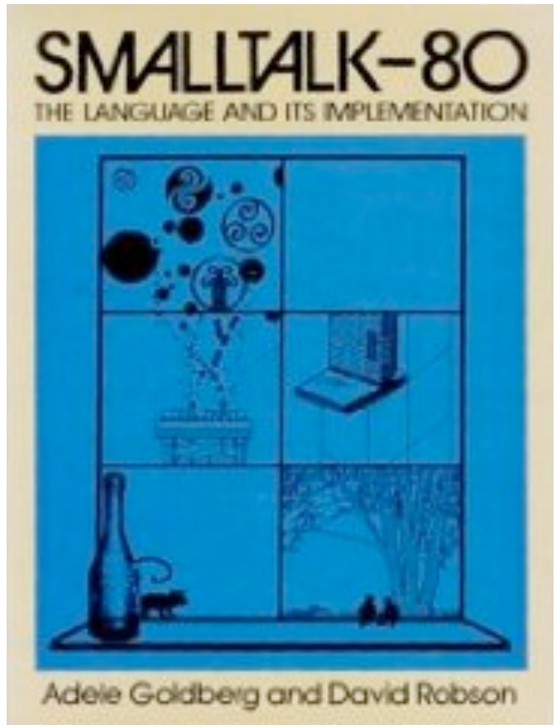
Simpler, safer though automation

MODERN OBJECTIVE-C

- Based on Smalltalk language
  - Messaging an object instead of ‘calling’ a function
- In "messaging"
  - The runtime decides which code gets executed
  - Performs a ‘lookup’ to identify method by name
  - Compiler does not check type of object being messaged
    - Dynamic binding at runtime

Looks up method to run

Code can compile without warnings/ errors and crash.



MODERN OBJECTIVE-C

- A strict-superset of ANSI C
  - Can use C code interspersed with objective-C code (in .m file)
    - Include header files using `#import`
  - Structs are used in many frameworks (for efficiency)
    - There is overhead to creating objects
    - Structs can only hold non object types (int, float, double, char, etc.)

```
struct CGRect {
  CGPoint origin;
  CGSize size;
};
typedef struct CGRect CGRect;
```

```
CGRect frame;
frame.origin.x = 0.0f;
frame.origin.y = 10.0f;
frame.size.width = 100.0f;
frame.size.height = 150.0f;
```

MODERN OBJECTIVE-C

OBJECTIVE-C ADD THE FOLLOWING SYNTAX AND FEATURES TO ANSI C

- Definition of new classes
  - Class and instance methods
  - Method invocation (called messaging)
  - Declaration of properties (synthesizing of accessor methods)
- Static and dynamic typing
- Blocks—encapsulated segments of code
- Extensions to the base language such as protocols and categories

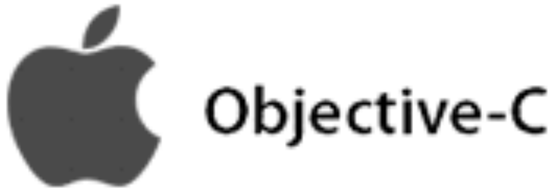
MODERN OBJECTIVE-C

- NEXT was the most significant adopter of Objective-C
- Next added
    - Full implementation of object-oriented C
    - Retain and Release memory management model
    - Interface library and project builder

MODERN OBJECTIVE-C

OBJECTIVE-C 2.0 (2006)

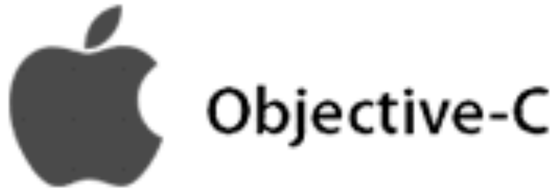
- Apple introduced Obj-C 2.0 and added
  - Properties
  - Fast Enumeration
  - Blocks
  - Literal syntax
  - ARC (automated memory management)



MODERN OBJECTIVE-C

- Many of the 2.0 features are strictly compiler automation
  - The underlying language/runtime features are the same

Simpler, safer though automation



OBJECTIVE-C TYPES

OBJECTIVE-C

TYPES

```
typedef struct objc_object {
    Class isa;
} *id;
```

isa variable of what class it “isa” an instance of

- "id"
  - General type for any kind of object (e.g. instance of a class or class objects)
  - Pointer to an object data structure

OBJECTIVE-C

TYPES

```
if ( [anObject isKindOfClass:someClass] )
...

if ( [anObject isMemberOfClass:someClass] )
…
```

- “isa” variable allows objects to perform introspection
  - `isKindOfClass:` - includes subclasses
  - `isMemberOfClass:` - specific to a class (no subclasses)
  - `respondsToSelector:` - implements a selector (method); does not care what class it belongs to (polymorphism)

OBJECTIVE-C

TYPES

- `id` is the generic type for dynamic binding at runtime
    - Can be cast to a pointer (*)
    - Does not know (or care) what objects is until it needs it
    - Warning: Compiler does not check

OBJECTIVE-C

TYPES

- Use case examples for `id`
  - Identify which button is pressed in a group of buttons
  - Iterate through an NSArray of different objects
- Advantages of statically type objects (why not to use `id`)
  - Compile-time checking
  - Autocompletion in Xcode

OBJECTIVE-C

TYPES

- SEL (selector) type represents a way to refer to a method by name
- @selector() keyword return a SEL

```
// Test if a class implements a method
[object respondsToSelector:@selector(methodName)]

// Store a method and call on object
SEL method = @selector(methodName);
[object performSelector:method];

// Set method to call for target-action
[aButton addTarget:self action:@selector(eatCake)

forControlEvents:UIControlEventTouchDown];
```

OBJECTIVE-C

TYPES

```
if ([aObject respondsToSelector:@selector(myMethodName)])
{
    ...
}
```

Does aObject implement a method named ‘myMethodName’?

- `@selector` turns a method into a name

```
[button addTarget:self action:@selector(digitPressed:)]
```

Has a parameter

When button is tapped, execute method named ‘digitPressed’ that is implemented in this class (self)

OBJECTIVE-C

TYPES

- `nil` object pointer to nothing
- Example: Test if it points to an object

```
if (myObject == nil) return;
```

- Instance variables can be set to nil

```
myObject.delegate = nil;
```

OBJECTIVE-C

TYPES

SUBTITLE

- It is safe to send a message to nil
  - Invoking a method on nil returns a zero value

```
myObject = nil;
[myObject doSomething];
```

Warning!!! Would crash other languages

OBJECTIVE-C

TYPES

- Sending messages to nil simplifies expressions
  - Limits having to check for nil before doing anything
  - Use as a convenience feature of Obj-C

```
// For example, this expression...
if (name != nil && [name isEqualToString:@"Steve"]) { ... }

// ...can be simplified to:
if ([name isEqualToString:@"steve"]) { ... }
```

It was so much work to do this…

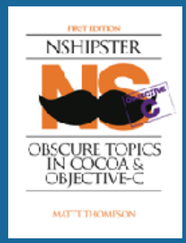
OBJECTIVE-C

TYPES

Symbol	Value	Meaning
`NULL`	`(void *)0`	literal null value for C pointers
`nil`	`(id)0`	literal null value for Objective-C objects
`Nil`	`(Class)0`	literal null value for Objective-C classes
`NSNull`	`[NSNull null]`	singleton object used to represent null

- Values representing nothing that every Objective-C programmer should know about

Collections can not hold nil, but can hold NSNull object



OBJECTIVE-C

TYPES

http://nshipster.com/

nil / Nil / NULL / NSNull

Written by Mattt Thompson on Jan 7th, 2013

Understanding the concept of nothingness is as much a philosophical issue as it is a pragmatic one. We are inhabitants of a universe of *somethings*, yet reason in a logical universe of existential uncertainties. As a physical manifestation of a logical system, computers are faced with the intractable problem of how to represent *nothing* with *something*.

In Objective-C, there are several different varieties of *nothing*. The reason for this goes back to a common NSHipster refrain, of how Objective-C bridges the procedural paradigm of C with Smalltalk-inspired object-oriented paradigm.

C represents *nothing* as `0` for primitive values, and `NULL` for pointers (which is equivalent to `0` in a

OBJECTIVE-C

TYPES

- `BOOL` types to encode ‘truth’
  - typedef of a signed char, with the macros YES and NO
  - YES = #define 1
  - NO = #define 0

```
// Test for true
if (myObject.value == YES) {...}
if (myObject.value) {...}

// Test for false
if (myObject.value == NO) {...}
if (!myObject.value) {...}

// Set BOOL type variable
myObject.updated = NO;
myObject.updated = YES;
```

OBJECTIVE-C

TYPES

- Truth types and values in Objective-C

Name	Typedef	Header	True Value	False Value
BOOL	signed char	objc.h	YES	NO
bool	_Bool(int)	stdbool.h	true	false
Boolean	unsigned char	MacTypes.h	TRUE	FALSE
NSNumber	__NSCFBoolean	Foundation.h	@(YES)	@(NO)
CFBooleanRef	struct	CoreFoundation.h	kCFBooleanTrue	k

OBJECTIVE-C

OBJECT MESSAGING

OBJECTIVE-C

OBJECT MESSAGING

- Send a message (method) to an object (receiver)

```
[receiver message];

[Car start];

[[[Car alloc] init] anotherMessage];
```

- When a message is sent, the runtime system selects method from the receiver
  - Method names in message are “selectors”

OBJECTIVE-C

OBJECT MESSAGING

MESSAGE EXPRESSION

MESSAGE

```
[receiver method:argument];
```

SELECTOR

OBJECTIVE-C

OBJECT MESSAGING

- Method parameters (arguments)
  - Not optional

```
[myObject setName:@”Jane”];
```

  - Order must be preserved

```
[myObject setName:@”Jane” andDate:[NSDate now] andEmail:jd@mail.com];
[myObject setName:@”Jane” andEmail:jd@mail.com andDate:[NSDate now]];
```

```
setName:andEmail:andDate
```

OBJECTIVE-C

OBJECT MESSAGING

- Polymorphism
  - Objects can respond differently to the same message

```
[personObject description];
```

  - Example: send a `description` message to an `id` variable

```
[carObject description];
```

NSObject provides ‘description’ to convert object to string. It varies depending on object. Good method to override in custom classes.

OBJECTIVE-C

OBJECT MESSAGING

```
myInstance.value = 10;
[myInstance setValue:10];


self.age = 10      // Access object's ivar
                   // using accessor methods
[self setAge:10]; // Equivalent to above
age = 10;          // Access ivar directly
```

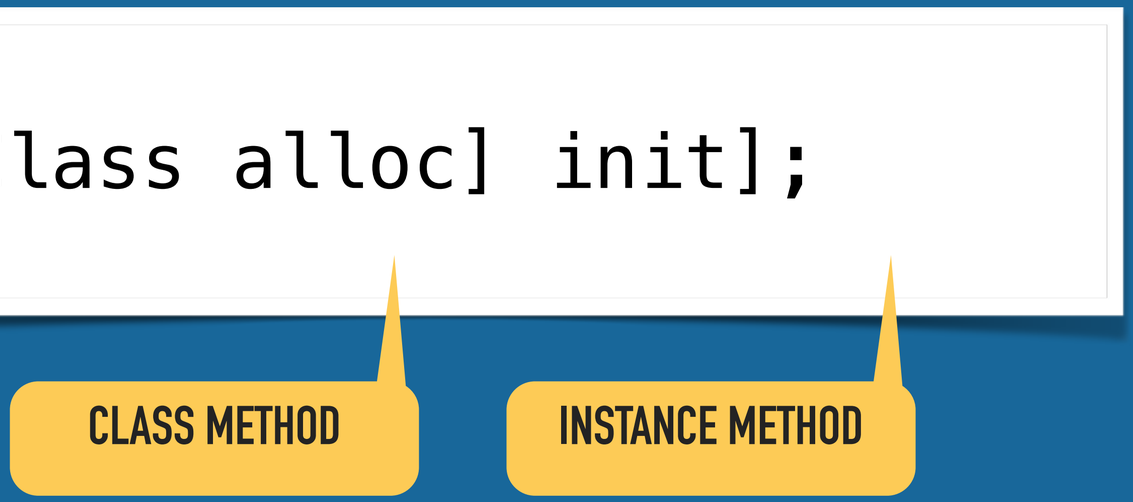
- Dot Syntax
  - Alternative to square bracket notation "[ ]"
  - Compiler transforms to accessor methods

OBJECTIVE-C

OBJECT MESSAGING

- Method types:
  - Class methods “+” directly to class
  - Instance methods “-” scoped to instance of a class

```
[[myClass alloc] init];
```



OBJECTIVE-C

OBJECT MESSAGING

- Method declaration

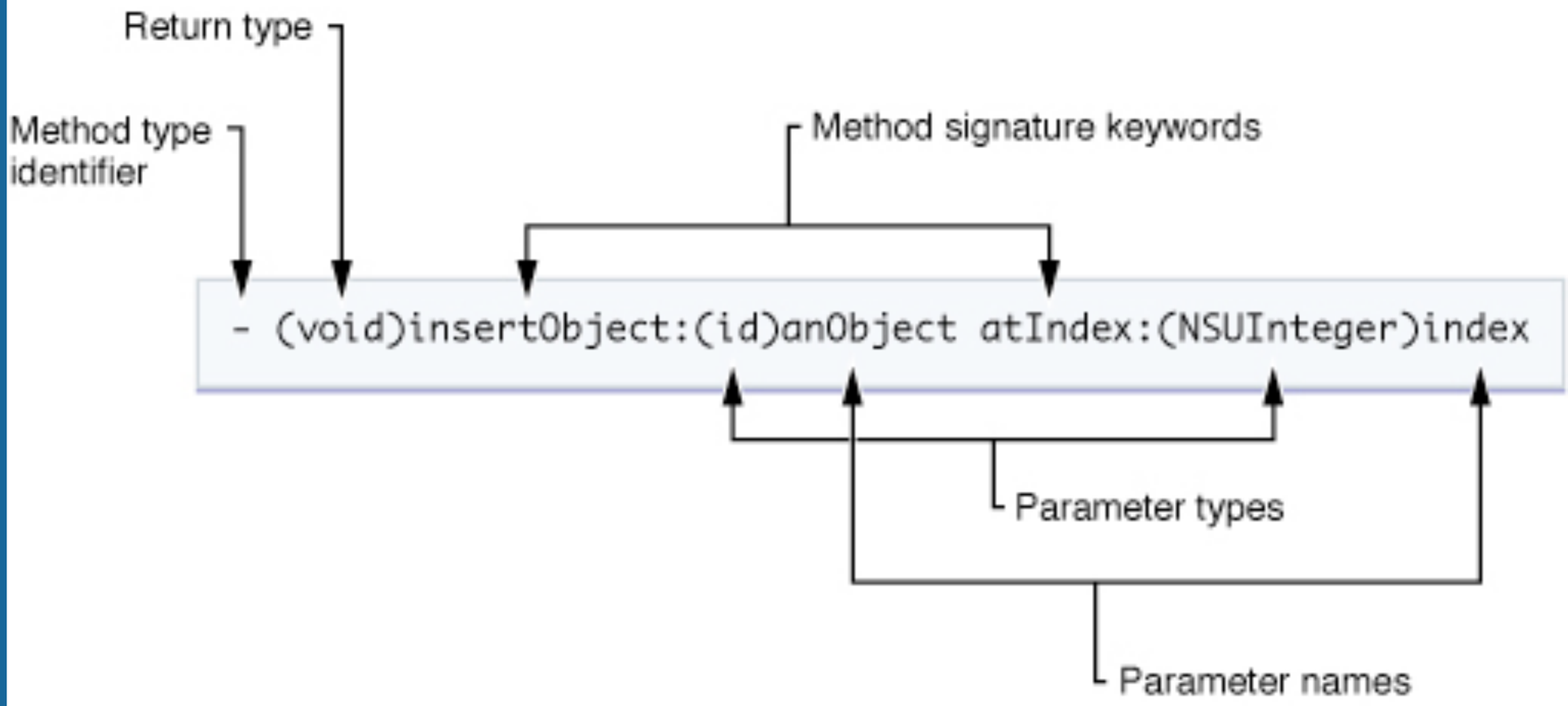


image via App

OBJECTIVE-C

OBJECT MESSAGING

```
// Allocate memory for the object (always pair w/init)
+ (id)alloc;

// returns the one and only, shared (singleton) instance
+ (id)sharedObject;
```

- Class methods
  - Used for allocation, singletons, utilities
  - No access to local variables

OBJECTIVE-C

OBJECT MESSAGING

```
// Setter method
-(void)setString:(NSString *)newString
{
...
}
```

- Instance methods
  - Most common methods to implement
  - Access variables as if they were local
  - Can call methods on `self` and `super`

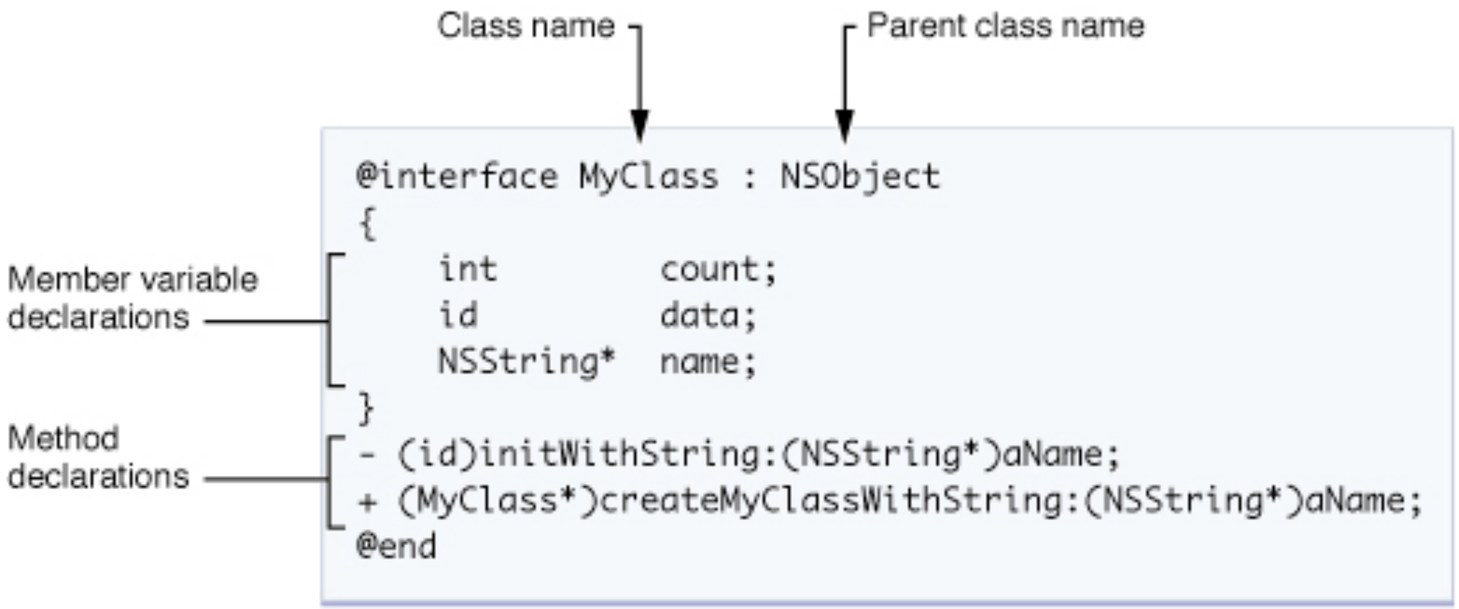
OBJECTIVE-C CLASSES

OBJECTIVE-C

CLASSES

- Classes declare state and behavior
  - State is maintained using instance variables
  - Behavior is implemented using methods

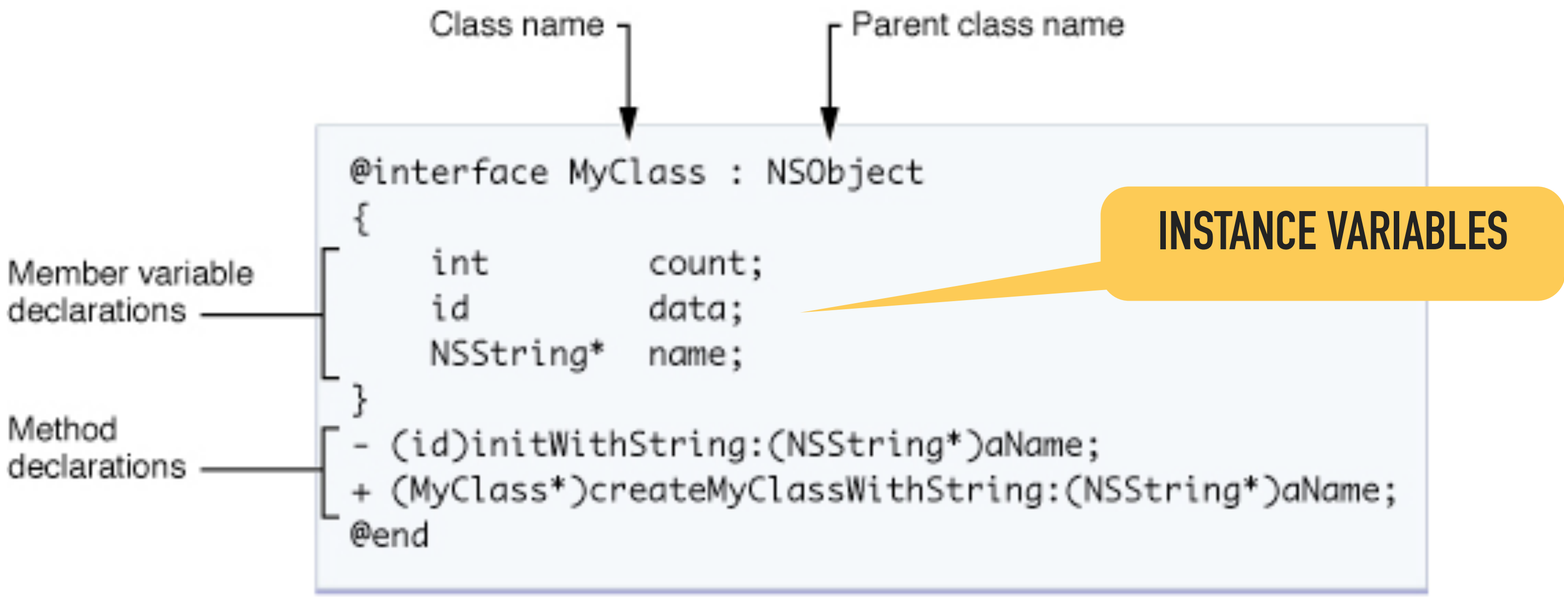
Interface files `.h`



OBJECTIVE-C

CLASSES

- Instance variables should only be accessed using getter/setter methods
  - Prefer use of @properties



OBJECTIVE-C

CLASSES

Subclass of UIViewController

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

@interface MyViewController : UIViewController {

}
@end
```

- Inheritance
  - Class definitions are additive
  - Methods and instance variables
  - Single inheritance
- Overwrite existing methods
  - Custom ‘description’ method

OBJECTIVE-C

CLASSES

- Every Foundation framework class inherits from `NSObject`
  - Memory management
  - Runtime cooperation

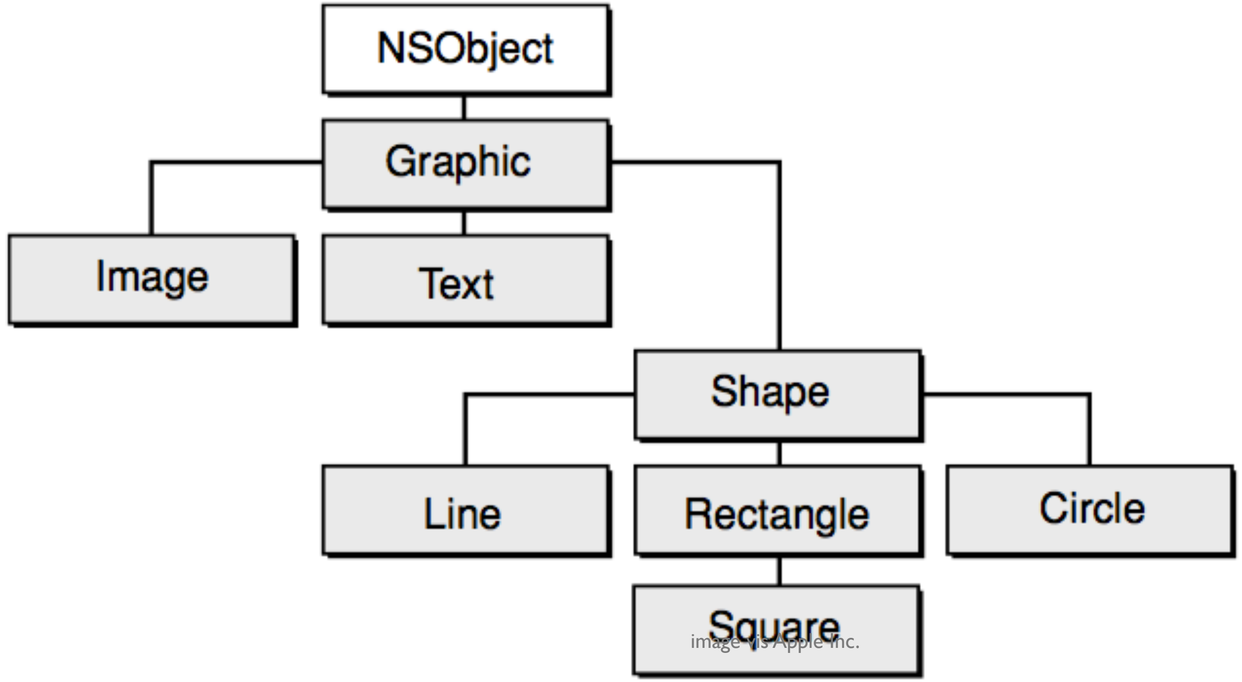


image © Apple Inc.

OBJECTIVE-C

CLASSES

- Creating a class instance by sending `alloc`
  - Dynamically allocates memory

```
//  Alloc and init
Class *myClass = [[myClass alloc] init];

//  Init with parameters
Class* myClass = [[myClass alloc] initWithInt:1];
```

- Initialize a class instance's instance variables by sending `init`
  - Can customize or override `init` method
  - Must call [super init]

OBJECTIVE-C

CLASSES

- All NSObjects implement the method -description
  - [NSString stringWithFormat:@"This is my class:%@", anObject];
- Called in format string using %@
  - [NSString stringWithFormat:@"This is my class:%@", anObject];
- Called in format string using %@
  - NSLog([anObject description]);
- Common to overwrite it for custom classes
-

```
-(NSString *)description
{
  return [NSString stringWithFormat:@"<GameObject: %@, Position: %f, %f>",
                  [self objectID], [self position].x, [self position].y];
}
```

OBJECTIVE-C

CLASSES

- Convenience methods to create a class
  - Automatically alloc, init and autorelease (pre-ARC) an object
  - Pro: Less code, will not leak (pre-ARC)
  - Cons: Less control

```
//  Convenience method
NSString* myString = [NSString string];

//  Manual
NSString* myString = [[NSString alloc] init];
```

OBJECTIVE-C

CLASSES

- Destroying a class by sending message `release`

```
//  Pair alloc-init with a release
Class *myClass = [[myClass alloc] init];
...
[myClass release];

//  Best practices
[myClass release], myClass = nil;
```

PRE-ARC

- Setting the variable to `nil` prevents bad access later
  - Sending a message to `nil does nothing

OBJECTIVE-C

CLASSES

- Class equalities of pointer values (e.g. class instances are identical)

```
if ([objectA class] == [objectB class]) { //...
```

- Class instance values are equivalent

```
if ([object1 isEqual: object2]) { //...
```

OBJECTIVE-C MEMORY MANAGEMENT

FOR HISTORICAL PURPOSES

OBJECTIVE-C

MEMORY MANAGEMENT

- Reference counting - keep track of all references to an object
  - Typical pattern: alloc, retain, release

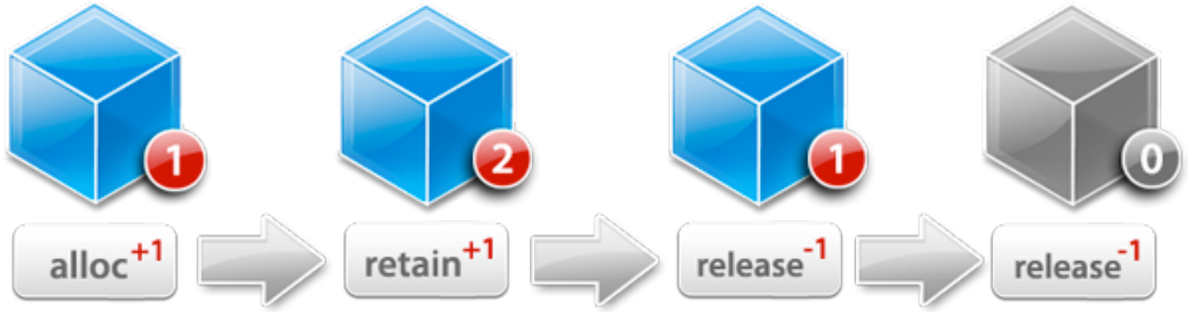
image via http://cocoadevcenttral.com

- If you create an object with `alloc` or `copy` you have to send it `release` or `autorelease`
  - Calls must be balanced or will leak memory or crash

OBJECTIVE-C

MEMORY MANAGEMENT

- Reference counting part of NSObject
  - Objects exist in memory if count > 0
- Object lifecycle
  - `+alloc` and `-copy` create objects with count = 1
  - `-retain` increments retain count
  - `-release` decrements retain count
- When retain count == 0
  - Object is destroyed and memory release
  - `-dealloc` method invoked automatically; do not call



OBJECTIVE-C

MEMORY MANAGEMENT

- Query the retain count of an object

```
NSString *myString = [[NSString alloc] init];
  // NSLog(@"Count:%@",[myString count]);
  // 1

  [myString retain];
  // NSLog(@"Count:%@",[myString count]);
  // 2

  [myString release];
  // NSLog(@"Count:%@",[myString count]);
  // 1

  [myString release];
  // NSLog(@"Count:%@",[myString count]);
  // 0 (Dealloced)
```

OBJECTIVE-C

MEMORY MANAGEMENT

- Query the retain count of an object

```
NSString *myString = [[NSString alloc] init];
  // NSLog(@"Count:%@",[myString count]);
  // 1

  [myString retain];
  // NSLog(@"Count:%@",[myString count]);
  // 2

  [myString release];
  // NSLog(@"Count:%@",[myString count]);
  // 1

  [myString release];
  // NSLog(@"Count:%@",[myString count]);
  // 0 (Dealloced)

  [myString release];
  // Crash
```

OBJECTIVE-C

MEMORY MANAGEMENT

```
@interface MyAppDelegate_iPhone : MyAppDelegate {
    UILabel *theLabel;
}
@property (nonatomic, retain) IBOutlet UILabel *theLabel;
```

- Balance retain and release of instance variables

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

OBJECTIVE-C

MEMORY MANAGEMENT

- Creating and initializing NSString

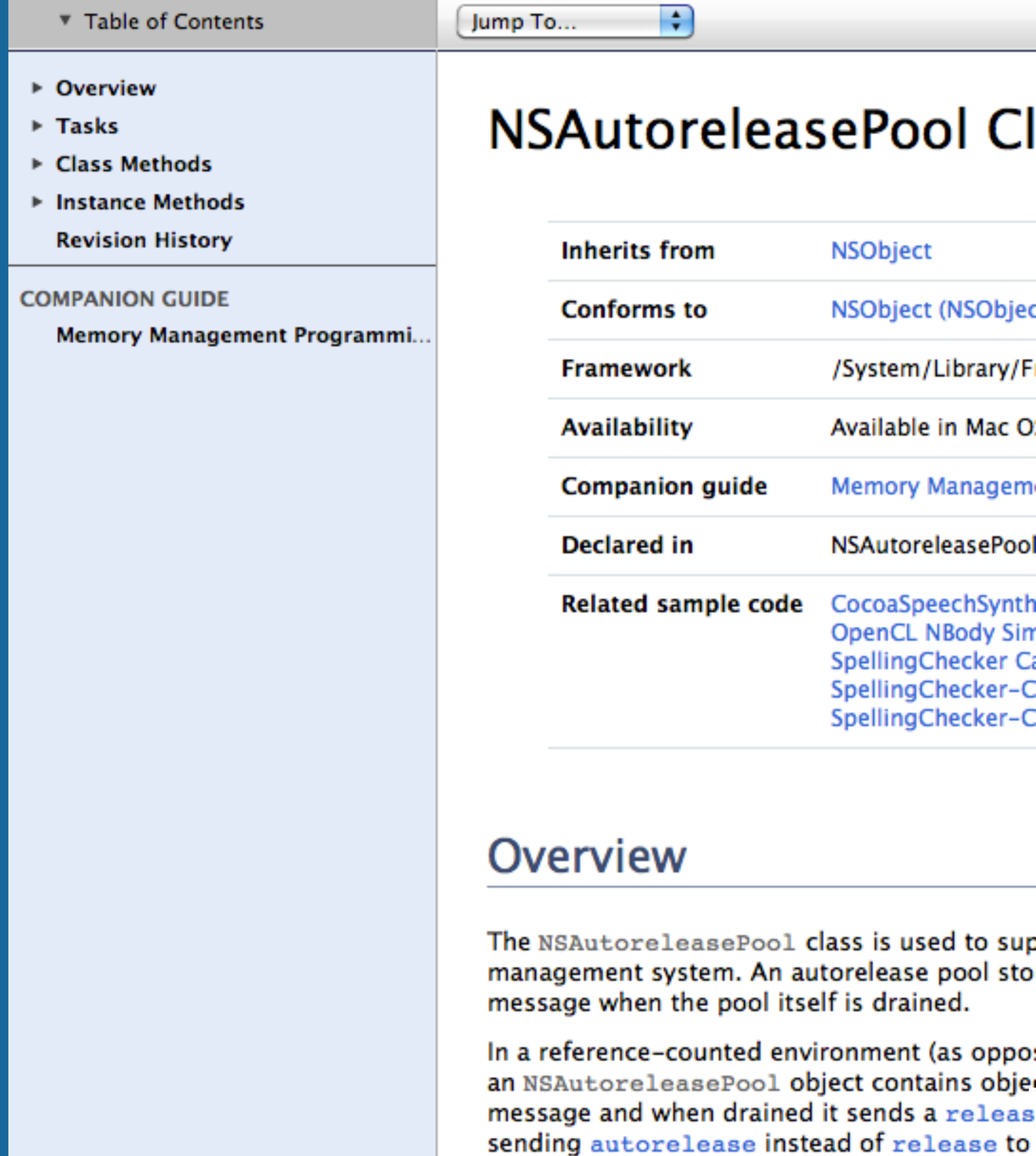
```
– initWithCharacters:length:
– initWithString:
– initWithCString:encoding:
– initWithUTF8String:
...
+ stringWithFormat:
+ localizedStringWithFormat:
+ stringWithCharacters:length:
+ stringWithString:
+ stringWithCString:encoding:
```

Convenience methods

OBJECTIVE-C

MEMORY MANAGEMENT

- Using `autorelease`
  - Objects are added to the autorelease pool
  - The pool is “drained” after each event
- Call autorelease on object you alloc-init
- Autorelease is built-in to convenience methods



OBJECTIVE-C

MEMORY MANAGEMENT

- Methods with alloc, copy or new in name need to be released or autorelease

```
[[NSString alloc] initWithString:@"hi"] autorelease];
```

- Convenience methods
  - Cover entire lifecycle of object
  - Autorelease built-in

```
[NSString stringWithString:@"Hi"];
```

OBJECTIVE-C

MEMORY MANAGEMENT

- Each application creates an NSAutoreleasePool in main.m
  - The pool sends a release message (e.g. drains) at the end of every event loop cycle
- Create local ones for use in loops, multithreading or other processes

```
int main(int argc, char *argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

OBJECTIVE-C AUTOMATIC REFERENCE COUNTING

OBJECTIVE-C

ARC

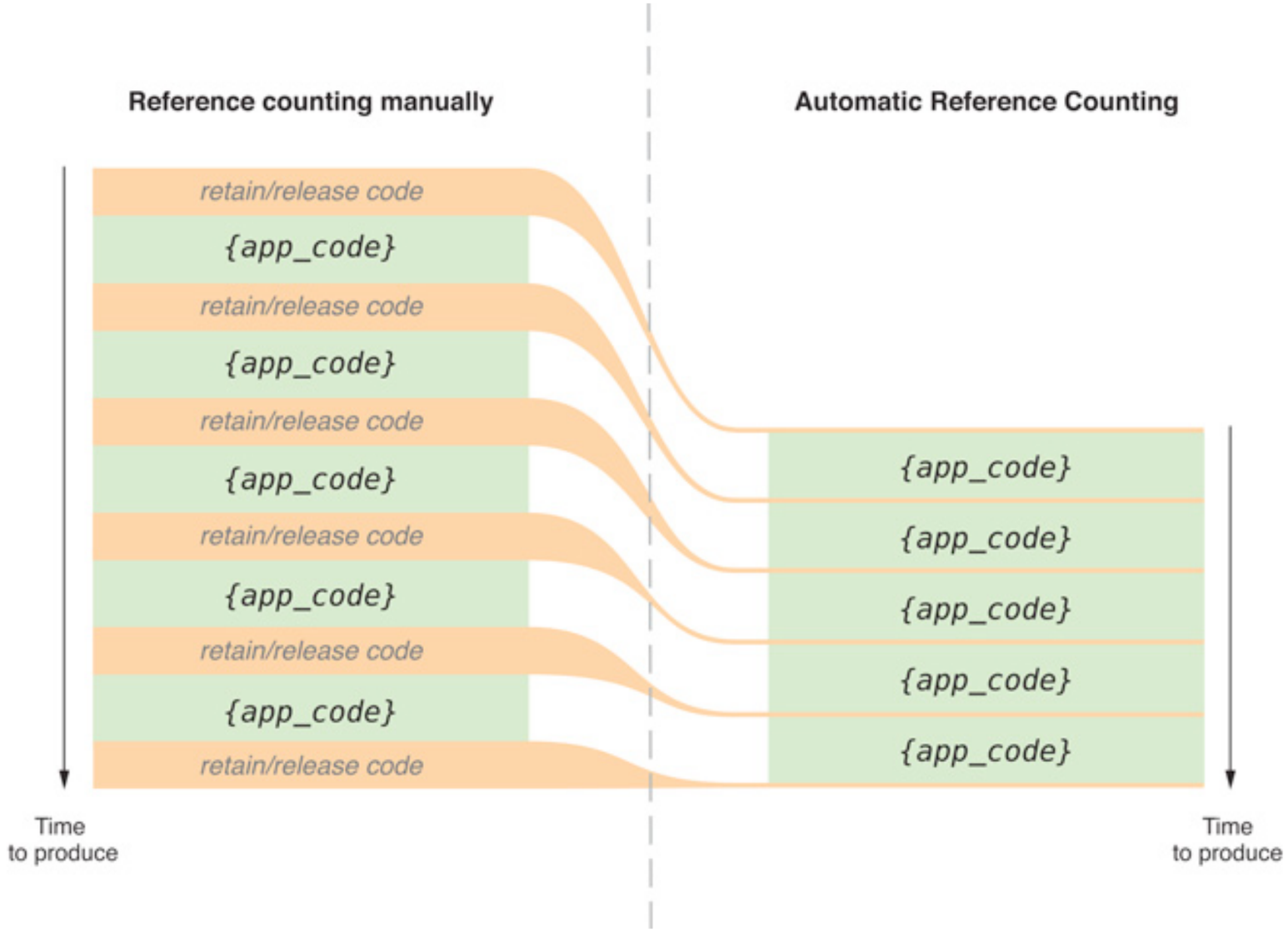
SUBTITLE

- Automatic Reference Counting
  - Introduced in iOS 5
  - Implements automatic memory management for Objective-C objects and blocks
  - Frees the programmer from the need explicitly insert retains and releases

OBJECTIVE-C

ARC

- ARC is a compiler feature that provides automatic memory management of objective-c objects



OBJECTIVE-C

ARC

- Follows the same memory management convention but “adds code at compile time”
- Evaluates the lifetime of an object
  - Inserts memory management into your compiled code
  - Generates dealloc methods
- How it works
  - Pointers keep objects in memory
  - If pointer gets a new value or is “zeroed” (nil) it is released

OBJECTIVE-C

ARC

```
@property (weak, nonatomic) IBOutlet UILabel *sliderValue;
@property (strong, nonatomic) NSString *name;
```

- Property attributes (pointer types)
  - strong
    - Keeps objects alive
  - weak
    - Point to object, but do not keep it around because another object is managing it
    - IBOutlets are weak because they are managed by parent (e.g. view controller's view)
    - Parent-child relationships (parent - strong; child - weak)

These concepts are in swift

OBJECTIVE-C

ARC

- New rules because of ARC
  - Cannot explicitly invoke retain, release, dealloc, retainCount, autorelease
  - Cannot use object pointers in C structs
  - No custom autorelease pools
  - Cannot prefix a property accessor with “new” (e.g. newLabel)

OBJECTIVE-C
DECLARED PROPERTIES

OBJECTIVE-C

DECLARED PROPERTIES

```
@property (strong, nonatomic) UIView *myView;
```

- `@property` is data encapsulated or stored by an object
  - Attributes of an object (e.g. label.text, view.backgroundColor)

OBJECTIVE-C

DECLARED PROPERTIES

- Properties simplify classes
  - Expose variable and methods through *accessor methods (e.g. getters and setters)*
  - Compiler automatically generates getters and setters
- Simplified memory management if using old method
  - Redundant if using ARC (but still common practice and seen in large frameworks)

```
@property (strong, nonatomic) UIView *myView;
```

OBJECTIVE-C

DECLARED PROPERTIES

```
@property (strong, nonatomic) UIColor *color;
```

- Class of the object defines an interface to set and get the values of the property
  - Property Name is `color`
  - Setter created named ‘setColor’
  - Getter created named ‘color’
  - Backing variable `_color`

Naming conventions

OBJECTIVE-C

DECLARED PROPERTIES

```
//
// Leaf.h
//

@interface Leaf : NSObject {
    NSString *_name; // ivar
}

// Declare accessor methods
- (void) name;
- (void) setName;

@end
```

```
//
// Leaf.m
//

@implementation Leaf

// Setter
- (void) setName:(NSString *)newName {
    if (_name != newString) {
        [_name release];
        _name = [newName copy];
    }
}

// Getter
- (NSString*)name {
    return _name;
}
@end
```

- Class creation in 2005

OBJECTIVE-C

DECLARED PROPERTIES

```
//
// Leaf.h
//

@interface Leaf : NSObject {
    NSString *_name; // ivar
}
@property(nonatomic, retain)
                    NSString *name;

// Declare accessor methods
- (void) name;
- (void) setName;

@end
```

```
//
// Leaf.m
//

@implementation Leaf
@synthesize name = _name;

// Setter
- (void) setName:(NSString *)newName {
    if (_name != newString) {
        [_name release];
        _name = [newName copy];
    }
}

// Getter
- (NSString*)name {
    return _name;
}
@end
```

SYNTHESIZE A PROPERTY “NAME”
IVAR NAME = _NAME
PROPERTY NAME = NAME
SETTER = SETNAME
GETTER = NAME

- Class creation in 2010 (Objective-C 2.0)

OBJECTIVE-C

DECLARED PROPERTIES

```
//
// Leaf.h
//

@interface Leaf : NSObject {
    NSString *_name; // ivar
}
@property(nonatomic,retain)
                  NSString *name;




@end
```

```
//
// Leaf.m
//

@implementation Leaf
@synthesize name = _name;
@end
```

- Class creation in 2010 (Objective-C 2.0)

OBJECTIVE-C

DECLARED PROPERTIES

```
//
// Leaf.h
//

@interface Leaf : NSObject {
    NSString *_name; // ivar
}
@property(strong)NSString *name;

@end
```

```
//
// Leaf.m
//

@implementation Leaf
@synthesize name = _name;
@end
```

- Class creation in 2012

OBJECTIVE-C

DECLARED PROPERTIES

```
//
// Leaf.h
//

@interface Leaf : NSObject
@property(strong)NSString *name;
@end
```

```
//
// Leaf.m
//

@implementation Leaf
@end
```

```
// @synthesize name = _name
// done by default
```

OBJECTIVE-C

DECLARED PROPERTIES

```
@property (nonatomic, weak) IBOutlet UILabel *theLabel;
```

- Atomicity
  - Multi-threading option
  - ALWAYS set to (`nonatomic`) unless you mean it; atomic is default
- Writeabilty
  - `readwrite` - default behavior; creates getters and setters
  - `readonly` - creates getter only
- Memory management:
  - Applies to setters only
  - Options: `strong`,`weak`, `copy`

OBJECTIVE-C

DECLARED PROPERTIES

```
// Generate getters/setters and retain the incoming object
@property (nonatomic, strong) NSString *name;

// Same
@property (nonatomic, strong) NSString *title;

// Generate getters/setters and assigns the incoming value
@property (nonatomic) int age;

// Generate getters/setters and assigns the incoming value
@property int age;

// Generate the getters only
@property (nonatomic, readonly) NSDate *created;

// Generate the getters/setters and create a new instance
// with the same values as the original
@property (nonatomic, copy) NSString *backupData;
```

OBJECTIVE-C

DECLARED PROPERTIES

```
// Generate getters/setters and retain the incoming object
@property (nonatomic, strong) NSString *name;

// Same
@property (nonatomic, strong) NSString *title;

// Generate getters/setters and assigns the incoming value
@property (nonatomic) int age;

// Generate getters/setters and assigns the incoming value
@property int age;

// Generate the getters only
@property (nonatomic, readonly) NSDate *created;

// Generate the getters/setters and create a new instar
// with the same values as the original
@property (nonatomic, copy) NSString *backupData;
```

copy is required when the object is mutable. Use this if you need the value of the object as it is at this moment, and you don't want that value to reflect any changes made by other owners of the object.

OBJECTIVE-C

DECLARED PROPERTIES

```objc
// Getter for name property
[self name];

// Call method "doSomething" in current class
[self doSomething];

// Self as parameter to method
[someClass doWorkOn:self];
```

- A class' `self`
  - Implicit local variable
  - Points to the object that was sent the message
  - Allows an object to send a message `[self addDate]`
- `self` uses the accessor methods to access instance variables

OBJECTIVE-C

DECLARED PROPERTIES

- Allows use of dot notation
  - Different notation `[myObject name]` or `myObject.name`

```
// Create property
@property int year;

// Bracket notation
[myCar year];          // Getter
[myCar setYear: 1962]; // Setter

// Dot notation
myCar.year = 1962;          // Setter
NSLog(@"%d", myCar.year);  // Getter
```

OBJECTIVE-C

DECLARED PROPERTIES

- Invoking accessor methods (i.e. getters and setters)

```
[[object theArray] insertObject:[myAppObject objectToInsert] atIndex:0];
[object.theArray insertObject:myAppObject.objectToInsert atIndex:0];
```

- Use for assignment

```
myAppObject.theArray = aNewArray;
[myAppObject setTheArray:aNewArray];
```

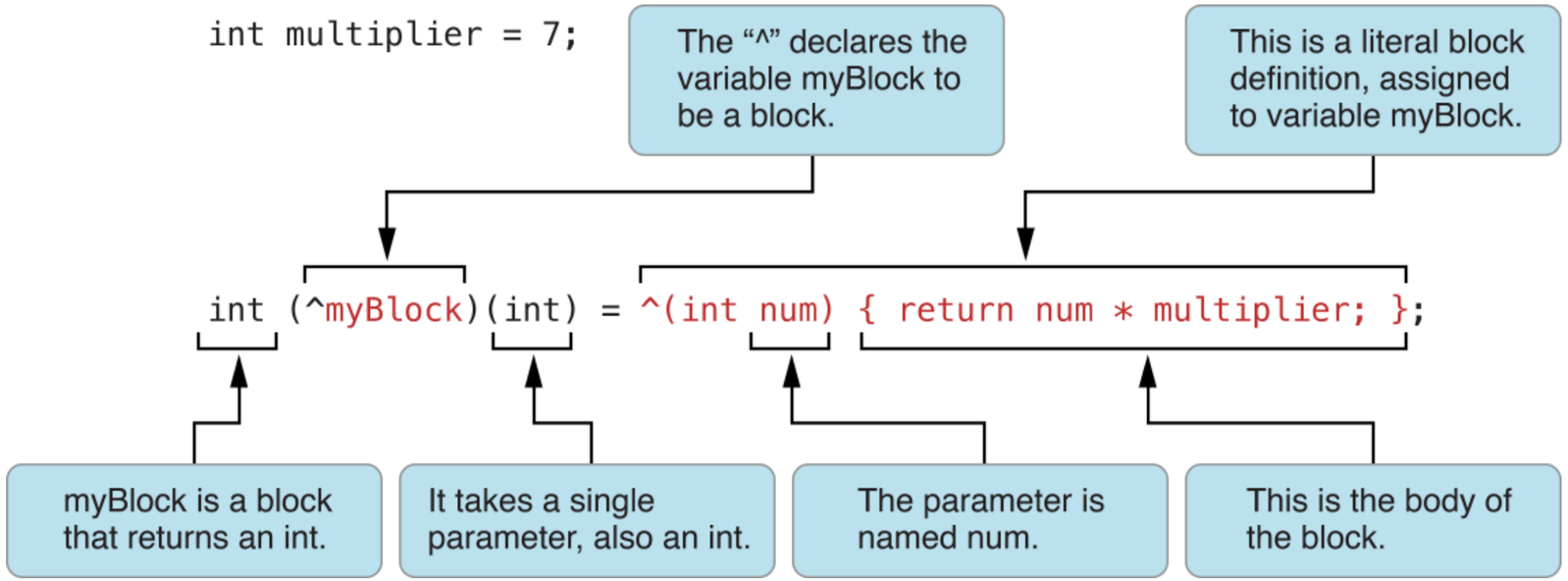
dot

dot

OBJECTIVE-C BLOCKS

BLOCKS

- Objects that encapsulate a unit of work
  - Anonymous functions
  - Passed as parameters to methods; returned from methods




```
int result = myBlock(4); // result is 28
```

Xcode helps

BLOCKS

- Simplify execution of a single task versus writing a collection of methods
  - Scoped locally and variables can be read-only or read-write
- Used throughout UIKit
  - Callbacks
  - Animations
  - Multi-threading
  - Parameters

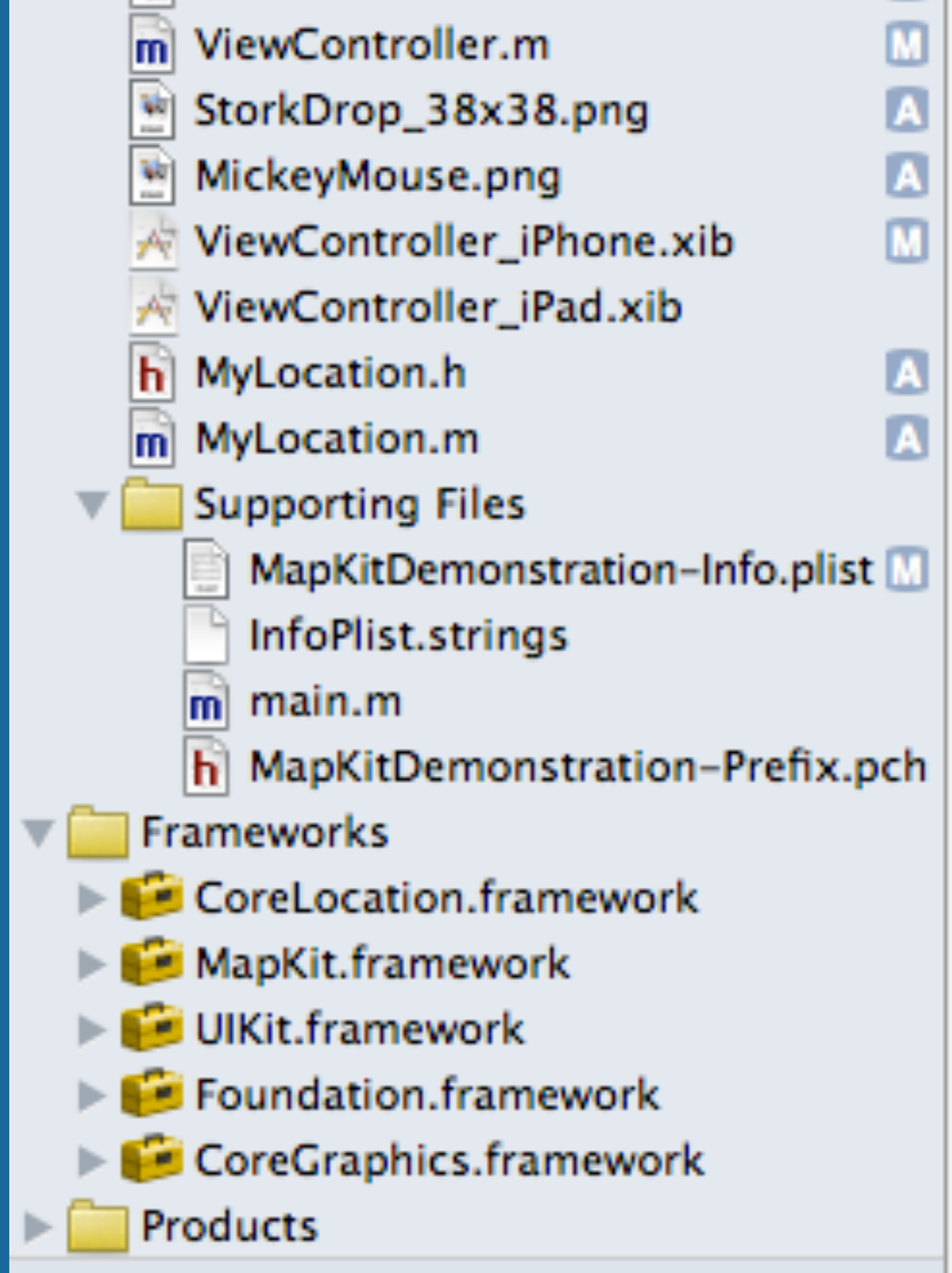
```
- (id)addObserverForName:(NSString *)name
                  object:(id)obj
                   queue:(NSOperationQueue *)queue
              usingBlock:(void (^)(NSNotification *note))block
```

OBJECTIVE-C CATEGORIES

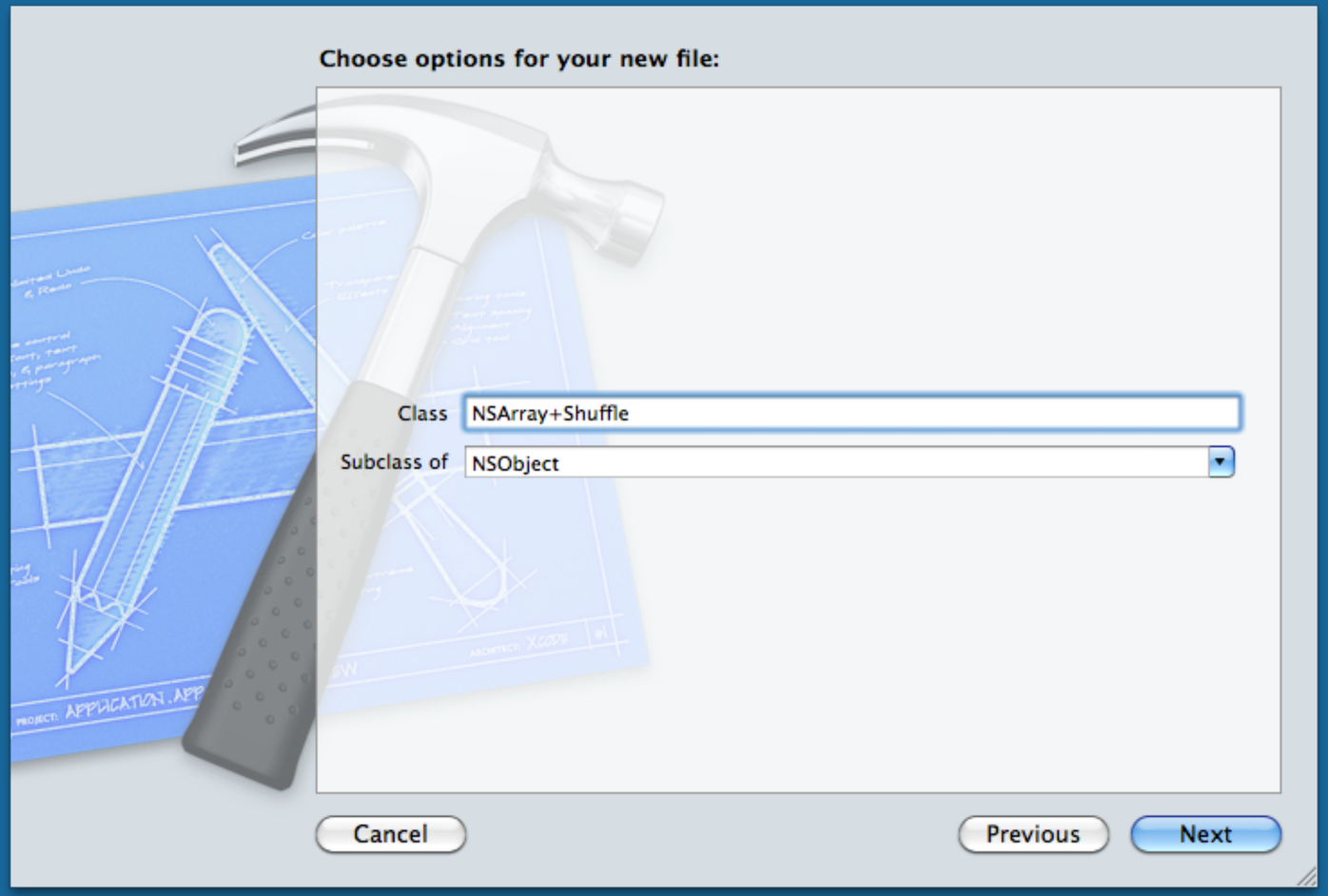
CATEGORIES

SUBTITLE

- Allow you to add methods to an existing class
- Why not subclass?
  - Would not have to change existing codebase
  - Can be applied to other methods that return NS and UI classes



CATEGORIES


Choose options for your new file:
Class
NSArray+Shuffle
Subclass of
NSObject
Cancel
Previous
Next

CATEGORIES

- Declaring the interface for a category

```
//
//  NSArray+Shuffle.h
//  MapKitDemonstration
//
#import <Foundation/Foundation.h>

@interface NSArray (Shuffle)
- (NSArray*) shuffle;
@end
```

CATEGORIES

```
//
//  NSArray+Shuffle.m
//  MapKitDemonstration

#import "NSArray+Shuffle.h"

@implementation NSArray (Shuffle)
- (NSArray*)shuffle
{
    // create temporary mutable array
    NSMutableArray *tmpArray = [NSMutableArray arrayWithCapacity:[self count]];

    for (id anObject in self) {
       NSUInteger randomPos = arc4random()%([tmpArray count]+1);
       [tmpArray insertObject:anObject atIndex:randomPos];
    }

    return [NSArray arrayWithArray:tmpArray];  // non-mutable autoreleased copy
}

@end
```

- Implementing a category

CATEGORIES

- Using a category

```
// Test NSSArray+Shuffle Category
   NSArray *places = [NSArray arrayWithObjects:@"Universal Studios",
                                               @"Disney World", @"Sea World", nil];
   NSLog(@"Places (before): %@",places);
   NSLog(@"Places (after): %@",[places shuffle]);
```

CATEGORIES

- Tips & Tricks
  - Categories are inherited (i.e. categories on NSObject are available to every class)
  - The names must be unique across all namespaces
- A couple useful categories
  - coredata-easyfetch
    - https://github.com/halostatue/coredata-easyfetch
  - Image utilities
    - http://mattgemmell.com/2010/07/05/mgimageutilities/

OBJECTIVE-C PROTOCOLS

PROTOCOLS

- Declare methods that can be implemented by any class
- Not a class; define an interface that other objects can implement
  - “Conforming to a protocol”
  - Implement in header file

```
@protocol MyProtocol

@required
- (void)myProtocolMethod;
@option
- (void)anotherMethod;

@end
```

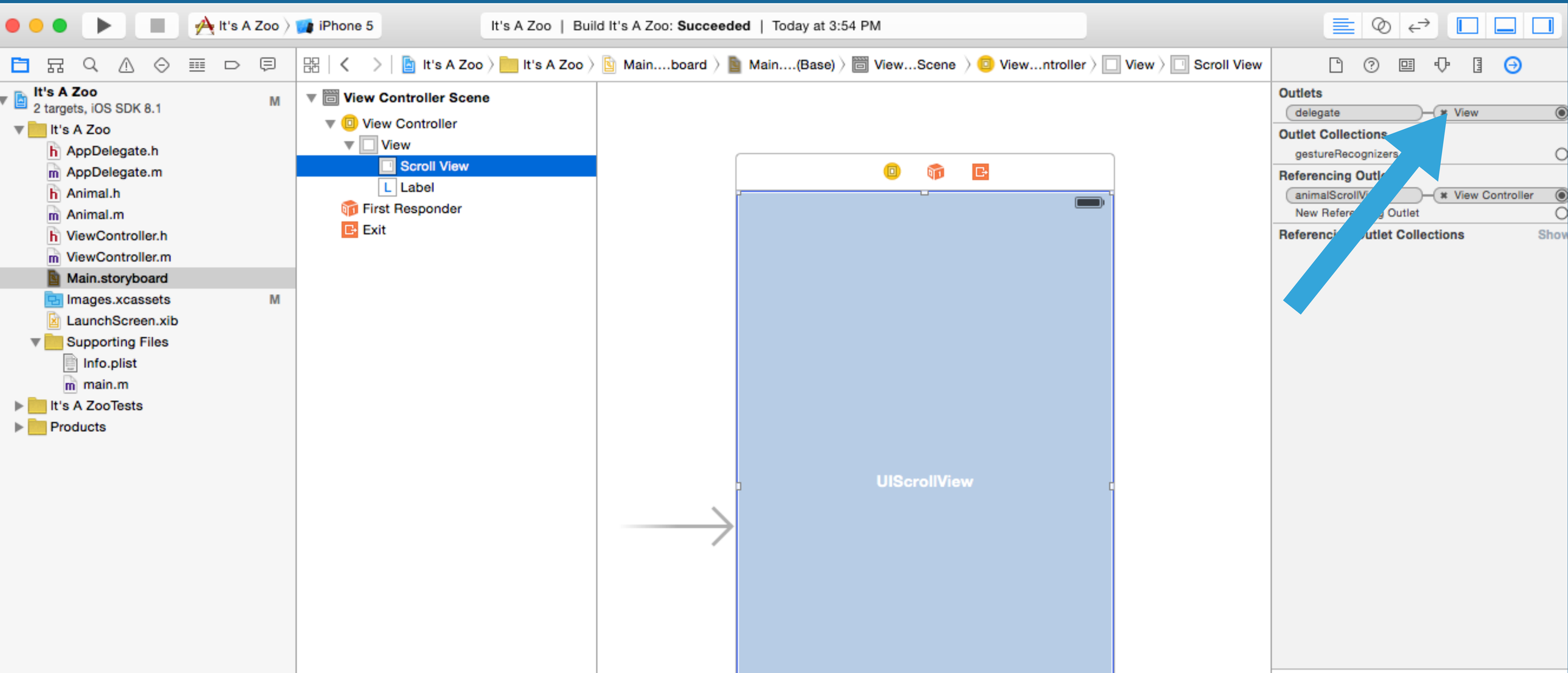
PROTOCOLS

- Classes that wish to implement a protocol declare it in their interface

```
@interface MyClass : NSObject <AProtocol> {}
...
@end
```

- All non-optional methods must be implemented.
- Used to specify the interface for a delegate object
  - Can be done programmatically or in Storyboard

PROTOCOLS


It's A Zoo 〉 iPhone 5
It's A Zoo | Build It's A Zoo: Succeeded | Today at 3:54 PM
It's A Zoo 〉 It's A Zoo 〉 Main....board 〉 Main....(Base) 〉 View...Scene 〉 View...ntroller 〉 View 〉 Scroll View
It's A Zoo
2 targets, iOS SDK 8.1
M
It's A Zoo
AppDelegate.h
AppDelegate.m
Animal.h
Animal.m
ViewController.h
ViewController.m
Main.storyboard
Images.xcassets
M
LaunchScreen.xib
Supporting Files
Info.plist
main.m
It's A ZooTests
Products
View Controller Scene
View Controller
View
Scroll View
Label
First Responder
Exit
UIScrollView
Outlets
delegate
View
Outlet Collections
gestureRecognizers
Referencing Outl
animalScrollV
View Controller
New Refere g Outlet
Referenci utlet Collections
Show

PROTOCOLS

```
@protocol UIScrollViewDelegate
@optional
- (void)scrollViewWillBeginDragging:(UIScrollView *)scrollView
- (void)scrollViewDidEndDragging:(UIScrollView *)scrollView
                  willDecelerate:(BOOL)decelerate
...
@end

@interface UIScrollView : UIView
@property (assign) id <UIScrollViewDelegate> delegate;
@end

@interface MyViewController : UIViewController <UIScrollViewDelegate> {}
@end

MyViewController *myVC = [[MyViewController alloc] init];
UIScrollView *scrollView = ...;
scrollView.delegate=myViewController;
```

Protocol definition

PROTOCOLS

```
@protocol UIScrollViewDelegate
@optional
- (void)scrollViewWillBeginDragging:(UIScrollView *)scrollView
- (void)scrollViewDidEndDragging:(UIScrollView *)scrollView
                  willDecelerate:(BOOL)decelerate
...
@end

@interface UIScrollView : UIView
@property (assign) id <UIScrollViewDelegate> delegate;
@end

@interface MyViewController : UIViewController <UIScrollViewDelegate> {}
@end

MyViewController *myVC = [[MyViewController alloc] init];
UIScrollView *scrollView = ...;
scrollView.delegate=myViewController;
```

Declare a delegate that adopts a protocol

PROTOCOLS

```
@protocol UIScrollViewDelegate
@optional
- (void)scrollViewWillBeginDragging:(UIScrollView *)scrollView
- (void)scrollViewDidEndDragging:(UIScrollView *)scrollView
                  willDecelerate:(BOOL)decelerate
...
@end

@interface UIScrollView : UIView
@property (assign) id <UIScrollViewDelegate> delegate;
@end

@interface MyViewController : UIViewController <UIScrollViewDelegate> {}
@end

MyViewController *myVC = [[MyViewController alloc] init];
UIScrollView *scrollView = ...;
scrollView.delegate=myViewController;
```

Conform to a protocol

PROTOCOLS

```
@protocol UIScrollViewDelegate
@optional
- (void)scrollViewWillBeginDragging:(UIScrollView *)scr
- (void)scrollViewDidEndDragging:(UIScrollView *)scroll
                  willDecelerate:(BOOL)decelerate
...
@end

@interface UIScrollView : UIView
@property (assign) id <UIScrollViewDelegate> delegate;
@end

@interface MyViewController : UIViewController <UIScrollViewDelegate> {}
@end

MyViewController *myVC = [[MyViewController alloc] init];
UIScrollView *scrollView = ...;
scrollView.delegate=myViewController;
```

Set delegate when methods are implemented

PROTOCOLS

```
@interface MyViewController : UIViewController <UIScrollViewDelegate> {}

- (void)loadView {
...
  // Add the scrollview
  scroll = [[UIScrollView alloc] initWithFrame:CGRectMake(0,30,320,400)];
}


 -(void)scrollViewDidScroll:(UIScrollView *)scrollView {
    static NSInteger previousPage = 0;
    CGFloat pageWidth = scrollView.frame.size.width;
    float fractionalPage = scrollView.contentOffset.x / pageWidth;
    NSInteger page = lround(fractionalPage);
    if (previousPage != page) {
        self.currentPage = page;
        previousPage = page;
        NSLog(@"PAGE %d",self.currentPage);
        // Do something with page
    }
}
```

Method declared in protocol. Sends messages every time the scrollview scrolls

OBJECTIVE-C CHEAT SHEET

OBJECTIVE-C

header File

```
//
//  MyClass.h
//  Objective-C Class
//

#import <Foundation/Foundation.h>

@interface MyClass : NSObject
{
    // Declare instance variables (optional)
}

// Define properties

// Define methods

@end
```

Implementation File

```
//
//  MyClass.m
//  Objective-C-hristmas
//

#import "MyClass.h"

@interface MyClass ()
// Define private properties
// Define private methods
@end



@implementation MyClass
// Implement methods
@end
```

OBJECTIVE-C

```objc
// Defining methods
- (void)message;
- (void)message:(id)parameter;
- (BOOL)message:(id)parameter1 anotherMessage:(id)parameter2;


// Implementing methods
- (void)message {
    // Do some stuff
}

- (BOOL)message:(id)parameter {
    // Do some stuff with parameter
    return YES;
}
```

OBJECTIVE-C

```
// Defining Properties
@property (attribute1, attribute2) propertyName;


// Synthesizing Properties
_propertyName   // Instance variable created
propertyName    // Accessor getter method
setPropertyName // Accessor setter method


// Bracket syntax
value = [MyClass propertyName];
// Dot syntax for getter
value = MyClass.propertyName;

// Bracket syntax
[myClass setPropertyName:value];
// Dot syntax for setter
myClass.propertyName = value;
```

OBJECTIVE-C

```objc
// Creating a class instance
MyClass *classObject = [[MyClass alloc] init];


// Creating a custom init method
- (id) initWithParameter:(id)parameter {
      if ( self = [super init]) {
           self.someProperty = parameter;
    }
     return self;
}
```

NSOBJECT DECLARES INIT PROTOTYPE

CALLS INIT ON SUPER CLASS

OBJECTIVE-C

```
// Messaging - invoking methods on an object

// Fancy objective-c terms
[receiver message];

// Message expression
[myClass method];

// Pass an argument as part of the message
[myClass message:argument];

// Pass two argument2 as part of the message
[myClass message:argument anotherMessage:anotherArgument];

// Nesting Methods
id output = [receiver message];
[myClass message:output];
[myClass message:[object method]];
```

OBJECTIVE-C

```objc
// Declare a custom protocol
@protocol MyProtocol
@required
- (void)myProtocolMethod;
@optional
- (void)anotherMethod;
@end
```

DECLARE PROTOCOL

```objc
// .h
@interface MyClass : NSObject <MyProtocol>
@end
```

CONFORM TO IT

```objc
// .m
@implementation MyClass
- (void)myProtocolMethod  {
   // Do some stuff
{
@end
```

IMPLEMENT PROTOCOL

THE UNIVERSITY OF CHICAGO

ADVANCED iOS APPLICATION DEVELOPMENT

MPCS 51032 • SPRING 2020 • SESSION 2A