Table of Contents

Objective-C Bootcamp

Welcome to the Objective-C Bootcamp

This is a two-day class that will prepare you for either the Cocoa or iOS five-day classes.

Daily Schedule at Banning Mills:

- Breakfast -- 8:30 AM to 9:00 AM
- Class Begins -- 9:00 AM
- Lunch -- 12:30 PM to 1:00 PM
- Optional walk -- around 2:00 PM , 30 to 45 minutes
- Class Ends -- 6:30 PM (Except Friday)
- Dinner -- 6:30 PM
- Open lab -- until 10 PM

Objective-C Bootcamp Goals

Learn how to write essential Objective-C code for the command line.

Objective-C:

- Extensions to C
- OOP Nomenclature
- Language Idioms
- Key concepts: pointers, memory management (ARC), objects, messages

Foundation:

- Fundamental OS X Library
- Commonly Used Objects
- Basic data types
- Collections

Class Structure

Chapter:

- Lecture or Demonstration
- Exercise
- Challenge (try them independently for best results)
- For the More Curious

Homework Assignment

Assigned Reading

Objective-C book Chapters 1 to 11.

Concepts learned:

- C Language fundamentals
- Basic data types: int, float, char, etc.
- Making decisions: booleans, conditional expressions, if / else
- Functions: writing functions, local variables, passing arguments by value and reference
- Loops: for, while, do-while
- Memory: pointers, stack, heap
- Structs

The Rest of the Objective-C Class

You will learn the Objective-C language and its object-oriented capabilities.

Objective-C is a superset of C: Any valid C code will compile as Objective-C.

New keywords: To avoid conflicts with C, any new keyword added to Objective-C begins with the @ character, such as *@interface* and *@selector*.

Including / importing files: In Objective-C you prefer to *#import* rather than *#include* header files. *#import* automagically prevents multiple includes of headers.

Objects

Classes and Instances 1

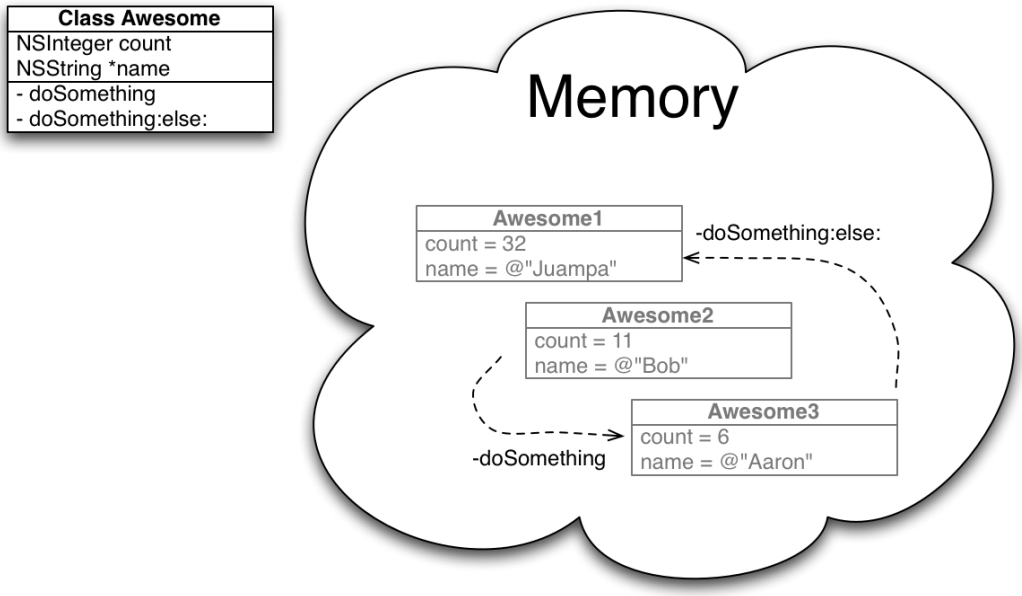
Class:

- Defines its Objects
- Defines Messages: what functionality will the object have?
- Implements Methods: make the functionality work

Instance (Object):

- Contains Data Only
- Knows its Class

Classes and Instances 2



Creating Objects

C:

```
int year;
year = 2011;
int hitPoints = 15;
```

Objective-C:

```
NSDate *today;
today = [NSDate date];

id anotherDate = [NSDate date];
```

Notice that in Objective-C a reference to an object is always a pointer.

id is a pointer to any object type.

Messages and Methods

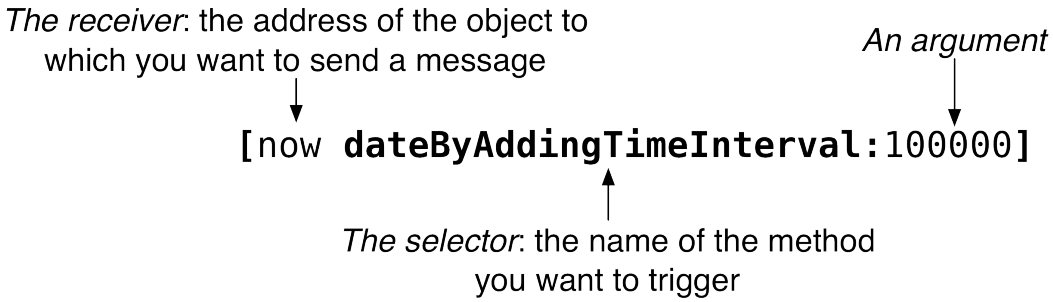
Message:

- Abstract concept: request sent to an object (instance)
- Concrete concept: call an object's method

Method: A function attached to a class - Has access to instance data

```
[obj1 doSomething];
int count = [obj2 itemCount];

[obj3 setYear:1984];
```

Message Anatomy



More Messages

Nested Messages

Messages always enclose the receiver and the selector inside a pair of square brackets. These brackets can be nested.

```
// The alloc method returns a pointer to a newly created instance:
id dateInstance = [NSDate alloc];


// Date is not ready to be used yet - it needs to be initialized:
dateInstance = [dateInstance init];


// These two messages can be combined into a common idiom:
id dateInstance = [[NSDate alloc] init];
```

Multiple Arguments

Messages can have zero, one, or more arguments.

```
// Zero arguments:
[obj updateCount];      // @selector(updateCount)


// One argument:
[obj setCount:13];      // @selector(setCount:)


// Two arguments:
[obj calculateSalariesWithPersonList:persons raise:amount];
//@Selector(calculateSalariesWithPersonList:raise:)


// Common formatting for multiple argument messages:
[obj calculateSalariesWithPersonList:persons
                               raise:amount];
```

Messages to nil

nil is a pointer to no object. It is the Objective-C equivalent of **NULL**.

In most languages, sending a message to a nil object is an error.

In Objective-C it is perfectly legal!

```
CoolClass *obj = nil;
[obj doSomething];     // Nothing happens -- no error


Froozle *frooz = [obj latestFroozle];
// frooz will be nil


int count = [obj tellMeTheCount];
// count will be zero
```

NSString

NSString

NSString is a class provided to you by the Foundation framework.

Use it instead of a C string whenever you need text.

```
// Constant literal NSString shortcut:
NSString *myStr = @"Hello ObjC students";


// NSStrings are immutable:
NSString *emptyString = [[NSString alloc] init];   // Stuck with empty string!


// Create one from another:
NSString *str = [NSString stringWithFormat:@"I love the number %d", 7];




// %-tokens are the same as in printf C-function.
```

Making More Strings

```
// Concatenate two strings:
NSString *one = @"Hello ";
NSString *two = @"world!";
NSString *three = [one stringByAppendingString:two];
// one and two are unchanged


// Divide a string:
NSString *list = @"Foo, Bar, Baz";
NSArray *items = [list componentsSeparatedByString:@", "];
```

Comparing Strings

```
// Remember: references to objects are pointers:
NSString *moon = @"The moon";
NSString *cheese = [NSString stringWithString:@"Cheese"];


if (moon == cheese) {
    // You will never get here! Pointer comparison!
}


if ([moon isEqualToString:cheese]) {
    // You may get here one day...
}
```

NSLog

Useful for simple debugging and sanity checks.

The **%@** token converts an object to a string representation.

```
NSDate *now = [NSDate date];
NSLog(@"The date is %@", now);

// Console output:
2011-10-10 17:59:05.498 date[40893:707] The date is 2011-10-10 22:59:05 +0000
```

```
// String representation is given by -description method.
```

NSArray

Arrays in Objective-C

An **NSArray** instance is just a list of pointers to objects.

You may think about it as an ordered list of **id**'s.

```
// Create an array from other objects:
Foo *foo = ...;
Bar *bar = ...;
Baz *baz = ...;


NSArray *listOfStuff = [NSArray arrayWithObjects:foo, bar, baz, nil];
// nil marks the end (a *fenced* list).


for (NSUInteger i=0; i<[listOfStuff count]; i++) {
    NSLog(@"Item at index %d is %@", i, [listOfStuff objectAtIndex:i]);
}
```

Fast Enumeration

```
// This is very common:

for (NSUInteger i=0; i<[list count]; i++) {
    id item = [list objectAtIndex:i];
    // Do something with item ...
}




// This is cleaner and extremely efficient:

for (id item in list) {
    // Do something with item ...
}
```

NSMutableArray

NSArray is immutable. Once you create one it can't be modified.

NSMutableArray is a mutable subclass of **NSArray**.

```
// Create an array from other objects:
Foo *foo = ...;
Bar *bar = ...;
Baz *baz = ...;


NSMutableArray *listOfStuff = [NSMutableArray array];   // Empty
[listOfStuff addObject:foo];
[listOfStuff addObject:baz];
[listOfStuff insertObject:bar atIndex:1];


[listOfStuff removeObjectAtIndex:2];
[listOfStuff removeAllObjects];     // Empty again!
```

Documentation

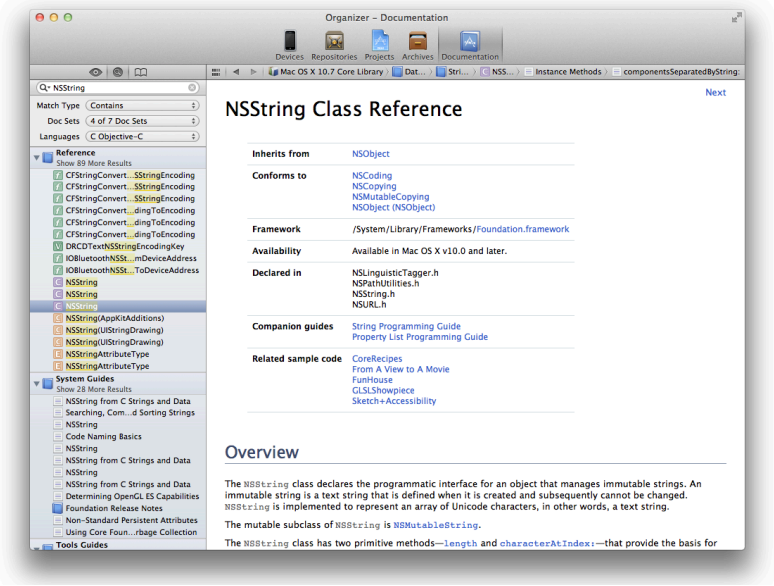
Documentation Is Your Friend

From now on you will use the documentation **constantly**.

The Foundation, Cocoa, CocoaTouch, CoreData, etc. frameworks are just too big. You will need to look-up the documentation as you work.

The Organizer Window

Xcode -> Window -> Organizer

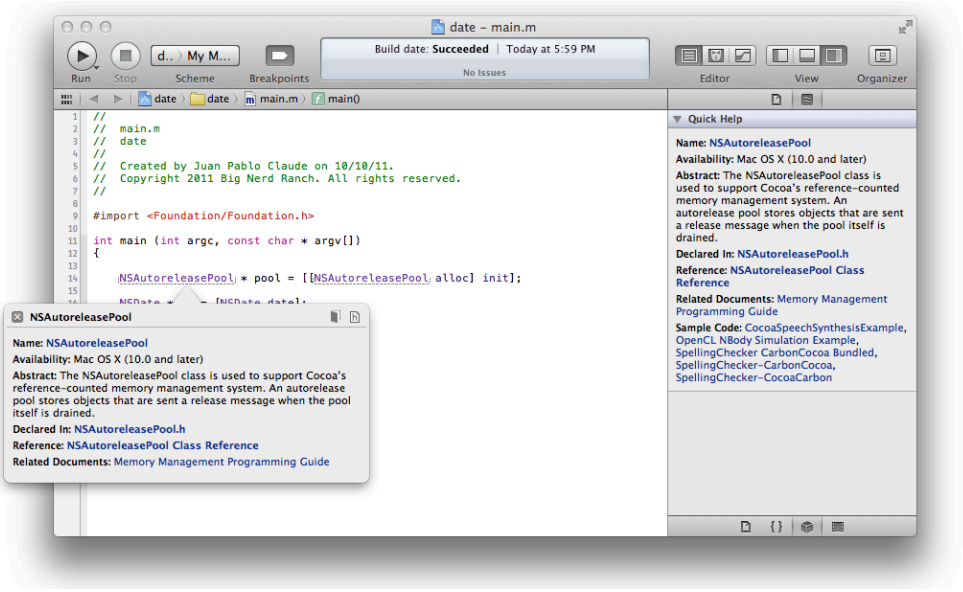


Shortcuts

- Command + double-click on symbol: opens the header file where defined
- Option + double-click on symbol: opens documentation
- Command + click on symbol: definition in standard window
- Option + Command + click on symbol: definition in assistant window

Quick Help

Option-click on a symbol



Your First Class

Classes Used So Far

You have been using classes supplied by the Foundation framework:

- NSData
- NSString
- NSArray
- NSMutableArray

Now you will create your own class!

Interface and Implementation

Interface: Declare your intentions. It is called an interface because just like in a real-life object it specifies how the user can interact with it (buttons and dials). Done in a **header (.h)** file.

- Declare instance variables (data)
- Declare properties (data/behavior)
- Declare methods (behavior)

Implementation: Make it work. Define methods. Done in an **implementation (.m)** file.

Interface

```
// ElectricGuitar inherits from NSObject
@interface ElectricGuitar : NSObject
{
    // Instance variables:
    float volume;   // Take it to 11!
    float tone;
    NSMutableArray *tuning;
}

// Methods:
- (void)playDString;
- (void)playDStringWithFretPosition:(NSUinteger)fret;
- (void)playHighwayStar;

@end
```

Implementation

```
@implementation ElectricGuitar

- (void)playDString
{
    // Do the deed!
}

- (void)playDStringWithFretPosition:(NSUinteger)fret
{
    // How is this done?
}

- (void)playHighwayStar
{
    Player *ritchie = [[Player alloc] initWithPlayer:@"Ritchie Blackmore"];
    [ritchie shredAway];
}

@end
```

Accessor Methods

Object-oriented dogma tells us that instance variables should not be accessed directly by others. Also, there may be other side-effects that you want to happen when an instance variable is changed. To this end, you implement **accessor methods**.

- Instance variable named foo
- Getter method: gets the value of an instance variable - named **foo**
- Setter method: sets the value of an instance variable - named **setFoo:**

The naming convention is important! You **will** break things if you don't follow it.

Implementing Accessor Methods 1

```
 @interface Froozler : NSObject
{
    double blurbleeflox;
}

- (void)setBlurbleeflox:(double)newBlurbleeflox;
- (double)blurbleeflox;

@end
```

Implementing Accessor Methods 2

```
@implementation Froozler

- (void)setBlurbleeflox:newBlurbleeflox
{
    blurbleeflox = newBlurbleeflox;
}

- (double)blurbleeflox
{
    return blurbleeflox;
}

@end
```

Properties

Writing accessors is boring. Anything boring tends to be error-prone.

```
// Beat the boredom, use properties!

@interface Froozler : NSObject
{
    double blurbleeflox;    // The ivar declaration is optional
}
@property double blurbleeflox;


// In the implementation:

@implementation Froozler

@synthesize blurbleeflox;   // Accessors are written for you!

@end
```

Self

Inside methods, you have an implicit local variable called **self**. **self** is a pointer to the current instance of the class.

You can use **self** to send messages to the same instance.

```
- (double)doThatThingYouDo
{
    float temp = [self calculateThatStuff];
    return temp * 123.0;
}
```

Inheritance

Overview

Inheritance is a powerful but sometimes abused object-oriented feature.

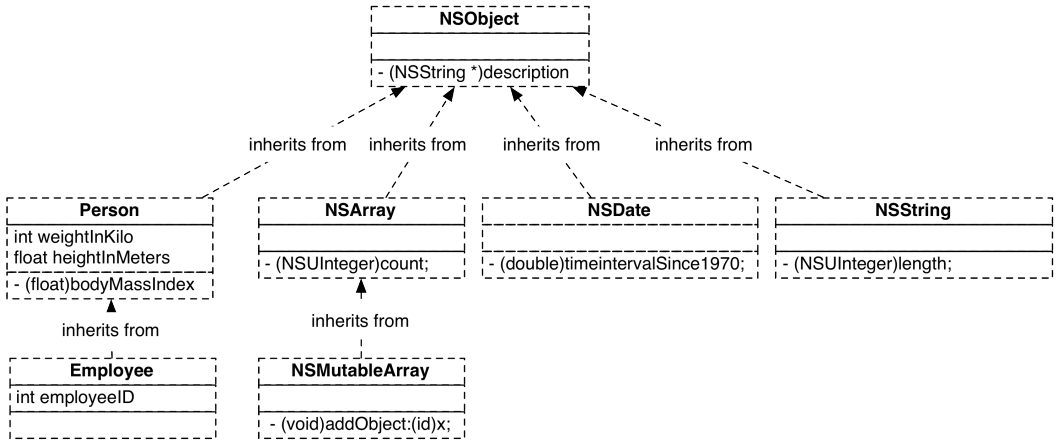
It allows a programmer to define a class as an extension of another.

When a class *inherits* from another, the child class (**subclass**) implicitly gets all the instance variables and methods of the parent class (**superclass**).

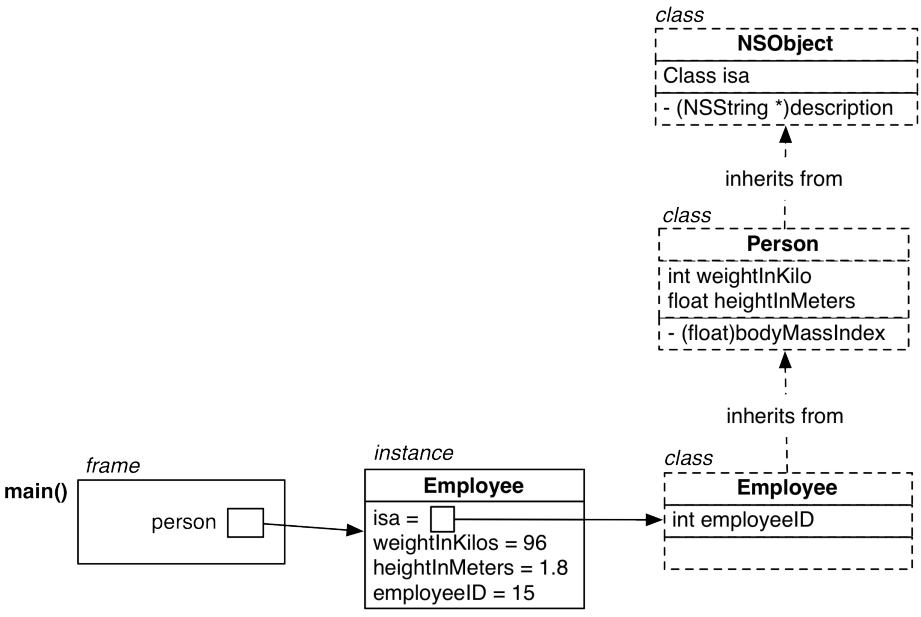
More specialized classes inherit from more generic ones.

Objective-C only supports *single inheritance*. You can only inherit from one superclass at a time.

Some Classes You Know



In This Chapter



Syntax

```
@interface MyClass : NSObject   // NSObject is the superclass
{
    // Declare only new ivars!
    double foo;
    NSArray *list;
}

// Declare only new properties and methods:
- (void) doSomethingNew;

@end
```

Overriding Methods

A subclass may re-implement a method that is already defined by a superclass (no matter how high up).

The new implemetation is used instead for that class and any subclass thereof.

Super

What if you want to explicitly invoke a method from a superclass? This is frequently done when you want to extend a parent's method.

```
@implementation MyClass

// calculateThatThing is overriden!
// How do you get a hold of the old code?
- (float) calculateThatThing
{
    float result = [super calculateThatThing];
    return result * [self correctionFactor];
}

@end
```

Message Dispatch

How is a method found for a given object?

1. What is the object's class? (isa pointer inherited from NSObject)
2. Does the class implement the method?
3. If YES, execute the method implementation.
4. If NO locate the parent of the object's class.
5. Does the parent class implement the method?
6. If YES, execute the method implementation.
7. If NO, go to 4.

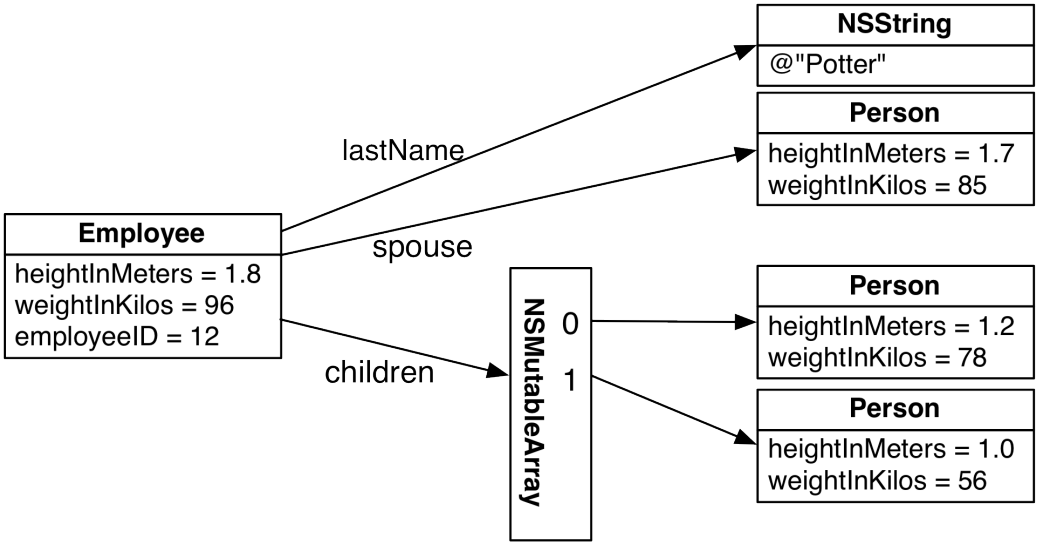
If NSObject is reached and the implementation is not found, an exception is raised.

Object Instance Variables

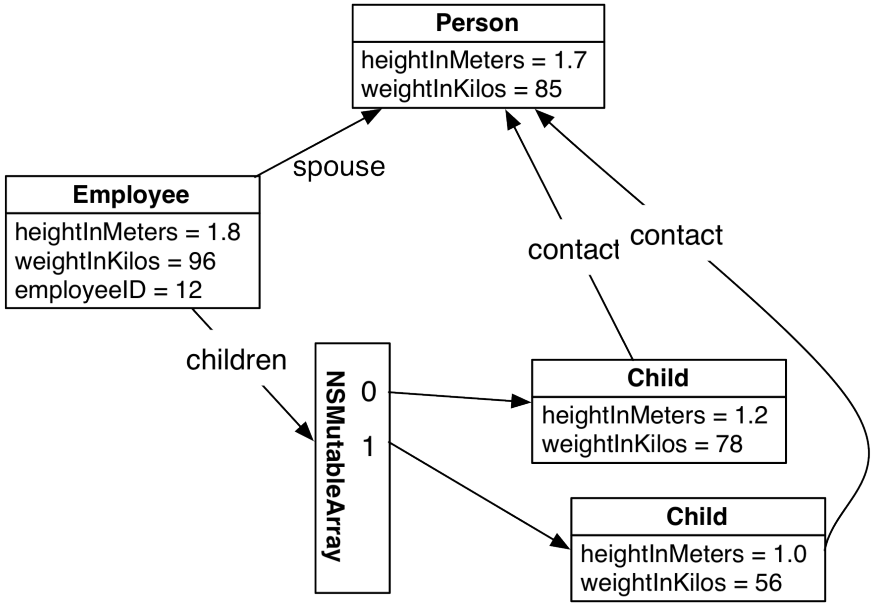
Instance Variables

```
@interface Employee : Person
{
    // C-type ivar - lives within the object:
    int employeeID;

    // Object ivars - this object only stores the address
    // Pointed-to instance is stored elsewhere in memory:
    NSString *lastName;
    Person *spouse
    NSMutableArray *children;
}

@end
```

Object Instance Variables



Objects Can Be Reused



Ownership

If an object has an instance variable to another object, the object **owns** the object that is being pointed to.

Objects keep a count of how many owners they have. When that count reaches zero, the object is *deallocated*.

The owner count is managed by *Automatic Reference Counting* or **ARC**. Apple introduced ARC with Xcode 4.2, OS X 10.7 (Lion) and iOS 5.

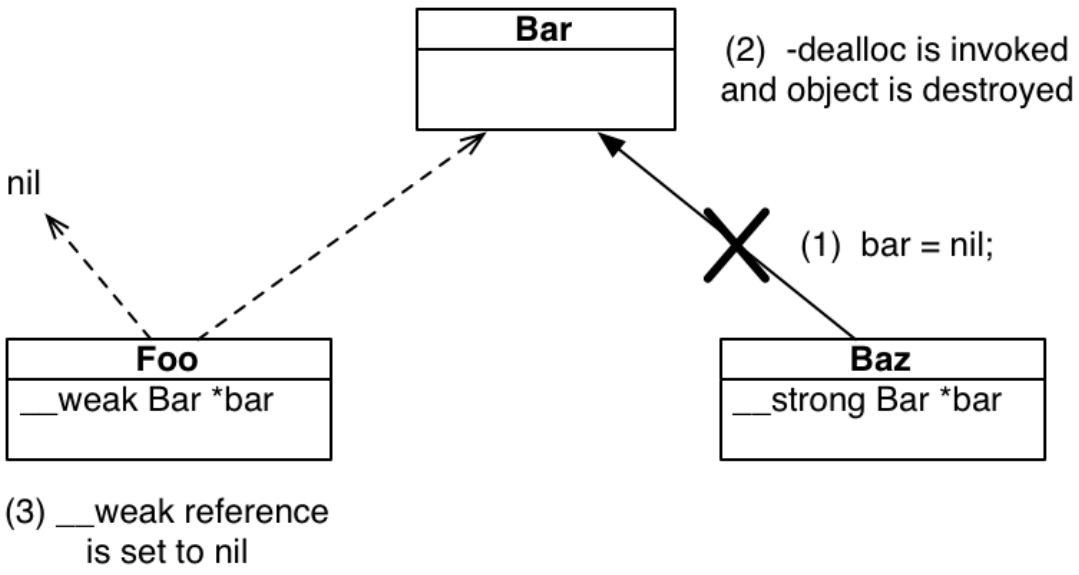
Reference Types

```
@interface AwesomeObject : NSObject
{
    // __strong is the default and can be omitted:
    __strong CoolObject *cool;

    // __weak does not increase owner count:
    __weak SplendidObject *splendid;
 }

 @end
```

Properties

```
@interface AwesomeObject : NSObject
{
    CoolObject *cool;
    __weak SplendidObject *splendid;
}

@property (strong) CoolObject *cool;
@property (weak) SplendidObject *splendid;

@end
```

ARC



@class

#import-ing a header file "pastes" the content.

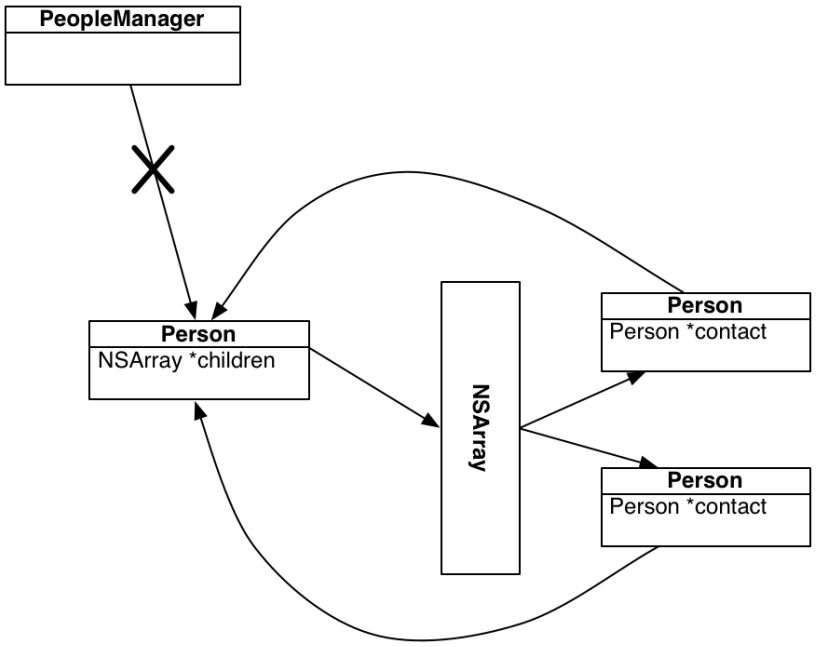
If possible, it is more efficient to use **@class**.

```
@class Dog;

@interface Human : NSObject
{
    Dog *dog;
}
@end
```

The Dog.h header will still have to be imported where the class is actually used.

Preventing Memory Leaks

Cyclical References

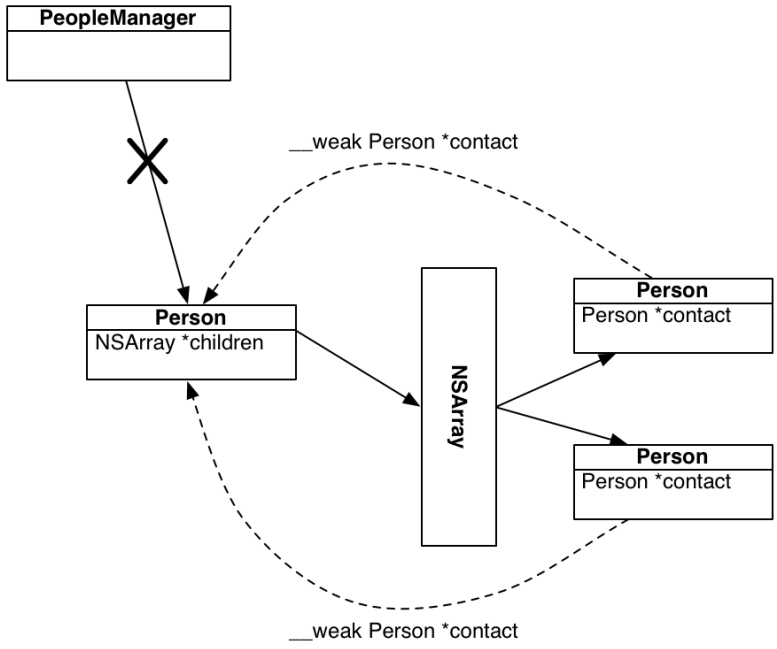


Parent-Child Relationships

When objects are arranged in parent-child relationships, parents should own the children, but children should not own the parents.

If you do not follow this rule, you may end-up with a *retain cycle*.

Avoiding Retain Cycles

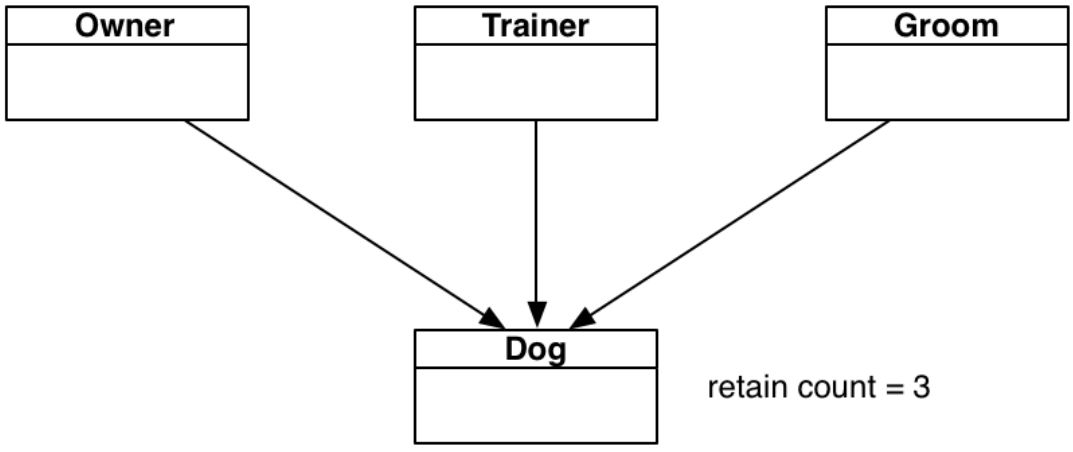


Manual Reference Counting

Before ARC, the programmer was responsible for managing an object's reference counts.

```
// Essential messages inherited from NSObject:

[anObject retain];      // Object gains an owner

[anObject release];    // Object loses an owner
```

Retain Counts



Manual Reference Counting Implementation

```
@interface Human : NSObject {
    Dog *dog;
}
@end

@implementation Human
- (void)setDog:(Dog *)newDog {
    [newDog retain];
    [dog release];
    dog = newDog;
}

- (void)dealloc {
    [dog release];
    [super dealloc];
}
@end
```

Methods Returning Objects

```
- (Foo *)giveMeAFoo
{
    Foo *result = [[Foo alloc] init];

    // Should the caller be in charge of releasing the result?
    // Should I pre-release it?
    // [result release]; ??

    return result;
}
```

Methods Returning Objects

```
- (Foo *)giveMeAFoo
{
    Foo *result = [[Foo alloc] init];

    // Correct solution:
    [result autorelease];

    return result;
}
```

Autorelease Pool

When is an autoreleased object actually released? *When the current autorelease pool is drained.*

```
// Create an autorelease pool:
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

// Create autoreleased objects!

[pool drain];   // All autoreleased objects are released.
```

Manual Retain Count Rules

Objects created with a method whose name contains **alloc**, **new** or **copy** are given to you with a retain count of one and are not in the autorelease pool. You own these objects and you are responsible for explicitly releasing them when you no longer need them.

Objects obtained by any other means are not owned by you. The are given to you with a retain count of one but are already in the autorelease pool.

If you do not own an object, you can take ownership by sending it a **retain** mesage.

If you own an object and no longer need it (want to relinquish ownership) you send it a **release** or **autorelease** message.

An object will continue to exist until its retain count reaches zero. Before an object is destroyed, it is sent a **dealloc** message.

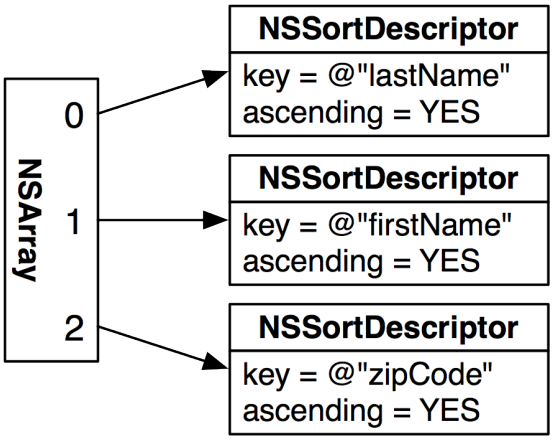
Collection Classes

NSArray and NSMutableArray

```
// Create a mutable array:
NSMutableArray *aList = [[NSMutableArray alloc] init];

// or:
NSMutableArray *aList = [NSMutableArray array];

// Add stuff:
[aList addObject:foo];

// Remove stuff:
[aList removeObjectAtIndex:0];

// Create an immutable array:
NSArray *anotherList = [NSArray arrayWithObjects:@"Spam",
                                                 @"Shrubbery",
                                                 @"Ni",
                                                 nil];
```

Sorting Arrays

```
// Method of NSMutableArray:
- (void)sortUsingDescriptors:(NSArray *)sortDescriptors;
```



Sorting Arrays Implementation

```
NSSortDescriptor *lastNameSD, *firstNameSD, *zipSD;

lastNameSD = [NSSortDescriptor sortDescriptorWithKey:@"lastName"
                                           ascending:YES];


firstNameSD = [NSSortDescriptor sortDescriptorWithKey:@"firstName"
                                            ascending:YES];


zipSD = [NSSortDescriptor sortDescriptorWithKey:@"zipCode"
                                      ascending:YES];


[employees sortUsingDescriptors:[NSArray arrayWithObjects:lastNameSD,
                                                          firstNameSD,
                                                          zipSD,
                                                          nil]];

// lastName, firstName, and zipCode are ivars of Employee.
```

Filtering Arrays

```
// Method of NSMutableArray:
- (void)filterUsingPredicate:(NSPredicate *)predicate;


// Method of NSArray, returns a new array:
- (NSArray *)filteredArrayUsingPredicate:(NSPredicate *)predicate;


// Predicates are SQL-like logical operators:
NSPredicate *predicate =
        [NSPredicate predicateWithFormat:@"(lastName like[c] %@) AND (birthday > %@)",
                        lastNameSearchString, birthdaySearchDate];









// Predicates can refer to ivars by name or use keypaths: foo.bar.lastName
```

NSSet and NSMutableSet

Sets are collections with no order and objects can be present only once. Useful when dealing with membership issues -- is the object in there?

```
// Create a set:
+ (id)setWithArray:(NSArray *)elements;
+ (id)setWithObjects:(id)firstObj ...;
+ (id)set;  // Useful in NSMutableSet case

// Use the set:
- (NSUInteger)count;    // How many elements in the set?
- (BOOL)containsObject:(id)anObject;    // Is it in there?

// ComparingSets:
- (BOOL)isEqualToSet:(NSSet *)aSet;
- (BOOL)isSubsetOfSet:(NSSet *)aSet;
- (BOOL)intersectsSet:(NSSet *)aSet;
```

NSMutableSet Methods

```
// Methods in NSMutableSet - they modify the receiver:
- (void)unionSet:(NSSet *)aSet;
- (void)minusSet:(NSSet *)aSet;
- (void)intersectSet:(NSSet *)aSet;
```

NSDictionary and NSMutableDictionary

Dictionaries are an extremely useful collection type. The are unordered and associate a **key** with a **value**.

The key can be any object that is copiable and unique within the collection. Typically, they are constant strings.

```
// Create a dictionary:
NSDictionary *dict = [NSDictionary dictionaryWithObjectsAndKeys:
                    @"value1", @"key1", @"value2", @"key2", nil];

// Create a mutableDictionary:
NSMutableDictionary *mDict = [NSMutableDictionary dictionary];

[mDict setObject:foo forKey:@"fooInstance"];
id obj = [dict objectForKey:@"key2"];
[mDict removeObjectForKey:@"foo"];
[mDict removeAllObjects];
```

NSDictionary Fast Enumeration

```
// Fast enumeration:
for (id key in dict) {
    // Do work!
}
```

C Primitive Types

Collections can only contain objects (pointers to objects actually). How do you put items like numbers in a collection?

```
NSMutableArray *numbers = [[NSMutableArray alloc] init];
[numbers addObject:[NSNumber numberWithInt:7]];
[numbers addObject:[NSNumber numberWithBool:YES]];

int seven = [[numbers objectAtIndex:0] intValue];
BOOL yn = [[numbers objectAtIndex:1] boolValue];
```

More Complex Non-Object Types

```
// Points:
typedef struct _NSPoint {
    CGFloat x;
    CGFloat y;
} NSPoint;

// Rectangles:
typedef struct _NSRect {
    NSPoint origin;
    NSSize size;
} NSRect;

// Ranges:
typedef struct _NSRange {
    NSUInteger location;
    NSUInteger length;
} NSRange;
```

NSValue

How do you put points, ranges, etc. in a collection?

```
NSValue *pointVal = [NSValue valueWithPoint:anNSPoint];
NSValue *rectVal = [NSValue valueWithRect:anNSRect];
NSValue *rangeVal = [NSValue valueWithRange:anNSRange];

NSArray *array = [NSArray arrayWithObjects: pointVal,
                                            rectVal,
                                            rangeVal,
                                            nil];

// Unpack:
NSPoint point = [[array objectAtIndex:0] pointValue];
NSPoint rect = [[array objectAtIndex:1] rectValue];
NSPoint range = [[array objectAtIndex:2] rangeValue];
```

NSNull

An collection cannot have a "hole" or nil value. But you can use **NSNull** as a placeholder.

```
// -null returns the singleton instance of NSNull:
+ (NSNull *)null;


NSMutableArray *array = [[NSMutableArray alloc] init];


// Add the placeholder:
[array addObject:[NSNull null]];


// Because null is a singleton, we can make direct comparisons:
if ([array objectAtIndex:0] == [NSNull null]) {
    // Do something
}
```

Constants

#define and Constants

```
// In Constants.h:
#define VIEW_WIDTH 700
#define VIEW_HEIGHT 350


// Wherever you need the constants:
#import "Constants.h"


// NSMakeRect is a convenience function to make NSRect structures from
// x-coord, y-coord, width, height:
NSRect viewRect = NSMakeRect(0.0, 0.0, VIEW_WIDTH, VIEW_HEIGHT);
```

Global Variables

Objective-C programmers use global variables frequently. Often for defining keys.

```
// In Constants.h:
extern NSString const *kItemCode;

// Elsewhere, but only one place (Constants.m perhaps):
NSString const *kItemCode = @"keyItemCode";

// Wherever the key is used:
#import "Constants.h"

[dict setObject:code forKey:kItemCode];

// kItemCode is better than @"keyItemCode" because if
// you misstype it, the compiler will not warn you
// in the latter case.
```

Enumerations

```
// Numbers are optional (if ommitted, begin at 0 and +1):
typedef enum {
    PhaserStun = 1,     //    1 b
    PhaserSinge = 2,    //   10 b
    PhaserBurn = 4,     //  100 b
    PhaserVaporize = 8  // 1000 b
} PhaserSetting;


// Elsewhere you can use the enum typedef:
- (void)setPhaserSetting:(PhaserSetting)newSetting;
```

Writing Files with NSString and NSData

Write Text to a File

```
// Get a string:
NSString *constitution = @"We the people of The United States ...";


// Dump to a file:
[constitution writeToFile:@"/Users/juampa/Desktop/Constitution.txt"
               atomically:YES
                 encoding:NSUTF8StringEncoding
                    error:NULL];


// Note method signature:
- (BOOL)writeToURL:(NSURL *)url
        atomically:(BOOL)useAuxiliaryFile
          encoding:(NSStringEncoding)enc
             error:(NSError **)error;
```

NSError

Many methods in Cocoa or CocoaTouch return a BOOL to indicate success or failure. The NSError instance passed by reference can hold more information about a failure.

If you pass **NULL** to a method taking an *NSError***, it means "I don't want to know what failed".

```
// If you care:
NSError *error = nil;

[constitution writeToFile:@"/Users/juampa/Desktop/Constitution.txt"
               atomically:YES
                 encoding:NSUTF8StringEncoding
                    error:&error];

if (error) {
    NSLog(@"Something went wrong: %@", [error localizedDescription]);
}
```

Reading Text From a File

```
NSError *error = nil;
NSString *path = @"/Users/juampa/Desktop/Constitution.txt";


NSString *constitution = [[NSString alloc]
                           initWithContentsOfFile:path
                                         ncoding:NSUTF8StringEncoding
                                           error:&error];


if (!constitution) {
    // Handle error
}
```

Writing Raw Data to a File

An **NSData** instance is a container of bytes. You can read and write these raw bytes from/to a file.

```
// Obtain a bunch of bytes.
// In the exercise you will get image data from the internet.
NSData *data = ...;
NSError *error = nil;


BOOL success = [data wtiteToFile:@"/tmp/image.png"
                         options:NSDataWritingAtomic
                           error:&error];


if (!success) {
    // Handle error
}
```

Reading Raw Data From a File

```
NSError *error = nil;


NSData *data = [NSData dataWithContentsOfFile:@"/tmp/image.png"
                                      options:NSdataReadingUncached
                                        error:&error];


if (!data) {
    // Handle error
}
```

Callbacks

The Hollywood Principle

"Don't call us, we'll call you."

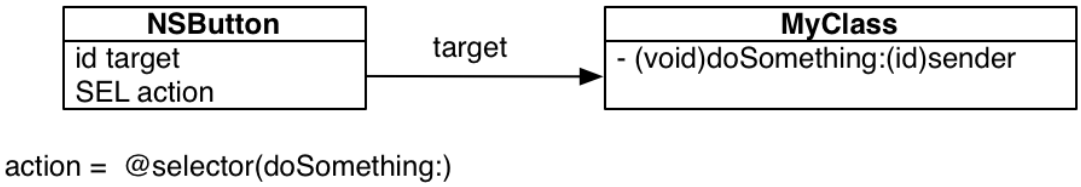
GUI software is not the boss, it is typically waiting for the user to tell it what to do.

Callbacks are methods or functions that are invoked when something happens.

As a Cocoa or CocoaTouch programmer you will deal essentially with three types of callback schemes:

- Target-action
- Delegates (helper objects)
- Notifications

Target-Action



action = @selector(doSomething:)

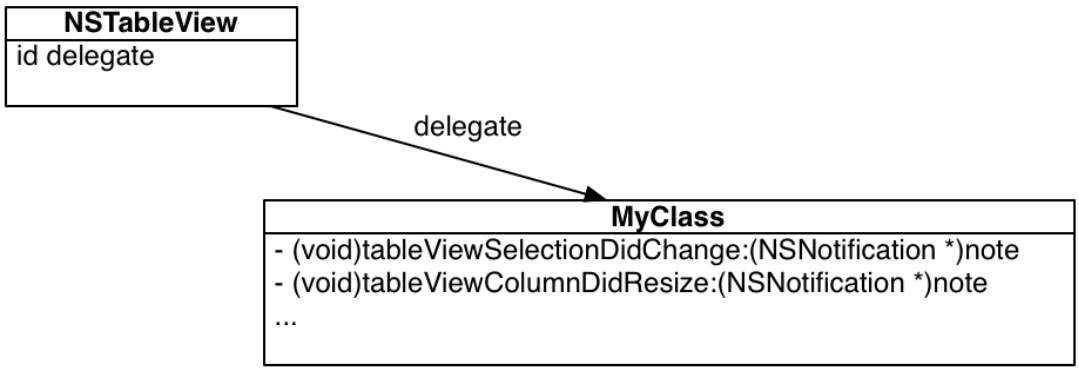
Delegates 1

Im many GUI frameworks, stock controls must be subclassed to attach the response methods.

In Cocoa and CocoaTouch, composition is favored. Custom functionality is attached to a control by setting a delegate. The delegate implements the custom functionality.

Methods that the delegate of a given class *must* or *may* implement are listed in a **protocol**.

Delegates 2



Notifications

NSNotificationCenter serves as a bulletin board.

An object can become an observer, and every time something specific happens, it will be notified.

An object can post a notification. Any observers for that event will be notified.

Observing

```
// To become an observer:
[[NSNotificationCenter defaultCenter]
                addObserver:self
                   selector:@selector(respondToNotification:)
                       name:@"schnitzelpusskrankengesheitmeyer"
                     object:nil];


// This object must implement:
- (void)respondToNitification:(NSNotification *)note;

// The method name is variable but the signature is not.
```

Posting

```
// Post a notification:
[[NSNotificationCenter defaultCenter]
            postNotificationName:@"schnitzelpusskrankengesheitmeyer"
                          object:self];


// At this point, any observers to this event will be notified.
```

Protocols

What Is a Protocol?

A **protocol** is a contract. It is a promise to the compiler that a given class will implement an given **interface**. An interface is a series of methods that provide a certain role or behavior.

Delegates or helper objects are expected to behave in a certain way. A protocol is a way to enforce that behavior.

Protocol Example

```
@protocol EspressoMachineDelegate <NSObject>
// This protocol inherits from NSObject. Delegate class
// must also implement methods required for NSObject.

@required
- (void)machineWillRequireCoffeeGrounds;
- (void)machineWillRequireCleaning;

@optional
- (void)machineDidMakeEspresso;
- (void)machileWillFoamMilk;

@end
```

Conforming to a Protocol

```
@interface MyEspressoDelegate : NSObject
        <EspressoMachineDelegate>
{
    //Ivars
}

// Properties and methods.

@end
```

Compiler Hints

Sometimes you want to tell the compiler an object implements methods listed in a protocol:

```
id <EspressoMachineDelegate> delegate = [controller espressoDelegate];

// Now the compiler knows that delegate implements certain methods:
[delegate machineWillFoamMilk];
```

Property Lists

Property Lists

Property lists are a file format used for serializing data. Allowed data types are converted to text and the entire file is written as XML. The file is readable both by computers and humans.

The allowed data types are:

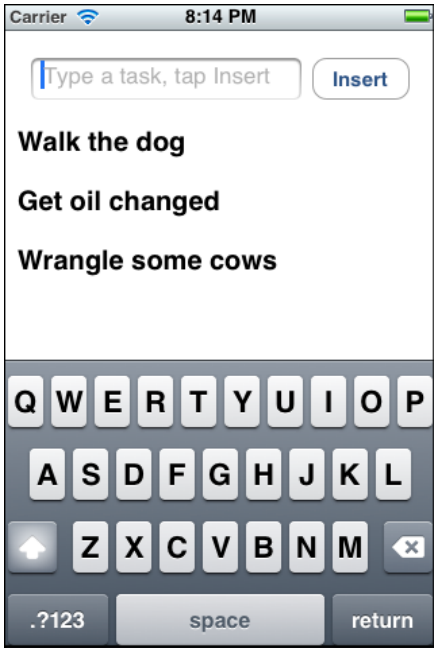
- NSArray
- NSDictionary
- NSString
- NSData
- NSDate
- NSNumber (integer, float or Boolean)

Plist Example

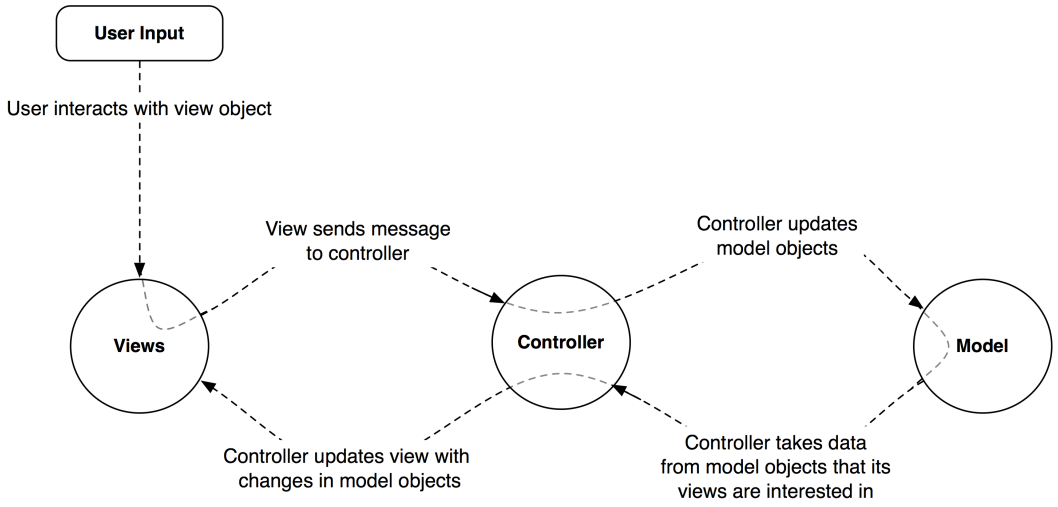
```
<?xml version="1.0" encoding="UTF-8"?>;
<!DOCTYPE plist PUBLIC
"-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">;
<plist version="1.0"></plist>
<array>
    <dict>
        <key>shares</key>
        <integer>200</integer>
        <key>symbol</key>
        <string>AAPL</string>
    </dict>
    <dict>
        <key>shares</key>
        <integer>160</integer>
        <key>symbol</key>
        <string>GOOG</string>
    </dict>
</array>
</plist>
```

Your First iOS Application

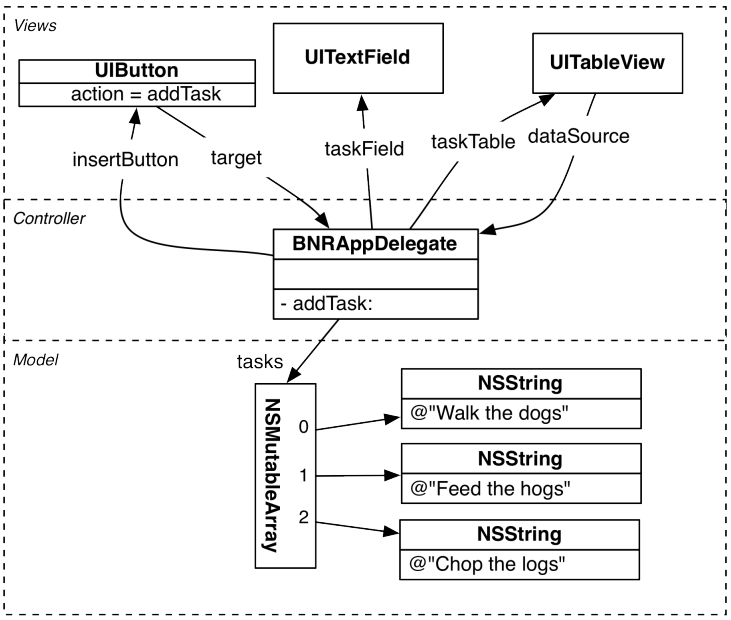
Completed iTahDoodle App



Model-View-Controller



Object Diagram



init

Object Creation

A very old pattern in the Cocoa frameworks is *two-stage object creation*. Objects are created by first allocating memory for them, and then preparing them for use.

```
NSMutableDictionary *dict = [[NSMutableDictionary alloc] init];
```

The alloc method inherited from NSObject zeroes the allocated memory. For that reason, clearing variables in custom initializers is rarely done.

Basic init Methods

```
- (id)init
{
    // The superclass needs to do its part:
    self = [super init];

    // Was super's initialization OK?
    if (self) {
        dog = @"Fido";
        count = 3;
    }

    // Return pointer to new object:
    return self;
}
```

init Safety

When can an initialization go wrong?

- **init** may make optimizations, deallocates original object and creates a new one. The address in self changes.
- An **init** method can actually fail and return nil.

Accessors in init

```
- (id)init
{
    // The superclass needs to do its part:
    self = [super init];

    // Was super's initialization OK?
    if (self) {
        // Mostly a matter of taste, but be careful about
        // side-effects before the object is ready!
        [self setDog:@"Fido"];
        [self setCount:3];
    }

    // Return pointer to new object:
    return self;
}
```

init With Arguments

```
- (id)initWithDogName:(NSString *)name
{
    // The superclass needs to do its part:
    self = [super init];

    // Was super's initialization OK?
    if (self) {
        [self setDog:name];
        [self setCount:3];
    }

    // Return pointer to new object:
    return self;
}

// Use it!
MyClass *obj = [[MyClass alloc] initWithDogName:@"Rex"];
```

Designated Initializers

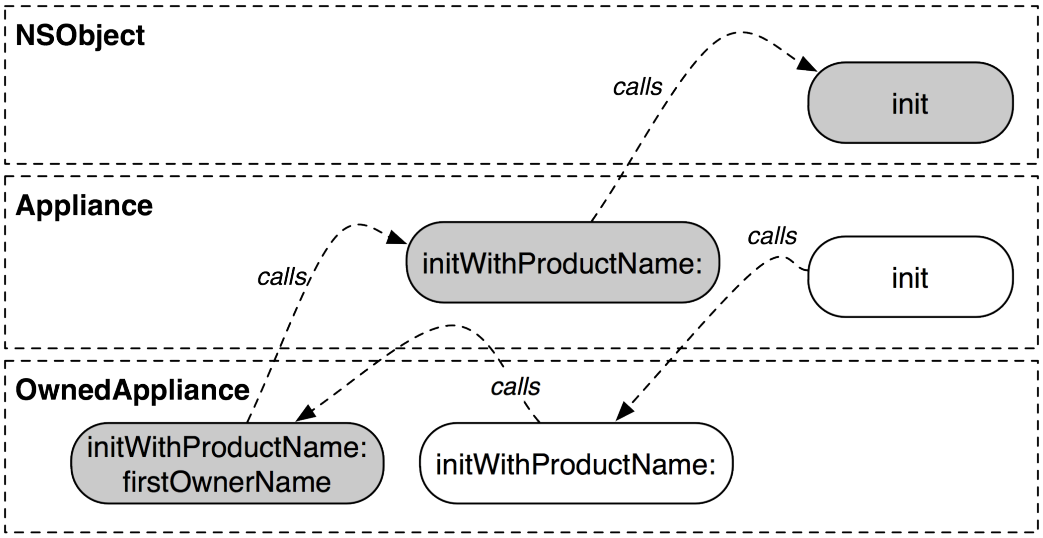
The Rules:

- Only one initializer should do the real work. This is the *designated initializer*. All other initializers should call it directly or indirectly.
- The designated initializer calls the superclass' designated initializer.
- If the designated initializer of your class has a different name than the superclass', you must override the older designated initializer to call the new one.
- Document your initializers carefully!

New Designated Initializer

```
// Designated initializer:
- (id)initWithDogName:(NSString *)name
{
    self = [super init];
    if (self) {
        [self setDog:name];
        [self setCount:3];
    }
    return self;
}

- (id)init
{
    return [self initWithDogName:@"Fido"];
}
```

Initializer Chain



Properties

Property Attributes

```
// Declare a property:
@property (<attribute_list>) <variable_type> <variable_name>;


// Implement the property:
@synthesize <variable_name>;
```

Mutability Attributes

- readonly
- readwrite (default)

```
@property (readonly) float volume;


// This synthesizes only the getter:
@synthesize volume;
```

Lifetime Attributes

- assign (default)
- unsafe_unretained
- strong
- weak
- copy

```
// You are on your own to manage the reference!
// Simple assignment:
@property (unsafe_unretained) Foo* foo;
```

Atomicity Attributes

- nonatomic -- faster
- atomic (default) -- thread-safe

Some Property History - 1

```
// In Foo.h:
@interface Foo : NSObject
{
    Bar *bar;
}
@property (strong, nonatomic) Bar *bar;

@end


// In Foo.m:
@implementation Foo

@synthesize bar;

@end
```

Some Property History - 2

```
// In Foo.h:
@interface Foo : NSObject
{
    // Ivar declaration becomes optional!
}
@property (strong, nonatomic) Bar *bar;

@end


// In Foo.m:
@implementation Foo

@synthesize bar;

@end
```

Some Property History - 3

```
// In Foo.h:
@interface Foo : NSObject
{
    // Ivar declaration becomes optional!
}
@property (strong, nonatomic) Bar *bar;

@end


// In Foo.m:
@implementation Foo

// @synthesize becomes optional as well!

@end
```

What is the Compiler Doing?

```
// In Foo.h:
@interface Foo : NSObject
{
    Bar *_bar;
}
@property (strong, nonatomic) Bar *bar;

@end


// In Foo.m:
@implementation Foo

@synthesize bar = _bar;

@end
```

Key-Value Coding

KVC allows you to access an object's properties by name.

If accessors are available, they are used. Otherwise the ivar is accessed directly.

```
// obj has an ivar named nickname:
[obj setValue:@"Bob" forKey:@"nickname"];
NSString *str = [obj valueForKey:@"nickname"];
```

Categories

Adding Methods to a Class

Wouldn't it be neat if NSString had a <*insert your wish here*> method?

```
// In NSString+VowelCounting.h:
#import <Foundation/Foundation.h>


@interface NSString (VowelCounting)
- (int)vowelCount;
@end


// In NSString+VowelCounting.m:
@implementation NSString (VowelCounting)
- (int)vowelCount
{
    // Will implement in exercise!
}
@end
```

Class Extensions

```
// Want to hide stuff from the header?
// Put it in the implementation!

@interface MyClass ()

- (void)doSomething;
@property (copy) NSMutableString *name;
int count;

@end


@implementation MyClass

// Implement!

@end
```

An Introduction to Blocks

Blocks Concepts

Blocks are:

Chunks of code

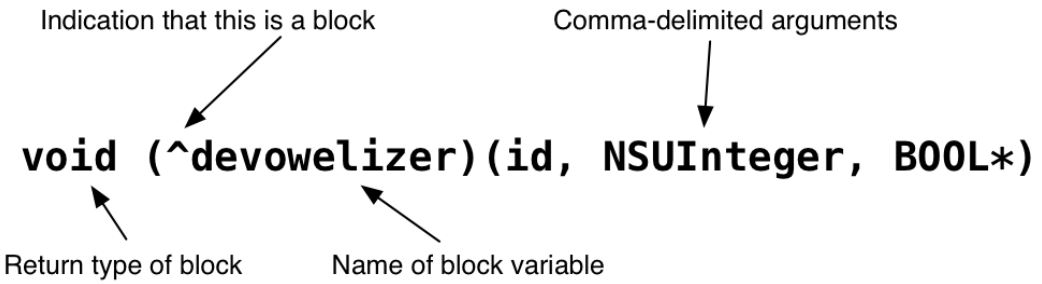
An alternative to callbacks.

Keep conceptually-related code together: seting a callback and callback code can be in the same place.

Closures, they remember variables in scope.

Blocks can be weird, but they are the future. Use them!

Block Variable Declaration



Using Blocks

```
// Variable in scope:
NSColor *favoriteColor = ...;

// Declare a block variable:
void (^operation)(id, NSUInteger, BOOL *);

// Assign a block to the variable:
operation = ^(id obj, NSUInteger i, BOOL *stop) {

    // Actual code here.
    // Can use favoriteColor!!
};

// Use the block later:
[array enumerateObjectsUsingBlock:operation];
```

Anonymous Blocks

```
// variable in scope:
NSColor *favoriteColor = ...;

[array enumerateObjectsUsingBlock:^(id obj, NSUInteger i, BOOL *stop) {
    // Actual code here.
    // Can use favoriteColor!!
}];
```

Switch Statements

Complicated if / else

```
int yeastType = ...;

if (yeastType == 1) {
    makeBread();
} else if (yeastType == 2) {
    makeBeer();
} else if (yeastType == 3) {
    makeWine();
} else {
    makeFuel();
}
```

Better Alternative

```
int yeastType = ...;

switch (yeastType) {
    case 1:
        makeBread();
        break;
    case 2:
        makeBeer();
        break;
    case 3:
        makeWine();
        break;
    default:
        makeFuel();
        break;
}
```