Stanford CS193p

Developing Applications for iOS
Fall 2011

Lessons from Walkthrough

- Xcode 4 - You've learned how to ...
  - Create a new project with a single MVC
  - Show various files in your project (via Navigator or clicking on bars at the top of files)
  - Show and hide the Navigator, Assistant Editor, Console, Object Library, Inspector
  - Drag objects into your view and edit their size, position and object-specific display attributes
  - Ctrl-drag to connect objects in your View to the code in your Controller (outlets and actions)
  - Show connections to outlet `@property`s and action methods (by mouse-over or right click)
  - Get quick help (option click) or full documentation (option double-click) for symbols in your code
  - Run your application in the simulator
  - Click on warning (yellow) and error (red) indicators to see problems in your code
  - Create a new class (like `CalculatorBrain`) using the File menu's New File ... item
  - Create browser-like tabs to organize your viewing of your project

Lessons from Walkthrough

- Objective C - You've learned how to ...
  - Define a class's public `@interface` and private `@implementation` in a `.h` and `.m` file respectively
  - Add a private `@interface` to `.m` file
  - Create a `@property`, both for a primitive type (like `BOOL`) and a pointer (like `NSMutableArray *`)
  - Use `nonatomic` in `@property` declarations
  - Use `strong` or `weak` in `@property` declarations of pointers to objects
  - Use `@synthesize` to create a `@property`'s setter and getter and backing instance variable
  - Use "`= _propertyname`" to choose the name `@synthesize` uses for its backing instance variable
  - For pointers to an object, use either the special type `id` or a static type (e.g. `UIButton *`)
  - Declare and define an Objective C method (e.g. `pushOperand:` or `popOperand`).
  - Declare local variables both of type "pointer to an object" (`id` or static type) and primitive type
  - Invoke an Objective C method (using square bracket `[]` notation)
  - Invoke a setter or getter using dot notation (e.g. `self.operandStack` or `self.display.text`)
  - Lazily instantiate an object by implementing your own `@property` getter (`operandStack` & `brain`)

Lessons from Walkthrough

- Objective C (continued) - You've learned how to ...
  - Wrap a primitive type (like `double`) in an object (using `NSNumber`)
  - Log formatted strings to the console using `NSLog()`
  - Use a "constant" `NSString` in your code using `@""` syntax (e.g. `@"+"`)
  - Add and remove an object from an `NSMutableArray` (the last object anyway☺).
  - Use `alloc` and `init` to create space in the heap for an object (well, you've barely learned this).
  - `#import` the `.h` file of one class into another's (`CalculatorBrain.h` into your Controller)
  - Create a string by asking a string to append another string onto it
  - Create a string with a printf-like format (e.g., `[NSString stringWithFormat:@"%g", result]`)

  Perhaps you've learned even more if you've done Assignment #1!

More on Properties

- Why properties?

  Most importantly, it provides safety and subclassability for instance variables.

  Also provides "valve" for lazy instantiation, UI updating, consistency checking (e.g. `speed < 1`), etc.

- Instance Variables

  It is not required to have an instance variable backing up a `@property` (just skip `@synthesize`).

  Some `@property`s might be "calculated" (usually `readonly`) rather than stored.

  And yes, it is possible to have instance variables without a `@property`, but for now, use `@property`.

- Why dot notation?

  Pretty.

  Makes access to `@property`s stand out from normal method calls.

  Synergy with the syntax for C structs (i.e., the contents of C structs are accessed with dots too).

  Syntactically, C structs look a lot like objects with `@property`s. With 2 big differences:

  1. we can't send messages to C structs (obviously, because they have no methods)
  2. C structs are almost never allocated in the heap (i.e. we don't use pointers to access them)

Dot Notation

- Dot notation

  `@property` access looks just like C struct member access

```
typedef struct {
    float x;
    float y;
} CGPoint;
```

Notice that we capitalize `CGPoint` (just like a class name).
It makes our C struct seem just like an object with `@property`s
(except you can't send any messages to it).

Dot Notation

- Dot notation

  `@property` access looks just like C struct member access

```
typedef struct {
    float x;
    float y;
} CGPoint;

@interface Bomb
@property CGPoint position;
@end

@interface Ship : Vehicle

@property float width;
@property float height;
@property CGPoint center;

- (BOOL)getsHitByBomb:(Bomb *)bomb;

@end
```

Returns whether the passed bomb would hit the receiving Ship.

Dot Notation

- Dot notation

  @property access looks just like C struct member access

```
typedef struct {
    float x;
    float y;
} CGPoint;

@interface Bomb
@property CGPoint position;
@end

@interface Ship : Vehicle

@property float width;
@property float height;
@property CGPoint center;

- (BOOL)getsHitByBomb:(Bomb *)bomb;

@end
```

```
@implementation Ship

@synthesize width, height, center;

- (BOOL)getsHitByBomb:(Bomb *)bomb
{
    float leftEdge = self.center.x - self.width/2;
    float rightEdge = ...;

    return ((bomb.position.x >= leftEdge) &&
            (bomb.position.x <= rightEdge) &&
            (bomb.position.y >= topEdge) &&
            (bomb.position.y <= bottomEdge));
}

@end
```

Dot notation to reference an object's @property.

Dot Notation

- Dot notation

  `@property` access looks just like C struct member access

```
typedef struct {
    float x;
    float y;
} CGPoint;

@interface Bomb
@property CGPoint position;
@end

@interface Ship : Vehicle

@property float width;
@property float height;
@property CGPoint center;

- (BOOL)getsHitByBomb:(Bomb *)bomb;

@end
```

```
@implementation Ship

@synthesize width, height, center;

- (BOOL)getsHitByBomb:(Bomb *)bomb
{
    float leftEdge = self.center.x - self.width/2;
    float rightEdge = ...;

    return ((bomb.position.x >= leftEdge) &&
            (bomb.position.x <= rightEdge) &&
            (bomb.position.y >= topEdge) &&
            (bomb.position.y <= bottomEdge));
}

@end
```

Dot notation to reference an object's `@property`.

Normal C struct dot notation.

strong vs weak

- strong "keep this in the heap until I don't point to it anymore"
  I won't point to it anymore if I set my pointer to it to `nil`.
  Or if I myself am removed from the heap because no one `strong`ly points to me!
- weak "keep this as long as someone else points to it `strong`ly"
  If it gets thrown out of the heap, set my pointer to it to `nil` automatically (if user on iOS 5 only).
- This is not garbage collection!
  It's way better. It's reference counting done automatically for you.
- Finding out that you are about to leave the heap
  A special method, `dealloc`, is called on you when your instance's memory is freed from the heap.
  You will rarely ever have to implement this method. It's "too late" to do much useful here.

```
- (void)dealloc
{
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
```

nil

- The value of an object pointer that does not point to anything

```
id obj = nil;
NSString *hello = nil;
```

- Like "zero" for a primitive type (`int`, `double`, etc.)

  Actually, it's not "like" zero: it is zero.

- All instance variables start out set to zero

  Thus, instance variables that are pointers to objects start out with the value of `nil`.

- Can be implicitly tested in an `if` statement

```
if (obj) { }  // curly braces will execute if obj points to an object
```

- Sending messages to `nil` is (mostly) okay. No code gets executed.

  If the method returns a value, it will return zero.

```
int i = [obj methodWhichReturnsAnInt]; // i will be zero if obj is nil
```

  Be careful if the method returns a C struct. Return value is undefined.

```
CGPoint p = [obj getLocation];  // p will have an undefined value if obj is nil
```

BOOL

- Objective-C's boolean "type" (actually just a `typedef`)

  Can be tested implicitly

```
if (flag) { }
if (!flag) { }
```

  `YES` means "true," `NO` means "false"

  `NO == 0`, `YES` is anything else

```
if (flag == YES) { }
if (flag == NO) { }
if (flag != NO) { }
```

Instance vs. Class Methods

- Starts with a dash

```
- (BOOL)dropBomb:(Bomb *)bomb
             at:(CGPoint)position
           from:(double)altitude;
```

- Starts with a plus sign

```
+ (id) alloc;
+ (Ship *)motherShip;
+ (NSString *)stringWithFormat:…
```

Bomb is a class.
It lives in the heap and we pass around pointers to it.

CGPoint is a C struct, not a class!
It looks like a class name, but notice no * because C structs are passed by value on the stack, not by reference in the heap.

Instance vs. Class Methods

- Starts with a dash

```
- (BOOL)dropBomb:(Bomb *)bomb
             at:(CGPoint)position
           from:(double)altitude;
```

- "Normal" Instance Methods
- Calling syntax

```
[<pointer to instance> method]
Ship *ship = …; // instance of a Ship
destroyed = [ship dropBomb:firecracker
                        at:dropPoint
                      from:300.0];
```

- Starts with a plus sign

```
+ (id) alloc;
+ (Ship *)motherShip;
+ (NSString *)stringWithFormat:…
```

- Creation & Utility Methods
- Calling syntax

```
[Class method]
Ship *ship = [Ship motherShip];
NSString *resultString =
    [NSString stringWithFormat:@"%g", result];
[[ship class] doSomething];
```

This ship is an instance.

doSomething is a class method.

This instance method (called "class") returns a Class.

Instance vs. Class Methods

- Starts with a dash

```
- (BOOL)dropBomb:(Bomb *)bomb
                 at:(CGPoint)position
               from:(double)altitude;
```

- "Normal" Instance Methods
- Calling syntax

```
[<pointer to instance> method]
Ship *ship = …; // instance of a Ship
destroyed = [ship dropBomb:firecracker
                        at:dropPoint
                      from:300.0];
```

- `self`/`super` is calling instance

`self` means "my implementation"
`super` means "my superclass's implementation"

- Starts with a plus sign

```
+ (id) alloc;
+ (Ship *)motherShip;
+ (NSString *)stringWithFormat:…
```

- Creation & Utility Methods
- Calling syntax

```
[Class method]
Ship *ship = [Ship motherShip];
NSString *resultString =
    [NSString stringWithFormat:@"%g", result];
[[ship class] doSomething];
```

- `self`/`super` is this class

`self` means "this class's class methods"
`super` means "this class's superclass's class methods"

Methods

- More Examples

```
- (double)performOperation:(NSString *)operation;
- (NSMutableArray *)operandStack;
- (NSString *)stringByAppendingString:(NSString *)otherString;
- (void)doSomething;
- (void)doSomethingWithThingOne:(Thing *)one
                   andThingTwo:(Thing *)two;
- (NSArray *)collectButterfliesWithSpotCount:(int)spots;
- (NSComparisonResult)compare:(NSString *)aString
                      options:(NSStringCompareOptions)mask
                        range:(NSRange)range;
(In Ship) + (double)yieldForPhotonTorpedoOfType:(PhotonTorpedoType)type;
```

Instantiation

- Asking other objects to create objects for you

  NSString's `- (NSString *)stringByAppendingString:(NSString *)otherString;`

  NSString's & NSArray's `- (id)mutableCopy;`

  NSArray's `- (NSString *)componentsJoinedByString:(NSString *)separator;`

- Not all objects handed out by other objects are newly created

  NSArray's `- (id)lastObject;`

  NSArray's `- (id)objectAtIndex:(int)index;`

  Unless the method has the word "copy" in it, if the object already exists, you get a pointer to it.

  If the object does not already exist (like the 3 examples above), then you're creating.

- Using class methods to create objects

  NSString's `+ (id)stringWithFormat:(NSString *)format, ...`

  UIButton's `+ (id)buttonWithType:(UIButtonType)buttonType;`

  NSMutableArray's `+ (id)arrayWithCapacity:(int)count;`

  NSArray's `+ (id)arrayWithObject:(id)anObject;`

Instantiation

- Allocating and initializing an object from scratch

  Doing this is a two step process: allocation, then initialization.

  Both steps must happen one right after the other (nested one inside the other, in fact).

  Examples:

```
NSMutableArray *stack = [[NSMutableArray alloc] init];
CalculatorBrain *brain = [[CalculatorBrain alloc] init];
```

- Allocating

  Heap allocation for a new object is done by the `NSObject` class method `+ (id)alloc`

  It allocates enough space for all the instance variables (e.g., the ones created by `@synthesize`).

- Initializing

  Classes can have multiple, different initializers (with arguments) in addition to plain `init`.

  If a class can't be fully initialized by plain `init`, it is supposed to raise an exception in `init`.

  `NSObject`'s only initializer is `init`.

Instantiation

- More complicated `init` methods

  If an initialization method has arguments, it should still start with the four letters `init`

  Example: `- (id)initWithFrame:(CGRect)aRect; // initializer for UIView`

  ```
  UIView *myView = [[UIView alloc] initWithFrame:thePerfectFrame];
  ```

- Examples of multiple initializers with different arguments

  From `NSString`:

  ```
  - (id)initWithCharacters:(const unichar *)characters length:(int)length;
  - (id)initWithFormat:(NSString *)format, ...;
  - (id)initWithData:(NSData *)data encoding:(NSStringEncoding)encoding;
  ```

- Classes must designate an initializer for subclassers

  This is the initializer that subclasses must use to initialize themselves in their designated initializer.

- Static typing of initializers

  For subclassing reasons, `init` methods should be typed to return `id` (not statically typed)

  Callers should statically type though, e.g., `MyObject *obj = [[MyObject alloc] init];`

Instantiation

- Creating your own initialization method

  We use a sort of odd-looking construct to ensure that our superclass inited properly.

  Our superclass's designated initializer can return nil if it failed to initialize.

  In that case, our subclass should return nil as well.

  This looks weird because it assigns a value to self, but it's the proper form.

  Here's an example of what it would look like if init (plain) were our designated initializer:

```
@implementation MyObject
- (id)init
{
    self = [super init];  // call our super's designated initializer
    if (self) {
        // initialize our subclass here
    }
    return self;
}
@end
```

Instantiation

- Example: A subclass of `CalculatorBrain` w/convenience initializer

  Imagine that we enhanced `CalculatorBrain` to have a list of "valid operations."

  We'll allow the list to be `nil` which we'll define to mean that all operations are valid.

  It might be nice to have a convenience initializer to set that array of operations.

  We'd want to have a `@property` to set the array of valid operations as well, of course.

  Our designated initializer, though, is still `init` (the one we inherited from `NSObject`).

```
@implementation CalculatorBrain

- (id)initWithValidOperations:(NSArray *)anArray
{
    self = [self init];
    self.validOperations = anArray; // will do nothing if self == nil
    return self;
}

@end
```

Note that we call our own designated initializer on `self`, not `super`!

We might add something to our designated initializer someday and we don't want to have to go back and change all of our convenience initializers too.

Only our designated initializer should call our `super`'s designated initializer.

Dynamic Binding

- All objects are allocated in the heap, so you always use a pointer

  Example …

```
NSString *s = ...;    // "statically" typed
id obj = s;           // not statically typed, but perfectly legal
```

  Never use "`id *`" (that would mean "a pointer to a pointer to an object").

- Decision about code to run on message send happens at runtime

  Not at compile time. None of the decision is made at compile time.

  Static typing (e.g. `NSString *` vs. `id`) is purely an aid to the compiler to help you find bugs.

  If neither the class of the receiving object nor its superclasses implements that method: crash!

- It is legal (and sometimes even good code) to "cast" a pointer

  But we usually do it only after we've used "introspection" to find out more about the object.

  More on introspection in a minute.

```
id obj = ...;
NSString *s = (NSString *)obj;  // dangerous … best know what you are doing
```

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
```

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];
```

No compiler warning.
Perfectly legal since `s` "isa" `Vehicle`.
Normal object-oriented stuff here.

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
```

No compiler warning.
Perfectly legal since s "isa" Vehicle.

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];
```

Compiler warning!

Would not crash at runtime though.
But only because we know `v` is a `Ship`.
Compiler only knows `v` is a `Vehicle`.

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
```

No compiler warning.

The compiler knows that the method `shoot` exists,
so it's not impossible that `obj` might respond to it.
But we have not typed `obj` enough for the compiler to be sure it's wrong.
So no warning.
Might crash at runtime if `obj` is not a `Ship`
(or an object of some other class that implements a `shoot` method).

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];
```

Compiler warning!

Compiler has never heard of this method.
Therefore it's pretty sure `obj` will not respond to it.

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];

NSString *hello = @"hello";
[hello shoot];
```

Compiler warning.
The compiler knows that NSString objects do not respond to shoot.
Guaranteed crash at runtime.

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];

NSString *hello = @"hello";
[hello shoot];
Ship *helloShip = (Ship *)hello;
```

No compiler warning.
We are "casting" here.
The compiler thinks we know what we're doing.

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];

NSString *hello = @"hello";
[hello shoot];
Ship *helloShip = (Ship *)hello;
[helloShip shoot];
```

No compiler warning!

We've forced the compiler to think that the `NSString` is a `Ship`. "All's well," the compiler thinks. Guaranteed crash at runtime.

Object Typing

```
@interface Vehicle
- (void)move;
@end
@interface Ship : Vehicle
- (void)shoot;
@end

Ship *s = [[Ship alloc] init];
[s shoot];
[s move];

Vehicle *v = s;
[v shoot];

id obj = ...;
[obj shoot];
[obj someMethodNameThatNoObjectAnywhereRespondsTo];

NSString *hello = @"hello";
[hello shoot];
Ship *helloShip = (Ship *)hello;
[helloShip shoot];
[(id)hello shoot];
```

No compiler warning!

We've forced the compiler to ignore the object type by "casting" in line. "All's well," the compiler thinks. Guaranteed crash at runtime.

Introspection

- So when do we use `id`? Isn't it always bad?
  No, we might have a collection (e.g. an array) of objects of different classes.
  But we'd have to be sure we know which was which before we sent messages to them.
  How do we do that? Introspection.
- All objects that inherit from `NSObject` know these methods
  `isKindOfClass:` returns whether an object is that kind of class (inheritance included)
  `isMemberOfClass:` returns whether an object is that kind of class (no inheritance)
  `respondsToSelector:` returns whether an object responds to a given method
- Arguments to these methods are a little tricky
  Class testing methods take a `Class`
  You get a `Class` by sending the class method `class` to a class :)

```
if ([obj isKindOfClass:[NSString class]]) {
    NSString *s = [(NSString *)obj stringByAppendingString:@"xyzzy"];
}
```

Introspection

- Method testing methods take a selector (SEL)

  Special `@selector()` directive turns the name of a method into a selector

```
if ([obj respondsToSelector:@selector(shoot)]) {
    [obj shoot];
} else if ([obj respondsToSelector:@selector(shootAt:)]) {
    [obj shootAt:target];
}
```

- SEL is the Objective-C "type" for a selector

```
SEL shootSelector = @selector(shoot);
SEL shootAtSelector = @selector(shootAt:);
SEL moveToSelector = @selector(moveTo:withPenColor:);
```

Introspection

- If you have a SEL, you can ask an object to perform it

  Using the performSelector: or performSelector:withObject: methods in NSObject

```
[obj performSelector:shootSelector];
[obj performSelector:shootAtSelector withObject:coordinate];
```

  Using makeObjectsPerformSelector: methods in NSArray

```
[array makeObjectsPerformSelector:shootSelector]; // cool, huh?
[array makeObjectsPerformSelector:shootAtSelector withObject:target]; // target is an id
```

  In UIButton, - (void)addTarget:(id)anObject action:(SEL)action …;

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

Foundation Framework

- NSObject

  Base class for pretty much every object in the iOS SDK

  Implements introspection methods, etc.

  `- (NSString *)description` is a useful method to override (it's `%@` in `NSLog()`).

```
- (id)copy;          // not all objects implement mechanism (raises exception if not)
- (id)mutableCopy;   // not all objects implement mechanism (raises exception if not)
```

Foundation Framework

- NSString

  International (any language) strings using Unicode.

  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.

  Usual usage pattern is to send a message to an `NSString` and it will return you a new one.

```
self.display.text = [self.display.text stringByAppendingString:digit];
self.display.text = [NSString stringWithFormat:@"%g", brain.operand];  // class method
```

  Tons of utility functions available (case conversion, URLs, substrings, type conversions, etc.).

- NSMutableString

  Mutable version of `NSString`. Somewhat rarely used.

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

```
NSMutableString *ms = [[NSMutableString alloc] initWithString:@"0."];
NSMutableString *ms = [NSMutableString stringWithString:@"0."]; // inherited from NSString
[ms appendString:digit];
```

Foundation Framework

- **NSNumber**

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

  ```
  NSNumber *num = [NSNumber numberWithInt:36];
  float f = [num floatValue]; // would return 36 as a float (i.e. will convert types)
  ```

  Useful when you want to put these primitive types in a collection (e.g. `NSArray` or `NSDictionary`).

- **NSValue**

  Generic object wrapper for other non-object data types.

  ```
  CGPoint point = CGPointMake(25.0, 15.0); // CGPoint is a C struct
  NSValue *pointObject = [NSValue valueWithCGPoint:point];
  ```

- **NSData**

  "Bag of bits." Used to save/restore/transmit data throughout the iOS SDK.

- **NSDate**

  Used to find out the time right now or to store past or future times/dates.

  See also `NSCalendar`, `NSDateFormatter`, `NSDateComponents`.

Foundation Framework

- NSArray

Ordered collection of objects.

Immutable. That's right, once you create the array, you cannot add or remove objects.

```
+ (id)arrayWithObjects:(id)firstObject, ...; // nil-terminated arguments
NSArray *primaryColors = [NSArray arrayWithObjects:@"red", @"yellow", @"blue", nil];
+ (id)arrayWithObject:(id)soleObjectInTheArray; // more useful than you might think!

- (int)count;
- (id)objectAtIndex:(int)index;
- (id)lastObject;  // returns nil (doesn't crash) if there are no objects in the array

- (NSArray *)sortedArrayUsingSelector:(SEL)aSelector;
- (void)makeObjectsPerformSelector:(SEL)aSelector withObject:(id)selectorArgument;
- (NSString *)componentsJoinedByString:(NSString *)separator;

- (BOOL)containsObject:(id)anObject; // could be slow, think about NSOrderedSet
```

Foundation Framework

- NSMutableArray

  Mutable version of NSArray.

```
+ (id)arrayWithCapacity:(int)initialSpace; // initialSpace is a performance hint only
+ (id)array;

- (void)addObject:(id)anObject; // at the end of the array
- (void)insertObject:(id)anObject atIndex:(int)index;
- (void)removeObjectAtIndex:(int)index;
- (void)removeLastObject;

- (id)copy; // returns an NSArray (i.e. immutable copy); and NSArray implements mutableCopy
```

  Don't forget that NSMutableArray inherits all of NSArray's methods
  (e.g. count, objectAtIndex:, etc.)

Foundation Framework

- NSDictionary

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

```
+ (id)dictionaryWithObjects:(NSArray *)values forKeys:(NSArray *)keys;
+ (id)dictionaryWithObjectsAndKeys:(id)firstObject, ...;

NSDictionary *base = [NSDictionary dictionaryWithObjectsAndKeys:
                          [NSNumber numberWithInt:2], @“binary”,
                          [NSNumber numberWithInt:16], @“hexadecimal”, nil];
- (int)count;
- (id)objectForKey:(id)key;
- (NSArray *)allKeys;
- (NSArray *)allValues;
```

Keys are sent `- (NSUInteger)hash` & `- (BOOL)isEqual:(NSObject *)obj` for unique identification.
NSObject returns the object's pointer as its `hash` and `isEqual:` only if the pointers are equal.
Keys are very often `NSString`s (they `hash` based on contents and `isEqual:` if characters match).

Foundation Framework

- NSMutableDictionary

  Mutable version of NSDictionary.

```
+ (id)dictionary; // creates an empty dictionary (don't forget it inherits + methods from super)

- (void)setObject:(id)anObject forKey:(id)key;
- (void)removeObjectForKey:(id)key;
- (void)removeAllObjects;
- (void)addEntriesFromDictionary:(NSDictionary *)otherDictionary;
```

Foundation Framework

- NSSet

  Immutable, unordered collection of distinct objects.

  Can't contain multiple "equal" objects at the same time (for that, use NSCountedSet).

```
+ (id)setWithObjects:(id)firstObject, ...;
+ (id)setWithArray:(NSArray *)anArray;
- (int)count;
- (BOOL)containsObject:(id)anObject;
- (id)anyObject;
- (void)makeObjectsPerformSelector:(SEL)aSelector;
```

- NSMutableSet

  Mutable version of NSSet.

```
- (void)addObject:(id)anObject; // does nothing if object that isEqual:anObject is already in
- (void)removeObject:(id)anObject;
- (void)unionSet:(NSSet *)otherSet;
- (void)minusSet:(NSSet *)otherSet;
- (void)intersectSet:(NSSet *)otherSet;
```

Foundation Framework

- NSOrderedSet

  Immutable ordered collection of distinct objects.

  Sort of a hybrid between an array and a set.

  Faster to check "contains" than an array, but can't store an (isEqual:) object multiple times.

  Not a subclass of NSSet! But contains most of its methods plus …

  - (int)indexOfObject:(id)anObject;
  - (id)objectAtIndex:(int)anIndex;
  - (id)firstObject; and - (id)lastObject;
  - (NSArray *)array;
  - (NSSet *)set;

- NSMutableOrderedSet

  Mutable version of NSOrderedSet.

  - (void)insertObject:(id)anObject atIndex:(int)anIndex;
  - (void)removeObject:(id)anObject;
  - (void)setObject:(id)anObject atIndex:(int)anIndex;

Enumeration

- Looping through members of a collection in an efficient manner

  Language support using `for-in`.

  Example: `NSArray` of `NSString` objects

```
NSArray *myArray = ...;
for (NSString *string in myArray) { // no way for compiler to know what myArray contains
    double value = [string doubleValue]; // crash here if string is not an NSString
}
```

  Example: `NSSet` of `id` (could just as easily be an `NSArray` of `id`)

```
NSSet *mySet = ...;
for (id obj in mySet) {
   // do something with obj, but make sure you don't send it a message it does not respond to
   if ([obj isKindOfClass:[NSString class]]) {
      // send NSString messages to obj with impunity
   }
}
```

Enumeration

- Looping through the keys or values of a dictionary

  Example:

```
NSDictionary *myDictionary = ...;
for (id key in myDictionary) {
    // do something with key here
    id value = [myDictionary objectForKey:key];
    // do something with value here
}
```

Property List

- The term "Property List" just means a collection of collections
  Specifically, it is any graph of objects containing only the following classes:
  `NSArray`, `NSDictionary`, `NSNumber`, `NSString`, `NSDate`, `NSData`
- An `NSArray` is a Property List if all its members are too
  So an `NSArray` of `NSString` is a Property List.
  So is an `NSArray` of `NSArray` as long as those `NSArray`'s members are Property Lists.
- An `NSDictionary` is one only if all keys and values are too
  An `NSArray` of `NSDictionary`s whose keys are `NSString`s and values are `NSNumber`s is one.
- Why define this term?
  Because the SDK has a number of methods which operate on Property Lists.
  Usually to read them from somewhere or write them out to somewhere.
  `[plist writeToFile:(NSString *)path atomically:(BOOL)]; // plist` is `NSArray` or `NSDictionary`

Other Foundation

- NSUserDefaults

  Lightweight storage of Property Lists.

  It's basically an NSDictionary that persists between launches of your application.

  Not a full-on database, so only store small things like user preferences.

  Read and write via a shared instance obtained via class method standardUserDefaults

```
[[NSUserDefaults standardUserDefaults] setArray:rvArray forKey:@"RecentlyViewed"];
```

  Sample methods:

```
- (void)setDouble:(double)aDouble forKey:(NSString *)key;
- (NSInteger)integerForKey:(NSString *)key;  // NSInteger is a typedef to 32 or 64 bit int
- (void)setObject:(id)obj forKey:(NSString *)key; // obj must be a Property List
- (NSArray *)arrayForKey:(NSString *)key; // will return nil if value for key is not NSArray
```

  Always remember to write the defaults out after each batch of changes!

```
[[NSUserDefaults standardUserDefaults] synchronize];
```