Stanford CS193p

Developing Applications for iPhone 4, iPod Touch, & iPad

Fall 2010

Today

- Objective-C
  - Methods (Class and Instance)
  - Instance Variables
  - Properties
  - Dynamic Binding
  - Introspection
  - `nil` and `BOOL`
- Foundation Framework (time permitting)
  - NSObject, NSString, NSMutableString
  - NSNumber, NSValue, NSData, NSDate
  - NSArray, NSDictionary, NSSet
  - Enumeration
  - Property Lists

Method Syntax

```
- (NSArray *)shipsAtPoint:(CGPoint)bombLocation withDamage:(BOOL)damaged;
```

Dash for instance method.
Plus for class method.
Will explain difference in a moment.

Method Syntax

```
- (NSArray *)shipsAtPoint:(CGPoint)bombLocation withDamage:(BOOL)damaged;
```

Return type in parentheses

Method Syntax

```
- (NSArray *)shipsAtPoint:(CGPoint)bombLocation withDamage:(BOOL)damaged;
```

First part of method name.
Full name is `shipsAtPoint:withDamage:`

Second part of method name.

Method Syntax

```
- (NSArray *)shipsAtPoint:(CGPoint)bombLocation withDamage:(BOOL)damaged;
```

Type of first argument in parentheses.
This one happens to be a C struct.

Type of second argument in parentheses.
This one is a BOOL (boolean value).

Method Syntax

```
- (NSArray *)shipsAtPoint:(CGPoint)bombLocation withDamage:(BOOL)damaged;
```

Name of first argument.
Use it like a local variable inside method definition.

Name of second argument.

Method Syntax

```
- (NSArray *)shipsAtPoint:(CGPoint)bombLocation withDamage:(BOOL)damaged;
```

Line up colons when there are lots of arguments (or argument names are long).

```
- (void)splitViewController:(UISplitViewController*)svc
     willHideViewController:(UIViewController *)aViewController
          withBarButtonItem:(UIBarButtonItem *)barButtonItem
       forPopoverController:(UIPopoverController *)popoverController;
```

Use `IBAction` (same as `void`) to alert Interface Builder of an action.

```
- (IBAction)digitPressed:(UIButton *)sender;
- (IBAction)digitPressed:(id)sender;
- (IBAction)digitPressed:sender;     // same as (id)sender version
- (IBAction)digitPressed;
```

Instance Methods

- Starts with a dash

```
- (IBAction)digitPressed:(UIButton *)sender;
```

- "Normal" methods you are used to
- Can access instance variables inside as if they were locals
- Can send messages to `self` and `super` inside

  Both dispatch the message to the calling object, but use different implementations

  If a superclass of yours calls a method on `self`, it will your implementation (if one exists)

- Example calling syntax:

```
BOOL destroyed = [ship dropBomb:bombType at:dropPoint from:height];
```

Class Methods

- Starts with a plus. Used for allocation, singletons, utilities

```
+ (id)alloc;                    // makes space for an object of the receiver's class (always pair w/init)
+ (id)motherShip;               // returns the one and only, shared (singleton) mother ship instance
+ (int)turretsOnShipOfSize:(int)shipSize; // informational utility method
```

- Can not access instance variables inside
- Messages to `self` and `super` mean something a little different

  Both invoke only other class methods. Inheritance does work.

- Example calling syntax (a little different from instance methods)

```
CalculatorBrain *brain = [[CalculatorBrain alloc] init];
Ship *theMotherShip = [Ship motherShip];
Ship *newShip = [Ship shipWithTurrentCount:5];
int turretsOnMediumSizedShip = [Ship turretsOnShipOfSize:4];
```

Instance Variables

- Scope

  By default, instance variables are `@protected` (only the class and subclasses can access).

  Can be marked `@private` (only the class can access) or `@public` (anyone can access).

- Scope syntax

```
@interface MyObject : NSObject
{
    int foo;
@private
    int eye;
@protected
    int bar;
@public
    int forum;
    int apology;
@private
    int jet;
}
```

Protected: `foo` & `bar`

Private: `eye` & `jet`

Public: `forum` & `apology`

Properties

- Forget everything on the previous slide!

  Mark all of your instance variables `@private`.

  Use `@property` and "dot notation" to access instance variables.

- Create methods to set/get an instance variable's value

```
@interface MyObject : NSObject
{
@private
    int eye;
}
- (int)eye;
- (void)setEye:(int)anInt;
@end
```

- Now anyone can access your instance variable using "dot notation"

```
someObject.eye = newEyeValue;    // set the instance variable
int eyeValue = someObject.eye;   // get the instance variable's current value
```

Properties

- Forget everything on the previous slide!
  Mark all of your instance variables `@private`.
  Use `@property` and "dot notation" to access instance variables.
- Create methods to set/get an instance variable's value

```
@interface MyObject : NSObject
{
@private
    int eye;
}
- (int)eye;
- (void)setEye:(int)anInt;
@end
```

Note the capitalization.
Instance variables almost always start with lower case.
In any case, the letter after "set" MUST be capitalized.

Otherwise dot notation will not work.

- Now anyone can access your instance variable using "dot notation"

```
someObject.eye = newEyeValue;    // set the instance variable
int eyeValue = someObject.eye;   // get the instance variable's current value
```

Properties

- @property

  You can get the compiler to generate set/get method declarations with @property directive

```
@interface MyObject : NSObject
{
@private
    int eye;
}
@property int eye;
- (int)eye;
- (void)setEye:(int)anInt;
@end
```

Properties

- @property

  You can get the compiler to generate set/get method declarations with @property directive

```
@interface MyObject : NSObject
{
@private
    int eye;
}
@property int eye;
@end
```

Properties

- @property

  You can get the compiler to generate set/get method declarations with @property directive

```
@interface MyObject : NSObject
{
@private
    int eye;
}
@property int eye;
@end
```

- If you use the readonly keyword, only the getter will be declared

```
@property (readonly) int eye;   // does not declare a setEye: method
```

Properties

- An @property does not have to match an instance variable name

  For example …

```
@interface MyObject : NSObject
{
@private
    int p_eye;
}
@property int eye;
@end
```

- In fact, you do not even have to have a matching variable at all

  The following is perfectly legal …

```
@interface MyObject : NSObject
{
}
@property int eye;
@end
```

Properties

- But whatever you declare, you must then implement

  For example, consider the following header (.h) file:

```
@interface MyObject : NSObject
{
@private
    int eye;
}
@property int eye;
@end
```

  The corresponding implementation (.m) file might look like this:

```
@implementation MyObject

- (int)eye {
    return eye;
}
- (void)setEye:(int)anInt {
    eye = anInt;
}

@end
```

Properties

- Or consider the case where the variable name is different

  Header (.h) file:

```
@interface MyObject : NSObject
{
@private
    int p_eye;
}
@property int eye;
@end
```

  Corresponding implementation (.m) file:

```
@implementation MyObject

- (int)eye {
    return p_eye;
}
- (void)setEye:(int)anInt {
    p_eye = anInt;
}

@end
```

Properties

- Or how about the "no corresponding variable" case?

  Header (.h) file:

```
@interface MyObject : NSObject
{
}
@property (readonly) int eye;
@end
```

  Implementation (.m) file:

```
@implementation MyObject

- (int)eye
{
    return <some calculated value for eye>;
}

@end
```

Properties

- Let the compiler help you with implementation using `@synthesize`!

Header (.h) file:

```
@interface MyObject : NSObject
{
@private
    int eye;
}
@property int eye;
@end
```

Implementation (.m) file:

```
@implementation MyObject
@synthesize eye;
- (int)eye {
    return eye;
}
- (void)setEye:(int)anInt {
    eye = anInt;
}
@end
```

Properties

- Let the compiler help you with implementation using `@synthesize`!

  Header (`.h`) file:

```
@interface MyObject : NSObject
{
@private
    int eye;
}
@property int eye;
@end
```

  Implementation (`.m`) file:

```
@implementation MyObject
@synthesize eye;
@end
```

Properties

- Let the compiler help you with implementation using `@synthesize`!

  Header (`.h`) file:

```
@interface MyObject : NSObject
{
@private
    int eye;
}
@property int eye;
@end
```

  Implementation (`.m`) file:

```
@implementation MyObject
@synthesize eye;
@end
```

- You can even get `@synthesize` to use a different variable

```
@synthesize eye = p_eye;
```

Properties

- If you use `@synthesize`, you can still implement one or the other

```
@implementation MyObject
@synthesize eye;
- (void)setEye:(int)anInt {
    if (anInt > 0) eye = anInt;
}
@end
```

The method `- (int)eye` will still be implemented for you by `@synthesize`

Your implementation of `setEye:` is the one that will count

If you implemented both the setter and the getter, the `@synthesize` would be ignored

Properties

- It's common to use dot notation to access ivars inside your class

  It's not the same as referencing the instance variable directly.

  For example, if `eye` is an instance variable …

  ```
  int x = eye;
  ```

  … inside a method is not the same as …

  ```
  int x = self.eye;
  ```

  The latter calls the getter method (which is usually what you want for subclassability).

- But occasionally things can go terribly wrong!

  What's wrong with the following code?

  ```
  - (void)setEye:(int)anInt
  {
      self.eye = anInt;
  }
  ```

  Infinite loop. Can happen with the getter too …

  ```
  - (int)eye { if (self.eye > 0) { return eye; } else { return -1; } }
  ```

Properties

- Why properties?

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

  But the syntax also makes code look more consistent with C structs.

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

@interface Bomb
@property Point position;
@end

@interface Ship : Vehicle {
    float width, height;
    Point center;
}
@property float width;
@property float height;
@property Point center;
- (BOOL)getsHitByBomb:(Bomb *)bomb;
@end
```

Notice that we capitalize `Point` (just like a class name).
It makes our C struct seem just like an object

Properties

- Why properties?

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

  But the syntax also makes code look more consistent with C structs.

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

@interface Bomb
@property Point position;
@end

@interface Ship : Vehicle {
    float width, height;
    Point center;
}
@property float width;
@property float height;
@property Point center;
- (BOOL)getsHitByBomb:(Bomb *)bomb;
@end
```

Instance variables may or may not exist here.
Remember that `@property` is just declaring the property.
`Bomb` would still have to implement setter and getter
(could use `@synthesize`, of course).

Properties

- Why properties?

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

  But the syntax also makes code look more consistent with C structs.

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

@interface Bomb
@property Point position;
@end

@interface Ship : Vehicle {
    float width, height;
    Point center;
}
@property float width;
@property float height;
@property Point center;
- (BOOL)getsHitByBomb:(Bomb *)bomb;
@end
```

Returns whether the passed bomb would hit the receiving Ship.

Properties

- Why properties?

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

  But the syntax also makes code look more consistent with C structs.

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

@interface Bomb
@property Point position;
@end

@interface Ship : Vehicle {
    float width, height;
    Point center;
}
@property float width;
@property float height;
@property Point 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
```

Notice access to instance variable using property of `self` instead of directly.

Properties

- Why properties?

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

  But the syntax also makes code look more consistent with C structs.

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

@interface Bomb
@property Point position;
@end

@interface Ship : Vehicle {
    float width, height;
    Point center;
}
@property float width;
@property float height;
@property Point 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.

Properties

- Why properties?

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

  But the syntax also makes code look more consistent with C structs.

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

@interface Bomb
@property Point position;
@end

@interface Ship : Vehicle {
    float width, height;
    Point center;
}
@property float width;
@property float height;
@property Point center;
- (BOOL)getsHitByBomb:(Bomb *)bomb;
@end
```

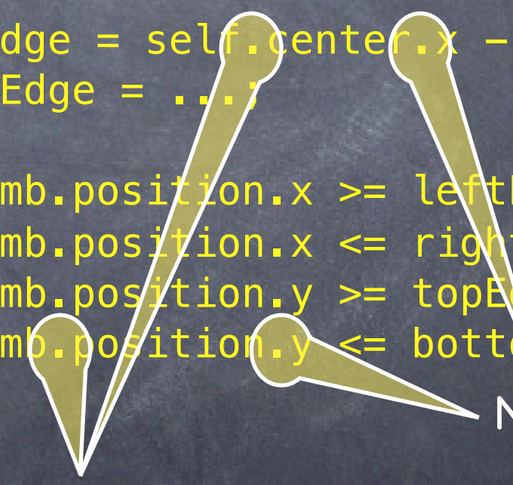
```
@implementation Ship

@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.

Private Properties

- Do all `@property`s have to be public?

  No. It is possible to declare a "private interface" to your class inside your implementation file.

  Example (this is all in `MyObject`'s `.m` file):

```
@interface MyObject()
@property double myEyesOnly;
@end

@implementation MyObject
@synthesize eye, myEyesOnly;
@end
```

  This is the "magic" to declare your private stuff. You can put properties and methods here, but not more instance variables.

  The property `myEyesOnly` can only be set/get via `self.myEyesOnly` since it is private.

Properties

- There's more to think about when a `@property` is an object
  We'll postpone that discussion to later on when we talk about memory management

Dynamic Binding

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

  Examples …

```
NSString *s = ...;    // "static" typed
id obj = s;
```

  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];
[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.

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];
}
```

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

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

  Target/action uses this, e.g. `[button addTarget:self action:@selector(digitPressed:)]`

- 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:moveToSelector withObject:coordinate];
```

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.

- `NSObject` sets all its instance variables 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) { }
```

Foundation Framework

- NSObject

  Base class for pretty much every object in the iOS SDK
  Implements memory management primitives (more on this later)
  Implements introspection methods
  `- (NSString *)description` is a useful method to override (it's `%@` in `NSLog()`).

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.

```
[display setText:[[display text] stringByAppendingString:digit]];
display.text = [display.text stringByAppendingString:digit];  // same but with properties
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`.

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

```
NSMutableString *mutString = [[NSMutableString alloc] initWithString:@"0."];
[mutString appendString:digit];
```

Foundation Framework

- NSNumber

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

```
NSNumber *num = [NSNumber numberWithFloat:36.5];
float f = [num floatValue];
```

  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);
NSValue *val = [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, you cannot add or remove objects to it once it's created.

  Important methods:

```
+ (id)arrayWithObjects:(id)firstObject, ...;
- (int)count;
- (id)objectAtIndex:(int)index;
- (void)makeObjectsPerformSelector:(SEL)aSelector;
- (NSArray *)sortedArrayUsingSelector:(SEL)aSelector;
- (id)lastObject;  // returns nil if there are no objects in the array (convenient)
```

- NSMutableArray

  Mutable version of `NSArray`.

```
- (void)addObject:(id)anObject;
- (void)insertObject:(id)anObject atIndex:(int)index;
- (void)removeObjectAtIndex:(int)index;
```

Foundation Framework

- NSDictionary

  Hash table. Look up objects using a key to get a value.

  Immutable. That's right, you cannot add or remove objects to it once it's created.

  Keys are objects which must implement `- (NSUInteger)hash` & `- (BOOL)isEqual:(NSObject *)obj`

  Keys are usually `NSString` objects.

  Important methods:

  ```
  - (int)count;
  - (id)objectForKey:(id)key;
  - (NSArray *)allKeys;
  - (NSArray *)allValues;
  ```

- NSMutableDictionary

  Mutable version of `NSDictionary`.

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

Foundation Framework

- NSSet

  Unordered collection of objects.

  Immutable. That's right, you cannot add or remove objects to it once it's created.

  Important methods:

```
- (int)count;
- (BOOL)containsObject:(id)anObject;
- (id)anyObject;
- (void)makeObjectsPerformSelector:(SEL)aSelector;
- (id)member:(id)anObject;  // uses isEqual: and returns a matching object (if any)
```

- NSMutableSet

  Mutable version of NSSet.

```
- (void)addObject:(id)anObject;
- (void)removeObject:(id)anObject;
- (void)unionSet:(NSSet *)otherSet;
- (void)minusSet:(NSSet *)otherSet;
- (void)intersectSet:(NSSet *)otherSet;
```

Enumeration

- Looping through members of a collection in an efficient manner

  Language support using `for-in` (similar to Java)

  Example: `NSArray` of `NSString` objects

```
NSArray *myArray = ...;
for (NSString *string in myArray) {
    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`

Next Time

- More Foundation
- Object Allocation/Initialization
- Memory Management
- Demo