Developing applications for iOS

Lecture 3:
Objective-C in Depth

Radu Ionescu
raducu.ionescu@gmail.com
Faculty of Mathematics and Computer Science
University of Bucharest

Content

- More on Dot Notation
- Instance Methods and Class Methods
- Object Typing
- Introspection
- Foundation Framework

Lessons from Labs 1 and 2

In Xcode you have learned how to ...

- Create a new project.
- 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 `@propertys` and action methods (by mouse-over or right click).

Lessons from Labs 1 and 2

In Xcode you have learned how to ...

- 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 Labs 1 and 2

In Objective-C you have 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.

Lessons from Labs 1 and 2

In Objective-C you have learned how to ...

- For pointers to an object, use either the special type `id` or a static type (e.g. `UIButton *`).
- Define an Objective C method (e.g. `presentGreeting:`).
- Declare local variables 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.display`).
- Lazily instantiate an object by implementing your own `@property` getter (`operandStack` and `brain`).
- Log formatted strings to the console using `NSLog()`.
- Wrap a primitive type (like `double`) in an object (using `NSNumber`).

Lessons from Labs 1 and 2

In Objective-C you have learned how to ...

- Use a “constant” `NSString` in your code using `@“”` syntax (e.g. `@“Hello”`).
- 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 have 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 (for example, `[NSString stringWithFormat:@“%f”, result]`)

More on Properties

Why properties?

- Most importantly, it provides safety and subclassability for instance variables.
- Also provides a good way 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 `@propertys` 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`.

More on Properties

Why dot notation?

- It's pretty.
- Makes access to `@propertys` 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 `@propertys`. 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

- `@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 `@propertys` (except you can't send any messages to it).

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

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

```
@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
```

Reference an object's `@property`.

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

Dot Notation

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

```
@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
```

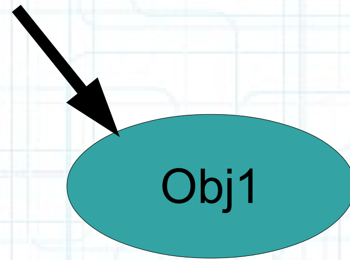
Normal C struct dot notation.

Normal C struct dot notation.

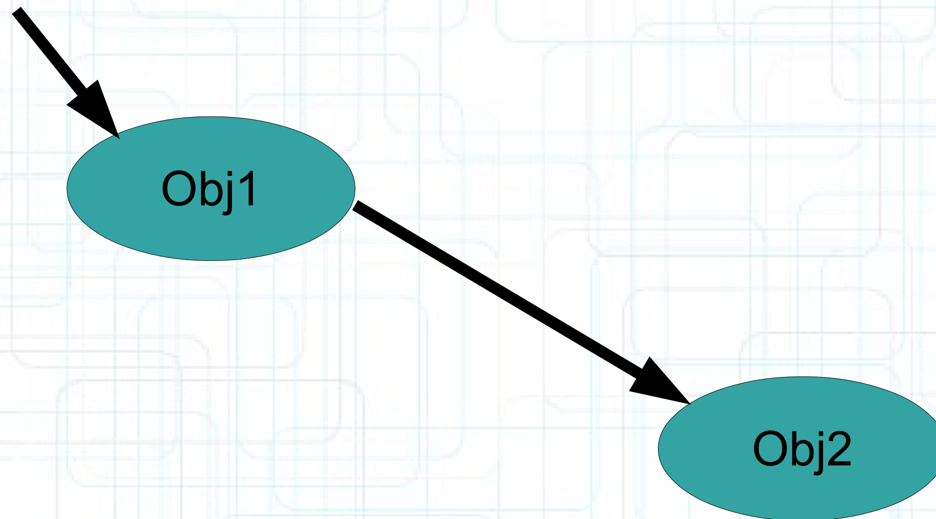
strong vs weak

- `strong` means "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` means "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 (ARC).
  It happens at compile time not at run time!

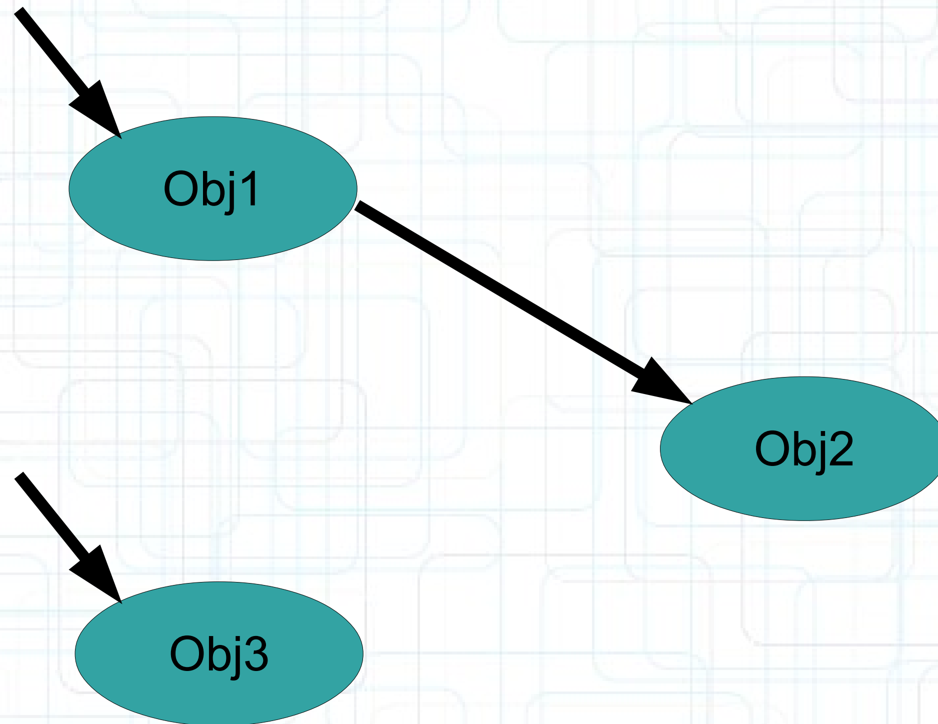
strong vs weak



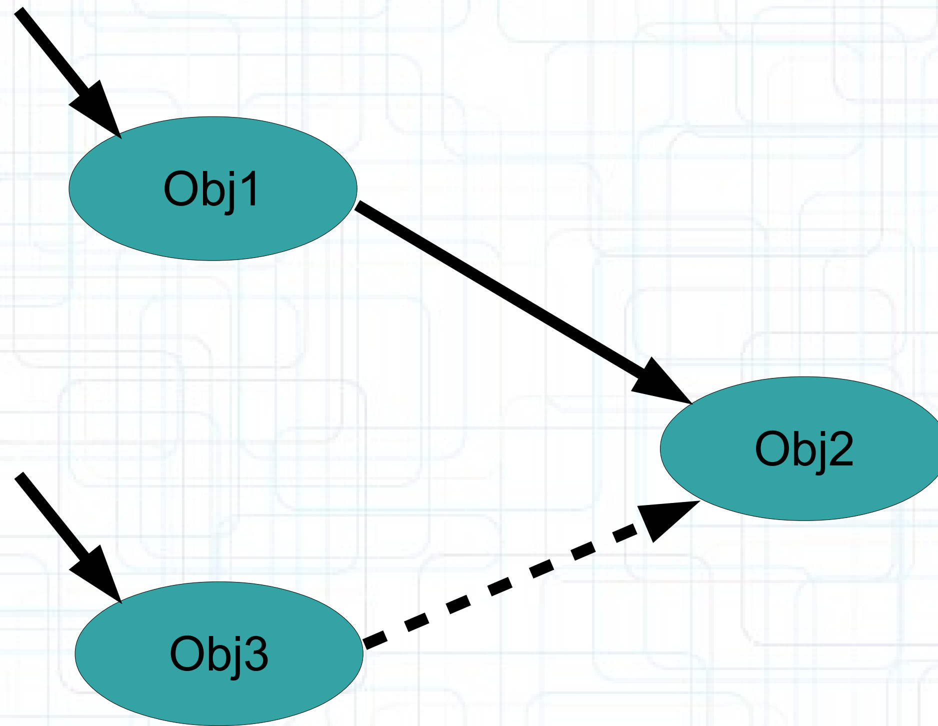
strong vs weak



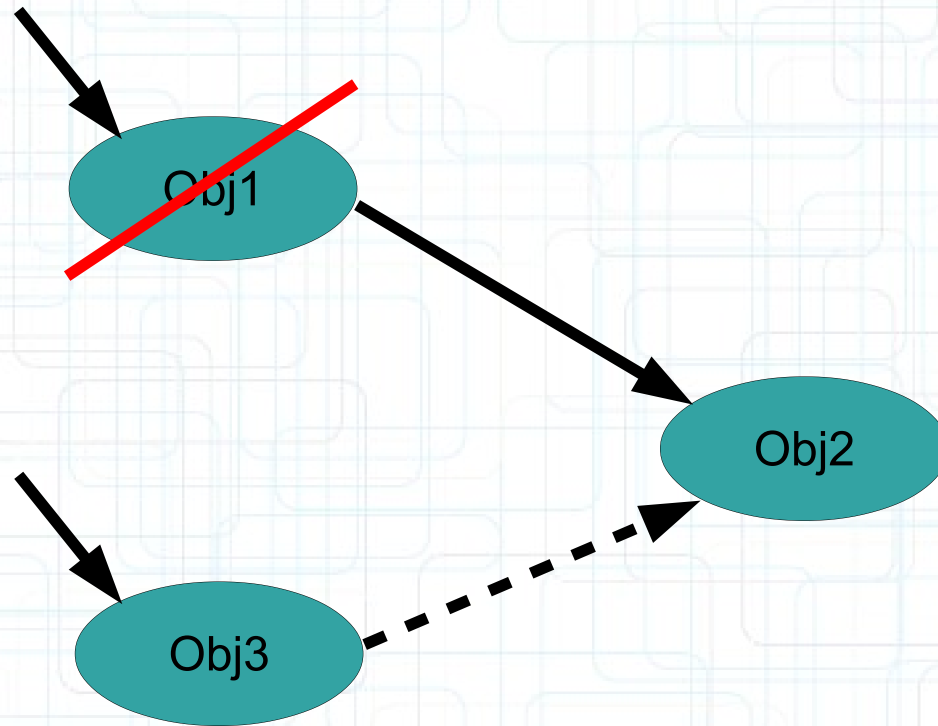
strong vs weak



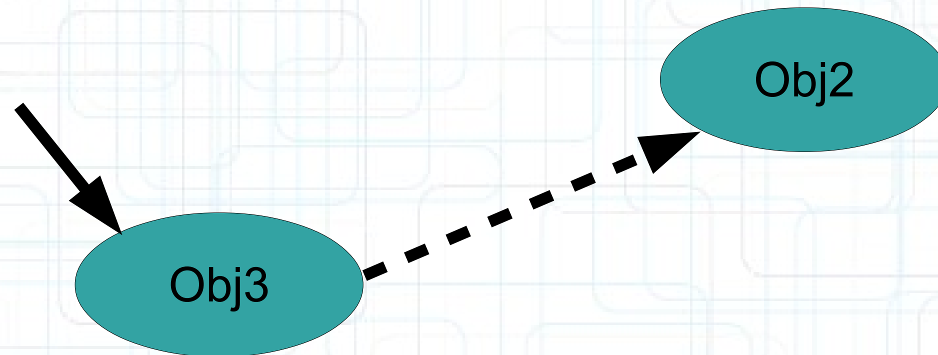
strong vs weak



strong vs weak

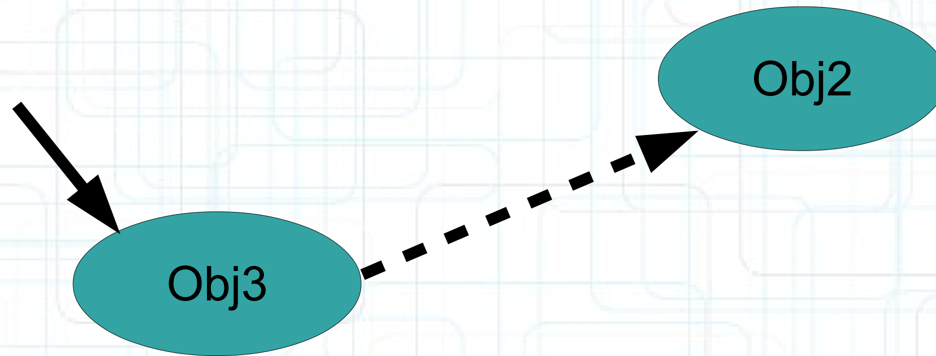


strong vs weak

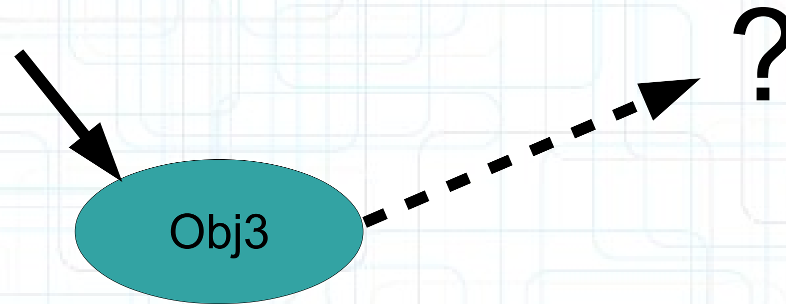


strong vs weak

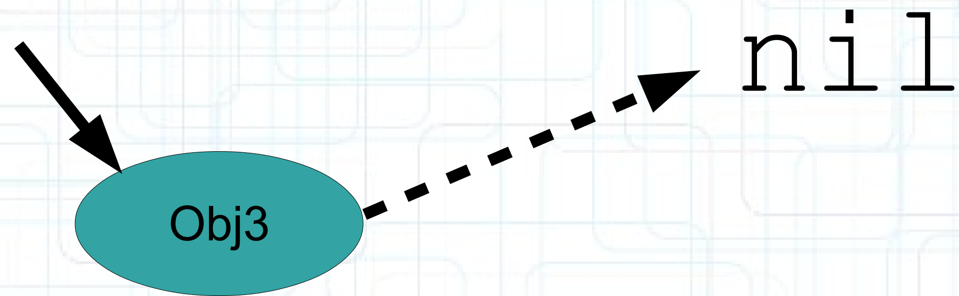
no `strong` reference to `Obj2`



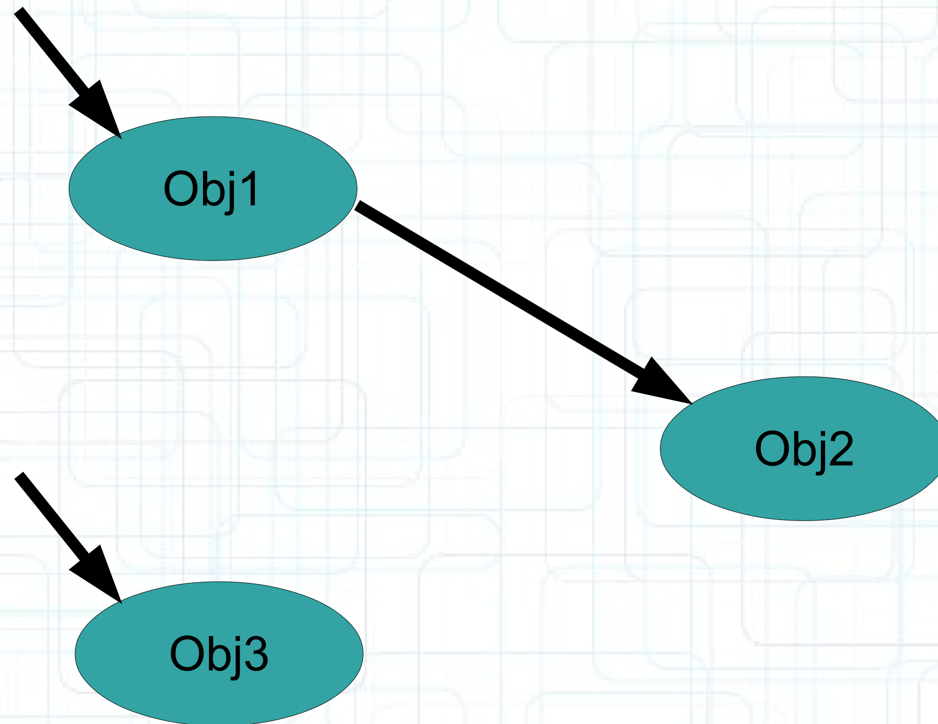
strong vs weak



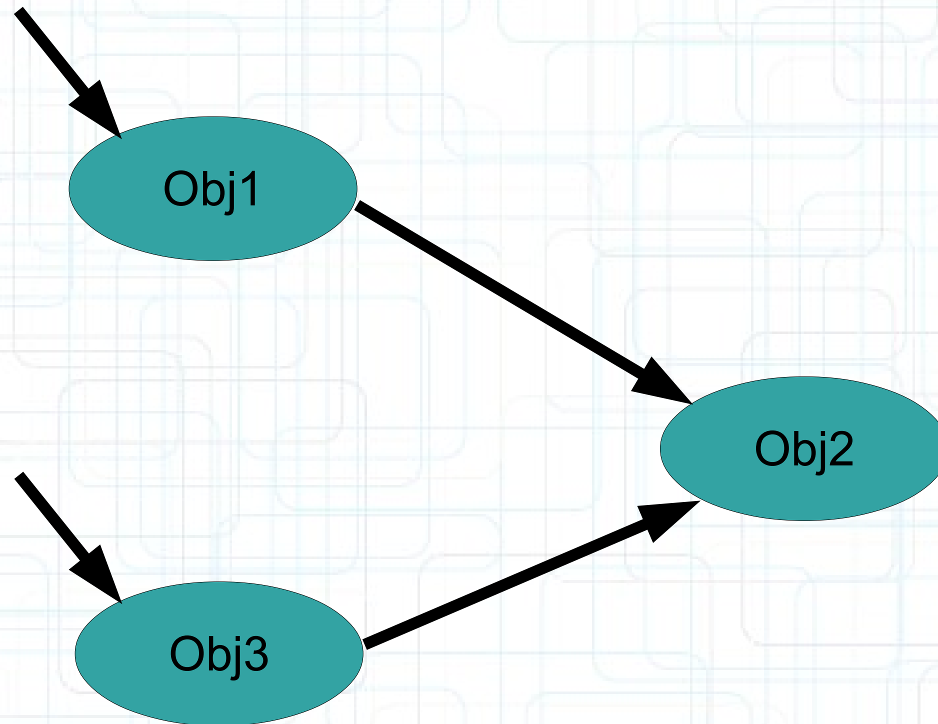
strong vs weak



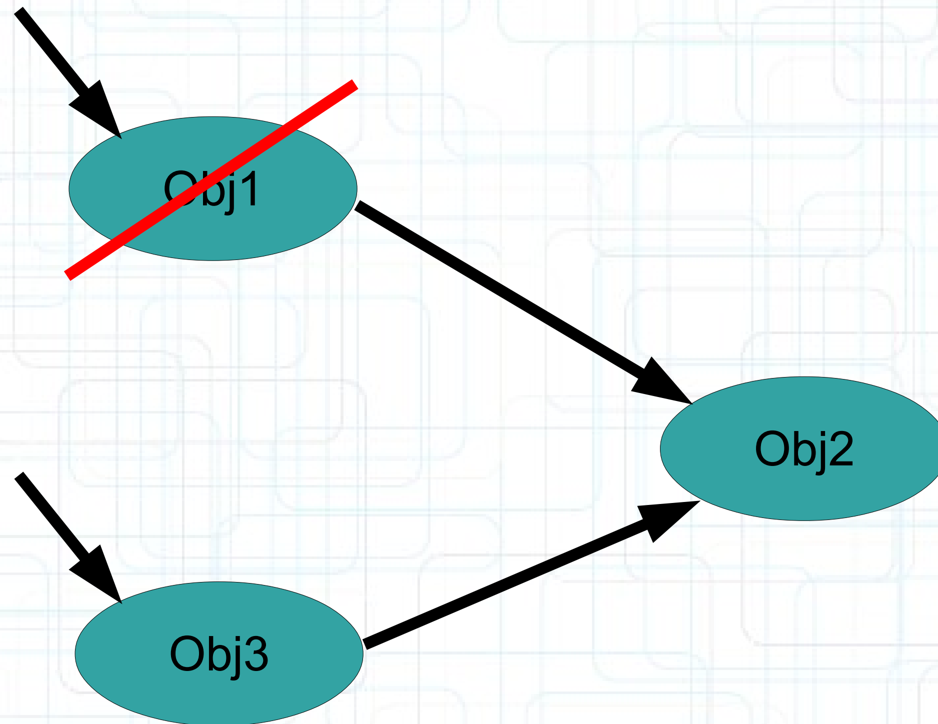
strong vs weak



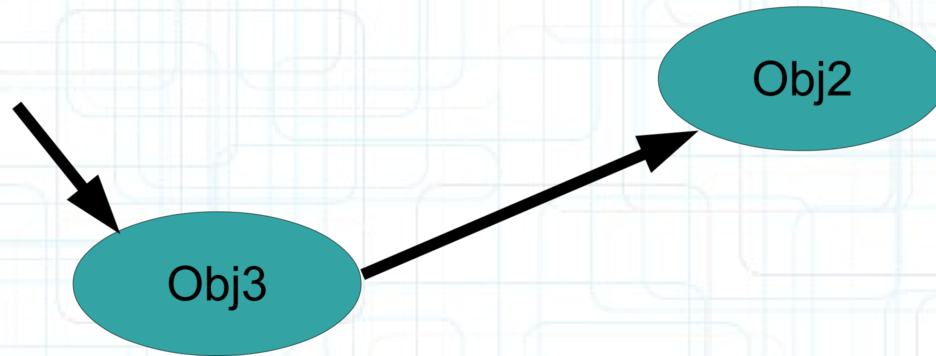
strong vs weak



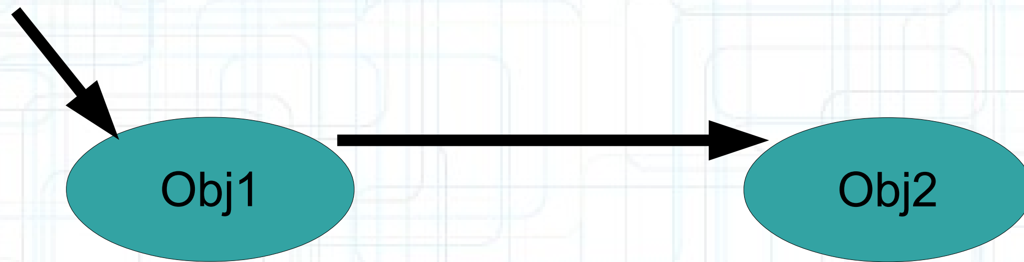
strong vs weak



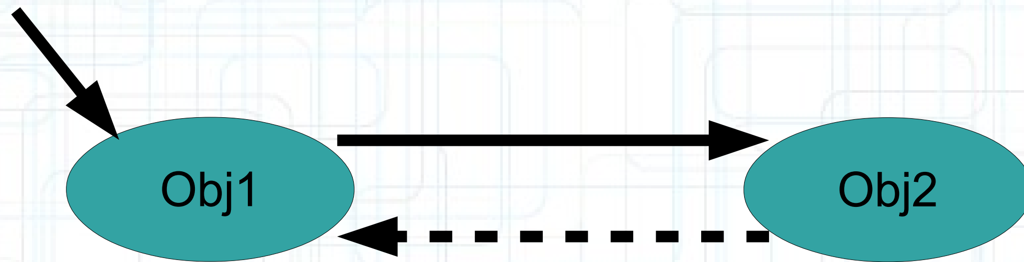
strong vs weak



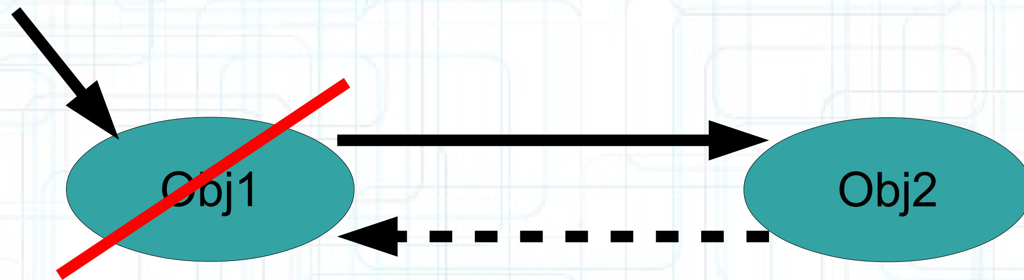
strong vs weak



strong vs weak

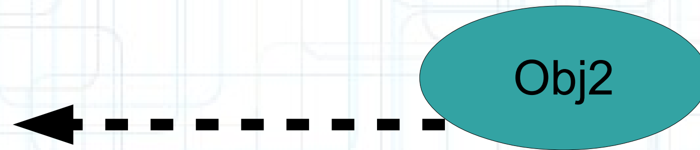


strong vs weak



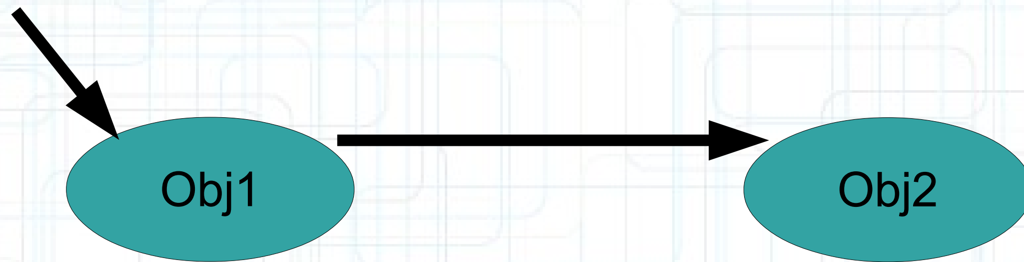
strong vs weak

no `strong` reference to `Obj2`

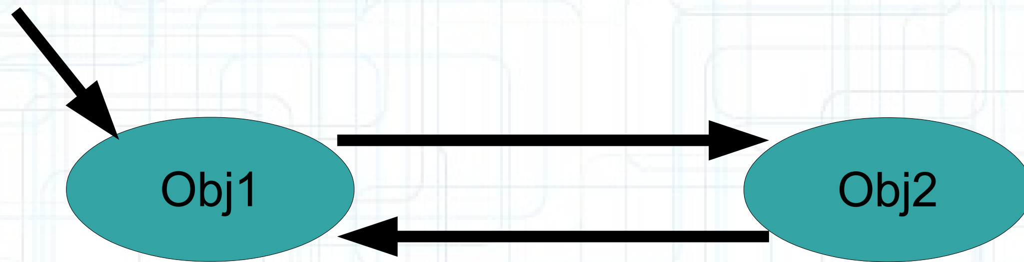


strong vs weak

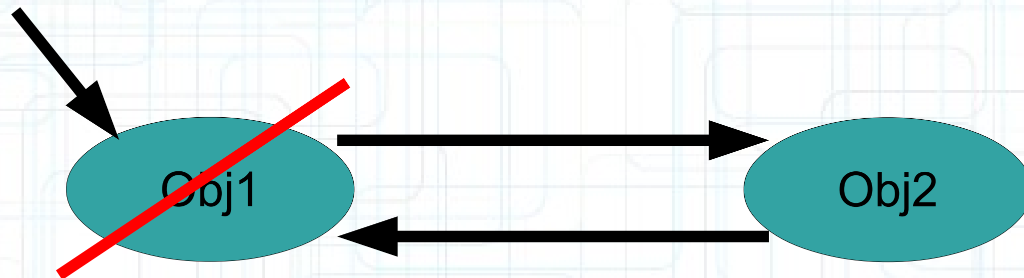
strong vs weak



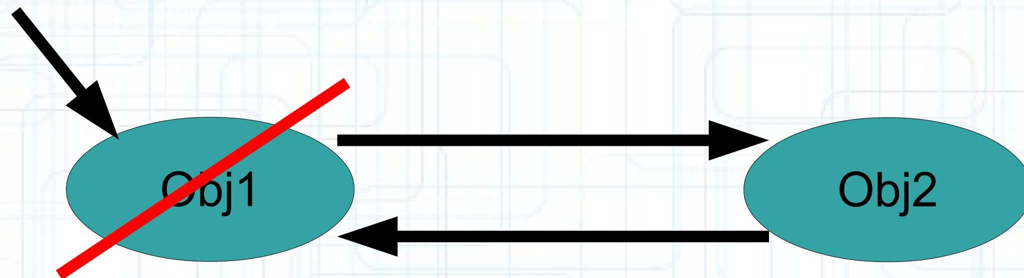
strong vs weak



strong vs weak



strong vs weak



Nothing gets deallocated. This problem is called a memory cycle. It can be solved using `weak` references.

strong vs weak

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) { /* do something with obj here */ }
  ```
  Curly braces will execute if `obj` points to an object.

`nil`

- Sending messages to `nil` is mostly okay. No code gets executed. If the method returns a value, it will return zero.
- For example:
```
int i = [obj methodWhichReturnsAnInt];
```
  In this case `i` will be zero if `obj` is `nil`.
- Be careful if the method returns a C struct. Return value is undefined.
```
CGPoint point = [obj getLocation];
```
  In this case `point` will have an undefined value if `obj` is `nil`.

BOOL

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

- Can be tested implicitly

```
if (flag) { /* flag is true */ }
if (!flag) { /* flag is false */}
```

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

`NO == 0`, `YES` is anything else (non-zero)

```
if (flag == YES) { /* flag is YES */ }
if (flag == NO) { /* flag is NO */ }
if (flag != NO) { /* flag is YES */ }
```

Instance vs Class Methods

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

Instance Methods

- Starts with a dash

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

- "Normal" Instance Methods

Class Methods

- Starts with a plus sign

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

- Creation & Utility Methods

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

Instance Methods

- Calling syntax

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

Class Methods

- Calling syntax

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

`doSomething` is a class method.

This `ship` is an instance.

This instance method (called “`class`”) returns a Class.

Instance vs Class Methods

Instance Methods

- `self/super` is calling instance
  `self` means “my implementation”
  `super` means “my superclass’s implementation”

Class Methods

- `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;
```

Instantiation

Asking other objects to create objects for you

- NSString's
  ```
  - (NSString *)stringByAppendingString:(NSString *)otherString;
  - (NSArray *)componentsSeparatedByString:(NSString *)separator;
  ```
- NSString's and NSArray's
  ```
  - (id)mutableCopy;
  ```
- NSArray's
  ```
  - (NSString *)componentsJoinedByString:(NSString *)separator;
  ```

Instantiation

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 examples from the previous slide), then you are creating.

Instantiation

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`).

Instantiation

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`.

More complicated `init` methods

- If an `initialization` method has arguments, it should still start with the four letters `init`
- Example (initializer for `UIView`):

```
- (id)initWithFrame:(CGRect)aRect;
UIView *myView = [[UIView alloc]
                  initWithFrame:thePerfectFrame];
```

Instantiation

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;
```

- From `NSNumber`:

```
- (id)initWithDouble:(double)value;

- (id)initWithInt:(int)value;
```

Instantiation

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, for example:

```
MyObject *obj = [[MyObject alloc] init];

NSNumber *intObject = [[NSNumber alloc]
                       initWithInt:57];
```

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.

Instantiation

Creating your own `initialization` method

- Here is an example of what it would look like if `init` (plain) were our designated initializer:

```
@implementation MyObject

- (id)init
{
 // first call our super's designated initializer
    self = [super init];

    if (self)
    {
        // initialize our subclass here
    }
    return self;
}

@end
```

Instantiation

Example: A subclass of `CalculatorBrain` with convenience initializer.

- Imagine that we enhanced `CalculatorBrain` to have a list of “valid operations”.
- We will allow the list to be `nil` which we will define to mean that all operations are valid.
- It might be nice to have a convenience initializer to set that array of operations.
- We would want to have a `@property` to set the array of valid operations as well, of course.

Instantiation

Example: A subclass of `CalculatorBrain` with convenience initializer.

- 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;
 // this 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

- This is a "statically" typed object:
  `NSString *s = ...;`
- This is not statically typed, but perfectly legal:
  `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**!

Dynamic Binding

- It is legal (and sometimes even good code) to “cast” a pointer
- But we usually do it only after we have used “introspection” to find out more about the object.
- More on introspection in a minute.
- For example, the following code ...

```
id obj = ...;
NSString *s = (NSString*)obj;
```

… is very 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 have forced the compiler to think that the `NSString` is a `Ship`. "All is 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 is 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 would 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.

Introspection

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.
- Here is an example:

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

Introspection

Arguments to these methods are a little tricky

- 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];
[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

- The Foundation framework defines a base layer of Objective-C classes.
- In addition to providing a set of useful primitive object classes, it introduces several paradigms that define functionality not covered by the Objective-C language.

  The Foundation framework is designed with these goals in mind:
- Provide a small set of basic utility classes.
- Make software development easier by introducing consistent conventions for things such as deallocation.
- Support Unicode strings, object persistence, and object distribution.
- Provide a level of OS independence, to enhance portability.

Foundation Framework

- The Foundation framework includes the root object class (`NSObject`), classes representing basic data types such as strings and byte arrays, collection classes for storing other objects, classes representing system information such as dates, and classes representing communication ports.
  `NSObject`
- Base class for pretty much every object in the iOS SDK.
- Implements methods for allocation, introspection , etc.
  `- (NSString *)description` is a useful method to override (it's `%@` in `NSLog()`).
- Some objects implement a copy mechanism:

```
- (id)copy;
- (id)mutableCopy;
```

  Not all objects implement this 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:@"%f", brain.operand];
    // class method
```

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

Foundation Framework

`NSMutableString`

- Mutable version of `NSString`. Not very often 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];
[ms appendFormat:@”%f”, brain.operand];
```

Foundation Framework

`NSNumber`

- Object wrapper around primitive types like `int`, `float`, `double`, `BOOL`, etc.
- `NSNumber *num = [NSNumber numberWithInt:36];`
  `float f = [num floatValue];`
  This 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 (for example, C structs).

```
CGPoint point = CGPointMake(25.0, 15.0);
NSValue *pointObject = [NSValue valueWithCGPoint:point];
```

Foundation Framework

`NSData`

- “Bag of bits”. Used to save/restore/transmit data throughout the iOS SDK.
- It has methods for writing data objects to the file system and reading them back.
- `NSData` and `NSMutableData` are typically used for data storage and are also useful in Distributed Objects applications, where data contained in data objects can be copied or moved between applications.

`NSDate`

- Used to find out the time right now or to store past or future times/dates.
- `[NSDate date]` returns the current date and time.
- 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 list of 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 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).
   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 *b = [NSDictionary dictionaryWithObjectsAndKeys:
                   [NSNumber numberWithInt:2], @“binary”,
                   [NSNumber numberWithInt:16], @“hexadecimal”,
                   nil];
- (int)count;
- (id)objectForKey:(id)key;
- (NSArray *)allKeys;
- (NSArray *)allValues;
```

- For unique identification, keys are sent `– (NSUInteger)hash` and `– (BOOL)isEqual:(NSObject *)obj`.
- `NSObject` returns the object’s pointer as its `hash` and `isEqual:` only if the pointers are equal. Keys are very often `NSStrings` (they `hash` based on contents and `isEqual:` if characters match).

Foundation Framework

`NSMutableDictionary`

- Mutable version of `NSDictionary`. It inherits methods from `super`.

```
+ (id)dictionary; // creates an empty dictionary

- (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;
```

Foundation Framework

`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 some others.

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

Foundation Framework

`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
     }
}
```

Enumeration

Looping through members of a collection in an efficient manner

- Looping through the keys or values of a dictionary is a bit different.
- Example:

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

Property Lists

- 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 `NSDictionarys` whose keys are `NSStrings` and values are `NSNumbers` is one.
- 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

- The remaining classes of the Foundation framework fall into various categories. The major categories of classes are described in the following slides.

  Operating-system services
- Many Foundation classes facilitate access of various lower-level services of the operating system.

  `NSUserDefaults`
- It provides a programmatic interface to a system database of global (per-host) and per-user default values (preferences).
- 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.

Other Foundation

`NSUserDefaults`

- Read and write via a shared instance obtained via class method `standardUserDefaults`.

```
[[NSUserDefaults standardUserDefaults] setArray:recArray
                                         forKey:@“Rec”];
```

- Sample methods:

```
- (void)setDouble:(double)aDouble
           forKey:(NSString *)key;
- (NSInteger)integerForKey:(NSString*)key;
//NSInteger is a type def 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];
```

Other Foundation

File system and URL

- `NSFileManager` provides a consistent interface for file operations such as creating, renaming, deleting, and moving files.
- `NSBundle` finds resources stored in bundles and can dynamically load some of them (for example, nib files and code).
- You use `NSURL` and related classes to represent, access, and manage URL sources of data.

Concurrency

- `NSThread` lets you create multithreaded programs, and various lock classes offer mechanisms for controlling access to process resources by competing threads.
- You can use `NSOperation` and `NSOperationQueue` to perform multiple operations (concurrent or non-concurrent) in priority and dependence order.

Other Foundation

Notifications

- Notification is a major design pattern in Cocoa. It is based on a broadcast mechanism that allows objects (called observers) to be kept informed of what another object is doing or is encountering in the way of user or system events.
- The object originating the notification can be unaware of the existence or identity of the observers of the notification.
- There are several types of notifications: synchronous, asynchronous, and distributed.
- The Foundation notification mechanism is implemented by the `NSNotification`, `NSNotificationCenter`, `NSNotificationQueue`, and `NSDistributedNotificationCenter` classes.

Other Foundation

Archiving and serialization

- The classes in this category make object distribution and persistence possible.
- `NSCoder` and its subclasses, along with the `NSCoding` protocol, represent the data an object contains in an architecture-independent way by allowing class information to be stored along with the data.

Objective-C language services

- `NSException` and `NSAssertionHandler` provide an object-oriented way of making assertions and handling exceptions in code.

XML processing

- `NSXMLParser` class is an object-oriented implementation of a streaming parser that enables you to process XML data in an event-driven way.

Other Foundation

JSON parsing

```
NSData *jsonData = [NSData dataWithContentsOfURL:
                   [NSURL URLWithString:@"http://..."]];
if (jsonData)
{
  id responseObj = [NSJSONSerialization
         JSONObjectWithData:jsonData
                    options:NSJSONReadingAllowFragments
                      error:nil];
  if ([responseObj isKindOfClass:[NSArray class]])
  {
     // show data
  }
}
```

Next Time

Views, Autorotation and Gestures:

- Views
- Drawing Paths
- Drawing Text
- Drawing Images
- Autorotation
- Protocols
- Gesture Recognizers