Mobile Application Development

Lecture 12
Introduction to Objective-C

Lecture Summary

- Objective-C language basics
- Classes and objects
- Methods
- Instance variables and properties
- Dynamic binding and introspection
- Foundation framework

References

iOS Developers Library

https://developer.apple.com/library/ios/documentation/cocoa/conceptual/ProgrammingWithObjectiveC

https://developer.apple.com/library/ios/navigation/

CS193P - Stanford class on iOS

http://www.stanford.edu/class/cs193p/cgi-bin/drupal/

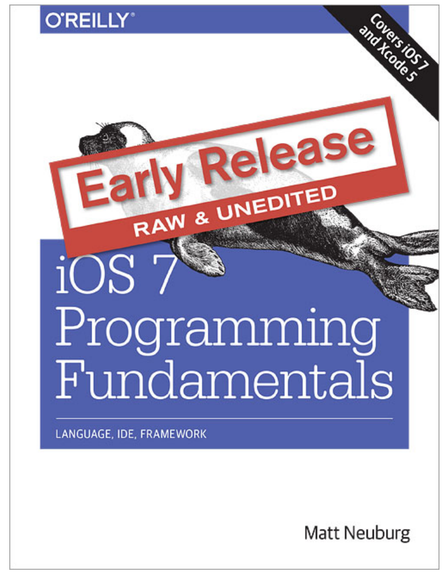
https://itunes.apple.com/us/course/coding-together-developing/id593208016

Programming iOS 7

by Matt Neuburg

O'Reilly Media - September 2013

http://shop.oreilly.com/product/0636920031017.do



iOS 7 Programming Fundamentals

by Matt Neuburg

O'Reilly Media - September 2013

http://shop.oreilly.com/product/0636920032465.do

Objective-C

- Objective-C is the primary programming language you use when writing software for OS X and iOS
- Objective-C is a strict superset of the C programming language and provides object-oriented capabilities and a dynamic runtime
  - it is possible to compile any C program with the Objective-C compiler
  - it is possible to include any C statement within an Objective-C program
  - it is even possible to include C++ code inside Objective-C, but the compiler must properly instructed to process C++ code
- Objective-C inherits the syntax, primitive types, and flow control statements of C and adds syntax for defining classes and methods
- All the syntax that is not related to object-oriented features of the language is the same as classical C syntax

Objective-C programs

- Objective-C programs are split into two main types of files:
  - ***.h*** files: define the interfaces, that is which functionalities are exposed to other parts of the code; classes, data structures, and methods are defined in the interface file
  - ***.m*** files: implement the interfaces defined in the corresponding .h file
- In case an implementation file contains C++ code, it must be named using the ***.mm*** extension
- Only the interface can be accessed from other parts of your code; the implementation file is opaque!

```objc
//
//  MDPoi.h
//  iMobdev
//
//  Created by Simone Cirani on 30/09/13.
//  Copyright (c) 2013 Università degli Studi
di Parma. All rights reserved.
//

#import <Foundation/Foundation.h>

@interface MDPoi : NSObject{

    NSString *_name;
    double _latitude;
    double _longitude;

}

- (id) initWithName:(NSString *)name latitude:
(double)latitude longitude:(double)longitude;

- (double) latitude;
- (double) longitude;

@property (nonatomic,copy) NSString *name;

- (void) log;

@end
```

.h

```objc
//
//  MDPoi.m
//  iMobdev
//
//  Created by Simone Cirani on 30/09/13.
//  Copyright (c) 2013 Università degli Studi
di Parma. All rights reserved.
//

#import "MDPoi.h"

@implementation MDPoi

@synthesize name = _name;

- (id) initWithName:(NSString *)name latitude:
(double)latitude longitude:(double)longitude{
    NSLog(@"%s", __FUNCTION__);
    if(self = [super init]){
        _name = [name copy];
        _latitude = latitude;
        _longitude = longitude;
    }
    return self;
}

- (double) latitude{
    return _latitude;
}

- (double) longitude{
    return _longitude;
}

- (void) log{
```

.m

Defining classes

- Classes are defined in the interface file, between the `@interface` directive and the corresponding `@end`
- Everything inside the interface block defines the class' structure (instance variables) and functionalities

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

@interface MDPoi : NSObject{

    NSString *_name;
    double _latitude;
    double _longitude;

}

- (void)setLatitude:(double)lat;

@end
```

MDPoi.h

Defining classes

- Let's define a new class called MDPoi, used to hold a point of interest, with name, latitude, and longitude

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

@interface MDPoi : NSObject{

    NSString *_name;
    double _latitude;
    double _longitude;

}

- (void)setLatitude:(double *)lat;

@end
```

Defining classes

- Let's define a new class called MDPoi, used to hold a point of interest, with name, latitude, and longitude

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

← Import the header for **NSObject** class

```
@interface MDPoi : NSObject{

    NSString *_name;
    double _latitude;
    double _longitude;

}

- (void)setLatitude:(double *)lat;

@end
```

Defining classes

- Let's define a new class called MDPoi, used to hold a point of interest, with name, latitude, and longitude

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

@interface MDPoi : NSObject{

    NSString *_name;
    double _latitude;
    double _longitude;

}

- (void)setLatitude:(double *)lat;

@end
```

Declare the **MDPoi** class

Defining classes

- Let's define a new class called MDPoi, used to hold a point of interest, with name, latitude, and longitude

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

@interface MDPoi : NSObject{

    NSString *_name;
    double _latitude;
    double _longitude;

}

- (void)setLatitude:(double *)lat;

@end
```

Extends the NSObject class

Defining classes

- Let's define a new class called MDPoi, used to hold a point of interest, with name, latitude, and longitude

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

@interface MDPoi : NSObject{

    NSString *_name;
    double _latitude;
    double _longitude;

}

- (void)setLatitude:(double *)lat;

@end
```

Instance variables must be declared between curly braces

Defining classes

- Let's define a new class called MDPoi, used to hold a point of interest, with name, latitude, and longitude

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

@interface MDPoi : NSObject{

    NSString *_name;
    double _latitude;
    double _longitude;

}

- (void)setLatitude:(double *)lat;

@end
```

Instance variables must be declared between curly braces

Defining classes

- Let's define a new class called MDPoi, used to hold a point of interest, with name, latitude, and longitude

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

@interface MDPoi : NSObject{

    NSString *_name;
    double _latitude;
    double _longitude;

}

- (void)setLatitude:(double *)lat;

@end
```

← Methods must be defined outside the curly braces

Implementing classes

- Now, let's implement the MDPoi class defined in the MDPoi.h interface file

```
#import "MDPoi.h"

@implementation MDPoi

- (void)setLatitude:(double *)lat{
    _latitude = lat;
}

@end
```

MDPoi.m

Implementing classes

- Now, let's implement the MDPoi class defined in the MDPoi.h interface file

```
#import "MDPoi.h"
```

← Import the interface header file

```
@implementation MDPoi

- (void)setLatitude:(double *)lat{
    _latitude = lat;
}

@end
```

Implementing classes

- Now, let's implement the MDPoi class defined in the MDPoi.h interface file

```
#import "MDPoi.h"

@implementation MDPoi

- (void)setLatitude:(double *)lat{
    _latitude = lat;
}

@end
```

The implementation of the interface must be between the `@implementation` directive and the corresponding `@end`

Implementing classes

- Now, let's implement the MDPoi class defined in the MDPoi.h interface file

```
#import "MDPoi.h"

@implementation MDPoi

- (void)setLatitude:(double *)lat{
    _latitude = lat;
}

@end
```

The declared methods must be implemented
(a warning is issued if not, but no error - only at runtime)

Special keywords: `id`, `nil`, `BOOL`, and `self`

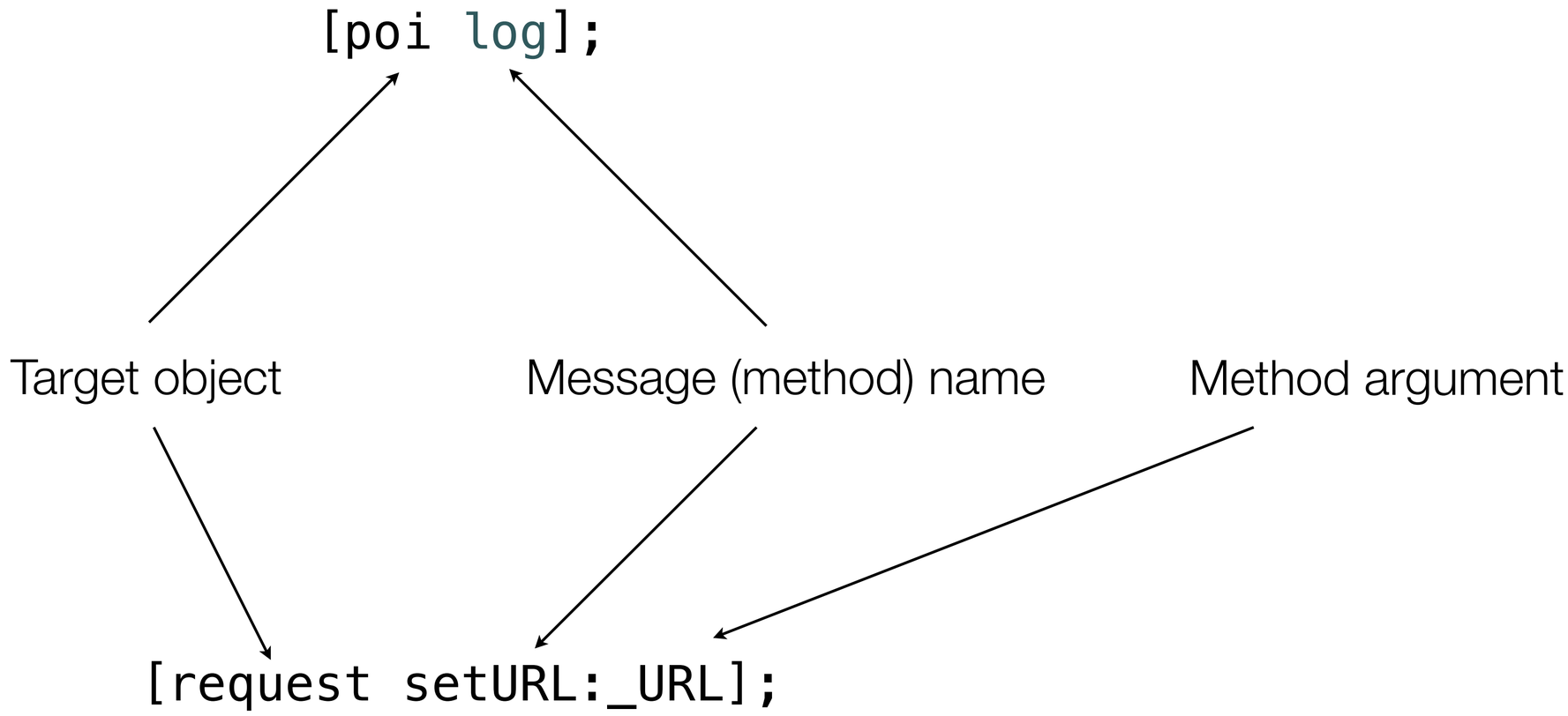
- In Objective-C, all objects are allocated in the heap, so to access them we always use a pointer to the object
- `id` means a pointer to an object of any kind (similar `void*` in C)
- `nil` is the value of a pointer that points to nothing (`NULL` in C)
- `BOOL` is the type defined (`typedef` in file objc.h) by Objective-C for boolean values
  - `YES` == 1 (true)
  - `NO` == 0 (false)
- `self` means a pointer to the current object (similar `this` in Java)

Objective-C methods

- Objective-C objects talks to each other by sending messages
- In Objective-C, method invocation is based on **message passing** to objects
- Message passing differs from classical method calls because the method to be executed is not bound to a specific section of code (in most cases at compile-time) but **the target of the message is dynamically resolved at runtime**, so it might happen that the receiver won't respond to that massage (no type checking can be performed at compile-time)
- If the receiver of a message does not respond to the message, an exception is thrown
- It is possible to send a message to `nil`: no NullPointerException is thrown (0 is returned)

Sending messages

- Square brackets are used to send messages

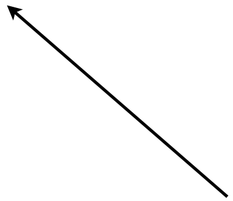


Method syntax

```
- (NSArray *)pointsWithinRange:(double)distance fromPoi:(MDPoi *)poi;
```

Method syntax

```
- (NSArray *)pointsWithinRange:(double)distance fromPoi:(MDPoi *)poi;
```



- **Instance methods** (relative to the particular object that is the target of the message) begin with a minus sign;
  - they can access instance variables
- **Class methods** (relative to the class itself) begin with a plus sign;
  - accessible through the class name (no need for an instance)
  - they cannot access instance variables
  - utility methods and allocation methods are typically class methods

Method syntax

```
- (NSArray *)pointsWithinRange:(double)distance fromPoi:(MDPoi *)poi;
```

Return type (between parentheses)

Method syntax

```
- (NSArray *)pointsWithinRange:(double)distance fromPoi:(MDPoi *)poi;
```

First part of method name

Second part of method name

This method's name is `pointsWithinRange:fromPoi:`

Method syntax

```
- (NSArray *)pointsWithinRange:(double)distance fromPoi:(MDPoi *)poi;
```

First argument of type `double`

Second argument of type `MDPoi*`

Methods in Objective-C

- Method names in Objective-C sound like sentences natural language

```
pointsWithinRange:fromPoi:
```

- Some methods have very long names:

```
- (BOOL)tableView:(UITableView *)tableView
 canPerformAction:(SEL)action
forRowAtIndexPath:(NSIndexPath *)indexPath
       withSender:(id)sender;
```

- In order to keep the code readable, it is better to align colons

Instance variables

- By default, instance variables are `@protected`
  - `@private`: can be accessed only within the class itself
  - `@protected`: can be accessed directly within the class itself and its subclasses
  - `@public`: can be accessed anywhere in the code (not a good OOP practice!)
- Good practice in OOP is to mark all instance variables as private and use getter and setter methods to access them

MDPoi class revised

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

@interface MDPoi : NSObject{

    @private
      NSString *_name;
      double _latitude;
      double _longitude;

}

- (double)latitude;
- (void)setLatitude:(double)lat;

- (double)longitude;
- (void)setLongitude:(double)lon;

@end
```

MDPoi class revised

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

@interface MDPoi : NSObject{

   @private
      NSString *_name;
      double _latitude;
      double _longitude;

}

- (double)latitude;
- (void)setLatitude:(double)lat;

- (double)longitude;
- (void)setLongitude:(double)lon;

@end
```

All instance variables are private

MDPoi class revised

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

@interface MDPoi : NSObject{

    @private
      NSString *_name;
      double _latitude;
      double _longitude;

}

- (double)latitude;
- (void)setLatitude:(double)lat;

- (double)longitude;
- (void)setLongitude:(double)lon;

@end
```

getter method for _latitude ivar

MDPoi class revised

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

@interface MDPoi : NSObject{

    @private
      NSString *_name;
      double _latitude;
      double _longitude;

}

- (double)latitude;
- (void)setLatitude:(double)lat;

- (double)longitude;
- (void)setLongitude:(double)lon;

@end
```

setter method for _latitude ivar

MDPoi class revised

```objc
#import <Foundation/Foundation.h>

@interface MDPoi : NSObject{

    @private
      NSString *_name;
      double _latitude;
      double _longitude;

}

- (double)latitude;
- (void)setLatitude:(double)lat;

- (double)longitude;
- (void)setLongitude:(double)lon;

@end
```

MDPoi.h

```objc
...

- (double) latitude{
  return _latitude;
}

- (void) setLatitude:(double)lat{
  _latitude = lat;
}

- (double) longitude{
  return _longitude;
}

- (void) setLongitude:(double)lon{
  _longitude = lon;
}
...
```

MDPoi.m

Accessing ivar with getters and setters

```
// get the value of _latitude
double lat = [poi latitude];
```

```
// set the value of _latitude
[poi setLatitude:12.0f];
```

Properties

- Properties are a convenient way to get/set the value of instance variables using dot notation
- The main advantage is to avoid the congestion of brackets in the code and to allow for more readable concatenated method invocations
- It is VERY IMPORTANT to pay attention to the lowercase and uppercase letters in the method name: **the compiler can understand the dot notation only if the names of the getter and setter methods are correct**!

```
double lat = [poi latitude];   <---->   double lat = poi.latitude;

[poi setLatitude:12.0f];       <---->   poi.latitude = 12.0f;
```

Compiler-generated properties

- It is possible to define all the getters and setters by hand, but let the compiler help you do the job
- The `@property` keyword instructs the compiler to generate the getter and setter definitions for you

```
- (double)latitude;
- (void)setLatitude:(double)lat;
```

⟷

```
@property double latitude;
```

- If only a getter method is needed it is possible to specify it to the compiler by adding the “readonly” keyword after `@property`

```
- (double)latitude;
```

⟷

```
@property (readonly) double latitude;
```

Properties

- Properties can also be defined without an instance variable that matches it

```
@property (readonly) NSString *detail;
```

.h

```
- (NSString *)detail{
  return [NSString stringWithFormat:@"%@ - (%f,%f)",
          _name, _latitude, _longitude];
}
```

.m

```
NSString *str = poi.detail;
```

usage

Compiler-generated property implementation

- It is possible to let the compiler implement (synthesize) a property matching an instance variable by using the `@synthesize` directive within the implementation of the class
- Using `@synthesize` doesn't forbid to implement either property directly

```
@synthesize latitude = _latitude;
```

the backed ivar can have a different name

getter/setter method depends on property name

```
- (double) latitude{
    return _latitude;
}

- (void) setLatitude:(double)lat{
    _latitude = lat;
}
```

Dynamic binding

- Objective-C is based on message passing for method invocation
- The resolution of the target of a message is done at runtime: no type checking can be performed at compile-time
- Specifying the type of an object when writing code does not let the compiler perform any type checking, but can only help to highlight possible bugs
- If an object receives a message which it cannot respond to (namely, the class or one of its superclasses does not implement that method) the application will crash
- Casting pointers is a way to “force” the execution of a method, but this should be done carefully (use **introspection**)
- **Make sure you always send messages to objects that can respond!**

Introspection

- Introspection allows to discover at runtime whether an object can respond or not to a message
- This is especially useful when all we can get are objects of type `id`, such as those that we can get from a collection
- How can we do these test? Any object that is a subclass of NSObject has the following methods:
  - `isKindOfClass:` can be used to check if the object's type matches a class or one if its subclasses
  - `isMemberOfClass:` can be used to check if the object's type matches a class, strictly
  - `respondsToSelector:` can be used to check whether the object can respond to a given selector (a selector is a special type that indicates a method name)

Introspection

- `isKindOfClass:` and `isMemberOfClass:` take a `Class` object as an argument
- A `Class` object can be obtained by using the `class` method which can be invoked on any Objective-C class

```
if([obj isKindOfClass:[NSString class]]){
    NSString *str = (NSString *)obj;
    // do something with string
    ...
}
```

Introspection

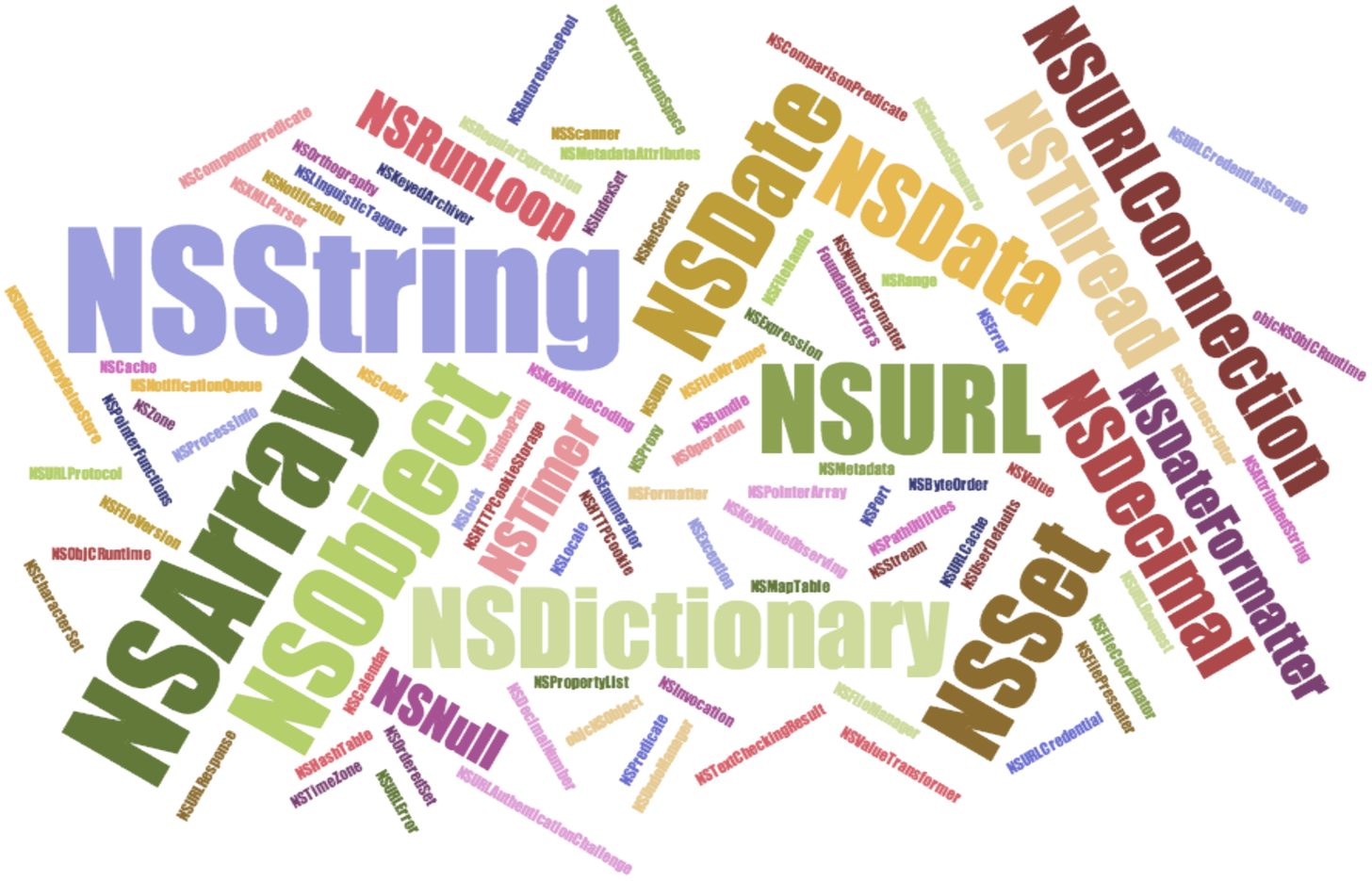
- `respondsToSelector:` takes a **selector** (SEL) as an argument
- A selector can be obtained with the `@selector` keyword:

```
if([obj respondsToSelector:@selector(lowercaseString)]){
    NSString *str = [obj lowercaseString];
    ...
}
```

- It is possible to ask an object to execute a selector:

```
SEL selector = @selector(lowercaseString);
if([obj respondsToSelector:selector]){
     NSString * str = [obj performSelector:selector];
     ...
}
```

Foundation framework



- The Foundation framework defines a base layer of Objective-C classes
- It provides a set of useful primitive object classes and introduces several paradigms that define functionality not covered by the Objective-C language
- It makes software development easier
- Provide a level of OS independence, to enhance portability (iOS/Mac OSX)

NSObject

- NSObject is the base class for almost anything in iOS (similar to Object class in Java)
- Everything is an (NS)Object
- Provides class and instance methods for memory management
- Provides methods for introspection (`isKindOfClass:` and `isMemberOfClass:` and `respondsToSelector:`)
- Basically, any custom class you will create will be directly or indirectly a subclass of NSObject

NSString & NSMutableString

- **NSString** is a class for managing Unicode strings in Objective-C instead C strings (**char***)
- Provides useful methods to manipulate strings
- `NSString *str = @"This is an Objective-C string";`
- Note the @ symbol at the beginning: it tells the compiler to handle it as **NSString**
- **NSString**s are immutable: typically, you get a new **NSString** from an existing one using **NSString** methods
- **NSMutableString** is a mutable **NSString** meaning that it can be manipulated at runtime
- **NSMutableString** is a **NSString**: all methods of **NSString** are available to **NSMutableString**
- With **NSMutableString** you can work with the same object without dealing with new ones

NSNumber

- `NSNumber` wraps primitive types into an object (`int`, `double`, `float`, ...)
- `NSNumber` objects can be used for storing them into collections, which require their elements to be objects

```
NSNumber *number = [NSNumber numberWithInt:2];
int i = [number intValue];
```

NSData & NSMutableData

- `NSData` (and its mutable version `NSMutableData`) are used to store bytes
- Typically, `NSData` objects are returned by remote connections since iOS cannot determine in advance what type of data is being transferred (binary or textual)

NSDate

- `NSDate` allows to perform operations on dates:
  - calculations
  - formatting
- `NSTimeInterval` (`double`) is also used with `NSDate` to manage operations on dates

```
NSDate *now = [NSDate date];
NSDate *nextHour = [NSDate dateWithTimeIntervalSinceNow:3600];
NSTimeInterval delta = [now timeIntervalSinceDate:nextHour];
```

NSArray & NSMutableArray

- **`NSArray`** (and its mutable version **`NSMutableArray`**) provides an ordered list of objects
- Objects of different types can be stored in **`NSArray`**: introspection should be used to determine the type of each object stored

```
NSArray *array = [NSArray arrayWithObjects:@"string", [NSNumber numberWithInt:10], nil];
int size = [array count];
id first = [array objectAtIndex:0];
id last = [array lastObject];
```

```
NSMutableArray *array2 = [NSMutableArray arrayWithObjects:@"str1", @"str2", nil];
[array2 addObject:@"str3"];
[array2 insertObject:@"first" atIndex:0];
[array2 removeObjectAtIndex:1];
```

NSDictionary & NSMutableDictionary

- **`NSDictionary`** (and its mutable version **`NSMutableDictionary`**) provides a map of key/value pairs
- Both keys and values are objects
- Keys must belong to a class that implements the **`hash`** and **`isEqual:`** methods
- Usually, keys are **`NSString`** objects
- Objects of different types can be stored in **`NSDictionary`**: introspection should be used to determine the type of each object stored

```
NSDictionary *dict = [NSDictionary dictionaryWithObjectsAndKeys:
            @"key1", @"value1", @"key2", @"value2", nil];
int size = [dict count];
id val = [dict objectForKey:@"key1"];


NSMutableDictionary *dict2 = [NSMutableDictionary dictionary];
[dict2 setObject:@"value" forKey:@"key"];
[dict2 removeObjectForKey:@"key"];
```

NSSet & NSMutableSet

- **NSSet** (and its mutable version **NSMutableSet**) provides an unordered list of objects
- Objects of different types can be stored in **NSSet**: introspection should be used to determine the type of each object stored

```
NSSet *set = [NSSet setWithObjects:@"obj1", @"obj2", @"obj3", nil];
int size = [set count];
id random = [set anyObject];


NSMutableSet *teachers = [NSMutableSet set];
[teachers addObject:@"Simone"];
[teachers addObject:@"Marco"];
[teachers removeObject:@"Simone"];
```

Iterating over collections

- It is possible to iterate over a collection as if it were a classical array
- Since collections can store objects of any type, it is a good practice to check against types before sending messages in order to avoid crashes

```
for(int i = 0; i < [alphabet count]; i++){
    id object = [alphabet objectAtIndex:i];
    // do something with object
    ...

    if([object isKindOfClass:[NSString class]]){
      // do something with string
      ...
    }
}
```

Enumeration

- A more convenient way to iterate over a collection is through **enumeration**
- Enumeration is performed through **for-in** statements

```
for(id object in alphabet){
    if([object isKindOfClass:[NSString class]]){
        NSString *string = (NSString *)object;
        // do something with string
        ...
    }
}
```

- It is possible to avoid explicit casting ONLY IF YOU KNOW WHAT YOU ARE DOING!

```
for(NSString *string in alphabet){
    // do something with string
    ...
}
```

Mobile Application Development

Lecture 12
Introduction to Objective-C