Objective-C

448460-1
Fall 2011
09/15/2011
Kyoung Shin Park
Multimedia Engineering
Dankook University

Overview

- Objective-C
- Foundation Framework

Objective-C

Objective-C

- Methods (Class and Instance)
- Instance Variables
- Class
- Properties
- Object Typing & Dynamic Binding
- Introspection
- Nil and BOOL

Object-C

- Strict superset of C
- A very simple language, but some new syntax
- Single inheritance, classes inherit from one and only one superclass
- Protocols define behavior that cross classes
- Dynamic runtime
- Loosely typed, if you'd like

Syntax Additions

- Small number of additions
- Some new types
  - Anonymous object
  - Class
  - Selectors
- Syntax for defining classes
- Syntax for message expressions

Dynamic Runtime

- Object creation
  - All objects allocated out of the heap
  - No stack based objects
- Message dispatch
- Introspection

Classes

- In Objective-C, classes and instances are both objects
- Class is the blueprint to create instances
- Classes declare state and behavior
  - State (data) is maintained using instance variables
  - Behavior is implemented using methods
- Instance variables typically hidden
  - Accessible only using getter/setter methods

Instance and Class Methods

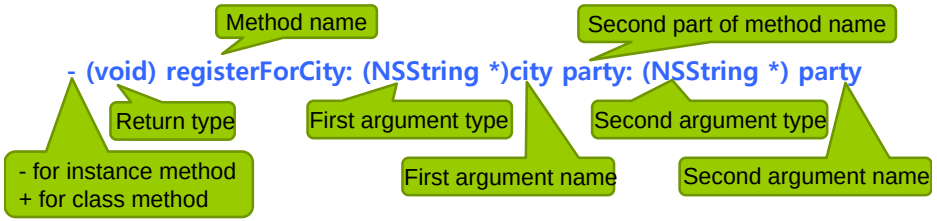
- Instance Method
  - Instances methods are normal methods you are used to

```
- (id) init;
- (float) height;
- (void) walk;
- (void) registerForCity: (NSString *)city party: (NSString *) party
```

- Class Method
  - Class methods are used for allocation, singletons, utilities

```
+ (id) alloc;
+ (id) person;
+ (Person *) sharedPerson;
```

Method Syntax



- Line up colons when there are lots of arguments:

```
- (void) slpitViewController: (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;
```

Message Syntax

```
[receiver message]

[receiver message:argument]

[receiver message:arg1 andArg:arg2]
```

Message Examples

```
Person *voter;      // assume this exists
[voter castBallot];
int theAge = [voter age];
[voter setAge:21];
if ([voter canLegallyVote]) {
        // do something
}
[voter registerForCity:@"Seoul" party:@"Independent"];
NSString *name = [[voter spouse] name];
```

Message Definition Examples

```
Person *voter;  // assume this exists
- (void) castBallot;
[voter castBallot];
- (int) age;
int theAge = [voter age];
- (void) setAge: (int)age;
[voter setAge:21];
- (BOOL) canLegallyVote;
if ([voter canLegallyVote]) {      // do something
}
- (void) registerForCity:(NSString *)city party:(NSString *) party;
[voter registerForCity:@"Seoul" party:@"Independent"];
- (Person *) spouse;
- (NSString *) name;
NSString *name = [[voter spouse] name];
```

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)

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

Protected: foo & bar
Private: eye
Public: forum

Class

- Header (Person.h)
  - Class interface declared in the header

```
#import <Foundation/Foundation.h>
@interface Person: NSObject
{
    NSString *name;        // instance variables
    int age;
}
- (NSString *) name;        // method declarations
- (void) setName: (NSString *) value;
- (int) age;
- (void) setAge: (int) age;
- (BOOL) canLegallyVote;
- (void) castBallot;
@end
```

Class

- Implementation (Person.m)
  - implement setter/getter methods
  - Implement action methods

```
#import "Person.h"
@implementation Person
- (int) age {                    // getter
    return age;
}
- (void) setAge: (int) value {   // setter
    age = value;
}
- (BOOL) canLegallyVote {        // action
    return ([self age] >= 18);   // calling your own methods
} // ... and other methods
@end
```

Class

- Superclass methods
  - Objects have an implicit variable named "self", like "this" in Java, C++ and C#.
  - Can also invoke superclass methods using "super".

```
- (void) doSomething {
    // call superclass implementation first
    [super doSomething];
    // then do our custom behavior
    int foo = bar;
}
```

Properties

- Objective-C 2.0 introduced dot syntax
- Convenient shorthand for invoking accessor methods

```
float height = [person height];     // get
float height = person.height;

[person setHeight:newHeight];     // set
person.height = newHeight;
```

- Follows the dots

```
[[person child] setHeight:newHeight];
person.child.height = newHeight;
```

Properties

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

```
@interface Person: NSObject
{
@private
    int p_height;
}
- (int) height;                    // getter method
- (void) setHeight: (int) value;   // setter method
@end
```

```
@implementation Person
- (int) height {                   // getter
      return p_height;
}
(void) setHeight: (int) value {   // setter
      p_height = value;
}
@end
```

- Access your instance variable using "dot notation"

```
aPerson.height = newHeightValue;          // set
int heightValue = anotherPerson.height;   // get
```

Properties

- @property

```
@interface Person: NSObject
{
    int age;
    int p_height;
}
@property int age;      // you can get the compiler to generate
@property int height; // set/get method declarations with @property
@end
```

- @synthesize

```
@implementation Person
@synthesize age;      // let the compiler generate implementation
@synthesize height = p_height; // when used a different variable
@end
```

Properties

- Read-only vs. Read-write
  - Use the readonly keyword, only the getter will be declared

```
@property int age;          // read-write by default
@property (readonly) int height;  // does not declare a setHeight
```

- Memory management policies (only for object properties)

```
@property (assign) NSString *name; // pointer assignment
@property (retain) NSString *name; // retain called
@property (copy) NSString *name; // copy called
```

Properties

- Mix and match synthesized and implemented properties
  - The 'setAge' setter method is explicitly implemented and the 'age' getter method is still synthesized.
  - The 'height' setter and getter method is synthesized.

```
@inplementation Person
@synthesize age;
@synthesize height = p_height;
- (void) setAge: (int) value {
    age = value;
    // do something with the new age value….
}
@end
```

Dot Syntax and self

- When used in custom methods, be careful with dot syntax for properties defined in your class.
- References to properties and ivars behave very differently

```
@interface Person : NSObject
{
    NSString *name;
}
@property (copy) NSString *name;
@end
@implementation Person
- (void) doSomething {
    name = @"Fred";       // accesses ivar directly!
    self.name = @"Fred";  // calls accessor method
}
@end
```

Common Pitfall with Dot Syntax

- What will happen when this code executes?

```
@implementation Person
- (void) setAge: (int) value {
    self.age = value;
}
@end
```

- This is equivalent to:

```
@implementation Person
- (void) setAge: (int) value {
    [self setAge: value];    // Infinite loop!
}
@end
```

Dynamic and Static Type

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

```
Person *anObject = ... ;     // Statically-typed object
id obj = anObject;           // Dynamically-typed object
// Never use "id *" (that would mean "a pointer to a pointer to an object").
```

- *Objective-C provides compile-time (not runtime) type checking & Objective-C always uses dynamic binding*
  - Decision about code to run on message send happens at runtime, not at compile time.
- It is legal to "cast" a pointer
  - But we usually do it only after we've used "introspection" to find out more about the object.

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.
[v shoot];       // Compiler warning! Would not crash at runtime though
                 // We know v is a Ship & Compiler knows v is a Vehicle
```

Object Typing

```
id obj =  ...;
[obj shoot]; // No compiler warning
             // Compiler knows the method shoot exists
             // it's not impossible that obj might respond to it
             // we have not typed obj, so no warning
             // might crash at runtime if obj is not a Ship

// Compiler warning! Compiler has never heard of this method
// Therefore, it's pretty sure obj will not respond to it
[obj someMethodNameThatNoObjectAnywhereRespondsTo];
```

Object Typing

```
NSString *hello = @"hello";
// Compiler warning
// Compiler knows NSString do not respond to shoot. Crash at runtime
[hello shoot];

// No compiler warning
// We are "casting". Compiler thinks we know what we're doing.
Ship *helloShip = (Ship *) hello;
// No compiler warning
// We've forced the compiler to think NSString is a Ship
[helloShip shoot];
// No compiler warning
// We've forced the compiler to ignore the object type by "casting".
// Guaranteed crash at runtime.
[(id)hello shoot];
```

Class Introspection

- You can ask an object about its class.
  - You get a **Class** by sending the class method **class** to a class

```
Class myClass = [myObject class];
NSLog(@"My class is %@", [myObject className]);
```

- Testing for general class membership (subclasses included):

```
if ([myObject isKindOfClass:[UIControl class]]) {
    // doSomething
}
```

- Testing for specific class membership (subclasses excluded)

```
if ([myObject isMemberOfClass:[NSString class]]) {
    // something string specific
}
```

Class 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

Selector

- Selectors identify methods by name
- A selector has type **SEL**.

```
SEL action = [button action];
[button setAction:@selector(start:)];
```

- Target/action uses this selector.

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

- Conceptually similar to function pointer.
- Selectors include the name and all colons:

```
- (void) setName:(NSString *)name age:(int)age;
SEL sel = @selector(setName: age:);
```

Selector

- You can determine if an object responds to a given selector.

```
id obj;
SEL startSelector = @selector(start:);
if ([obj respondsToSelector:startSelector]) {
    [obj performSelector:startSelector withObject:self]
}
```

- This sort of introspection and dynamic messaging underlies many Cocoa design patterns.

```
- (void) setTarget: (id) target;
- (void) setAction: (SEL) action;
```

The Null Object Pointer

- Test for nil explicitly
  ```
  if (person == nil) return;
  ```
- Or, implicitly
  ```
  if (!person) return;
  ```
- Can use in assignments and as arguments if expected
  ```
  Person = nil;
  [button setTarget:nil];
  ```
- Sending a message to nil is (mostly) okay.
  ```
  Person = nil;
  [person castBallot];          // No code gets executed
  int i = [obj methodReturnsAnInt]; // i will be 0 if obj is nil
  CGPoint p = [obj getLocation]; // CGPoint struct p will have an
  undefined value if obj is nil
  ```

BOOL typedef

- When Objective-C was developed, C had no boolean type. Objective-C uses a typedef to define BOOl as a type.
  ```
  BOOL flag = NO;
  ```
- Macros included for initialization and comparison: YES and NO
  ```
  if (flag == YES)
  if (flag)
  if (!flag)
  if (flag != YES)
  flag = YES;
  flag = 1;
  ```

Object Identity vs. Equality

- Identity – testing equality of the pointer values.
  ```
  if (object1 == object2) {
      // same exact object instance
  }
  ```
- Equality - testing object attributes
  ```
  if ([object1 isEqual: object2]) {
      // logically equivalent, but may be different object instances
  }
  ```

Description

- NSObject implements -description
  ```
  - (NSString *) description;
  ```
- Objects represented in format strings using %@
- When an object appears in a format string, it is asked for its description.
  ```
  [NSString stringWithFormat: @"The answer is: %@", myObject];
  ```
- You can log an object's description with:
  ```
  NSLog([anObject description]);
  ```
- Your custom subclasses can override description to return more specific information.

Foundation Framework

Foundation Framework

- Value and collection classes
- User defaults
- Archiving
- Notifications
- Undo manager
- Tasks, timers, threads
- File system, piles, I/O, bundles

NSObject

- Root class for all UIKit and Foundation classes
- Implements many basics
  - Memory management (retain, release)
  - Introspection (isKindOfClass, isMemberOfClass)
  - Object equality (isEqual)

NSString

- General-purpose Unicode string support
- Consistently used throughout Cocoa Touch instead of "char *"
- Without doubt the most commonly used class
- Easy to support any language in the world with Cocoa
- Constant strings are NSString instances

```
NSString * aString = @"Hello World!";
```

NSString

- Similar to printf, but with %@ added for object

```
NSString *aString = @"Johnny";
NSString *log = [NSString stringWithFormat: @"It's '%@'.", aString];
=> It's 'Johnny'.
```

- Also used for logging

```
NSLog(@"I am a %@, I have %d items.", [array className], [array count]);
=> I am a NSArray, I have 5 items.
```

NSString

- Often ask an existing string for a new string with modifications.

```
- (NSString *)stringByAppendingString: (NSString *)string;
- (NSString *)stringByAppendingFormat: (NSString *)string;
- (NSString *)stringByDeletingPathComponent;
```

- Example

```
NSString *myString = @"Hello";
NSString *fullString;
fullString = [myString stringByAppendingString:@" world!"];
=> Hello world!
```

NSString

- Common NSString methods.

```
- (BOOL)isEqualToString: (NSString *)string;
- (BOOL)hasPrefix: (NSString *)string;
- (int)intValue;
- (double)doubleValue;
```

- Example

```
NSString *myString = @"Hello";
NSString *otherString = @"449";
if ([myString hasPrefix:@"He"]) {
    // true
}
if ([otherString intValue] > 500) {
    // false
}
```

NSMutableString

- NSMutableString subclasses NSString.
- Allows a string to be modified.
- Common NSMutableString methods.

```
+ (id)string;
- (void)appendString: (NSString *)string;
- (void)appendFormat: (NSString *)format, ...;
NSMutableString *newString = [NSMutableString string];
[newString appendString:@"Hi"];
[newString appendFormat:@", my favorite number is: %d",
    [self favoritNumber]];
```

Collections

- NSArray – ordered collection of objects
- NSDictionary – collection of key-value pairs
- NSSet – unordered collection of unique objects
- Common enumeration mechanism
- Immutable and mutable versions
  - Immutable collections can be shared without side effect
  - Prevent unexpected changes
  - Mutable objects typically carry a performance overhead

NSArray

- Common NSArray methods

```
+ arrayWithObjects: (id)firstObject, ...; // nil terminated!
- (unsigned)count;
- (id)objectAtIndex: (unsigned) index;
- (unsigned) indexOfObject: (id) object;
```

- NSNotFound returned for index if not found

```
NSArray *myArray = [NSArray arrayWithObjects:@"Red",
@"Blue", @"Green", nil];
if ([myArray indexOfObject:@"Purple"] == NSNotFound) {
    NSLog(@"No color purple");
}
```

- **Be sure to include the nil termination!**

NSMutableArray

- NSMutableArray subclasses NSArray
- Common NSMutableArray methods

```
+ (NSMutableArray *) array;
- (void)addObject: (id)object;
- (void)removeObject: (id)object;
- (void)removeAllObjects;
- (void)insertObject: (id)object atIndex: (unsigned)index;
NSMutableArray *array = [NSMutableArray array];
[array addObject:@"Red"];
[array addObject:@"Green"];
[array addObject:@"Blue"];
[array removeObjectAtIndex:1];
```

NSDictionary

- Hash table. Look up objects using a key to get a value.
- Common NSDictionary methods

```
+ dictionaryWithObjectsAngKeys: (id)firstObject, ...;
- (unsigned)count;
- (id)objectForKey: (id) key;
```

- nil returned if no object found for given key

```
NSDictionary *colors = [NSDictionary
dictionaryWithObjectsAndKeys:@"Red", @"Color 1", @"Green",
@"Color 2", @"Blue", @"Color 3", nil];
NSString *firstColor = [colors objectsForKey:@"Color 1"];
if ([colors objectForKey:@"Color 8"]) {
    // won't make it here
}
```

NSMutableDictionary

- NSMutableDictionary subclasses NSDictionary
- Common NSMutableDictionary methods

```
+ (NSMutableDictionary *) dictionary;
- (void)setObject: (id)object forKey: (id)key;
- (void)removeObjectForKey: (id)key;
- (void)removeAllObjects;

NSMutableDictionary *colors = [NSMutableDictionary dictionary];
[colors setObject:@"Orange" forKey:@"HighlightColor"];
```

NSSet

- Unordered collections of objects
- Common NSSet methods

```
+ setWithObjects: (id)firstObject, ...; // nil terminated
- (unsigned)count;
- (BOOL)containsObject: (id) object;
```

NSMutableSet

- NSMutableSet subclasses NSSet
- Common NSMutableSet methods

```
+ (NSMutableSet *) set;
- (void)addObject: (id)object;
- (void)removeObject: (id)object;
- (void)removeAllObjects;
- (void)intersectSet: (NSSet *)otherSet;
- (void)minusSet: (NSSet *)otherSet;
```

Enumeration

- Consistent way of enumerating over objects in collections
- Use with NSArray, NSDictionary, NSSet, etc

```
NSArray *array = [NSArray arrayWithObjects:Person1, Person2,
                  Person3, nil]; // assume an array of People objects
Person *person;
int count = [array count];
for (i = 0; i<count; i++) {
    person = [array objectAtIndex: i];
    NSLog([person description]);
}
// or
for (Person *person in array) {
    NSLog([person description]);
}
```

NSNumber/NSValue

- NSNumber
  - In Objective-C, you typically use standard C number types
  - NSNumber is used to wrap C number types as objects
  - Subclass of NSValue
  - No mutable equivalent!
  - Common NSNumber methods

```
+ (NSNumber *) numberWithInt: (int)value;
+ (NSNumber *) numberWithDouble: (double)value;
- (int)intValue;
- (double)doubleValue;
```

- NSValue
  - Generic object wrapper for other non-object data types

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

NSData/NSMutableData
NSDate/NSCalendarDate

- NSData/NSMutableData
  - Arbitrary sets of bytes
  - Used to save/restore/transmit data throughout the iOS SDK
- NSDate/NSCalendarDate
  - Times and dates
  - Used to find out the time right now or to store past or future times/dates
  - See also, NSCalendar, NSDateFormatter, NSDateComponents

Property List

- The term "Property List" just means a collection of collections, 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.

```
// plist is NSArray or NSDictionary
[plist writeToFile: (NSString *)path atomically: (BOOL)];
```

Getting Some Objects

- Use class factory methods
  - NSString +stringWithFormat
  - NSArray +array
  - NSDictionary +dictionary
- Or any method that returns an object except alloc/init or copy

References

- Lecture 2 & 3 Slide from iPhone Application Development (Winter 2010) @Stanford University