CS 474—Spring 2013
Introduction to Objective C (Part II)

Ugo Buy
Dept. of Computer Science
University of Illinois at Chicago

February 27, 2014

Objective C: History

- Hybrid object-oriented language blending ANSI C with Smalltalk
  - Defined in early 1980's by Brad Cox and Tom Love at Stepstone
  - Popularized by Steve Jobs at NeXT
  - Moved to Apple when Jobs returned from NeXT
  - Main language of OSX and iOS, and their APIs, Cocoa and Cocoa Touch
  - Consequence: Native language of iPhone, iPad, iPod apps

Objective C: Basic structure

- A strict superset of ANSI C, with addition of Smalltalk model for object-oriented paradigm
- Resulting syntax looks weird
  - C code with Smalltalk expressions freely interspersed into it
  - Additional constructs for specifying memory management
- We'll cover the basics, enough to write iOS apps
  - Knowledge of ANSI C is assumed here too

Objective C vs. Smalltalk

Left from Smalltalk:

1. Classes, variables, methods and messages
   - Same model as in Smalltalk
   - Instance variables sometimes called *iVars* or *member variables* (similar to C++)
2. Single (vs. multiple) inheritance
   - Class hierarchy rooted in Object class, called NSObject here (unlike C++)
   - No interfaces (unlike Java), but something similar called *protocols*

Objective C vs. Smalltalk

Left from Smalltalk:

3. Class objects (as in Smalltalk)
   - But don't worry about the metaclasses this time; they exist but hidden… phew!
4. The class pointer
   - Here it is called the isa variable present in every instance
   - Consequence: Every instance "knows" its data type
   - But: Possible to restrict identifier to certain class

Objective C vs. Smalltalk

Left from Smalltalk:

5. Strict information hiding
   - Only receiver's variables accessed in a method
   - Need calls to accessor/modifier methods to get to another object's variables
6. Message expressions
   - Unary and keyword expressions, e.g.,

     [bigFrame width: 100.0 height: 50]
   - Use C constructs for binary expressions

Objective C vs. Smalltalk

Left from Smalltalk:

7. Dynamic binding of messages and methods (message polymorphism)
   - Method executed depends on class of receiver
8. Message dispatching
   - First, follow isa (class) pointer
   - Look up dispatch table (similar to method dictionary)
   - Follow superclass pointer, if method not found
   - Implementation note: This search performed by C function objc_msgSend()

7

Objective C vs. Smalltalk

Left from Smalltalk:

9. Dynamic function linking
   - New methods can be added dynamically as program is running
   - Done with C function class_addMethod() and method resolveInstanceMethod: (resolveClassMethod:)
10. Dynamic class loading
   - New classes can be added dynamically at run-time
   - Done either with C function objc_loadModules() or NSBundle class

8

Objective C vs. Smalltalk

Left from Smalltalk:

11. Inheritance model
   - Subclasses inherit all methods and variables (both instance and class) from superclass (transitively)
   - Identifiers self and super (for method refinement)
   - Abstract classes (but no convenient way to specify, such as subclassResponsibility)
   - Class names as type identifiers
   - Run-time type checking: Methods isMemberOfClass: and isKindOfClass:

9

Objective C vs. Smalltalk

Left from Smalltalk:

12. Class initialization
   - RT system sends message initialize to every class object when class object created
   - This is identical to class message initialize in Smalltalk, used to initialize class variables there
   - Do not confuse with instance message initialize in Smalltalk (used to initialize instances, not classes)

10

Objective C vs. Smalltalk

Different from Smalltalk:

1. Statically typed as a whole
   - All identifiers have a data type
   - ANSI C code: Typing as usual in C
   - Smalltalk code: All objects have a special data type called id

     id anObject
   - Replace int as default data type for class instances
   - Can also use class identifiers as data types

11

Objective C vs. Smalltalk

Different from Smalltalk:

2. Syntax for creating classes and methods
   - Done textually, no VW IDE
3. Protocols
   - Similar to Java interfaces, don't confuse with Smalltalk protocols (i.e., synonymous with APIs)
4. Properties (new feature for declaring instance vars)
5. Associative references (new feature)
6. Static behavior (new feature)

12

Objective C vs. Smalltalk

Different from Smalltalk:

7. Memory management by reference counting
   - Hybrid solution between C++ (entirely programmer managed) and Smalltalk/Java (entirely garbage collected)
   - Also, support for weak references
8. No user-defined class variables, use file-scope variable in .m file
   - Optimal use: Declare those variables static, and define accessors and modifiers as class methods

13

Objective C vs. Smalltalk

Different from Smalltalk:

9. Messages sent to nil have no effect (instead of causing RT error)
10. alloc instead of new
11. Class names are in the same space as global variable names
    - Can't use same identifier for a variable and a class

14

Class definitions

- Two components, **interface** and **implementation**, usually in different text files
  - Suffixes: .h for header and .m for code files
- Syntax of class interface (.h file):

```
@interface ClassName : SuperclassName
    // Method and property declarations
@end
```

- Declare a new class, its superclass, methods, variables and properties

15

Example of class definition

- Header for Stack class:

```
@interface Stack: NSObject
    // Method and property declarations
@end
```

16

Method declaration

- **Methods must be declared in class interface (both for class and instance methods)**
- Declaration syntax (.h file) depends on whether class or instance method declared
  - Instance methods: Declaration starts with a minus sign —
  - Class methods: Declaration starts with a plus sign +

17

Method declaration (continued)

- Syntax depends on whether unary or keyword method
  - Unary: ± (*returnType*) *methodName* ;
  - Keyword methods add parameter list:

    ± (*returnType*) *keyword_parameter_list* ;
  - Parameter list: Blank separated list of triples:

    keyword: (*argumentType*) *parameterName*
- Caveats:
  - Return types and argument types are optional
  - Default is id

18

Method declaration (continued)

- Examples of method declarations for Stack class
  - — (id) pop ;
  - — (id) push: (id) item ;
  - + (Stack*) new: (int) initialSize ;
- Additional examples
  - — (void) setWidth: (float) newX height: (float) newY ;
  - + (Stack*) new: (int) initialSize
- Class methods, instance methods and instance variables can have the same name (as in Smalltalk)

19

Variable declarations

- Old syntax (discontinued): sequence of declarations in curly braces
- New syntax: Declared with @property clauses in header file and generated automatically by compiler
- Will see syntax later when discussing properties

20

Visibility of declarations

- Declarations in an interface (.h) file must be imported in files using those declarations
- ANSI C's #include directive is still available
- New #import directive avoids multiple inclusion problem of #include directive
- Example code

```
#import "Queue.h"
…
@interface PriorityQueue : Queue
…
```

21

Class declarations

- Use class identifier without importing interface declaration
- New syntax: @class declaration
- Example:

```
@class Queue, Stack ;
```

- Semantics: Allow use of class identifier in file containing @class declaration without importing (e.g., for type declarations)
- However, importing needed when creating instances or sending messages, etc.

22

Class implementation

- Syntax: Declared with @implementation … @end clauses (typically in .m file)

```
@implementation ClassName
{
  // variable declarations
}
// method definitions
@end
```

- Omit braces if variables declared only with @property clauses in .h file
- Must import class definition (from .h file)

23

Access levels for instance variables

Four access levels for variable identifiers:

1. @private — accessible only inside defining class
2. @protected — accessible inside defining class and subclasses
3. @public — accessible everywhere
4. @package — accessible inside defining class, and in classes within the same package (image) as the defining class

Default is @protected

24

Syntax of access levels

Keyword introduces sections of declarations at same access level, e.g.,

```
@interface Person: NSObject
{
        NSString* name ;
  @private
        long ssn ;
        id medicalHistory ;
  @public
        Person* emergencyContact ;  } …
```

25

Method definitions

- Method definition = Method header + body
- Header: Same as in method declaration, except omit semicolon at end of declaration while defining method
- Body: C + Smalltalk syntax enclosed in curly braces
- Example (push method for Stack class):

```
— (Stack*) push: (id) anElement {
      items = [items add: anElement] ;
      return self ;
}
```

26

A class example: Date

```
@interface Date: NSObject
  @property int day ;
  @property int month ;
  @property int year ;

  — (BOOL) leapYear ;
  +   (id) newDay: (int) day month: (int) month year: (int) year ;

@end
```

27

A class example: Date

```
@implementation Date
  @synthesize day ;
  @synthesize month ;
  @synthesize year ;

  — (BOOL) leapYear  {
        if (year % 4 != 0) return false ;
        if (year % 100 != 100) return true ;
        if (year % 400 == 0) return true;
        return false ; }
          …
@end
```

28

self and super

- Same behavior as in Smalltalk
- Both denote original receiver
- self starts method search in receiver's class
- super starts method search in superclass of class containing method that uses identifier super
- Only difference: Can rebind self (?!)
  - Example: self = [Stack new: anObject] ;
  - Seems like a really bad idea! Please don't do this…

29

Protocols

- Protocol = Group of method declarations, not associated with any particular class
- Protocol can be "implemented" by one or more classes
- Class implementing protocol *P* is said to **adopt** *P*
- Important: Protocol identifier can be used as a type identifier
  - Note similarity with Java **interface** construct

30

Protocols

- Useful in various situations
  1. To define common APIs for classes not related by inheritance
  2. To declare interfaces to remote objects
  3. To hide class hierarchy from clients of a given class

31

Protocol syntax

- Syntax of protocol definition

```
@protocol protocolName
  // method declarations (as in @interface definition)
@end
```

- Example of protocol definition

```
@protocol Printable
   — (OutStream*) printOn: (OutStream) aStream ;
   — (String*) printString ;
@end
```

- Typically defined in .h files

32

Adopting a protocol

- Class that implements protocol *P* must declare *P* in its header

```
@interface className : superclassName <protocol_list>
…
@end
```

- Example

```
@interface Student : Person < Printable, Formattable >
```

- Important: Adopting class must provide definitions for all required methods declared in all adopted protocols

33

Conforming to a protocol

- A Class conforms to a protocol either if:
  1. It adopts the protocol, or
  2. One of its superclasses adopted the protocol
- Method conformsToProtocol: tests whether an object belongs to a class that conforms to a given protocol
  - This is similar to isKindOfClass: method of Objective-C and isKindOf: method of Smalltalk

34

Using a protocol

- Declaring an identifier of a protocol class

  Type declaration uses angle bracket notation again:

  type_id <protocol_list>
- Example

```
id <Printable> anObject ;
```

- Forward protocol declaration (useful when two protocols are mutually referential)
- @protocol *protocol_name*

35

Properties

- Way to declare instance variables with automatic (but controlled) generation of accessor and modifier methods
- Syntax (simplest form):

  @property *typeName variableName*
- Example:

```
@property id items ;
```

- Declared in **@interface … @end** construct

36

Properties

- Semantics: New instance variable declared, along with accessor and modifier method, by **@synthesize** directive in implementation file
- Example: Following declaration…

  @property id items ;

  …automatically generated methods:

  — (id) items ;
  — (void) setItems: (id) newValue ;
- By default accessor for property prop is unary method prop and modifier is setProp:

37

Property attributes

- Modify behavior of property declarations
- Syntax: Comma-separated list of attribute specifiers, between parentheses and before list of variables

  @property (attr1, attr2, …) *typeName variableName*
- Attribute could be keyword or key/value pair
- Example:

  @property (weak) int total;

  @property (getter=getItems) id items ;

38

Examples of relevant properties

- **setter=*methodName***
- **getter=*methodName***
- **readonly** (no setter defined)
- **readwrite** (default)
- **strong** (default)
- **weak** (var reference is weak)
- **nonatomic** (non default, relax support for multithreading)

Weak Reference: Not counted for GC purposes, object is deallocated if accessible only through weak references

39

Properties in implementation files

- **@synthesize**—Tells compiler to generate automatically getter and setter methods for specified properties
- Syntax: Two forms

  @synthesize property1, property2, … ;

  @synthesize property1=varName1, … ;
- Examples:

  @synthesize name, dateOfBirth, socialSecurityNumber ;
  @synthesize dateOfBirth=dob ,

  @synthesize socialSecurityNumber = ssn ;
- Optional: specify name of iVar for property

40

Categories and class extensions

- Categories support addition of new methods to existing classes (good to make methods private)
- Syntax of declaration is similar to class definition, but with category name in parentheses (and without superclass name)

  @interface className (categoryName)

  // method declarations

  @end
- Implementation consists of two files, categoryName.h and categoryName.m (with method implementations)

41

Associative references

- Allow addition of instance variables to existing classes
- Cumbersome syntax
- Beyond our scope

42

Static typing

- Goal: Enhance language efficiency by limiting dynamic typing features
- Obtained by restricting the type of an identifier to something less general than id
- Syntax: Use pointer to class name for statically-typed identifier, e.g.,

  ClassName variableName* ;
- By **identifier polymorphism** instances of subclasses of ClassName can also be bound to *variableName*

43

Example of static typing

- Declare identifier of class Person

  Person* aPerson;
- Now only instances of Person class or its subclasses can be bound to aPerson
- This does not change implementation of objects bound to aPerson
- However, compiler attempts to enforce type correctness, e.g., by checking that only messages understood by Person instances are sent to aPerson's referent

44

More on static typing

- Person methods are still dynamically bound, e.g.,

```
Shape* aShape;
aShape = [[Rectangle alloc] init] ;
// Invoke Rectangle isFilled, not Shape isFilled
BOOL solid = [aShape isFilled] ;
```

- This behavior allowed only for methods defined in superclass and refined in subclass
- Error if rectangleMethod defined in Rectangle class

```
BOOL solid = [aShape rectangleMethod] ;
```

- Compiler thinks that only Shape objects bound to aShape

45

Memory management

- Until recently, hybrid approach between C/C++ (programmer controlled) and Smalltalk/Java (garbage collected)
- Basic language uses standard messages for allocating and deallocating objects
- Example:

```
id aPerson = [[Person alloc] init ] ;
[ aPerson dealloc ] ;
```

46

Reference counting

- Foundation framework adds reference counting
- All objects have a reference count
- All objects start with a count of 1
- Programmer invokes methods for incrementing and decrementing an object's reference count

```
— retain       // Increments an object's count
— release      // Decrements an object's count
— retainCount  // Answers current reference count
```

- Object destroyed when its reference count drops to zero

47

More on reference counting

- Programmer is still responsible for maintaining reference count
- Typically references that are automatically deallocated are not retained or released
  - Example: Stack allocated objects (bound to method parameters or local variables)
- Instance variables and global variables are so-called *owning* references; must be retained or released

48

Example with reference counting

- Example of code

```
— (void) setStringValue: (NSString*) aString   {
      [string release] ;
      string = [aString retain] ;   }
```

- Problem with code if string == aString
- Corrected code:

```
— (void) setStringValue: (NSString*) aString {
     id tmp = [aString retain] ;
     [string release] ;
     string = tmp ;   }
```

49

Automatic Reference Counting (ARC)

- Newest compilers (Lion OS, 10.7, and beyond) insert retain and release calls automagically for you
- Supported in Xcode 4.3 and above
- Never worry about inserting retain and release calls in your code

50