Objective C 2.0
Programming Language

Final Exam
C SC 520
(Principles of Programming Languages)

Report prepared by:
Bhavin Mankad
Karthik Ravichandra

Table of Content

1. History

With languages like Simula and SmallTalk, the importance of Object Oriented Programming was becoming apparent in early 1980s. Object Oriented Programming made development and maintenance of large scale projects a lot easier as compared to Structured Programming. Also, thinking in Object Oriented terms was closer to real world situations. Brad Cox, working in a company called StepStone was much interested in Software Reuse and Object Oriented development and was greatly influenced by simplicity and Object Oriented features of SmallTalk. In early 1980s, while working at StepStone, he incorporated SmallTalk like Object Oriented features to C compiler and developed a new language called Objective-C. In 1986, Cox put much of the description of his new language in a book named 'Object Oriented Programming, An evolutionary approach'.

The company started by Steve Jobs, NeXT, licensed Objective-C from StepStone and released their own version of compiler and APIs in 1988. NeXTStep, their Object Oriented multi-tasking operating system was based on Objective-C. OpenStep was developed by NeXT in partnership with Sun Microsystems.

Work on GNU's version of NeXTstep clone started in 1992 and it was given the name 'GNUstep'. The first compiler written by Dennis Glatting in 1992 was followed by another version written by Richard Stallman.

After NeXT was acquired by Apple, the OpenStep was used in their operating system Mac OS-X. Apple's one of the most important frameworks Cocoa is developed in OpenStep interface objects. Thus Apple is the single most influential force behind existence of Objective-C in the market place today.

2. Design Objectives

a. Object Oriented Programming
   The primary design goal of Objective-C was Object Oriented Programming. The Object Oriented syntax is much influenced by SmallTalk.
b. Simple extension to C
   The language is a strict super-set of C. Thus it is backward compatible with C. It was designed to add Object Oriented support to C. Only those Object Oriented concepts which were absolutely essential were added into the language. This has made the language much simpler to learn and program in.
c. Light Weight
   The language does not need any run-time platform like Virtual Machine for Java. Language Pre-processor translates this syntax into C source code. The language just adds a SmallTalk style messaging and some syntactical sugar for defining and working with objects in C language's environment. This keeps the language light weight at run time. The language does not have additional syntactical and conceptual quirks like virtual functions, friend functions, templates, multiple inheritance etc in C++.

d. Flexibility
As Objective-C is a strict superset of C, it can support pure structural programming as well. This gives flexibility to programmers who may want to use structural programming like C for systems level code and OO programming for application development.

e. Dynamic Behavior
Objective C is the most dynamic of all OO languages. It defers most of the decisions till run time. It supports Dynamic Typing, Dynamic Binding and Dynamic Loading. These concepts are further explained in section 6.

f. Reflection
Objective C supports reflection. It can observe and modify its structure at runtime using the 'Class' object methods and also paradigms like 'selector'. The concept is similar to Reflection APIs in Java.

3. Interface and Implementation

In Objective C, the code is organized in two types of source files. Interface files declare a class with its instance variables and methods. Implementation files contain the implementation of these methods. The objective behind this separation is to achieve *'Data Abstraction'* – an important concept in Object Oriented Programming. Interface describes *contract* for the given class stating which methods (messages) this class is responsible for responding to and what data it holds. This hides the implementation details and exposes only the contract to other classes which use this class. The users of this class don't have to worry about implementation. Thus any updates in a class method's implementation will not affect the user classes.

Interface

- This declares a class named Rectangle which inherits from the class Shape
- length and width are instance variables. Instance variables are declared within { }
- Methods are declared outside the { }
- In Method declaration, (+) means it's a class method and (–) means it is an instance method. First ( ) declares return type of the method. For example (void) or (float). Then comes the method name followed by ':' incase the method has arguments. Arguments follow the ':' with types given in ( )

Rectangle.h

```
@interface Rectangle : Shape
 {
   float length;
   float width;
 }
 + (void)alloc;
 -(float) setLength: (int)len  andWidth: (float) width;
@end
```

Implementation

- Implementation file contains definition of methods for the Rectangle class
- Similar to the interface, (+) shows it is a class method and (–) shows that it is an instance method

Rectangle.m

```
#import "Rectangle.h"
@implementation Rectangle
 + alloc
 {
implementation
 }

 - (float) setLength: (int)len  andWidth: (float) width
 {
   // implementation
 }
 @end
```

4. Class and Object

Class

These are the building blocks of any Object Oriented language. Objective C defines classes in the interface header files as explained in the previous section. For example in the previous section, *Rectangle.h* file defines a class named *Rectangle*. Each class can have instance and class variables and methods.

Method definitions for a class are provided in an implementation file such as *Rectangle.m* shown above. Implementation file imports the header file with the statement

```
#import "Rectangle.h"
```

#import is different from #include in that it does not import the content of the header file again if it is already imported unlike include which does not put such a check

Instance and Class variables

By default, the class has access to all its instances variables. No special access specifiers like a '.' or '→' is needed.

Class variables are declared as 'static' variables. Static variables can not be inherited by sub-classes. These variables retain their values across multiple objects and they can be manipulated by class level methods.

Scope of Instance Variables

Instance variables have scope of either private, public or protected. They have exactly the same meaning as in other OO languages like Java. Private variables can be accessed only from within the class that defines it, protected

variables by the class it defines and its subclasses and the public variables can be accessed by any class

Objects

Classes are instantiated as *objects.* This is when memory for instance variables is allocated. Objects are declared either with dynamic or static type. For example

```
id rectangle;                  //declares a variable called rectangle which can point to an
                               //instance of any class at run time
Rectangle *rectangle;  //declares a variable called rectangle
                               //which can point to only the
                               //objects of class Rectangle or its subclasses
```

Allocation and Initialization

```
Rectangle *rectangle;
rectangle = [Rectangle alloc];

if (rectangle)
        rectangle = [rectangle init];
```

The code given above shows two important steps in instantiation of objects. The first line declares an object of type Rectangle class. The next line allocates memory for this object – in other words actual instantiation happens here. We send 'alloc' message (a method call in other words) to Rectangle class and it returns a pointer to an object of its type. This object is then passed a message (message passing explained in later sections) 'init' which initializes the instance variables.

5. Inheritance

Another important concept of OO programming is implemented using keyword 'extends' as shown in the example below.

As figure below shows, Rectangle is the class which inherits from the class Shape. It provides convenient way of reusing code. All the public and protected member variables and methods of Shape are now part of Rectangle too. A class can override a method that belongs to the parent class by providing its own implementation.

Multiple inheritance is not allowed in this language just like Java and unlike C++. A class can only inherit from one class. *NSObject* is the root class of all the classes. In other words, any class that a user defines, normally derives from *NSObject*. It is a base implementation for all objects and provides important instance variables and methods useful for all the classes that derive from it. The most basic functionality of *NSObject* class is to allocate instances and connecting instances to their classes.

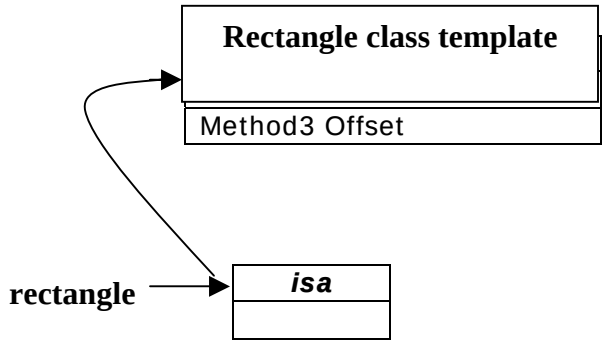
NSObject defines the ***'isa'*** instance variable that connects every object with its class. When an object is allocated, its '***isa'*** is made to point to its class

template. This template defines the offsets of ***variables*** and ***methods*** into memory and code section of a program respectively.

```
@ interface Rectangle : Shape

        //declarations here

@end
```



6. Dynamic Behavior

In languages like C++ and C, many decisions related to types and method calls are taken at compile time. In Objective-C most of these decisions are taken dynamically at runtime. This gives a lot of flexibility to programs. There are mainly three types of dynamic behavior in Objective-C.

Dynamic typing

The class (type) of an object is decided at runtime. The compiler does not associate an object variable with its class type statically. There are two types of dynamic typing available. One displayed by Dynamic Polymorphism and the other displayed by a type called 'id'. Dynamic polymorphism allows a variable, at runtime, to point to any object of its own type or any of its subclass types as shown in the first line in the code below. The second line in the code shows id type. Here *obj* is defined as of type *'id'* and at runtime, it can point to an object of any class type.

```
Parent *parent = [[Child alloc] init];
id obj = [[SomeClass alloc] init];
```

Dynamic binding

Dynamic typing facilitates binding of methods with the objects at runtime. A variable can point to any object at run time and therefore a method is bound to the class only at run time. As shown in example below a receiver can be of any class type at runtime and therefore which 'aMethod' to invoke is decided only at runtime.

```
[receiver aMethod]
```

Dynamic loading

Objective-C runtime can dynamically load classes while a program is running. It is possible to create methods on the fly and load them to the program at runtime. These features adds flexibility to programs. All the modules do not have to be statically loaded at compile time. Modules can be loaded as and

when needed. Also, a developer need not have classes written by other developers at compile time. The interface can be negotiated between developers and classes can be loaded at runtime when needed.

7. Messaging

In Objective C's terminology, methods belonging to classes are called 'messages' and 'calling methods' is termed as 'passing messages'.

```
[rectangle init]                                  //No arguments

[rectangle initWithLength: 80 andWidth: 90];      //With two arguments
```

As shown in figure 7, we are passing '*init*' message to a '*rectangle'* object. In the first case, the message is passed without any arguments. In the second case message is passed with two arguments; Length and width. The name of the implemented method in Rectangle class is

```
- (void) initWithLength: (float) len andWidth: (float) width;
```

Dynamic Binding of method

The messages are bound to the objects at runtime. Internally, message passing is converted into a call to a routine named *objc_msgSend (receiver, selector,…).* This routine is called at runtime with at least two arguments. 'receiver' is the object that is passed the message to and 'selector' is the name of the message (method).

Implementation detail of message passing

When a message is sent to an object, it follows *'isa'* field of the object to locate the class template. Once it has the correct class template it locates the current message in the 'method offset table' and finds its offset into the code section. It then moves the program counter to that instruction and starts executing the method.

Forwarding

Because there is no check at compile time whether an object can respond to a message passed to it, it is runtime's responsibility to deal with a situation when an object is passed a message it doesn't respond to. In such cases, the runtime gives an object a chance to decide what to do with such a message that it has received. It sends the object a *forwardInvocation*: message with an *NSInvocation* object as its argument The *NSInvocation* object encapsulates the original message and the arguments.

Program can implement a *forwardInvocation*: method to give a default response to the message, or to avoid the error in some other way. As its name implies; *forwardInvocation*: is commonly used to forward the message to another object if the receiver object does not want to handle that message.

8. Exception Handling

Exception Handling in Objective C is very similar to C++ and Java. There are 4 compiler directives, *@try, @catch, @throw, @finally*, which form the crux of exception handling in Objective-C. Code that could possibly raise an exception is enclosed in the @try block. The @catch is used to handle the exception generated in @try block. The @finally block is used for cleanup immaterial of an exception being raised or not. The @throw block is used to throw back an exception which can be caught in subsequent @catch blocks. Since the exception handling is very similar to Java and C++, we will not discuss this in detail in this report.

9. Categories

Categories are an easy way to add methods to a class, even to the ones to which you don't have the source. They are a powerful feature of extending the functionality of a class. In other words, categories can sometimes be a good alternative to sub-classing, and can avoid sub-classing when it is done just for the sake of extending a class with new methods. It is important to note that, you can only add new "methods" to a class. No new instance variables can be added to a class. The syntax for declaring a Category is:

```
#import "ClassName.h"

@interface ClassName ( CategoryName )

// method declarations

@end
```

As we see above, the declaration of a category is very similar to a class interface declaration, except that a "Category name" is listed after the class name. Once declared, the implementation is defined as:

```
#import "CategoryName.h"

@implementation ClassName ( CategoryName )

// method definitions

@end
```

Once new methods are added to a class through categories, all the sub-classes can now access the new methods as if they were part of the original class interface.

Categories can also be used to split the implementation of a single huge class across many source files, and may simplify code management.

Another big advantage of Categories is that, the base class code doesn't have to be re-compiled at all. Since, we are only adding new methods to an

existing class, the only ones to be recompiled are those new methods that we added.

10. Properties

Properties is another useful feature that was introduced as part of Objective-C 2.0. As we have seen earlier, the standard way of accessing class variable is through messages. Properties provide a concise means of accessing class variables. Properties can be accessed using the dot-syntax. There are 2 parts to using Properties - Declaration and Implementation.

Properties are declared with the help of the @property compiler directive.

The property implementation uses the @synthesize and @dynamic directives to tell the compiler how to handle the actions associated with a property.

With @synthesize, the compiler will generate the setter and getter methods for the property. With @dynamic, the programmer will have to supply the getter and setter methods with the help of getter= and setter= attributes.

Properties also allow the programmer to associate *readwrite* and *readonly* attributes with a class variable. Below is an example, which illustrates the use of Properties in Objective-C.

```
@interface Square : NSObject{
   ....
   @property(readwrite) int size;
   ....
}

@implementation Square : NSObject
   ....
   @synthesize size;
}
```

The above piece of code, declares "size" as a property, and directs the compiler to generate the getter and setter methods automatically. Further, the size variable can now be accessed as

```
Square*sq = [[Square alloc] init];
sq.size = 10; // This is the equivalent of saying [sq setSize: 10]
```

11. Memory Management

Objective-C provides two options for memory management. Until the 2.0 version, the memory had to be managed manually by the programmer. Objective-C has compiler directives like "release", "retain", "autorelease" that we will discuss below, to help the programmer manage memory. With Objective-C 2.0, the concept of a garbage collector was introduced, which automates the task of memory management. Garbage Collection uses the concept of Generation-based collection.

Objective-C uses a reference counting mechanism to keep track of the liveness of an object. This count is called the retainCount. When an object is allocated, using the alloc method, the retainCount is set to 1. A release method is used to decrement the retainCount by 1. When the retainCount reaches 0, the object is dealloced. If the object is referenced in the program, the programmer can indicate this to the compiler using the retain message. The retain message will increment the retainCount by 1.

Consider the example below:

```
Rectange *rect = [[Rectangle alloc] init];     // retainCount is 1.

Rectangle *tmprect = rect;                     // Now there is another reference to rect.

[rect retain]                                  // So we increment the retainCount

[rect release]

[tmprect release]                              // the retainCount is now 0 and dealloc is called.
```

The retainCount of an object can be obtained by passing it a "retainCount" message.
The *autorelease* message is similar to release, except that it defers the release of the object to a later point in time. *autorelease* is generally used in conjunction with Autorelease Pools. These pools contain other objects that have received an "*autorelease*" message. Thus sending an *autorelease* to an object instead of a release extends the lifetime of the object till the Pool itself is released. An *autorelease pool* is created as an instance of a NSAutoreleasePool Object. Consider the example below:

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
Rectangle *rect;
for (i=0;i <10; ++i){
   rect = [[Rectangle alloc] init];
   [rect autorelease];
}
[pool release];
```

In the above example, the 10 instances of Rectangle are released only when the pool is released.
One of the biggest disadvantages of reference counting mechanism is the problem of reference cycles. Consider the example below

```
[[tmpA setNext] tmpB]     // tmpA->next = tmpB
[tmpB retain]             // increment number of references to B.

[[tmpB setNext] tmpA]     // tmpB->next = tmpA
[tmpA retain]             // increment number of references to A.
```

Object tmpA has a reference to tmpB, and vice-versa. Hence the retainCount on both these objects is 1. However, neither of these objects can be reached

from the root set. So, ideally these objects must be released. But the reference counting mechanism fails to address this scenario.

12. Interesting Facts about Objective-C

- Interest in Objective-C is still very much alive due to Apple's endorsement of the language.
- Games like NuclearAttack and Quake are developed in Objective-C.
- Today there are four major Objective-C compilers: Stepstone, GNU, Apple and Portable Object Compiler (POC).
    - Stepstone and POC are preprocessors that that generate C code which is then compiled.
    - GNU and Apple act like compilers by creating intermediate code for the GNU code generator directly from the source.
- An Objective-C class allows a method and a variable with the exact same name.
- The language also allows same name for a class and an instance method.

13. Comparison with C++

Features	Objective C	C++
Binding	Dynamic	Static
Modifying a Class	Subclassing and Categories	Only Subclassing
Operator Overloading	No	Yes
Templates	Not Needed because of Dynamic Binding	Yes
Namespace	No	Yes
Inheritance	Single	Multiple
Language Complexity	Only a few syntactical additions to C	Lot of additions to C like virtual and friend functions, templates etc.
Reflection	In-built support for reflection	No in-built support
Typing	Dynamic	Static
Influenced By	SmallTalk and C	Simula and C
Compiler	Pre-processed or Compiled	Compiled

14. References

a. The Objective-C Programming Language 2.0. Cocoa framework from Apple Inc. http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf
b. Memory Management Guide for Objective-C. Cocoa Guide http://developer.apple.com/documentation/Cocoa/Conceptual/MemoryMgmt/MemoryMgmt.pdf
c. Beginner's Guide to Objective-C http://www.otierney.net/objective-c.html
d. Objective-C Wikipedia Article http://en.wikipedia.org/wiki/Objective-C