Session 2

Introduction to Objective-C

MODULE OBJECTIVES

At the end of this module, you will be able to:

- Demonstrate an understanding of Objective-C
- Explain memory management in Objective-C
- Describe advanced concepts in Objective-C

SESSION OBJECTIVES

At the end of this session, you will be able to:

- Explain the basics of Objective-C
- Declare classes and instancing objects
- Describe the concept of managing memory
- Describe the Model-View-Controller design pattern
- Provide an overview of blocks
- Explain the error handling patterns in Objective-C

INTRODUCTION

To develop an iPhone application, such as your **Bands** app, you must have a good knowledge of the iOS environment and its interface. The iOS application development requires a thorough knowledge of **Objective-C**. Objective-C is the programming language used to develop both **Mac** and **iOS applications**.

Objective-C is a compiled language, meaning that it gets compiled down to raw machine code as opposed to being interpreted at runtime. As its name implies, it is based on the C programming language. It is actually a superset of C that adds **object-oriented programming** (OOP) methodologies. Because it is a descendant of C, its syntax and concepts are similar to other C-based languages. In this session, you will revise the basics of Objective-C by comparing it to Java and C#.

EXPLAINING THE BASICS OF OBJECTIVE-C

Objective-C was developed in the early 1980s by a company called **Stepstone.** The company was working on a legacy system built using C, but wanted to add reusability to the code base by using objects and messaging. The concept of object-oriented programming (OOP) had been around for a while. During that period, the Smalltalk language developed by Xerox was the most prominent object-oriented language in use. Objective-C got its start by taking some of the concepts and syntax of Smalltalk and adding it to C. This can be seen in the syntax of message passing in Objective-C and Smalltalk.

Message passing or **method calling** is a method in OOP languages in which an object can send a message to or call a method of another object. All object-oriented languages include this feature. The following code shows how message passing is done in Smalltalk, Objective-C, and Java and C#.

```
Smalltalk: anObject aMessage: aParameter secondParameter: bParameter
Objective-C: [anObject aMessage:aParameter secondParameter:bParameter];
Java and C#: anObject.aMessage(aParameter, bParameter);
```

BIG Picture

Objective-C is very different from C. The code of both languages look similar and many apps combine elements of both languages. However, using Objective-C makes you think about the design of your app in a more abstract way, which is less tied to computer hardware. Objective-C is also a much looser language than C. You can use its features to change how your app responds to events as it runs. With Objective-C, the goal is not just the code that works, but an intelligent code with a wide range of possible behaviors. You can use Objective-C to create a simple, linear code. Its powerful features can help you add smart features to your apps.

In every programming language, **primitive types** are the basic types used to build more complex objects and data structures. They are typically the same from language to language. The primitive types in Objective-C are the same as those in C and most C variants. **Table 1** lists these types.

Table 1: Primitive Data Types in Objective-C

Data Type	Description
`bool`	Single byte representing TRUE/FALSE or YES/NO
`char`	Integer value the size of 1 byte
`short`	Integer value the size of 2 bytes
`int`	Integer value the size of 4 bytes
`long`	Integer value the size of 8 bytes
`float`	Single precision floating-point type the size of 4 bytes
`double`	Double precision floating-point type the size of 8 bytes

As with C and C++, Objective-C includes the `typedef` keyword. If you are coming from C# or Java background, and have never programmed using C or C++, you may not be familiar with `typedef`. It gives you a way of creating and naming a new data type using primitive data types. You can then refer to this new data type by the name you assigned it anywhere in your code.

Given below is a simple example of `typedef` that creates a new data type called `myInt`, which is the same as the primitive `int` data type.

```
// Objective C
typedef int myInt;
myInt variableOne = 5;
myInt variableTwo = 10;
myInt variableThree = variableOne + variableTwo;
// variableThree == 15;
```

A typical use of `typedef` in Objective-C is **naming a declared enumerated type**. An enumerated type, or `enum`, is a primitive type that is made up of a set of constants. Here, each constant is represented by an **integer**.

The enumerated types are used in pretty much every OOP language, including Java and C#. They give you a way to declare a variable with the enumeration's name and assign its value using the constant name. It helps to make your code easier to read. The four cardinal directions are often declared as an enumeration.

To use an `enum` in C, you would need to add the `enum` keyword before its name. Instead of this, you can `typedef` it, so you no longer need the `enum` keyword. You do not have to do this, but it is a common practice.

Another common practice in Objective-C is to **prepend the constants with the name of the enumeration**. Again, you do not have to do this but it helps in the readability of your code.

The following example demonstrates how you declare and `typedef` an `enum` in Objective-C as well as Java and C# (their syntax is identical) and how you would use it later in code.

```
// Objective C
typedef enum {
   CardinalDirectionNorth,
   CardinalDirectionSouth,
   CardinalDirectionEast,
   CardinalDirectionWest
} CardinalDirection;
CardinalDirection windDirection = CardinalDirectionNorth;
if(windDirection == CardinalDirectionNorth)
// the wind is blowing north

// Java and C#
public enum CardinalDirection {
   NORTH,
   SOUTH,
   EAST,
   WEST
}
CardinalDirection windDirection = CardinalDirectionNorth;
if(windDirection == NORTH)
// the wind is blowing north
```

In C, C++, C#, and Objective-C, you can also create a **struct**. Java does not have this because everything is a class. A struct is not a class. A **struct** is a compound data type made up of primitive data types, which are used to encapsulate similar data. Similar to `enums`, developers often use `typedef` to avoid having to use the keyword structs when using them in code.

There are three common structs in Objective-C:

- CGPoint
- CGSize
- CGRect

The following code shows how these structs are defined.

```
struct CGPoint {
   CGFloat x;
   CGFloat y;
};
typedef struct CGPoint CGPoint;

struct CGSize {
   CGFloat width;
   CGFloat height;
};
typedef struct CGSize CGSize;

struct CGRect {
   CGPoint origin;
   CGSize size;
};
typedef struct CGRect CGRect;
```

An Overview of Object and Class

As in any other object-oriented language, objects in Objective-C are the building blocks of applications. They have **member variables** that describe the object, **methods** that can manipulate the member variables, and any parameters that may be passed to them. The member variables can be **public** or **private**.

Objects are defined in a **class**. The class acts as the template for how an object is created and behaves. A class in Objective-C consists of the following two files (much like classes in C++):

- **Header File (.h):** This file contains the interface of the class. This is where you declare the member variables and the method signatures.
- **Implementation File (.m):** In this file, you write the code for the actual methods.

The following code shows how to define a class and its implementation in Java.

```
package SamplePackage;
public class SimpleClass {
   public int firstInt;
   public int secondInt;

   public int sum() {
```

```
            return firstInt + secondInt;
    }

    public int sum(int thirdInt, int fourthInt) {
            return firstInt + secondInt + thirdInt + fourthInt;
    }

    private int sub() {
            return firstInt - secondInt;
    }
}
```

The following code shows how to define a class and its implementation in C#.

```
namespace SampleNameSpace
{
    public class SimpleClass
    {
            public int FirstInt;
            public int SecondInt;

            public int Sum()
            {
                    return FirstInt + SecondInt;
            }

            public in Sum(int thirdInt, int fourthInt)
            {
                    return FirstInt + SecondInt + thirdInt + fourthInt;
            }

            private int Sub()
            {
                    return FirstInt - SecondInt;
            }
    }
}
```

The following code shows how you declare a header file in Objective-C.

```
@interface SimpleClass : NSObject
{
    @public
    int firstInt;
    int secondInt;
}
- (int)sum;
- (int)sumWithThirdInt:(int)thirdInt fourthInt:(int)fourthInt;
@end
```

The following code shows how to define a class implementation in Objective-C.

```
#import "SimpleClass.h"
@implementation SimpleClass
- (int)sum
{
   return firstInt + secondInt;
}
- (int)sumWithThirdInt:(int)thirdInt fourthInt:(int)fourthInt
{
   return firstInt + secondInt + thirdInt + fourthInt;
}
- (int)sub
{
   return firstInt - secondInt;
}
@end
```

All three of these classes (in Java, C#, and Objective-C) are conceptually the same. Each class has the following items:

- Two integer member variables that are public
- A public method that adds the member variables together
- A public method that adds the member variables plus additional two integers passed in as parameters
- A private method that subtracts Y from X

The following sections discuss the key differences between classes in Java and C# compared to Objective-C.

ADDITIONAL KNOWHOW

The @ symbol is special in Objective-C, though it has no meaning in C. Because Objective-C is a superset of C, a C compiler can be modified to also compile Objective-C. The @ symbol tells the compiler when to start and stop using the Objective-C compiler versus the straight C compiler.

QUICK TIP

When creating your own classes, methods, and variables, it is a good idea to follow a couple of standard naming conventions. For example, class names (such as View) should start with a capital letter.

Classes in Objective-C versus Classes in Java and C#

This section compares the classes in Java and C# with the Objective-C classes. Objective-C differs from Java and C# through the following points:

1. There are no namespaces in Objective-C
2. Methods are visible to other classes in Objective-C
3. Most classes inherit from `NSObject` in Objective-C
4. Objective-C uses long and explicit signatures

Now let's consider each point in detail.

No Namespaces in Objective-C

A **namespace** is a way of grouping related classes together. C# uses `namespace` as the keyword for this concept, whereas Java uses packages to accomplish the same thing. The namespace and packages keywords in the C# and Java code, respectively, do not correspond to anything in the Objective-C code. This is because Objective-C does not have the concept of namespaces.

If a class in a namespace or package wants to use a class in another namespace or package, it needs to link to that namespace or package. In Java, this is done using the `import` keyword followed by the name of the package. In C# this is done with the `using` keyword. The class must also be declared as public.

In Objective-C, a class is made visible to another class simply by importing the header file in which the class's interface is declared. The `public` keyword is also used to declare if member variables are visible to other classes.

QUICK TIP

Classes in Java, C#, and Objective-C can declare their member variables to be public. However, this is not a common practice in modern Objective-C. You will learn more about this in the *Adding Properties to a Class* section of this session.

Visible Methods to Other Classes

Methods in a Java or C# class can also be declared public, making them visible to other objects. This is not the case with Objective-C.

In Objective-C, all methods that are declared in the interface are visible to other classes. If a class needs a private method, it just adds the method to the implementation file. All code within the implementation file can call that method, but it will not be visible to any classes that import the header file.

Classes Inherit from `NSObject`

Another key concept of object-oriented languages is the capability of one class to **inherit** from another class. In Java, all classes inherit from the `Object` class. C# is similar with the `System.Object` class. In Objective-C, virtually every class inherits from `NSObject`. `NSObject` defines the code for managing the memory of an object in Objective-C.

All three languages have a common root class so that they can provide methods and behaviors that can be assumed as members. The difference in the syntax is that in Java and C# classes that do not explicitly define their superclass, the root class is assumed. In Objective-C, you must always declare the superclass by following the name of the class with a colon and then the name of the superclass, as shown in the following example:

```
   @interface Simple Class : NSObject
```

Use of Long and Explicit Signatures

The last difference among these classes is the signatures of the methods themselves. Both Java and C# use the same type of method signatures. The return type of the method is listed first, followed by its name, and then by any parameters and their type listed within parentheses. Java and C# also use method overloading, where the same method name is used but is distinguished by the list of parameters. In Objective-C, the signature is a bit different, which reflects its roots in Smalltalk.

Objective-C method signatures tend to be very long and explicit. Many developers coming from other languages may dislike this at first but learn to love it, as it enhances the readability of the code. Instead of needing to know which method of "Sum" to use when you want to pass in two parameters, you know by just looking that the `sumWithThirdInt:fourthInt: method` is the one you want. Actually, that is the full name of that method. The colons in the method name denote the number of parameters the method takes. By having the parameters listed inline with the method name, you can create signatures that are easier to understand. As you proceed through this course, you will see how this type of method signature is used throughout Objective-C and Cocoa Touch.

Instantiating Objects

A class is only a template for creating an object. To use an object, it needs to be **instantiated** and **created in memory**. In Java and C#, you use the `new` keyword and a constructor. Both languages have a default constructor that is part of their base object class.

The following code shows how you instantiate an object in Java or C#.

```
SimpleClass simpleClassInstance = new SimpleClass();
```

In Objective-C, the `NSObject` class has something similar to a default constructor, but there is a slight difference. Instead of a single step to instantiate an object, you do it in two steps, as follows:

1. The first step uses the static method `alloc`, which finds enough available memory to hold the object and assign it.
2. The second step uses the `init` method to set the memory.

The following code shows an example of instantiating an object in Objective-C.

```
SimpleClass *simpleClassInstance = [[SimpleClass alloc] init];
```

If you have ever written code in C or C++, you should recognize the *** operator**. This is the **pointer dereference operator**. It acts the same in Objective-C by dereferencing the pointer and getting or setting the object in memory.

Pointers are part of C. The memory of a computer can be thought of as a bunch of little boxes that hold values. Each of these boxes has an address. In this example, the variable `simpleClassInstance` is a pointer. The value it holds in memory is not the object, but the address of where the object exists in memory. It "points" to the object in another part of the memory.

Figure 1 illustrates how this works.

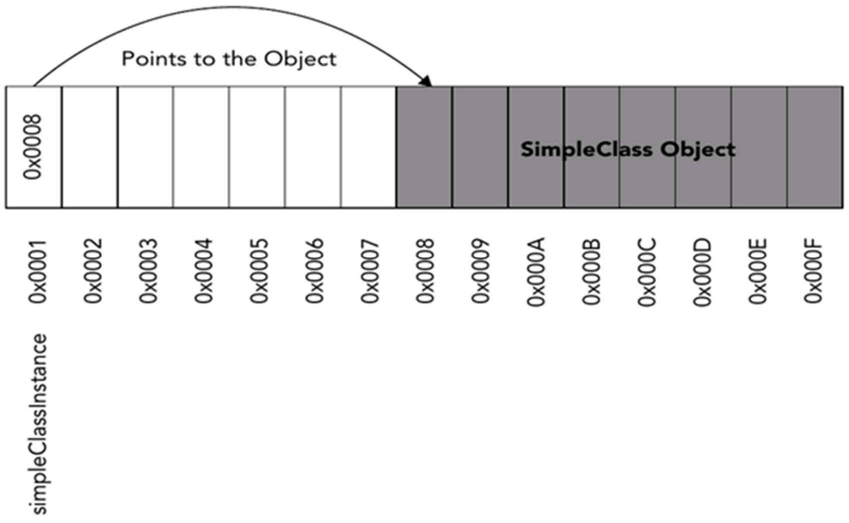


Figure 1: Working of Pointers in C

In Java and C#, you can also declare your own constructors that can take a list of parameters and set the member variables of the object. The following code adds these constructors to the Java example classes with sample code of how they are called.

```
package SamplePackage;
public class SimpleClass {
   public int firstInt;
   public int secondInt;
   public SimpleClass(int initialFirstInt, int initialSecondInt)
```

```
    {
          firstInt = initialFirstInt;
          secondInt = initialSecondInt;
    }

    // other methods discussed earlier
}
// sample code to create a new instance
SimpleClass aSimpleClassInstance = new SimpleClass(1, 2);
```

The following code defines a constructor in C#.

```
namespace SampleNameSpace
{
    public class SimpleClass
    {
          public int FirstInt;
          public int SecondInt;

          public SimpleClass(int firstInt, int secondInt)
          {
                FirstInt = firstInt;
                SecondInt = secondInt;
          }
          // other methods discussed earlier
    }
}
// sample code to create a new instance
SimpleClass aSimpleClassInstance = new SimpleClass(1, 2);
```

To implement the same type of constructor in Objective-C, you would add your own `init` method, as shown below.

```
// in the SimpleClass.h file
@interface SimpleClass : NSObject
{
    @public
    int firstInt;
    int secondInt;
}
- (id)initWithFirstInt:(int)firstIntValue secondInt:(int)secondIntValue;
// other methods discussed earlier
@end
// in the SimpleClass.m file
@implementation SimpleClass
- (id)initWithFirstInt:(int)firstIntValue secondInt:(int)secondIntValue
{
    self = [super init];
    if(!self)
```

```
        return nil;
        firstInt = firstIntValue;
        secondInt = secondIntValue;

        return self;
    }
    // other methods discussed earlier

    @end

    // sample code to create a new instance
    SimpleClass *aSimpleClassInstance =
     [(SimpleClass alloc) initWithFirstInt:1 secondInt:2];
```

Now, let's discuss some key points about the above code example:

- **Use of the id Type:** Objective-C is a dynamic language, meaning that you do not have to give a specific type to an object at compile time. The `id` type can hold a pointer to any object. It is similar to the `dynamic` type in C#. Java does not have anything similar. When you use the `id` type, its actual type is determined at runtime. Returning `id` from an `init` method is part of a Cocoa convention, which you should always follow.
- **Use of the self Variable:** The `self` variable is the same as in Java or in C#. It refers to the instance of the object that is receiving the message. When you are initializing an object, you need to first call `init` on its parent class. This ensures that objects are created correctly through their hierarchy.

The `if` statement, `if(!self)`, introduces another concept that is prevalent throughout Objective-C. If a variable does not point to anything, it has a value of `nil` or essentially 0. **Figure 2** illustrates how this works.

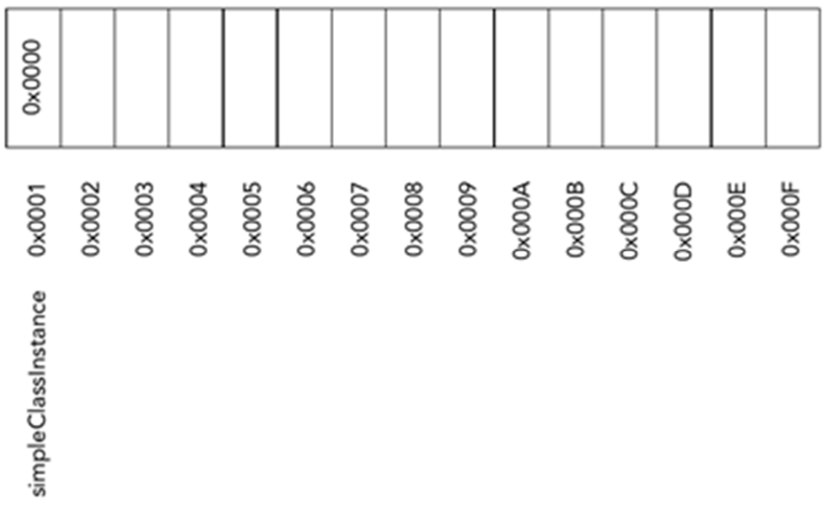


Figure 2: `Nil` Value in Objective-C

In C the value 0 is synonymous with `FALSE`, while any value greater than 0 is synonymous with `TRUE`. If an object has failed to initialize, then its pointer will be `nil`. You need to check if the parent class has returned `nil` because of defensive coding. If a pointer is `nil` and you send it a message, it is treated as a no op, meaning nothing will happen. In Java and C#, this throws an exception.

The rest of the `init` method works the same as the constructors in Java and C#. The member variables are set using the values passed in before returning the pointer.

Another approach to instantiate objects is to use a **factory method**. These methods are **static methods**. A static method does not need an instance of the class to call it. It also does not have access to any of the member variables of a class.

If you have written code in Java or C#, you must be familiar with the static methods and factory methods. In Objective-C they are more of a convenience method than anything else, which is different from their use in Java and C#. The basic idea remains the same, though.

The following code shows how you would implement a factory constructor in Java.

```
package SamplePackage;
public class SimpleClass {
   public int firstInt;
   public int secondInt;
   public static SimpleClass Create(int initialFirstInt, int initialSecondInt)
   {
         SimpleClass simpleClass = new SimpleClass();
         simpleClass.firstInt = initialFirstInt;
         simpleClass.secondInt = initialSecondInt;
         return simpleClass;
   }
   // other methods discussed earlier
}
// sample code to create a new instance
SimpleClass aSimpleClassInstance = SimpleClass.Create(1, 2);
```

The following code shows how you would implement a factory constructor in C#.

```
namespace SampleNameSpace
{
   public class SimpleClass
   {
         public int FirstInt;
         public int SecondInt;

         public static SimpleClass Create(int firstInt, int secondInt)
         {
               SimpleClass simpleClass = new SimpleClass();
               simpleClass.FirstInt = firstInt;
               simpleClass.SecondInt = secondInt;

               return simpleClass;
         }

         // other methods discussed earlier
   }
}
// sample code to create a new instance
SimpleClass aSimpleClassInstance = SimpleClass.Create(1, 2);
```

The following code demonstrates how you would define a factory method in Objective-C. These types of convenience methods are often used in many of the basic data structures of Objective-C.

```
// in the SimpleClass.h file
@interface SimpleClass : NSObject
{
   @public
   int firstInt;
   int secondInt;
}
+(id)simpleClassWithFirstInt:(int)firstIntValue secondInt:(int)secondIntValue;
// other methods discussed earlier

@end

// in the SimpleClass.m file
@implementation SimpleClass

+ (id)simpleClassWithFirstInt:(int)firstIntValue secondInt:(int)secondIntValue
{
   SimpleClass *simpleClass = [[SimpleClass alloc] init];
   simpleClass->firstInt = firstIntValue;
   simpleClass->secondInt = secondIntValue;

   return simpleClass;
}

// other methods discussed earlier

@end

// sample code to create a new instance
SimpleClass *aSimpleClassInstance =
[SimpleClass simpleClassWithFirstInt:1 secondInt:2];
```

The Java and C# implementations rely on the default constructors defined in their respective root classes to instantiate a new object. This is the same as the `init` method defined in the `NSObject` class. They next set the member variables of the new instance before returning it. Syntactically, the difference is how a method is declared as static. In Java and C#, the `static` keyword is added to the method signature. In Objective-C, a `static` method is declared by using the + (plus) symbol instead of the – (minus) symbol, which would define it as an instance method.

MANAGING MEMORY

Now, let's discuss memory management, which is an important feature in Objective-C. Memory is a finite resource, which means there is only so much of it that can be used. This is particularly true for mobile devices. When a system runs out of memory, it can no longer perform any more instructions, which is obviously a bad thing. Running low on memory will also have a dramatic impact on performance. The system has to spend a lot more time finding available memory to use, which slows down every process. **Memory**

management is a method to control which objects need to remain in memory and which ones to remove, so their memory can be reused.

Memory leaks are a classic problem in computer programming. A **memory leak**, in the most basic of definitions, is when **memory is allocated but never deallocated**.

The opposite of a leak is when memory is deallocated before it is completely used. This concept is referred to as a **dangling pointer**. In Objective-C, it is common to refer to these dangling pointers as **zombie objects**. These types of memory issues are usually easier to find because the program will most likely crash if it tries to use a deallocated object.

Languages and runtimes handle memory management in two different ways. Java and C# use **garbage collection**. This method was first introduced in Lisp in the late 1950s. The implementation of garbage collection is detailed and different depending on the runtime, but the idea is basically the same. As a program executes, it allocates objects in memory that needs to continue. Periodically another process runs that looks for objects that are no longer reachable, which means they have no references left in any code that is executing. This type of a system works very well; though contrary to popular belief, your code can still leak memory by creating objects and keeping a reference to them but never using them again. This system also has a bit of overhead associated with it. The system needs to continue executing the program being used while running the garbage collection process in parallel. This can create performance problems on systems that have limited computational power.

Objective-C on iOS does not use garbage collection. Instead, it uses **manual reference counting**. Each object that is allocated has a reference or retain count. If a piece of code requires that the object be available, it increases its retain count. When it is done and no longer needs the object, it decrements the retain count. When an object's retain count reaches 0, it can be deallocated and the memory is returned to the system.

Manual reference counting can be difficult to understand if you are not used to thinking about the life cycle of objects in use. Because languages like Java and C# use garbage collection, it can be even more difficult for developers who have used those languages for an extended period to make the transition to Objective-C. Apple recognized this and made significant improvements to the Objective-C compiler that will be discussed later in this section. Though these improvements make manual reference counting much easier, it is still important for the developers to understand exactly how retain counts work and the rules around them.

BIG Picture

Zombie objects are a feature of Cocoa/CoreFoundation used for debugging, to help you catch memory errors. Normally, when the reference count of an object drops to zero, it is freed immediately, but that makes debugging difficult. In Objective-C, the object's memory is not instantly freed, it is just marked as a zombie, and any further attempts to use it will be logged. You can track down where in the code the object was used past its lifetime if zombie objects are enabled.

The first thing to understand is how retain counts work and how they can lead to memory leaks and zombie objects. The following code shows one way a memory leak can occur using the `SimpleClass` instance described previously in this session.

```
- (void)simpleMethod
{
   SimpleClass *simpleClassInstance = [[SimpleClass alloc] init];

   simpleClassInstance->firstInt = 5;
   simpleClassInstance->secondInt = 5;

   [simpleClassInstance sum];
}
```

A Java or C# developer would not see anything wrong with this code. To them there is a method that creates an instance of the `SimpleClass`. When the method is done executing, the `simpleClassInstance` no longer has a reference to it and is eventually deallocated by the garbage collector. This is not the case in Objective-C.

When the simpleClassInstance is instantiated using alloc, it has a retain count of one. When the method is done executing, its pointer goes out of scope but keeps a retain count of one. Because the retain count stays above zero, the object is never deallocated. With no pointer still referencing the object, there is no way to decrement its retain count so that it can be deallocated. This is a classic memory leak, which is illustrated in **Figure 3**.

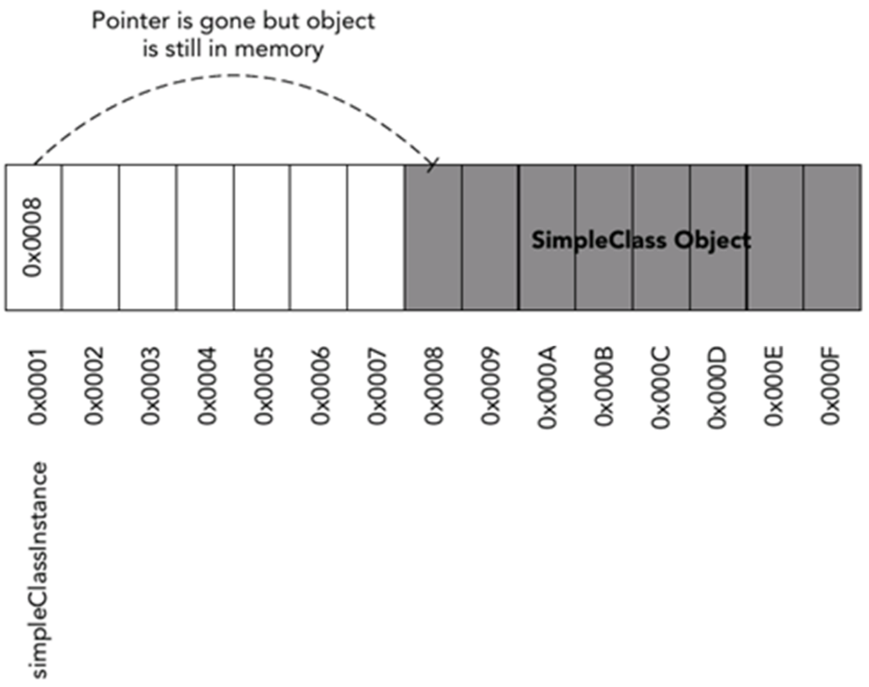


Figure 3: An Example of Memory Leak in Objective-C

To fix this memory leak in Objective-C, you need to explicitly decrement the retain count before leaving the method. You do this by calling the release method of the NSObject class, as shown in in the following example. The release method decrements the count by one, which in this example sets the count to zero, which in turn means that the object can be deallocated.

```
- (void)simpleMethod
{
   SimpleClass *simpleClassInstance = [[SimpleClass alloc] init];

   simpleClassInstance->firstInt = 5;
   simpleClassInstance->secondInt = 5;

   [simpleClass sum];

   [simpleClass release];
}
```

Another way to fix this memory leak is to use the **autorelease pool**. Instead of explicitly releasing the object when you are done with it, you can call autorelease on it. This puts the object into the autorelease pool that keeps track of objects within a particular scope of the program. When the program has exited that scope, all the objects in the autorelease pool are released. This is referred to as **draining the pool**. The following code shows how you would implement this.

```
- (void)simpleMethod
{
   SimpleClass *simpleClassInstance = [[SimpleClass alloc] init];

   [simpleClassInstance autorelease];

   simpleClassInstance->firstInt = 5;
   simpleClassInstance->secondInt = 5;

   [simpleClass sum];
}
```

Using `alloc` generally means that the code creating the object is its owner, which is why it gets a retain count of one. There are other times in your code where an object was created elsewhere but the code using it needs to take the ownership. The most common example of this is using factory methods to create the object. Factory methods of Objective-C core classes always return an object that has autorelease called on it before it is returned. The following code shows how the `simpleClassInstance` variable would generally be implemented now that you understand the `autorelease` method.

```
    + (id)simpleClassWithFirstInt:(int)firstIntValue secondInt:(int)secondIntValue
    {
        SimpleClass *simpleClass = [[SimpleClass alloc] init];
        simpleClass.firstInt = firstIntValue;
        simpleClass.secondInt = secondIntValue;

        [simpleClass autorelease];
        return simpleClass;
    }
```

A piece of code, which creates a `simpleClass` using the factory method, may want that object to stay in memory even after the autorelease pool has been drained. To increase its retain count and make sure it stays in memory, you would call `retain` on the instance.

If you explicitly retain an object, it is important to always call `release` sometime later in your code, or else it will cause a memory leak. The following code shows a simple example of this; though in a real program, you would probably call release in some other place.

```
    - (void)simpleMethod
    {
        SimpleClass *simpleClass =
        [SimpleClass simpleClassWithFirstInt:1 secondInt:5];

        [simpleClass retain];

        // do things with the simple class knowing it will not be deallocated

        [simpleClass release];
    }
```

The other memory management issue you can run into is referencing an object that has already been deallocated. This is called a zombie object. **Figure 4** illustrates this.

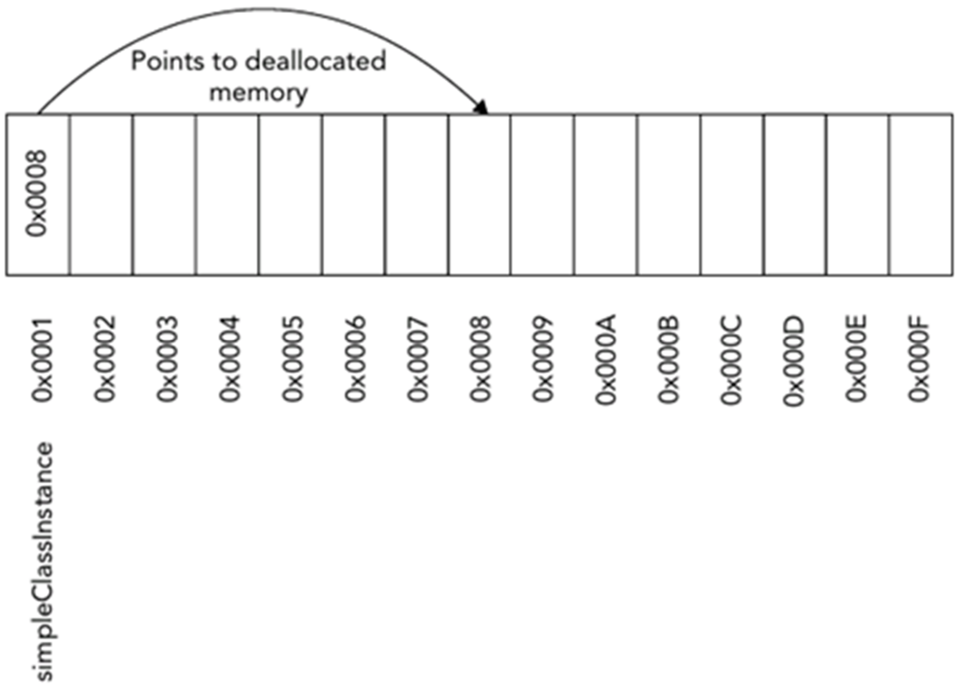


Figure 4: A Zombie Object

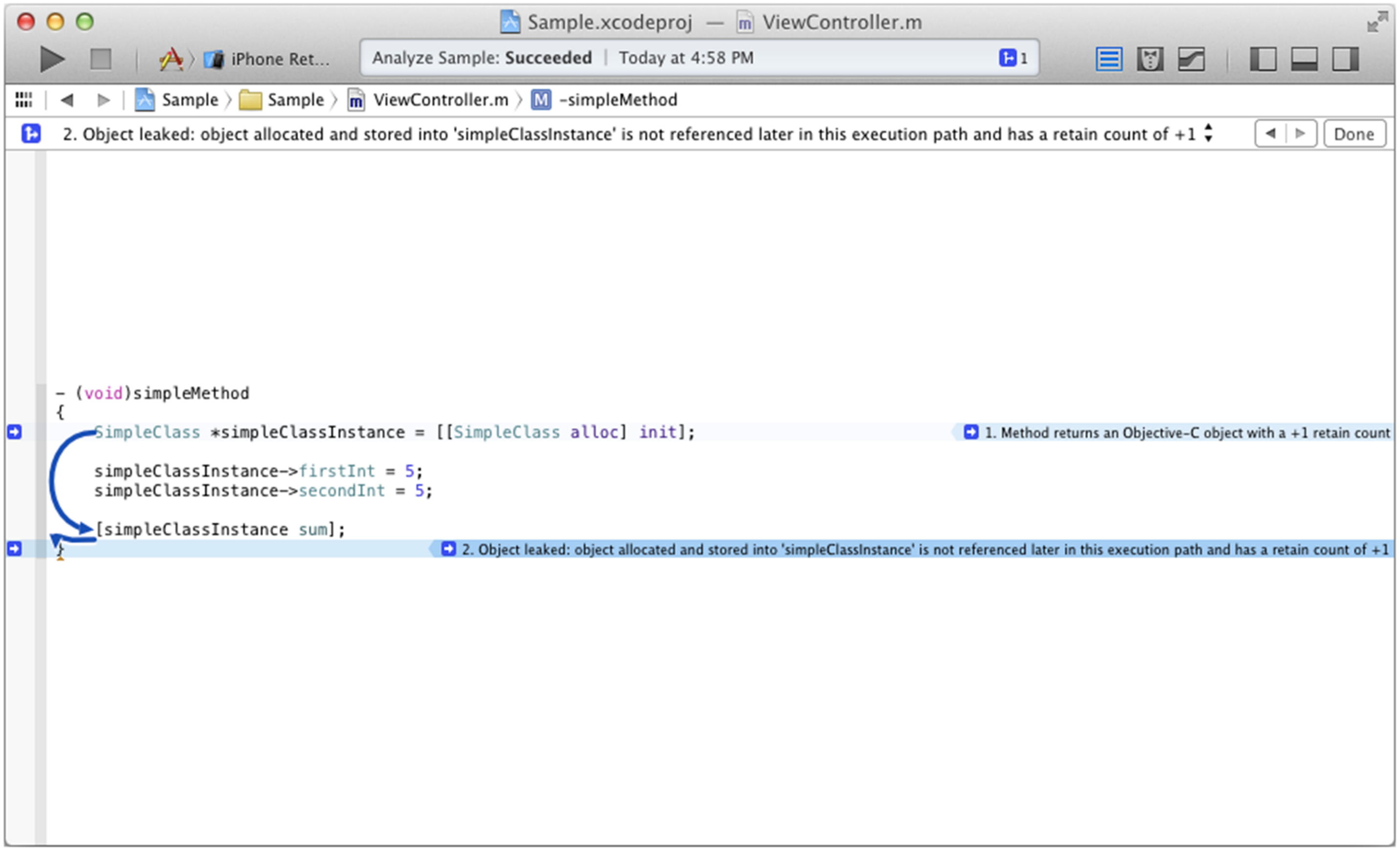
If this happens during runtime, the execution of your app can become unpredictable or just simply crash. It depends on whether the memory address is still part of the memory allocated to your program or if it has been overwritten with a new object. It is conceivable that the memory is still there and your code would execute just fine even though the memory has been marked for reuse. The following example demonstrates how a zombie object can occur.

```
- (void)simpleMethod
{
   SimpleClass *simpleClassInstance = [[SimpleClass alloc] init];
   [simpleClassInstance release];

   simpleClass.firstInt = 5;
   simpleClass.secondInt = 5;

   int sum = [simpleClass sum];
}
```

Introducing Automatic Reference Counting

Manual reference counting differs from garbage collection by setting when objects are allocated and deallocated at compile time instead of runtime. The compiler that Apple uses in its development tools is part of the **LLVM Project** (*llvm.org*) and is called **Clang** (*clang.llvm.org*). Developers using manual memory management can follow a strict set of rules to ensure that memory is neither leaked nor deallocated prematurely. The developers working on the Clang compiler recognized this and set out to build a tool that could analyze a code base and find where objects could potentially be leaked or used after they have been released. The tool they released is called the **Clang Static Analyzer** *(clang-analyzer.llvm.org)*.

The Clang Static Analyzer is both a standalone tool as well as integrated in Apple's Xcode development environment. As an example of how it works, **Figure 5** shows the results of running the static analyzer on the first example of a memory leak.


Sample.xcodeproj — ViewController.m
iPhone Ret...
Analyze Sample: Succeeded | Today at 4:58 PM
Sample 〉 Sample 〉 ViewController.m 〉 -simpleMethod
Done

2. Object leaked: object allocated and stored into 'simpleClassInstance' is not referenced later in this execution path and has a retain count of +1

```
- (void)simpleMethod
{
    SimpleClass *simpleClassInstance = [[SimpleClass alloc] init];

    simpleClassInstance->firstInt = 5;
    simpleClassInstance->secondInt = 5;

    [simpleClassInstance sum];
```

1. Method returns an Objective-C object with a +1 retain count

2. Object leaked: object allocated and stored into 'simpleClassInstance' is not referenced later in this execution path and has a retain count of +1

Figure 5: Clang Static Analyzer

After building the static analyzer, the compiler developers realized that they could insert the required `retain` and `release` calls at compile time by detecting rule violations in manual reference counting. They implemented this in a compiler feature called **Automatic Reference Counting (ARC)**. Because Xcode uses the Clang compiler, iOS developers and Mac developers could use this new feature by simply enabling it in the compiler settings and then removing all their explicit calls to `retain`, `release`, and `autorelease`.

BIG Picture

Using ARC is becoming a standard practice for Mac and iOS developers. This course uses ARC in all sample code. By using ARC you, as a new developer, do not need to worry about most of the details of memory management, though there are cases in which you do need to know how an object will be treated in memory.

Adding Properties to a Class

The `SimpleClass` example in this session has used public instance variables for its two integer values. This is considered a bad practice in all three languages. Declaring instance variables as public leaves your class vulnerable to bad data. There is no way to validate the value being assigned. Instead for all three languages, it is a common practice to keep the instance variables private and then implement the getter and setter methods. By doing this, you can validate the value in the setter method before actually setting the value of the instance variable. Though the concept is the same, the implementation and syntax are different.

The following code example demonstrates how a class should be defined in Java. It is the most straightforward implementation. The instance variable is declared as private, and two additional methods are added to the class to get and set their values.

```
package SamplePackage;
public class SimpleClass {
   private int firstInt;
   private int secondInt;

   public void setFirstInt(int firstIntValue) {
         firstInt = firstIntValue;
   }

   public int getFirstInt() {
         return firstInt;
   }

   public void setSecondInt(int secondIntValue) {
         secondInt = secondIntValue;
   }

   public int getSecondInt() {
         return secondInt;
   }

   // other methods discussed previously
}

// sample code of how to use these methods
SimpleClass simpleClassInstance = new SimpleClass();

simpleClassInstance.setFirstInt(1);
simpleClassInstance.setSecondInt(2);

int firstIntValue = simpleClassInstance.getFirstInt();
```

In C#, this type of implementation is done by using **properties**. A property can be referred to as if it were an instance variable but still uses getters and setters. The following code demonstrates properties in C#.

```
namespace SampleNameSpace
{
   public class SimpleClass
   {
         private int firstInt;
         private int secondInt;

         public int FirstInt
         {
               get { return firstInt; }
               set { firstInt = value; }
         }
```

```
            public int SecondInt
            {
                  get { return secondInt; }
                  set { secondInt = value; }
            }
            // other methods discussed earlier
      }
   }
   // sample code of how to use these properties
   SimpleClass simpleClassInstance = new SimpleClass();

   simpleClassInstance.FirstInt = 1;
   simpleClassInstance.SecondInt = 2;

   int firstIntValue = simpleClassInstance.FirstInt
```

The Objective-C implementation is a bit of a mix of the Java and C# implementations. Like C# it has an idea of properties, but like Java the implementation of the getters and setters are the actual methods in implementation. The following code shows how properties are declared in the interface, implemented, and used.

```
   // in the SimpleClass.h file
   @interface SimpleClass : NSObject
   {
      int _firstInt;
      int _secondInt;
   }

   @property int firstInt;
   @property int secondInt;

   // other methods discussed earlier

   @end

   // in the SimpleClass.m file
   @implementation SimpleClass

   - (void)setFirstInt:(int)firstInt
   {
      _firstInt = firstInt;
   }

   - (int)firstInt
   {
      return _firstInt;
   }

   - (void)setSecondInt:(int)secondInt
```

```
{
   _secondInt = secondInt;
}

   - (int)secondInt
{
return _secondInt;
}

@end

// sample code to create a new instance
SimpleClass *simpleClassInstance = [[SimpleClass alloc] init];

[simpleClassInstance setFirstInt:1];
[simpleClassInstance setSecondInt:2];

int firstIntValue = [simpleClassInstance firstInt];
```

Because properties in Objective-C are the preferred way of exposing data members as public, its properties have been made easier to use as Objective-C has evolved. The **@synthesize** keyword was one of those enhancements. By using it, the getter and setter methods are generated for you at compile time instead of you needing to add them into your implementation. You can still override the getter or setter method if you need to. As the Objective-C language progressed, the `@synthesize` keyword became optional. Now, you do not need to declare the private instance variables. Instead, they are generated for you. The instance variables are named the same as the property but with a leading underscore. The following code shows what a modern Objective-C class looks like.

```
// in the SimpleClass.h file
@interface SimpleClass : NSObject

@property int firstInt;
@property int secondInt;

- (int)sum;

@end

// in the SimpleClass.m file
@implementation SimpleClass

- (int)sum
{
   return _firstInt + _secondInt;
}

@end
```

Another enhancement made around properties was the introduction of **dot notation**. Instead of needing to message an object using brackets, you can simply follow the instance of the class with a "." and the name of the property, as shown in the following code.

```
SimpleClass *simpleClassInstance = [[SimpleClass alloc] init];

simpleClassInstance.firstInt = 5;
simpleClassInstance.secondInt = 5;

int firstIntValue = simpleClassInstance.firstInt;
```

Properties can also be pointers to other objects. These properties need a little more attention in Objective-C. In C# you declare properties to other objects in the same way as you declare properties to primitive data types. In Objective-C, you also need to tell the compiler if your object is the owner of the other object as well as how its value should be set and retrieved in a threaded environment. For example, assume there is another class called `SecondClass`. The `SimpleClass` interface in the following code example has two properties for this class.

```
// in the SimpleClass.h file
@interface SimpleClass : NSObject

@property (atomic, strong) SecondClass *aSecondClassInstance;
@property (nonatomic, weak) SecondClass *anotherSecondClassInstance;

@end
```

In this example, the first property has the **atomic attribute**, whereas the second property is **nonatomic**.

Properties by default are **atomic**. Atomic properties have synthesized the getter and setter methods, which guarantee that the value is fully retrieved or fully set when accessed simultaneously by different threads. Nonatomic properties do not have this guarantee. Atomic properties have extra overhead in their implementations to make this guarantee. Though it may sound like this makes them thread safe, that is not the case. A solid threading model with proper locks also creates the same guarantee, and so the use of atomic properties is limited. Most often you declare your properties as nonatomic to avoid the extra overhead.

The other attributes, **strong** and **weak**, are much more important to understand. As you learned earlier in this session, an object that has a retain count of zero will be deallocated. The concept of strong and weak with properties is similar. An object that does not have a strong reference to it will be deallocated. By declaring an object property as strong, it will not be deallocated as long as your object points to it. This implies that your object owns the other object. An object property that is declared weak remains in memory as long as some other object has a strong pointer to it.

Weak properties are used to avoid strong reference cycles. These occur when two objects have properties that point at each other. If both objects have a strong reference to each other, they will never be deallocated, even if no other objects have a reference to either of them. **Figure 6** illustrates the issue. The strong references between the objects are represented by the solid line arrows.

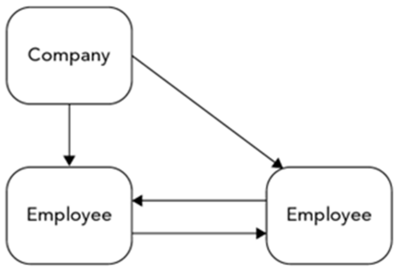


Figure 6: An Example of Strong Properties

In this example, there are three objects.

The first is a `Company` object that represents some company. Every company has employees.

In this example, the company has two employees represented by the two `Employee` objects.

Within a company there are bosses and workers. The bosses need a way to send messages to their workers in the same way as the workers needs a way to send messages to their bosses. This means both need to have a reference to each other.

If the `Company` object gets deallocated, its references to both `Employee` objects go with it. This should result in the `Employee` objects also being deallocated. But if the references between the boss `Employee` object and the worker `Employee` object are also strong references, then they will not be deallocated. If the references are weak, as illustrated in **Figure 7**, with dashed lines, then losing the strong reference to each `Employee` object from the `Company` object will mean there are no longer any strong references pointing to them so they will be properly deallocated. In this example, the `Company` object is the owner of the `Employee` objects, so it would use a strong reference to indicate this ownership.

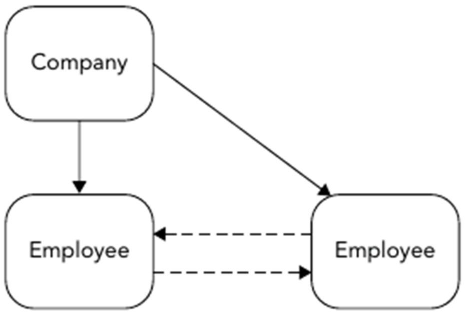


Figure 7: An Example of Weak Properties

Explaining Strings

Java, C#, and Objective-C have special classes for **strings**. Objective-C strings can be a bit confusing when you are first learning to use them, but comparing their use to Java and C# should help reduce the learning curve. This section is a brief overview of strings in Objective-C just to get you started. When you understand the basics and limitations, you can easily learn how to deal with specific situations using the class documentation provided by Apple at *https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSString_Class/Reference/NSString.html.*

The following code examples show how you would declare a basic string in Java, C#, and Objective-C. These examples are, of course, just one way to create a string instance. All three languages have many other methods, but the purpose here is just to show how a string would be commonly created.

```
// declaring a string in Java
String myString = "This is a string in Java";

// declaring a string in C#
String myString = "This is a string in C#";

// declaring a string in Objective-C
NSString *myString = @"This is a string in Objective-C";
```

In Java and C#, the class that represents a string is simply called `String`. In Objective-C, it is called `NSString`. Because Objective-C is a superset of C, you cannot use only quotes to create an `NSString`. Quoted strings in C create a C string. By adding the @ symbol before the quoted text, you tell the compiler to treat the text as an `NSString` and not a C string.

The C language does not permit overloading operators, which means that you cannot do it in Objective-C, either. Overloading an operator is a way to use an operator in your code instead of calling a method to do the work. The compiler knows that when it sees that operator, it should use the method instead. For example, you have two `Company` objects, each with their own set of employees, and you want to "merge" them into a new `Company` object with all the `Employee` objects combined. In a language like C#, you can override the + operator and then write code to do the merge, as shown below.

```
Company firstCompany = new Company();
Company secondCompany = new Company();

// you could write a merge method as part of the Company class
// then create a new company using it
Company thirdCompany = new Company();
thirdCompany.merge(firstCompany);
thirdCompany.merge(secondCompany);

// or you could override the + operator and do the same in a single line
Company thirdCompany = firstCompany + secondCompany;
```

You cannot do this in Objective-C. This difference becomes stark when comparing the `NSString` class to the `String` classes in Java or C#. In all languages, a text string is represented in memory as an array of characters. The classes you use to handle strings in code are more or less convenience classes so that you do not deal with raw character arrays, but a single object. The implementation of the classes hides that they are character arrays. In languages that allow operator overloading, you can do things like concatenating strings by using the overloaded + operator, as shown below, or calling a method like `concat` in Java.

```
// this code is correct in Java
String firstString = "This is a";
String secondString = "full sentence";
String combinedString = firstString + " " + secondString;

// or you could use concat
String combinedString = new String();
combinedString.concat(firstString);
combinedString.concat(secondString);
```

You cannot do either of these in Objective-C using the `NSString` class. First, because you cannot overload the + operator. Second, because an `NSString` cannot be changed after it is created. Objective-C has no methods like `concat`. It is an immutable object. To do the same in Objective-C, you would use the `NSMutableString` class, a subclass of `NSString`. The following code shows how you would use the `NSMutableString` class to append other strings.

```
// this code is correct in Objective-C
NSString *firstString = @"This is a";
NSString *secondString = @"full sentence";

NSMutableString *combinedString = [[NSMutableString alloc] init];
[combinedString appendString:firstString];
[combinedString appendString:@" "];
[combinedString appendString:secondString];
```

The idea between mutable and immutable objects is used throughout Objective-C. All data structures, which are discussed in the next section, have mutable child classes and immutable parent classes. The reason behind this is to guarantee that the object will not change while you are using it.

Another difference with strings in Objective-C is string formatting. In Java and C#, you can create a string using text and integer values simply by using the + operator, as shown below.

```
int numberOne = 1;
int numberTwo = 2;
String stringWithNumber = "The first number is " + numberOne +
" and the second is " + numberTwo;
```

The above code will produce the text string `"The first number is 1"`. In Objective-C, you do string formatting the way it is done in C using the IEEE printf specification (*http://pubs.opengrouporg/onlinepubs/009695399/functions/printf.html*).

The following code example shows how to create the same text string in Objective-C. It uses the `stringWithFormat:` static convenience method that allocates the string, instead of you needing to call `alloc`.

```
int numberOne = 1;
NSString *stringWithNumber = [NSString stringWithFormat:
@"The first number is %d and the second is %d", numberOne, numberTwo];
```

The `stringWithFormat:` method looks for format specifiers in the actual text and then matches them with the list of values that follows. You get a compile error if the formatter does not match the value at the same index in the value list.

QUICK TIP

For a full list of format specifiers, refer to the Apple's documentation found at *https://developer.apple.com/library/ios/documentation/cocoa/conceptual/Strings/Articles/formatSpecifiers.html.*

One last thing to be mindful of when using strings in Objective-C is **string comparison**. The difference again goes back to the ability to overload operators. In Java, you can compare two strings using the == operator. This ensures that every char at every index is the same between the two strings.

In Objective-C, you need to remember that the `NSString` instance variable holds only a pointer to the object in memory. So using == between two `NSString` instances will evaluate if both instances are pointing to the same place in memory. Instead, you would use the `isEqualToString:` method, as shown below.

```
NSString *firstString = @"text";
NSString *secondString = @"text";
if(firstString == secondString)
{
   NSLog(@"this will never be true");
}
else if ([firstString isEqualToString:secondString])
{
   NSLog(@"this will be true in this example");
}
```

The above code uses the `NSLog()` function. This is how you write debugging information to the console in Xcode. It also uses format specifiers followed by a list of values, which are the same as `stringWithFormat:`. It is comparable to `System.out.println()` in Java or `Console.WriteLine()` in C#.

Using Basic Data Structures

Data structures in programming languages are ways to **organize data**. An object is a data structure. You are more likely to think of data structures like arrays, sets, or dictionaries. All languages support these types of data structures in one way or another. There are a handful of data structures available in Objective-C. This section covers only the ones you will use while building the **Bands** app.

The most basic data structure is an **array**.

The simple definition of an **array** is a list of items stored in a particular order and referenced by their index number. Arrays can be either 0-based, with their start item at index number 0, or 1-based. C based languages use 0-based arrays, whereas languages such as Pascal use 1-based arrays. The following code example shows how you would create an array of integers in a C-based language and set the values at each index.

```
int integerArray[5];
integerArray[0] = 101;
integerArray[1] = 102;
integerArray[2] = 103;
integerArray[3] = 104;
integerArray[4] = 105;
```

In Java or C#, you can also create arrays of objects using a similar syntax. This is not possible in Objective-C. Instead, you use the `NSArray` class and its subclass `NSMutableArray`. `NSArray` is like `NSString`; it is immutable. After it is created, its objects cannot be changed and you cannot add or remove objects. For arrays that need to change or be dynamic, you need to use the `NSMutableArray` class.

`NSArray` is a little different from your typical array in Java or C#. In those languages, you always declare the type of each object in the array. You do not do this with `NSArray`. Instead, it will hold any object whose root class is `NSObject`. The following code shows the different ways you can create an `NSArray` instance in Objective-C. The syntax is a bit different with each, but they all create the same thing. Also, keep in mind that you can create an `NSString` using `@"my string"` syntax and that `NSString` is a descendant of `NSObject`.

```
NSArray *arrayOne = [[NSArray alloc] initWithObjects:@"one", @"two", nil];
NSArray *arrayTwo = [NSArray arrayWithObjects:@"one", @"two", nil];
NSArray *arrayThree = [NSArray arrayWithObject:@"one"];
NSArray *arrayFour = @[@"one", @"two", @"three"];
NSString *firstItem = [arrayOne objectAtIndex:0];
```

The first array is created using the `alloc/init` pattern. The second array is created using the `arrayWithObjects:` convenience method. Both methods take a C array of objects, which is basically a list of objects followed by a `nil`. The third array is also a convenience method but takes only one object. The last array uses the `NSArray` literal syntax, which does not need a `nil` at the end.

Getting an object from an `NSArray` is slightly different from arrays in other languages. In the above code example, the items are referenced by their index number by following the name of the array with brackets and the index number. Because brackets are used in Objective-C to send messages to an object, you cannot use this approach. Instead you use the `objectAtIndex:` method. You can also search for objects in an `NSArray` using `indexOfObject:`, or sort them by using the `sortedArrayUsingSelector:`. You will learn how to use these later in this course.

As you now know, the `NSArray` class is immutable, so you cannot change the values or the size of the array after it has been created. Instead you use an `NSMutableArray` class. **Table 2** lists the additional methods in the `NSMutableArray` class that you can use to modify the array.

Table 2: Methods in the NSMutableArray Class

METHOD	DESCRIPTION
`addObject:`	Add an object to the end of the array
`insertObject:atIndex:`	Insert an object into the array at a specific index
`replaceObjectAtIndex:withObject:`	Replaces the object at a specific index with the object passed in
`removeObjectAtIndex:`	Remove an object at the specific index
`removeLastObject`	Remove the last object in the array

Similar to the `NSArray` data structure are the `NSSet` and `NSMutableSet` classes. This type of data structure holds a group of objects but not in any particular order. A set is used when you do not need to access individual objects, but instead need to interact with the set as a whole. There are no methods in the `NSSet` class that return an individual object in the set. However, there are ways to get a subset of a greater set. Sets are particularly useful if you need to check only if an object is included in it. You will not use sets while building the **Bands** app, so there is no need to discuss them further in this session.

The last common data structure in Objective-C is a dictionary or hash table. It uses a key/value storage paradigm. The `NSDictionary` and `NSMutableDictionary` classes represent this type of data structure in Objective-C. An `NSDictionary` has a set of keys that are an instance of an `NSObject` descendant. Often, you will use an `NSString` as the key. The value in a dictionary is also a descendant of `NSObject`.

For example, a company has several employees. Because employees may have the same first and last names, each employee is assigned a unique identification (ID) number. When employees get a new title, their information needs to be updated, but the only information you have for employees is their ID number. If you were to use only arrays or sets to keep track of employees, you would need to iterate through all employees, checking their IDs until you find the correct employee. With a dictionary, you can simply look up employees using their unique IDs.

The following code example demonstrates how this would be done in Objective-C using an `NSMutableDictionary`. In this example, there are three `Employee` objects, each with a unique ID. The `NSMutableDictionary` is created by using the convenience method dictionary. The `Employee` objects are added to the dictionary by using the `setObject:forKey:` method. To get the `Employee` object whose unique ID is `"E2"`, the code can get it quickly using the `objectForKey:` method.

```
NSString *employeeOneID = @"E1";
NSString *employeeTwoID = @"E2";
NSString *employeeThreeID = @"E3";

Employee *employeeOne = [Employee employeeWithUniqueID:employeeOneID];
Employee *employeeTwo = [Employee employeeWithUniqueID:employeeTwoID];
Employee *employeeThree = [Employee employeeWithUniqueID:employeeThreeID];

NSMutableDictionary *employeeDictionary = [NSMutableDictionary dictionary];
[employeeDictionary setObject:employeeOne forKey:employeeOneID];
[employeeDictionary setObject:employeeTwo forKey:employeeTwoID];
[employeeDictionary setObject:employeeThree forKey:employeeThreeID];

Employee *promotedEmployee = [employeeDictionary objectForKey:@"E2"];
promotedEmployee.title = @"New Title";
```

QUICK TIP

Both `NSArray` and `NSDictionary` store `NSObject` instances. Therefore, you cannot set primitive types such as integers or booleans as values. Instead, you can use the `NSNumber` class. Since it is a descendant of `NSObject`, it can be used with both `NSArray` and `NSDictionary` and can hold any primitive data type.

ADVANCED CONCEPTS

Objective-C is similar in syntax to other C-variant programming languages. Some of the concepts and patterns, though, are different. This section discusses these concepts and patterns so that you can see how they are used in practice while building the **Bands** app.

These advanced concepts of Objective-C include:

- Model-View-Controller (MVC) design pattern
- Protocols and delegates
- Blocks
- Error handling

Model-View-Controller Design Pattern

The Model-View-Controller design (MVC) pattern is another influence of Smalltalk. It is a high-level design pattern that can be applied to almost any programming language. In Mac and iOS programming, it is a predominant pattern and is ingrained deeply in Cocoa and Cocoa Touch. To begin developing iOS applications, you should be familiar with how it works so that the classes and user interface design of iOS apps makes sense.

The MVC design pattern is based on the concept of separating all components of a piece of software into one of the following three roles:

- Model
- View
- Controller

This separation helps to make the components independent of one another as well as configurable and reusable. If done correctly, it can greatly reduce the amount of code that needs to be written as well as make the overall architecture of the software easy to understand. This helps the new developers coming to a project in getting up to speed quickly.

Now, let's discuss each role in this design pattern in detail.

The Model

The **model** is the knowledge or data of an application as well as the rules that define how pieces of data interact with each other. It is responsible for loading and saving data from persistent storage as well as validating the data. The model role does not care or store any information about how its data is to be displayed.

The View

The **view** is the visual representation of the model. It does not act on the model in any way or save data to the model. It is strictly the user interface of the software.

The Controller

The **controller** is the link between the model and the view. When the model changes, the controller is notified and knows if there is a view that needs to update its visual display. If a user interacts with a view, the view notifies the controller about the interaction. The controller then decides if it needs to update the model.

For example, administrative assistants in a company use software to keep track of employees and update their information. **Figure 8** illustrates how the MVC pattern can be used to build this software.

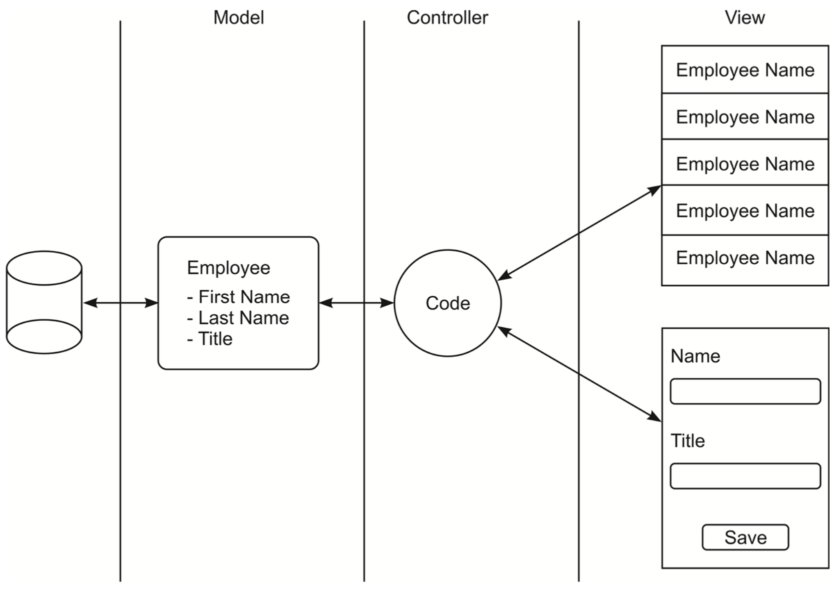


Figure 8: An Example of the Model-View-Controller Pattern

In **Figure 8**, the model would be the Employee on the left side and its underlying database. The employee has three properties:

- Employee's name
- Employee's title
- Employee's unique ID within the database

The model is responsible only for loading the employee information from the database and keeping its values in memory.

The views in this example are the list of employees and the employee detail view on the right of **Figure 8**. The list of employees shows the name of each employee. The administrative assistant can use this view to select an employee. The employee detail view again shows the name and title of the employee, but gives the administrative assistant the ability to change the values displayed. It also has a **Save** button, which the administrative assistant can click when he/she is done updating the information.

The controller is the circle that connects the employee model to the employee list screen and the employee details screen. When the employee list screen needs to be displayed to the administrative assistant, the employee list asks the controller for names of all the employees. It does not care about the employee's title or unique ID because it does not display those. It acts as a filter of the model data, so the administrative assistant sees only the information he/she needs to select the correct employee.

When the administrative assistant selects an employee, that interaction is passed again to the controller. The controller then changes the screen to the employee details screen. When this screen is loaded, it asks the controller for the employee's name and title.

The controller retrieves this information from the model and passes it back to the detail view for display. The administrative assistant then changes the title of the employee in the detail view. At this point, it is only a visual change in the employee detail view. It has not changed the value in the model, nor has the database been updated.

When the **Save** button is clicked, the view informs the controller about the user interaction. The controller then looks at the data in the view and determines that the model needs to be updated. It passes the new value to the model and tells it to save it to the database.

The controller can also update either of the views if another administrative assistant has made a change. For instance, there are two administrative assistants looking at the same employee detail view. The first changes the name of the employee from "Tom" to "Ted" and then clicks the **Save** button, which tells the controller to update the model. Because the model has changed, it notifies the controller that its values have been updated. The controller gets this notification and determines that the value being displayed to the second administrative assistant is out of date, so it tells that detail view to update the visual display of the value.

Protocols and Delegates

The MVC design pattern is a great high-level programming pattern, but to use it in a programming language, you need the tools to make it work. Objective-C has these tools, as follows:

- The model layer is implemented using classes and properties to create reusable objects.
- The view layer is implemented in the Cocoa and Cocoa Touch application programming interfaces (APIs), including reusable views and subviews such as buttons and text fields.
- The controller layer is done by using delegates and data sources to facilitate communication.

Delegates and **data sources** are used heavily in Objective-C. It is a way of having one part of the software ask for the work to be done by another part. This fits the MVC design pattern with the communication between views and controllers. Because views interact only with controllers and never with the model, they need a way of asking for the model data from the controller. This is the role of the data source. When a user interacts with a view, the view needs to tell the controller about it. **Delegates** are used for this role. Typically, your controller will perform both of these roles.

A view needs to define all the questions it may ask its data source and what type of answer it expects in return. It also needs to define all the tasks it may ask the delegate to perform in response to user interaction. In Objective-C, this is done through protocols. You can think of a **protocol** as a contract between the view and its data source or delegate.

Consider the employee list view in the company example. This is a simple example in this session to illustrate how a protocol and a delegate are coded. In this example, the employee list is called the `EmployeeListView`. For the `EmployeeListView` to show all the employees in the company, it needs to ask its data source for them. When the administrative assistant selects an employee's details to view, the `EmployeeListView` needs to tell its delegate which employee was selected. This calls for two different protocols to be defined: one for the data source and one for the delegate. These protocols would be declared in an `EmployeeListView.h` file.

For the `EmployeeListView` to ask its data source for employees or tell its delegate that an employee was selected, the view needs a reference to both the delegate and the data source. The references are added as properties with a type of `ID`. This is because the view does not care what type of class it is, just as long as it implements the protocols. The following example shows how all this would be coded.

```
@protocol EmployeeListViewDataSource

- (NSArray *)getAllEmployees;

@end

@protocol EmployeeListViewDelegate

- (void)didSelectEmployee:(Employee *)selectedEmployee;

@end

@interface EmployeeListView : NSObject
@property (nonatomic, weak) id dataSource;
@property (nonatomic, weak) id delegate;

@end
```

The controller in this example is called the `EmployeeListViewController`. To communicate with the `EmployeeListView`, the `EmployeeListViewController` needs to declare that it will implement the `EmployeeListViewDataSource` and `EmployeeListViewDelegate` protocols. Controllers often have a reference to the view they are controlling in a property as well. In this example, the reference to the `EmployeeListView` is set through the `initWithEmployeeListView:` method. The following example shows how to do this in code.

```
#import "EmployeeListView.h"

@interface EmployeeListViewController : NSObject <EmployeeListViewDataSource,
EmployeeListViewDelegate>

- (id)initWithEmployeeListView:(EmployeeListView *)employeeListView;

@property (nonatomic, weak) EmployeeListView *employeeListView;

@end
```

In the implementation of the `EmployeeListViewController`, you need to add the actual implementation of the methods listed in the protocols it declares. You also need to set it as both the data source and the delegate of the `EmployeeListView`. There are a few ways to do this in Xcode, but as a simple example, it is done in the `initWithEmployeeListView:` method. The following example shows what the `EmployeeListViewController.m` file would look like.

```
#import "EmployeeListViewController.h"

@implementation EmployeeListViewController
- (id)initWithEmployeeListView:(EmployeeListView *)employeeListView
{
    self.employeeListView = employeeListView;
    self.employeeListView.dataSource = self;
    self.employeeListView.delegate = self;
}

- (NSArray *)getAllEmployees;
{
    // ask the model for all the employees
    // create an NSArray of all the employees
    // return the array
return allEmployeesArray;
}

- (void)didSelectEmployee:(Employee *)selectedEmployee;
{
    // display the employee detail view with the selected employee
}

@end
```

Now when the `EmployeeListView` needs to get the employees it needs to show or when an employee is selected, it can simply call the methods of the protocols using its references, as shown below.

```
// in the implementation of the EmployeeListView it would
// get the array of employees using this code

NSArray *allEmployees = [self.dataSource getAllEmployees];

// to tell its delegate that an employee was selected
// it would use this code

[self.delegate didSelectEmployee:selectedEmployee];
```

Using Blocks

Blocks are a relatively new programming construct in iOS. They were first introduced to iOS programming with the release of iOS 4. The concept behind them has been in other languages for quite some time. C and C++ have function pointers, C# has lambda expressions, and JavaScript has callbacks. If you have used any of these, the idea of blocks should be relatively easy to grasp. If you have not, you may want to research them to get a better understanding. This section gives a quick overview of the syntax and a high-level explanation that should be enough for you to understand how to use them when needed while building the **Bands** app.

A **block,** in its most simple definition, is a chunk of code that can be assigned to a variable or used as a parameter to another method. The ^ operator is used to declare a block, while the actual code is contained in between curly brackets. Blocks can have their own list of parameters as well as their own return type. Blocks in Objective-C have the special capability to use local variables declared within the scope they are defined.

You use blocks in the context of completion handlers to other methods. When you call these methods, you pass them whatever parameters they need and then also declare a block of code that gets invoked at some point within the method. For example, imagine a method that retrieves a string from a URL. The following example uses a fake object called `networkFetcher` that has a fake method called `stringFromUrl:withCompletionHandler:`, so you can get a feel for the syntax of blocks. This is just a quick overview of the block syntax. The code for this simple example is shown below.

```
NSString *websiteUrl = @"http://www.simplesite.com/string";

[networkFetcher stringFromUrl:websiteUrl
withCompletionHandler:^(NSString* theString) {

NSLog("The string: %@ was fetched from %@, theString, websiteUrl);
}];
```

This code example defines an inline block that is passed as a parameter to the `stringFromUrl:withCompletionHandler:` method. The ^ character signifies the start of the block. The (`NSString* theString`) portion is the parameter list being passed into the block. The code of the block is contained in the {} that follows. The code simply prints the string that was retrieved along with the URL string. When the code is executed, the `networkFetcher` class makes the network connection and does all the hard work of actually getting the string. When it is done, it calls the block of code with that string. This example shows how you will use blocks in this course but barely scrapes the surface of them. Blocks are a powerful tool that can change how software is designed and implemented.

Handling Errors

The last advanced concept that needs to be discussed is **error handling** in Objective-C. This is particularly important to developers coming from the Java background. Objective-C does have exceptions and `try`/`catch`/`finally` statements; although they are rarely used. You may have the urge to continue using them as you learn Objective-C but you should not, as it is not a common practice.

An exception in most OOP languages is a special object that gets created when an error has occurred. This object is then “thrown” to the system runtime. The runtime, in turn, looks for the code that “catches” the exception and handles the error in some manner. If nothing catches the exception, it is considered an uncaught exception and generally will crash the program that is running.

In Java and C#, exceptions are a normal part of the coding process and execution. A common use is when you want to do something but you are not sure if you will have everything you need to accomplish it at runtime, such as reading from a file. If the file does not exist, that is an error and the user should be notified. In Java or C# you would use a `try`/`catch`/`finally` statement to detect this situation. The following example shows some pseudo code of what this might look like in either of those languages without going into the implementation details.

```
public void readFile(string fileName)
{
   // create the things you would need to read the file
   FileReaderClass fileReader = new FileReaderClass(fileName);

   try
   {
         FileReaderClass.ReadFile();
   }
   catch(Exception ex)
   {
         // uh-oh, something happened. Alert the user!
   }
   finally
   {
         // clean up anything that needs to be cleaned up
   }
}
```

In Objective-C, this type of coding is rarely used. Instead, it is preferred to use the `NSError` class. Reading a file into a string in Objective-C takes this approach, as shown in the following code example.

```
- (void) readFile:(NSString *)fileName
{
   NSError *error = nil;
   NSString *fileContents = [NSString stringWithContentsOfFile:fileName
   encoding:NSASCIIStringEncoding error:&error];
   if(error != nil)
   {
         // uh-oh, something happened. Alert the user!
   }
}
```

In this example, the code first creates an `NSError` pointer with a `nil` value. It then passes the `error` instance to the method by reference instead of by value. The idea of passing a parameter by reference or by value is in both C# and Java. For typical method calls, the object is passed by value, which means the method gets a copy of the object and not the object itself. If the method modifies the object, it modifies only its own copy of the object and not the original. When the method returns, your object is the same and does not reflect those changes.

In the above code, you use the & (ampersand) symbol to pass the address of the object and not the object itself. If an error occurs while reading the file, the method creates a new `NSError` object and assigns it to that address. When the method returns, you check to see if your error points to an actual `error` object now or if it still points to `nil`. You know when to pass a parameter by reference when you see ** in the method signature. The full signature in this example is as follows:

```
+(instancetype)stringWithContentsOfFile:(NSString*)path
   encoding:(NSStringEncoding)enc error:(NSError **)error
```

Though using `try`/`catch` is not recommended, it is possible. There are base classes that throw exceptions. Using the `NSString` class again, you can get an exception if you try to get the character at an index that is out of bounds for the underlying character array. The following code example shows what this looks like.

```
- (void)someMethod
{
   NSString *myString = @"012";
   @try
   {
         [myString characterAtIndex:3];
   }

   @catch(NSException *ex)
   {
         // do something with the exception
   }
   @finally
   {
         // do any cleanup
   }
}
```

The Objective-C way to handle this situation would be to add a check to the code, making sure that the method will not fail before calling it. The following code example shows how you would use this approach instead of a `try`/`catch` statement.

```
- (void)someMethod
{
   NSString *myString = @"012";

   if([myString length] >= 3)
   {
         [myString characterAtIndex:3];
   }

   else
```

```
        {
                // nope, can't make that method call!
        }
    }
```

ADDITIONAL KNOWHOW

The `NSObject` class also has a new method that performs both the `alloc` and `init` method calls and then returns the pointer. If you use this method, any overridden `init` methods that take parameters will not be called. Instead, you need to set the values of the member variables after the object is instantiated.

LAB CONNECT

During the lab hour of this session, you will install the Objective-C development environment and create a basic application on palindrome.

Cheat Sheet

- Objective-C is the language used to develop both Mac and iOS applications. It is an OOP language with similarities to Java and C#.
- The basic building blocks of all OOP languages are objects and the classes that define them.
- There are two mainstream approaches to memory management. Java and C# use Garbage Collection, whereas Objective-C uses Manual Reference Counting where the developer is responsible for the life cycle of objects in use.
- Automatic Reference Counting (ARC) is a modern approach to manual reference counting. ARC takes the responsibility of keeping track of reference counts out of the hands of developers and passes it instead to the compiler.
- OOP languages recommend using the public getter and setter methods to change member values while keeping the actual member variables private. Objective-C implements this concept using class properties.
- Higher-level programming languages are designed with built-in data structures such as arrays and dictionaries that developers use to organize data to make their code fast and efficient. Objective-C includes the basic data structures: `NSArray`, `NSSet`, and `NSDictionary`.
- The Model-View-Controller (MVC) design pattern is a high-level design pattern that separates all the components in a piece of software into three roles: model, view, and controller. These roles help to promote independence and reusability.
- The Cocoa framework uses the concept of delegates and protocols to facilitate the MVC design pattern by formalizing how components in different roles communicate and pass data to each other.
- All OOP languages include exceptions and errors. The way they are used, however, can vary widely. Understanding how Objective-C and Cocoa approach them is fundamental to writing iOS applications.