iPhone Objective-C Exercises

About These Exercises

The only prerequisite for these exercises is an eagerness to learn. While it helps to have a background in object-oriented programming, that is not a requirement. The exercises explain new concepts when they are needed, in a step-by-step, easy-to-understand, manner.

If you are an existing C/C++, C#, Flash, Java, PHP, Python, Ruby or similar developer, then you'll enjoy how Objective-C 2.0 handles object-oriented programming. Objective-C was influenced by SmallTalk, one of the early object-oriented programming languages. In turn, Objective-C influenced the Java language designers, which in turn influenced the C# language designers.

iOS uses Objective-C 2.0 as its programming language, which is a superset of the C programming language. As you work through these exercises, the Objective-C 2.0 concepts will be covered as you come to them. It's OK if you do not have experience with Objective-C 2.0; just start Exercise 1 and work your way forward.

About the iOS SDK

As a point of reference, I performed all of these steps in a couple of hours, and that included downloading and installing the iOS SDK.

Step 1. Get an Intel-based Mac.

The lowest entry-level MacBook will do fine for iPhone programming. (Note that an older Mac that used the PowerPC processor will not work for the iPhone SDK; it must be an Intel-based Mac. All of the Macs over the last several years have been Intel-based, so this should not be an issue.)

Step 2. Join Apple Developer Connection, or ADC.

To join, point your browser to https://developer.apple.com/programs/register/ and register as an Apple developer (free).

Step 3. Download the iOS SDK (Software Developers Kit).

Once you have registered on the ADC, you can download the iOS SDK from http://developer.apple.com/iphone/.

The download is quite large, so it may take some time depending on your network speed.

Step 4. Install the SDK.

After you download the iOS SDK, install by clicking on the "yellow box" that says "Xcode and iPhone SDK".

Step 5. Startup Xcode.

After installation, you can startup Xcode, the tool you will use to perform these exercises.

To startup Xcode, click on Finder (the "Smiling Face" icon at the lower left of your desktop), then select Macintosh HD near the top-left of the Finder window, then select the Developer folder on the right-side of the Finder window, then select the Applications folder, then double-click on Xcode.

Stated differently, the path to Xcode is this: /Volumes/Macintosh HD/Developer/Applications/Xcode.app.

Exercise 1: Create an iPhone Window-based App

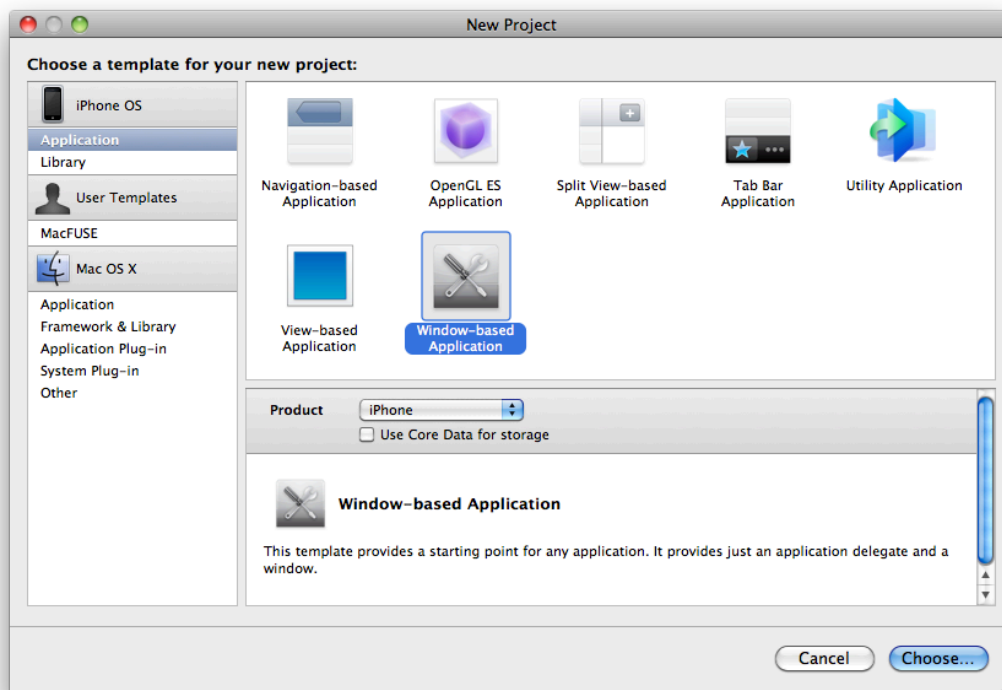
In this exercise, you will create an iPhone Window-based App named ObjCDemo. The ObjCDemo will serve as the foundation for the exercises in this book. At the end of all the exercises, you will submit the completed ObjCDemo Project.

Step 1. Startup Xcode:

1. Click on Finder (the "Smiling Face" icon at the lower left of your desktop).
2. Select Macintosh HD near the top-left of the Finder window.
3. Select the Developer folder on the right-side of the Finder window.
4. Select the Applications folder.
5. Double-click on Xcode.

Stated differently, the path to Xcode is this: /Volumes/Macintosh HD/Developer/Applications/Xcode.app.

Step 2. Use the Xcode main menu to select File > New Project.
You will see the Xcode New Project window appear:



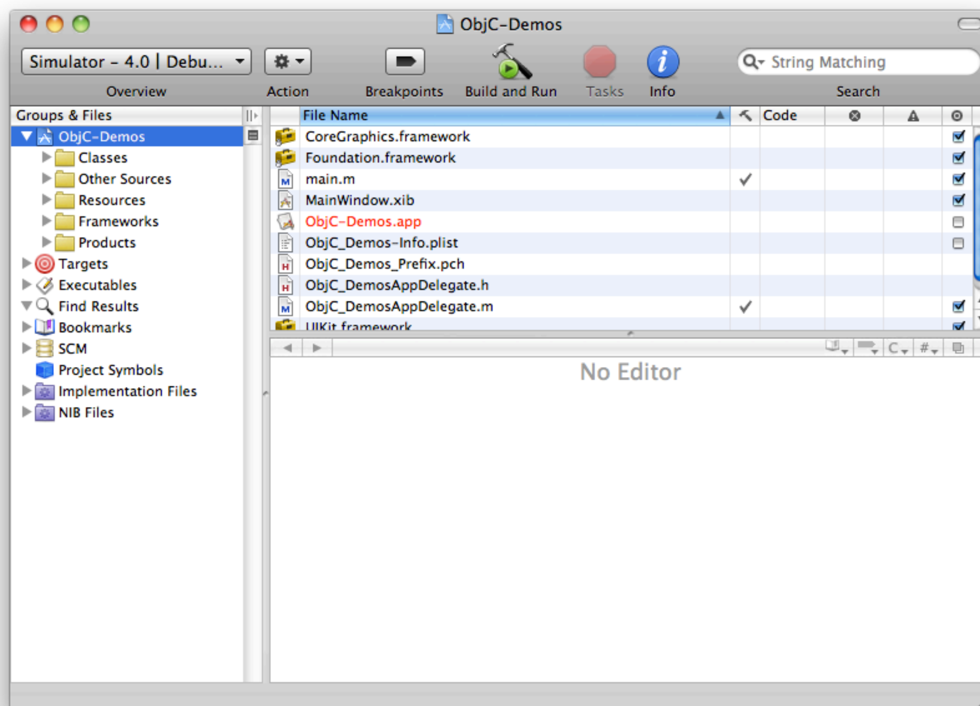
Step 3. On the left-side of the New Project window, under the iPhone OS heading, make sure that Application is selected.

On the right-side of the New Project window, make sure that Window-based Application is selected.

Step 4. Click on the Choose button.

Step 5. In the Save As field, enter ObjC-Demos for the project name.

Step 6. Click on the Save button. Xcode will display the project files for the ObjC-Demos project.



Step 7. Near the top-middle of Xcode is a green button labeled Build and Run. Click on that button to build and run the default iPhone Window-based app.

You should see the default iPhone Window-base app running in the iPhone Simulator.

Step 8. Press the HOME key on the iPhone Simulator to stop the default application from running.

Exercise 2: main(), printf(), "C String"

In this exercise, you will add your first code to the existing Objective-C 2.0 code that was generated by the iPhone Window-based application template.

The skills you will learn in this exercise include:

- main()
- printf()
- "C String"

Step 1. With Xcode still open in the ObjC-Demos project from the previous exercise, observe the folder titled Other Sources.

Step 2. Click on the small triangle to the left of the Other Sources folder. You should see the list of files displayed.

Step 3. Observe the file named main.m. A file with a .m file extension is an Objective-C source-code file.

Step 4. Click on the file named main.m, displaying it in the Xcode editor pane. You will see code that looks like this:

```
// main.m
// ObjC-Demos

#import <UIKit/UIKit.h>

int main(int argc, char *argv[]) {

    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

This Objective-C code is common for all iPhone applications. While it is code that you will rarely, if ever change, it is important to understand each line of code, for the concepts used in this file are similar to concepts used in other iPhone Objective-C source files.

Any lines that have a // in them are comments. These lines are ignored during the build process; they are of use only to humans who are reading the source code.

```
// main.m
```

```
// ObjC–Demos
```

The next line imports, or includes, pre-written code into this file. The details of the pre-written code are not important at this time. Rather, just know that a import statement includes pre-written code into the file.

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

Note for C/C++ developers: The import is an improved version of the C/C++ include directive. It works similar to include, except that the import ensures that a file is included only once; there is no need for ifndef/define protection.

The next line of code is the entry-point for all iPhone applications; the code starts running when the main() function is called.

```
int main(int argc, char *argv[]) {
```

Note for C/C++ developers: iPhone programs do not use the input parameters, i.e. `argc`, and `argv`, nor the return value, i.e., `int`.

Unless you are a Objective-C software developer, the next line of code looks strange:

```
NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
```

We will come back to this line of code in a later exercise. No worries if you do not understand what that line does, for we will cover the specifics in a later exercise.

For now, here is everything the line of code does:

1. Declares a variable named pool that is a pointer, *, to a NSAutoreleasePool.
2. Sends the alloc method to the NSAutoreleasePool class: [NSAutoreleasePool alloc]. The result is that memory for that class is allocated, and a pointer to that memory is returned.
3. Using the pointer to the object that was returned from the alloc method, sends the init method to tell the object to initialize itself.
4. Assigns the newly allocated (alloc) and initialized (init) pointer to the pool pointer.

Again, no worries if you do not fully the NSAutoreleasePool line of code; more to come on that in a later exercise!

The next line of code calls the UIApplicationMain() function; the result is that your application starts receiving method calls from the UIApplication object. You can ignore the details of the UIApplication object at this point.

```
int retVal = UIApplicationMain(argc, argv, nil, nil);
```

Recall a few lines back when the pool object was allocated and initialized? The next line of code sends a release method to the pool object, which results in the memory for the pool object being released.

```
[pool release];
```

Again, the details of the release are not important now; a separate exercise covers object alloc, init, and release in detail.

The last line of code returns the value retval to the program that started this program.

```
return retVal;
```

Step 5. Use Xcode to edit main.m to match the code shown below.

```
// main.m
// ObjC-Demos

#import <UIKit/UIKit.h>

int main(int argc, char *argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    // Display message on Xcode Console
    printf("Hello C-String\n");

    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

Step 6. After entering the code shown above, use the Xcode main menu to select Run > Console. You should see the Xcode console appear.

Step 7. In the Xcode console, click on the Clear Log button to clear the output on the console.

Step 8. In the Xcode console, click on the green Build and Run button to build and run the code, observing the output on the console:

```
Hello C-String
```

An important point: This exercise used a C string, which is represented by the double-quotes around the string, i.e., "C String". In the next exercise, you will see how to represent an NSString.

Exercise 3: NSLog(), @"NSString"

In this exercise, you will use the NSLog() function, and NSString.

Step 1. Assuming that you are continuing from the previous exercise, add the NSLog() function as shown below.

```
// main.m
// ObjC-Demos

#import <UIKit/UIKit.h>

int main(int argc, char *argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    // Display message on Xcode Console
    printf("Hello C-String\n");

    NSLog(@"Hello NSString");

    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

In this exercise, you are using the NSLog() function, which comes from the Foundation framework.

A framework provides a set of pre-written code that other applications can call as needed.

The Foundation framework is used in all iOS Objective-C programs; it provides code that every application can use, e.g., the NSLog() function.

The NSLog() function displays a NSString on the Xcode console.

The NS is the prefix used in code from the Foundation framework. The NS stands for Next Step, the company that Apple purchased to obtain the Next Step code base, which was then used for both Mac OS X and iOS.

An important point: Notice that the NSString has a @ at the beginning of a string, while a C string does not:

- C Text String: "Hello C String"
- NSString: @"Hello NSString

In iPhone Objective-C programming, you will almost always use an NSString string, e.g. @"String", instead of a C string, e.g., "String".

Step 2. After entering the code, use the Xcode main menu to select Run > Console, then click on the green Build and Run, observing the output in the console.

```
Hello C-String
2010-08-15 21:57:56.097 ObjC-Demos[59366:207] Hello NSString
```

From the above output, observe that the first line of console output, created by the printf() function, displayed the string output without any date/time/program-name stamp.

The second line of console output was created by the NSLog() function. Observe how output generated by NSLog() includes a date, time, and program name as a prefix to the output.

Exercise 4: @interface, @implementation, @end

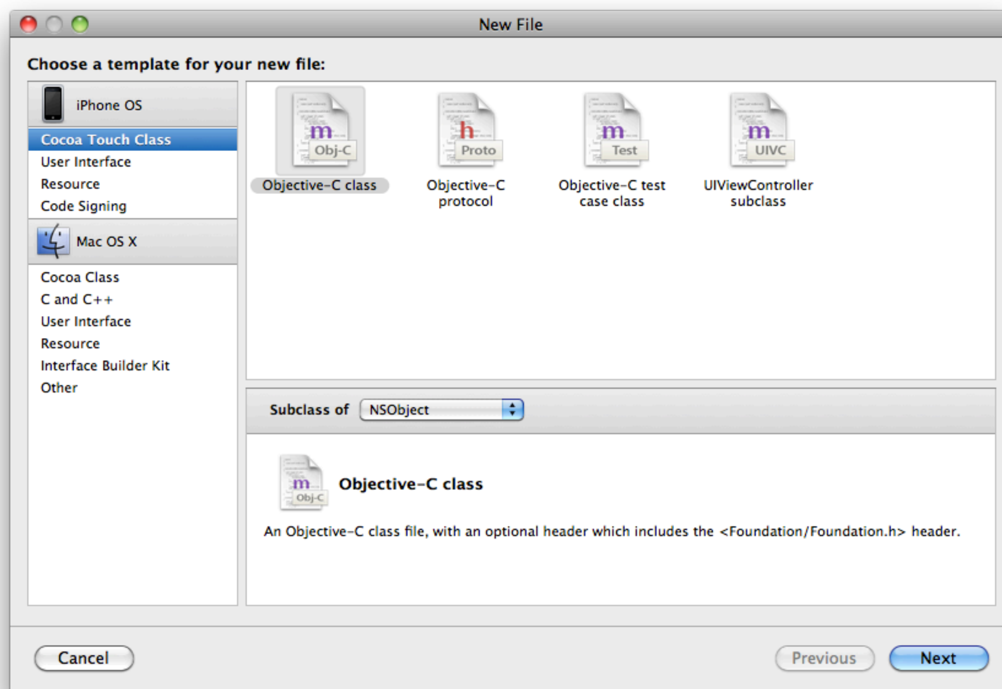
In this exercise, you will use Xcode to generate a new Objective-C class named Hello.

Step 1. With the same ObjC-Demos project open from the previous exercises, click on the Xcode folder named Classes to select it.

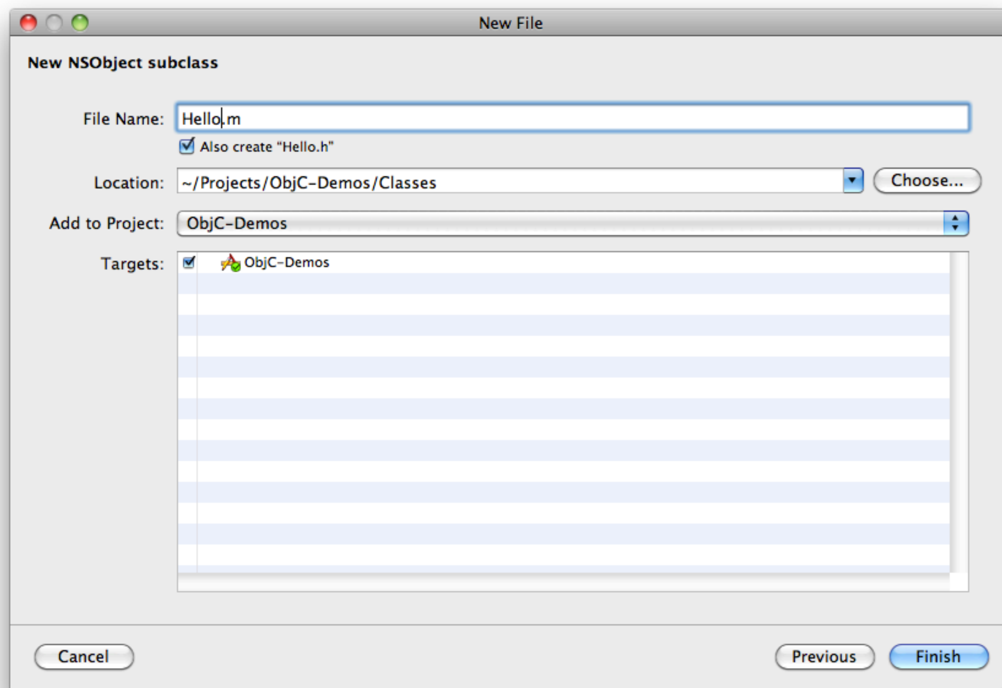
Step 2. Use the Xcode File > New File main menu selection to display the New File window.

Step 3. Select iPhone OS / Cocoa Touch Class on the left-side of the New File window, then select Objective-C class in the right-side of the New File window, then select Subclass of NSObject, then click the Next button.

NOTE: Be sure that you select Objective-C class and Subclass of NSObject before clicking on the Next button.



Step 4. In the New NSObject Subclass window, enter Hello.m as the filename, making sure that "Also create Hello.h" is checked, then click the Finish button.



Step 5. In Xcode, under the Classes folder, click on the Hello.h file, or the header file, to display it in the Xcode editor pane.

```
// Hello.h
// ObjC-Demos

#import <Foundation/Foundation.h>

@interface Hello : NSObject {

}

@end
```

Step 6. Observe that the file begins with comments:

```
// Hello.h
// ObjC-Demos
```

Step 7. The next line imports, or includes, pre-written code from the Foundation framework. This includes the declaration of NSObject, which is used later in this file.

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

Step 8. The next several lines of code declare the interface to a class named Hello:

```
@interface Hello : NSObject {

}

@end
```

A class is a named template, e.g. Hello. The named template is used when creating new instances of the class; all instances of the class will be similar, for they follow the same template.

NOTE: At this point, the class template is empty; it does not have any instance variables or methods. Both instance variables and methods will be added in later exercises.

For iPhone Objective-C programming, all classes inherit from a parent class. In this case, the parent class is NSObject, which is provided by the Foundation framework. Using the Objective-C inheritance operator, :, the Hello class gains the properties and methods that are already defined by Foundation class NSObject.

The line of code with the Objective-C keyword @interface declares that a class named Hello will inherit, :, from the existing class named NSObject.

For now, the interface is empty, and ends with the Objective-C @end keyword.

To repeat an important point: A class is a template that defines a set of properties and methods. The interface declares how those properties and methods can be accessed. For this exercise, the properties and methods of Hello are empty; later exercises will add both properties and methods to the Hello interface.

Step 9. Now select the Hello.m, or implementation file, into the Xcode editor pane.

```
// Hello.m
// ObjC-Demos

#import "Hello.h"

@implementation Hello
@end
```

Step 10. This file also begins with comments:

```
// Hello.m
// ObjC-Demos
```

Step 11. The next line imports, or includes, the contents of the Hello.h header file.

```
#import "Hello.h"
```

The next two lines begin and end the implementation of the Hello class:

```
@implementation Hello
@end
```

The Objective-C keyword @implementation begins the implementation of a class. As you will see in later exercises, all the code between the @implementation and the @end implements the Hello class. For now, the class is empty, i.e, it uses the default implementation as provided by the parent class, NSObject.

Step 12. Review the previous steps to this exercise. Makg sure that you understand each step.

Especially make sure you understand the Objective-C @interface/@end keywords for declaring the interface to a new class.

Also make sure that you understand that the Hello class inherits from the existing NSObject class; the Hello class inherits the functionality provided by NSObject.

Finally make sure that you understand the @implementation/@end keywords, for defining the implementation; for this exercise, the implementation for this class is empty.

Exercise 5: +alloc, -init, -release

In this exercise, you will continue your journey into Objective-C object-oriented programming:

You will use the +alloc class method to allocate an instance of the Hello class.You will use the -init instance method to initialize the instance of the Hello class.You will use the -release instance method to release the memory held by the object.

Step 1. Assuming that you are continuing from the previous exercise, edit main.m to match the following:

```
// main.m
// ObjC-Demos

#import <UIKit/UIKit.h>
#import "Hello.h"
...
```

The above line of code imports, or includes, the interface declaration for the Hello class.

Step 2. Continue editing the code:

```
...
int main( int argc, char *argv[] ) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    // Display message on Xcode Console
    printf("Hello C-String\n");
    NSLog(@"Hello NSString");

    // _Declare_ a pointer to an instance of a Hello class
    Hello *ptr;
```

As stated by the comment, the added code declares a variable named ptr, that is a pointer, *, to an instance of a class named Hello.

An instance of a class is an object; so the variable ptr is a pointer to an object of type Hello.

An important point: While the variable ptr has been declared, it has not yet been initialized.

Step 3. Enter the next lines of code:

```
// _Allocate_ an instance of a Hello class
ptr = [Hello alloc];
```

The [Hello alloc] Objective-C syntax says to send the Hello class the alloc class method. The result is that the Hello class allocates an instance of itself, and returns a pointer to itself, which is stored into the ptr variable.

An important point: The Objective-C syntax for sending a class method to a class is this: [ClassName classMethod];

Step 4. Continue by entering code:

```
// _Initialize_ the instance of the Hello class
ptr = [ptr init];
```

The [ptr init] Objective-C syntax says to send the object pointed to by ptr the init instance method. The result is that the ptr object does whatever it needs to do to initialize itself; it then returns a pointer to itself, which is stored in ptr.

An important point: In Objective-C, creating objects (instance of classes) is always a two-step process: allocate, then initialize. This is so common that you'll often see this syntax: ptr = [[Hello alloc] init], which allocates, then initializes, all in one statement.

Step 5. Enter these additional lines of code:

```
// Display info about the instance of the class.
// Four ways of saying the same thing
NSLog( @"%@", ptr );
NSLog( @"%@", [ptr description] );
NSLog( @"%p", ptr );
printf( "%p", ptr );
```

Here the NSLog function will display a formatted string, where the format is determined by the "%@" formatting sequence, along with the value of the ptr object.

An important point: The %@ syntax is a placeholder in an Objective-C string. The value of the placeholder is replaced with an Objective-C string that is return by the object when sent the description method. That is, internal to the NSLog() function, it does the equivalent of [ptr description], which returns an Objective-C string that describes the object.

Step 6. Enter this final line of code, which sends the release message to the object. Sending the release method tells the object that you are finished with it, and it can release its memory.

```
        // Release the object's memory
        [ptr release];
        int retVal = UIApplicationMain( argc, argv, nil, nil );
        [pool release];
        return retVal;
    }
```

Step 7. Confirm that your code matches the following:

```
    // main.m
    // ObjC-Demos

    #import <UIKit/UIKit.h>
    #import "Hello.h"

    int main( int argc, char *argv[] ) {
        NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
        // Display message on Xcode Console
        printf( "Hello C-String\n" );
        NSLog( @"Hello NSString" );
        // _Declare_ a pointer to an instance of a Hello class
        Hello *ptr;
        // _Allocate_ an instance of a Hello class
        ptr = [Hello alloc];
        // _Initialize_ the instance of the Hello class
        ptr = [ptr init];
        // Display info about the instance of the class.
        // Four ways of saying the same thing
        NSLog( @"%@", ptr );
        NSLog( @"%@", [ptr description] );
        NSLog( @"%p", ptr );
        printf( "%p\n", ptr );
        // Release the object's memory
        [ptr release];
        int retVal = UIApplicationMain( argc, argv, nil, nil );
        [pool release];
        return retVal;
    }
```

Step 8. Use the Xcode Run > Console command to run the code in the console. Your output should be similar to this:

```
Hello C-String
2010-08-16 00:24:54.765 ObjC-Demos[10603:207] Hello NSString
2010-08-16 00:24:54.767 ObjC-Demos[10603:207] <Hello: 0x4707100>
2010-08-16 00:24:54.768 ObjC-Demos[10603:207] <Hello: 0x4707100>
2010-08-16 00:24:54.769 ObjC-Demos[10603:207] ptr: 0x4707100
ptr: 0x4707100
```

Observe the lines of output that show the class name, Hello, followed by the pointer value, 0x4707100, of where the object resides in memory.

An important point: As you have seen, by default output of the object is to display the class name, along with the value of the pointer to the object.

Step 9. Review the previous steps to this exercise. Make sure that you understand each step.

Especially make sure that you understand the Objective-C syntax of both [ClassName classMethodName] and [objectPointer instanceMethodName].

Also make sure that you understand that the methods used in this exercise, alloc and init, were inherited by the Hello class from the NSObject class.

Finally make sure that you understand the importance of sending your objects the release method when you are finished with them.

To repeat an important point: The alloc, init, and release methods were inherited from the NSObject.

Exercise 6: Class Methods (+) and Instance Methods (-)

In this exercise, you will continue your journey into Objective-C object-oriented programming:

You will add and test a class method to the Hello class.You will add and test an instance method to the Hello class.

At the end of this exercise, you will have a Hello class that has a class method named demoClassMethod and an instance method named demoInstanceMethod.

Step 1. Assuming that you are continuing from the previous exercise, edit Hello.h to match the following:

```
// Hello.h
// ObjC-Demos

#import <Foundation/Foundation.h>

@interface Hello : NSObject {
}

+(void)demoClassMethod;
-(void)demoInstanceMethod;

@end
```

The + character is the Objective-C syntax for a class method. A class method can be called at anytime; you do not not need an instance of the class to call the class method: [Hello demoClassMethod].

The generic format of the class method declaration that does not have any input parameters is this: +(ReturnType)classMethodName;

Here, the return type is void, which says do not return anything. The class method name is demoClassMethod.

An important point: This method does not have any input parameters. The exercise after this one will cover the syntax for input parameters.

The - character is the Objective-C syntax for a instance method. To call an instance method, you need a pointer to the instance. That is, a pointer to the object:
[ptr demoInstanceMethod].

The generic format of the instance method declaration that does not have any input parameters is this: -(ReturnType)instanceMethodName;

Step 2. Edit Hello.m to match the following:

```
// Hello.m
// ObjC-Demos

#import "Hello.h"

@implementation Hello

+(void)demoClassMethod {
    NSLog( @"%s", __FUNCTION__ );
}

-(void)demoInstanceMethod {
    NSLog( @"%s", __FUNCTION__ );
}

@end
```

Step 3. Edit main.m to match the following:

```
    ...same code as before...
    // Send Hello a class method
    [Hello demoClassMethod];
    // Send ptr a instance method
    [ptr demoInstanceMethod];
    // Release the object's memory
    [ptr release];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

Observe how the code sends the class method demoClassMethod to the Hello class. There was no need to create an instance of the class before sending the class method.

The [ptr demoInstanceMethod] Objective-C syntax says to send the object pointed to by ptr the demoInstanceMethod.

Step 4. Use the Xcode Run > Console menu to build and run. Your output should be similar to the following:

```
Hello C-String
... Hello NSString
...
...
... ptr: 0x4707120
```

```
ptr: 0x4707120
... +[Hello demoClassMethod]
... -[Hello demoInstanceMethod]
```

Observe the last two lines of output that show the results of calling both the class method (+), and the instance method (-). As you see, the __FUNCTION__ used in the NSLog method displays the name of the method: NSLog(@"%s", __FUNCTION__). (A %s is a C-string while a %@ is an NSString.)

Step 5. Review the previous steps to this exercise. Make sure that you understand each step.

Especially make sure that you understand the Objective-C syntax for class methods and instance methods.

Also make sure that you understand that the __FUNCTION__ can be used in NSLog() to show the name of the method name.

Exercise 7: Method Parameters

In this exercise, you will continue your journey into Objective-C object-oriented programming, adding two class methods to your Hello class.

You will add a demoOneParam: method to the Hello class. You will add a demoTwoParams:paramTwo: to the Hello class.

At the end of this exercise, you will have a Hello class that has a class method named demoClassMethod that has no parameters, a class method named demoOneParam: that has one parameter, and a class method named demoTwoParams:paramTwo: that has two parameters. In addition, the demoTwoParams:paramTwo: will return an integer result.

An important point: The : used in an Objective-C method name means that a parameter follows the colon (:). A method with a single parameter has a single colon (:) in the name; a method with two parameters has two colons (:) in the name.

Step 1. Use your text editor to edit Hello.h:

```
// Hello.h
// ObjC-Demos

#import <Foundation/Foundation.h>

@interface Hello : NSObject {
}

+(void)demoClassMethod;
-(void)demoInstanceMethod;
+(void)demoOneParam:(int)theParam;
+(int)demoTwoParams:(int)numOne paramTwo:(int)numTwo;

@end
```

The demoOneParam: method takes a single int (integer) parameter named theParam; the method does not return a value, i.e., void.

The demoTwoParams:paramTwo: takes two int (integer) parameters, and returns an int result.

Although this exercise uses class methods (+), the same syntax is used for instance methods (-).

The generic format of the class method declaration with one parameter is this:

```
+(ReturnType)classMethodName:(TypeOfParameter)theParameterName;
```

The generic format of the class method declaration with two parameters is this:

```
-(ReturnType)instanceMethodName:(Type)paramOneName
                 nameOfParamTwo:(Type)paramTwoName;
```

Step 2. Now edit Hello.m to match the following:

```
// Hello.m
// ObjC-Demos

#import "Hello.h"

@implementation Hello

+(void)demoClassMethod {
    NSLog(@"%s", __FUNCTION__);
}

-(void)demoInstanceMethod {
    NSLog(@"%s", __FUNCTION__);
}

+(void)demoOneParam:(int)theParam {
    NSLog(@"%s theParam: %d", __FUNCTION__, theParam);
}

+(int)demoTwoParams:(int)numOne paramTwo:(int)numTwo {
    NSLog(@"%s numOne: %d, numTwo: %d",
            __FUNCTION__, numOne, numTwo);
    return numOne + numTwo;
}

@end
```

Observe how the demoOneParam: method uses NSLog() to display the value of theParam:

```
NSLog(@"theParam: %d", theParam);
```

The %d is a placeholder for the value of theParam; the result is that the value of theParam will appear on the terminal.

The demoTwoParams:paramTwo: method also uses NSLog() to display the input parameters. It also adds the two input values, and returns the sum to the caller of the method.

```
return numOne + numTwo;
```

Step 3. Next modify main.m:

```
// main.m
// ObjC-Demos

#import <UIKit/UIKit.h>
#import "Hello.h"

int main(int argc, char *argv[]) {
    ...same code as before...

    // Send Hello a class method
    [Hello demoClassMethod];
    // Send ptr a instance method
    [ptr demoInstanceMethod];
    // Send class methods with parameters
    [Hello demoOneParam:456];
    int sum = [Hello demoTwoParams:10 paramTwo:20];
    NSLog(@"sum: %d", sum);
    // Release the object's memory
    [ptr release];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

Observe how the code sends the class method demoOneParam: to the Hello class: [Hello demoOneParam:456]. There was no need to create an instance of the class before sending the class method.

Also observe how the demoTwoParams:paramTwo: method is called, storing the results into an int (integer) variable named sum.

The NSLog() is used once again, use the formatting string @"sum: %d" to format the value of sum before display on the terminal.

Step 4. Do a Run > Console and observe the results.

```
...same output as before...
... +[Hello demoClassMethod]
... -[Hello demoInstanceMethod]
... +[Hello demoOneParam:] theParam: 456
... +[Hello demoTwoParams:paramTwo:] numOne: 10, numTwo: 20
... sum: 30
```

Observe the lines of output that show the results of calling the methods with parameters.

Step 5. Review the previous steps to this exercise. Make sure that you understand each step.

Exercise 8: Instance Variables

In this exercise, you will continue your journey into Objective-C object-oriented programming, adding a instance variable to your Hello class.

At the end of this exercise, you will have a Hello class that has a class method named demoClassMethod that has no parameters, a class method named demoOneParam: that has one parameter, and a class method named demoTwoParams:paramTwo: that has two parameters, and a instance variable named debugMode. Finally you will add two instance methods to that you can get and set the values of the instance variable.

Step 1. Edit Hello.h to match the following:

```
// Hello.h
// ObjC-Demos

#import <Foundation/Foundation.h>

@interface Hello : NSObject {
    BOOL debugMode;
}

+(void)demoClassMethod;
-(void)demoInstanceMethod;
+(void)demoOneParam:(int)theParam;
+(int)demoTwoParams:(int)numOne paramTwo:(int)numTwo;
-(BOOL)debugMode;
-(void)setDebugMode:(BOOL)value;

@end
```

Observe the Objective-C syntax for adding an instance variable named debugMode. A BOOL data type has a value of either NO or YES. In iPhone Objective-C programming, you will often use the BOOL data type, checking if it is NO or YES, or setting to one of those values yourself.

An important point: An instance variable is a value that is part of the instance data. Every instance of the Hello class gets a separate, private copy of the instance variables.

Another important point: While instance variables can be accessed inside the class instance methods, the instance variables cannot be accessed outside the class instance methods, unless you provide instance methods for getting/setting the values.

For this exercise, two instance methods are provided. The setDebugMode: instance method is for setting the value, and the debugMode method is for getting the values.

Step 2. Now edit Hello.m to implement two new methods:

```
// Hello.m
// ObjC-Demos

#import "Hello.h"

@implementation Hello

+(void)demoClassMethod {
    NSLog(@"%s", __FUNCTION__);
}

-(void)demoInstanceMethod {
    NSLog(@"%s", __FUNCTION__);
}

+(void)demoOneParam:(int)theParam {
    NSLog(@"%s theParam: %d", __FUNCTION__, theParam);
}

+(int)demoTwoParams:(int)numOne paramTwo:(int)numTwo {
    NSLog(@"%s numOne: %d, numTwo: %d",
            __FUNCTION__, numOne, numTwo);
    return numOne + numTwo;
}

-(BOOL)debugMode {
    return debugMode;
}

-(void)setDebugMode:(BOOL)value {
    debugMode = value;
}

@end
```

Observe how the setDebugMode: method accepts a BOOL input parameter named value, and stores it in the debugMode instance variable.

Also, observe how the debugMode method returns the value of the debugMode instance variable.

Step 3. Edit main.m to match the following:

```
        ...same code as before...
        // Test debugMode
        NSLog(@"debugMode: %d", [ptr debugMode]);
        [ptr setDebugMode:YES];
        NSLog(@"debugMode: %d", [ptr debugMode]);
        // Release the object's memory
        [ptr release];
        int retVal = UIApplicationMain(argc, argv, nil, nil);
        [pool release];
        return retVal;
    }
```

An important point: The value of a instance variable is set to zero when created, so in this case the default value of debugMode is NO, or 0. (NO is 0 and YES is 1.)

The [ptr debugMode] Objective-C syntax sends the debugMode method to the ptr object, which returns the current value of debugMode. The value returned is formatted by NSLog() as a decimal value: @"debugMode: %d".

The [ptr setDebugMode:YES] Objective-C syntax sends the setDebugMode: method to the >ptr object, which sets the instance variable to the new value.

Step 4. Issue a Run > Console and observe the result.

```
...
... debugMode: 0
... debugMode: 1
```

Observe the last two lines of output, showing the values of debugMode.

Step 5. Review the previous steps to this exercise. Make sure that you understand each step.

Exercise 9: Properties

In this exercise, you will continue your journey into Objective-C object-oriented programming, adding properties to your Hello class, by using the Objective-C 2.0 @property and @synthesize keywords.

An important point: The @property and @synthesize keywords were added in Objective-C 2.0.

At the end of this exercise, you will have:

- a Hello class that has a class method named demoClassMethod that has no parameters
- a class method named demoOneParam: that has one parameter
- a class method named demoTwoParams:paramTwo: that has two parameters
- an instance variable named debugMode
- two methods to get/set the instance variable
- a property named count

Step 1. Edit Hello.h to match the following:

```
// Hello.h
// ObjC-Demos

#import <Foundation/Foundation.h>

@interface Hello : NSObject {
    BOOL debugMode;
    NSInteger count;
}

@property (nonatomic) NSInteger count;

+(void)demoClassMethod;
-(void)demoInstanceMethod;
+(void)demoOneParam:(int)theParam;
+(int)demoTwoParams:(int)numOne paramTwo:(int)numTwo;
-(BOOL)debugMode;
-(void)setDebugMode:(BOOL)value;

@end
```

Observe that an additional instance variable was added: NSInteger count. A NSInteger is a type defined by the Foundation framework that is an int (integer).

```
typedef int NSInteger;
```

Observe the @property Objective-C 2.0 syntax. This syntax declares that the member variable will have two instance methods, one to set the value, the other to read the value. As you will see in the next step, the @property used in the @interface section works together with the @synthesize used in the @implementation section.

Step 2. Edit Hello.m to match the following:

```
// Hello.m
// ObjC-Demos

#import "Hello.h"

@implementation Hello
@synthesize count;

...same code as before...
```

The @synthesize Objective-C 2.0 syntax instructs the build process to generate two methods: a getter method named count and a setter method named setCount. The result is that you can access the instance variable using the dot, ".", to access the instance variable:

```
NSLog(@"ptr.count: %d", ptr.count);
```

An important point: In the previous exercise, you manually created a getter (debugMode) and setter (setDebugMode) instance method to access the instance variable debugMode. The use of @property and @synthesize also gives you access to the instance variable, but using the often more familiar dot notation.

Step 3. Edit main.m to match the following:

```
    ...same code as before...
    // Access count using Objective-C 2.0
    NSLog(@"ptr.count: %d", ptr.count);
    ptr.count = 123;
    NSLog(@"ptr.count: %d", ptr.count);
    // Access count using Objective-C 1.0
    NSLog(@"[ptr count]: %d", [ptr count]);
    [ptr setCount:456];
    NSLog(@"[ptr count]: %d", [ptr count]);
    // Release the object's memory
    [ptr release];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

Observe the use of the Objective-C 2.0 dot notation to access the instance variable as a property: prt.current. This dot notation is used to both get and set the instance variable.

Also observe the equivalent operation using Objective-C 1.0. The [ptr count] gets the value and the [ptr setCount:456] sets the value.

An important point: You will find some iPhone Objective-C developers prefer the Objective-C 2.0 property syntax, while others prefer the Objective-C 1.0 syntax. Sometimes you will even find code that mixes the two methods. In any case, you now know that regardless of syntax, the end result is the same.

Step 4. Use the Run > Console to build and run the code, observing the output:

```
...same output as before...
... debugMode: 0
... debugMode: 1
... ptr.count: 0
... ptr.count: 123
... [ptr count]: 123
... [ptr count]: 456
```

Observe the last four lines of output, showing the values of the property currentCount.

Step 5. Review the previous steps to this exercise. Make sure that you understand each step.

Exercise 10. @protocol (Delegates)

Often in iOS programming, one object needs to call methods that are implemented in another object.

For example, when you start your iOS application, an instance of a UIApplication object is created. The UIApplication object then calls methods in your Application Delegate Object.

The purpose of the @protocol keyword is to declare a protocol. In iOS, the name of the protocol always has the name delegate somewhere in the name.

For example, the UIApplication object calls your Application Delegate object by using the UIApplicationDelegate protocol.

In this exercise, you will add a HelloDelegate protocol. The protocol will have a single method named sayHello.

Step 1. Edit Hello.h to match the following:

```
// Hello.h
// ObjC-Demos

#import <Foundation/Foundation.h>

@protocol HelloDelegate
-(void)sayHello;
@end

@interface Hello : NSObject {
    BOOL debugMode;
    NSInteger count;
    id delegate;
}

@property (nonatomic, assign) id delegate;
@property (nonatomic) NSInteger count;

+(void)demoClassMethod;
-(void)demoInstanceMethod;
+(void)demoOneParam:(int)theParam;
+(int)demoTwoParams:(int)numOne paramTwo:(int)numTwo;
-(BOOL)debugMode;
-(void)setDebugMode:(BOOL)value;

@end
```

Step 2. Edit Hello.m to match the following:

```
// Hello.m
// ObjC-Demos

#import "Hello.h"

@implementation Hello
@synthesize count;
@synthesize delegate;
...same code as before...
```

Step 3. Use Xcode to do a Build > Build to confirm that you do not have any syntax errors.

Step 4. Click on the Classes folder to select, then select the Xcode menu File > New File. Then create a new class named Demo. Be sure that both the Demo.m and Demo.h files are created.

Demo.h
Demo.m

Step 5. Edit Demo.h to match the following:

```
// Demo.h
// ObjC-Demos

#import <Foundation/Foundation.h>
#import "Hello.h"

@interface Demo : NSObject
<HelloDelegate> {
}

-(void)sayHello;

@end
```

Step 6. Edit Demo.m to match the following:

```
// Demo.m
// ObjC-Demos

#import "Demo.h"

@implementation Demo
-(void)sayHello {
    NSLog(@"%s", __FUNCTION__);
}

@end
```

Step 7. Edit main.m to match the following:

```
// main.m
// ObjC-Demos

#import <Foundation/Foundation.h>
#import "Hello.h"
#import "Demo.h"

int main(int argc, char *argv[]) {
    ...same code as before...
    // Demo Hello with HelloDelegate
    // Create instance of Hello Object
    Hello *hello = [[Hello alloc] init];
    // Create instance of Demo object
    Demo *demo = [[Demo alloc] init];
    // Set Hello object delegate to demo
    hello.delegate = demo;
    // Hello object calls the delegate
    [hello.delegate sayHello];
    // All done
    [demo release];
    [hello release];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

Step 8. Use Xcode to do Run > Console then build and run and observe the results:

```
...same output as before...
...
...
... -[Demo sayHello]
```

Exercise 11. Memory Management

The iPhone OS does not provide automatic memory management. When you create an object, you have to remember to release it when done.

You may have heard some developers say that iOS memory management is "hard", but it is only "hard" if you do not understand the rules. Once you know the rules, the memory management is clear.

In this exercise, you will explore iOS memory management.

Step 1. First and foremost, memorize the #1 rule for iOS memory management:

Memory Management Rule 1: If you alloc, you must release.

For example, if you do this:

```
Hello *ptr = [[Hello alloc] init];
```

then later in your code you must do this:

```
[ptr release];
```

Step 2. Also memorize the #2 rule for iOS memory management:

Memory Management Rule 2: If you need to use a pointer to an object, but you did not alloc/init the object, then you must do a retain to ensure that the object stays around until you are done with it. Once you are done with it, send it a release.

Step 3. Edit main.m to match the following:

```
...same code as before...
// Allocate an object
ptr = [[Hello alloc] init];
// Show retain count
NSLog(@"[ptr retainCount]: %d", [ptr retainCount]);
// Increment the retain count
[ptr retain];
NSLog(@"[ptr retainCount]: %d", [ptr retainCount]);
// Decrement retain count
[ptr release];
NSLog(@"[prt retainCount]: %d", [ptr retainCount]);
// All done with this object
```

```
        [ptr release];
        int retVal = UIApplicationMain(argc, argv, nil, nil);
        [pool release];
        return retVal;
    }
```

Step 4. Use Xcode Run > Console to build and run, observing the output:

```
...same output as before...
... [ptr retainCount]: 1
... [ptr retainCount]: 2
... [prt retainCount]: 1
```

As shown be the output, when you alloc, the retain count is set to 1. When you retain, the retain count is incremented. When you release, the retain count is decremented. When the retain count goes to 0, the system releases the memory.

An important point: Memory management is the management of the retain count. As long as the retain count is 1 or higher, then the object stays around.

Step 5. Edit main.m to match the following:

```
        ...same code as before...
        // Allocate an object
        ptr = [[Hello alloc] init];
        NSLog(@"After alloc/init: %@", [ptr description]);
        // Send it autorelease.
        // This says to release the object soon but not now
        [ptr autorelease];
        NSLog(@"After autorelease: %@", [ptr description]);
        int retVal = UIApplicationMain(argc, argv, nil, nil);
        [pool release];
        return retVal;
    }
```

The autorelease says to release the object in the future, but not at this time. Use of autorelease is useful when you need to release an object, but at some point in the future; just not now.

An important point: When you have a method that allocates an object, and passes the object back to the caller, it is common to use autorelease in your method; the calls is responsible for doing a retain to keep the object until they are done with it.

Step 6. Use Xcode Run > Console and then build and run to view the output:

```
...same output as before...
... After alloc/init:
... After autorelease:
```

Another important point: From the output, observe that the pointer could still be accessed even after the autorelease. At some point in the future the system will release the pointer, just not now.

Submission

Step 1. Use Xcode Build > Clean All Targets.

Step 2. From Finder, locate your project directory, Ctrl-Click (or Right-Click) on the project directory, and select Compress <directoryName>.

Step 3. Submit the archive to the Angel Drop box for this assignment.