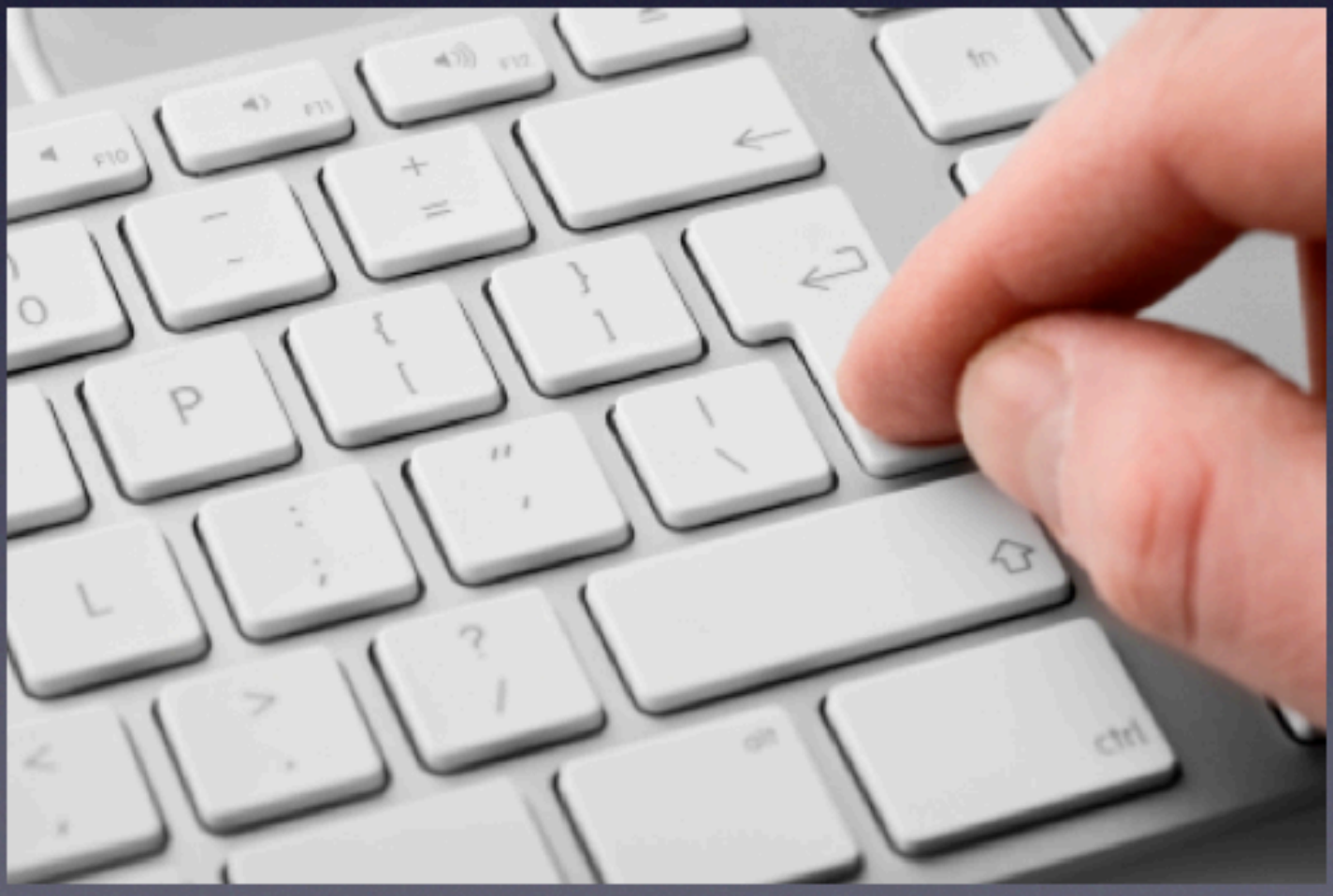
Beginning Objective-C Programming

By Matthew Campbell



Forward

A little over ten years ago, I had just started learning object-oriented programming after spending a few years as a mental health counselor (yes, really!). What I had initially thought would be a dry and technical topic based on my days at the university turned out to be the key to an intriguing hidden world of codes and virtual universes.

Programming is the key to an intriguing hidden world of codes and virtual universes

What was really cool was that since computers had become so powerful and could be found everywhere in our world, programmers had suddenly become ***creators***. Learning object-oriented programming quickly opened up my career while I was working at my 9-5 in ways a psychology major would never expect.

Programming also served as my escape route once I decided to leave my 9-5 job and start my own company. All these things are why I want to share this exciting world of programming with you in this book, ***Beginning Objective-C Programming***.

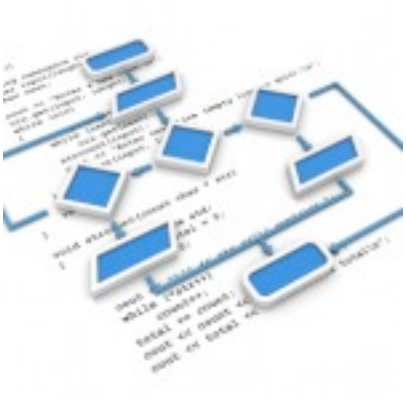
Introduction

The focus of ***Beginning Objective-C Programming*** is to teach basic programming in the simplest way possible. Furthermore, this book is geared toward people who want to eventually make iOS and Mac apps and so starts

from the beginning teaching you using XCode 4.2 and iOS 5 which are the latest technologies available.

My hope is that this book will be the first place you will look when you are ready to learn the programming skills you will need to master the iOS SDK so you can make your own app down the road.

Chapter One: Introduction to Programming



Programming is the process of giving instructions to a computer. Apps on your iPhone work because someone took the time to write instructions that have been packaged into an app that you download and use on your iPhone.

Here's an example of how you would give instructions to an iPhone app. The code below will show us a alert that says Hello World:

```
UIAlertView *alertView = [[UIAlertView alloc] initWithTitle:@"Alert View"
                                                    message:@"Hello World"
                                                   delegate:nil
                                          cancelButtonTitle:@"Ok"
                                          otherButtonTitles:nil];
[alertView show];
```

The code above will present an alert view to the user with the text Hello World. See figure 1.1 below for an example.

Figure 1.1 - Hello World

These instructions probably don't make much sense to you yet but that's ok – this book is all about how you can use instructions like this to get things done in your apps.

Programming Languages

Programming languages are used to communicate instructions to computers like the iPhone. Like natural languages, programming languages have a vocabulary and must follow certain rules. We must follow these rules strictly so that a computer can understand what we intend.

Writing instructions with programming languages is called coding or sometimes simply writing code. Coding for iOS apps takes place with an editor called XCode that is specifically tailored to make writing code easier. The programming

language that we will be using and learning about is called Objective-C. The first half of this book is about learning the rules and vocabulary of Objective-C because that is what iPhone apps are made of.

Programming Essentials

Let's go over some the essential rules and things you can do with programming.

Code Execution

Code execution refers to when instructions are given to the computer. Computers execute instructions one at a time as they are received. When you are looking at a screen full of code, each line of code will get executed one at a time from top to bottom in the same way a person would read a book. Top to bottom. Left to right.

A line of code is an instruction that is ended by a semi-colon `;`

For example:

```
[alert show];
```

It is very important to remember the semi-colon when you end a line of code.

Code Regions

Code regions are used to separate a set of instructions from the rest of the program. You will see this in different contexts as you learn programming but the general idea is that code that is in a code block belongs together. Code regions start with a left curly brace **{** and end with a right curly brace **}**.

Here is an example of curly braces defining a code region:

```
if(alertView){
      NSLog(@"UIAlertView was used");
      NSLog(@"as an example");
      NSLog(@"and it was great.");
}
```

Code Documentation

As you get into coding you may start to lose track of what your code was supposed to do. One way to remind yourself what you were trying to accomplish is to document your code. This just means to write a short note called a comment that is included with the rest of your code.

This code comment will let others know what you were intending on doing but it won't be considered an instruction to the computer. Code comments are ignored. You can use a comment anywhere in your program by enclosing your comment text with these characters: **/*** and ***/**.

Here is an example of code documentation:

```
/* This program is going to calculate monthly loan payments
for auto loans with different interest rates and payment schedules */
```

This type of documentation works best when applied to a big section of your program where we want to remember what the broad overview of what we are doing. To document smaller pieces of your program it is better to apply this notion of self-documentation. What we mean by self-documentation is that the words we choose for each piece of the program should describe what are intentions were. For example, take a look at these two lines of code:

```
float x = 199;
float interestRate = 4.25;
```

Even though you do not know the programming syntax that we are using here yet I bet that you can guess what the second line of code is supposed to be. The programmer probably choose the word interestRate because the code had something to do with interest rates right? Who knows what x is supposed to be? Compared to x, interestRate is much more meaningful. Hopefully, if you remember to write your code like you will save yourself some work down the road.

Debugging

Debugging is the process of finding mistakes in your code and fixing them. In practice, more time is spent on debugging than coding itself. Generally, when I am writing some code I will quick type it out and then try to run the program, see if what I did works and then test the results. Usually, I will need to go back and fix something. In fact, I would say that most programmers spend 90% of their

programming time debugging. This is why I want to now show you how XCode makes this process as painless as possible for us.

Debugger Screens

The debugger is a special area in XCode that has the tools you need to debug your program. To see the debugger select **View > Debug Area > Show Debugger Area**. You should see a screen like Figure 1.2 appear at the bottom of XCode.

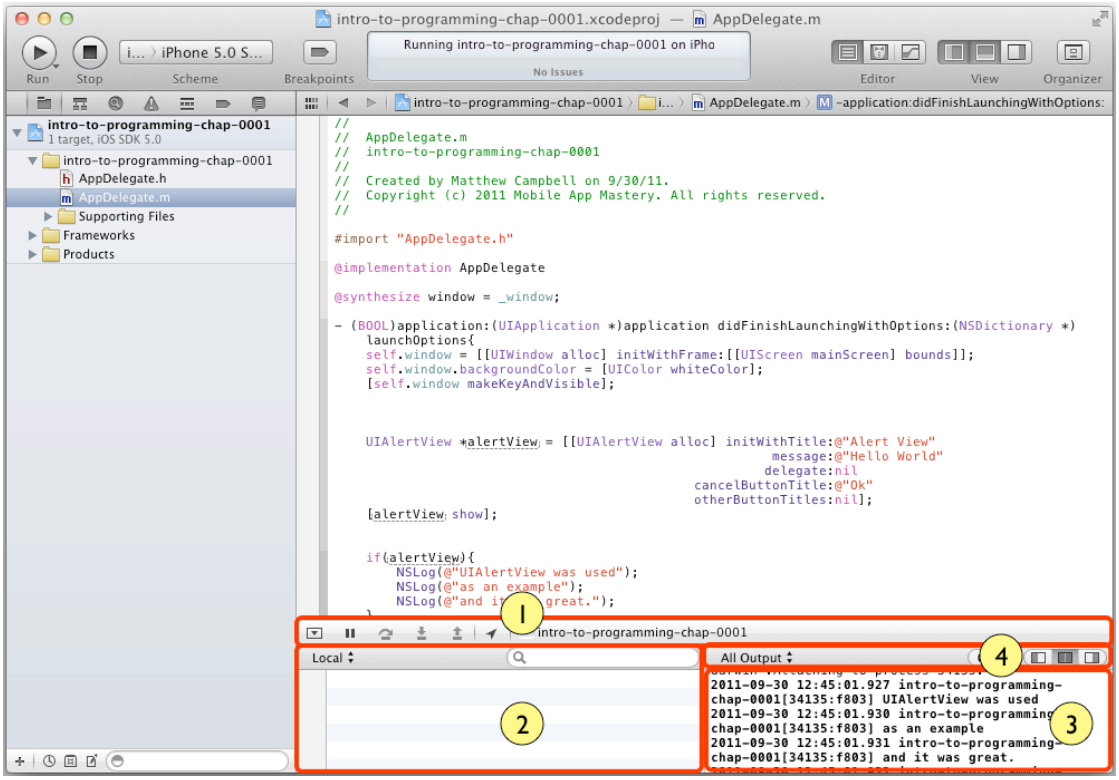


Figure 1.2 - XCode Debugger

In figure 1.2 you can see that I marked the most important sections of the debugger. Area 1 is used to control the app's execution when it is running in the

simulator. You can pause and resume the app. This figures in more when we talk about setting breakpoints below. You use area 2 to inspect your code while the app is executing. This is what you use when you want to see what is happening while your app is running.

The area marked 3 is the console. This is a special window that the program will write messages to giving us status updates. We can write message to this window ourselves using NSLog. In area 4 you will find a control that you can use to show or hide the various debugger panes.

NSLog

NSLog is used to write messages to the console screen. This is a simple way to see what is going on in your program and it is also a way to produce output. Output is simply information created by our program and presented to us. You use **NSLog** like this:

```
NSLog(@"This is my message");
```

Everything that you include in the parenthesis will be printed out to the console. You can also include substitute other items into your **NSLog** messages. What you need to do is insert special placeholders into your NSLog messages followed by a comma-separated list of things to substitute into your message. For instance, if we wanted to add our alert object from the Hello World example into our message we would do this:

```
NSLog(@"This is my alert object: %@", alert);
```

Setting Breakpoints

Breakpoints are flags that you can attach to a line of code that will stop your app from executing when that line of code is reached. When the app stops you get a chance to use the debugger screen to look at your code to see what is going on. To set a breakpoint all you need to do is click in the gutter (the area right to the left of the code editor). The gutter is the area marked in red in figure 1.3. Also, if you look closely you will see a blue icon which is an example of a breakpoint.

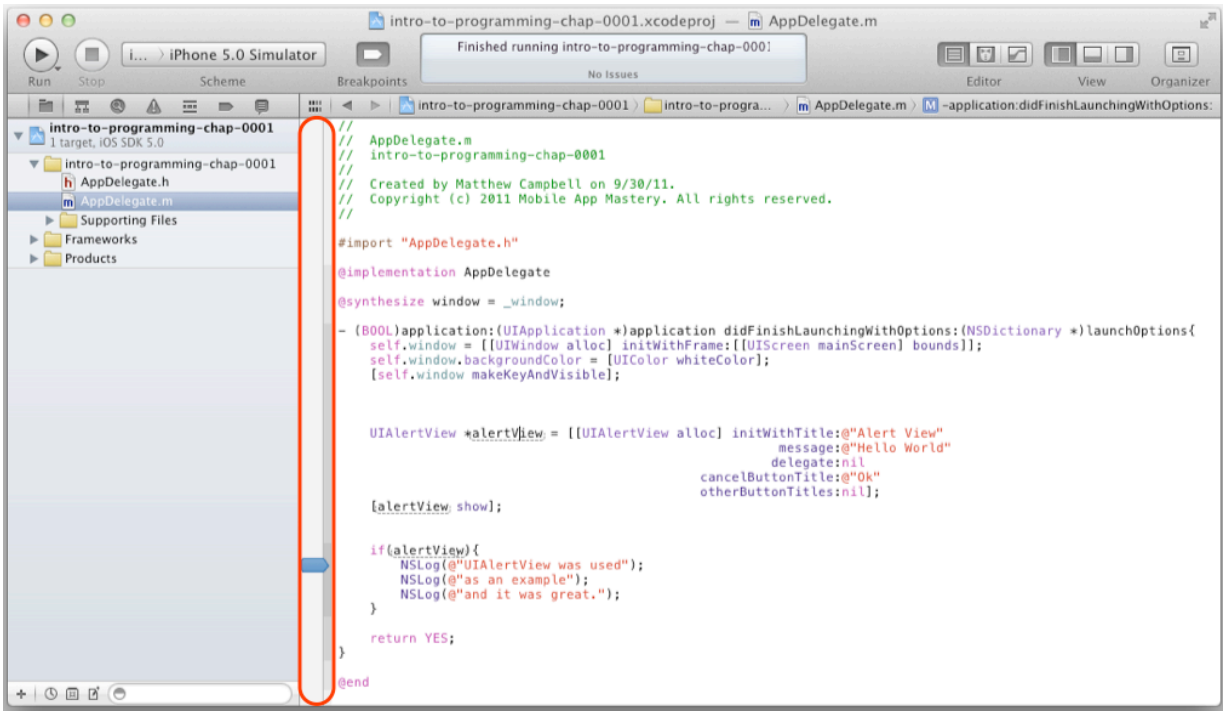


Figure 1.3 - Breakpoints and the Gutter

Now when I build and run this app everything will stop when the program reaches the blue breakpoint. If I look at the debugger screen I'll see tons of information about my program in it's current state and I will see the message I wrote to the console screen.

You can see all kinds of information on this screen, as you learn more about programming it will become more meaningful to you.

Summary

Programming is how we get computers to do the tasks required for iPhone apps. The thing with programming is that we need to be very literal and strictly follow the rules of the programming language that we are using to get our work done. XCode is the tool that Apple provides developers to help us get our work done and becoming intimately familiar with this tool will help you out a lot. In the next chapter we start going over the essential programming rules that we use to get things done in iPhone apps.

Please note that this chapter is just an introduction to the tools and general process of programming. The remaining chapters of this book will make the process of programming much clearer so please stick with me and jump right into chapter two where we talk about ***if statements***.

Chapter Two: If-Statements

If statements are used to execute code based on whether a condition is present in the state of the program. Simply put, an if statement is a way for us to tell a computer to do something if a condition is met and something else if the condition is not met. Below is the general form that an if statement takes.

```
BOOL condition = YES;
if(condition){
     //take action when condition is YES
}
else{
     //take action when condition is NO
}
```

The if statement above has three important components: the test to see if the condition is true, the code that will execute if the condition is true and the code that will execute if the condition is false.

BMI Calculator Example

Let's take a look at how an if statement works in the context of something you might find in a real iPhone app like MyNetDiary or LoseIt!. The example that I am thinking of is a BMI calculator. BMI stands for Body Mass Index. BMI is a statistic used to determine if you are underweight, normal weight, overweight or obese.

This statistic is included in many health apps because it is used to determine if someone is at risk of developing health conditions like heart disease.

To calculate BMI you need to know your height and weight. The formula for BMI is weight divided by height squared. Once you have calculated BMI you can figure out whether you are underweight, normal weight, overweight or obese based on your BMI value.

BMI	Weight Status
Below 18.5	Underweight
18.5-24.9	Normal
25-29.9	Overweight
30 & Above	Obese

```
What we are going to do in our program is first write code to figure out BMI based on height and
weight values that a user will supply to us. Then we will use an if statement to find out
whether the BMI stat indicates that the person is underweight. Let's declare the variables that
we need now and set some initial values so that we can test.
float heightInMeters = 1.8796;
float weightInKilograms = 117.934016;
float BMI;
```

Now we can add in the BMI calculation and assign the result to the BMI float variable.

```
float heightInMeters = 1.8796;
float weightInKilograms = 117.934016;
float BMI;

BMI = weightInKilograms / (heightInMeters * heightInMeters);
```

The result that we get with the test data above is 33.3816795. Now what we want to do is compare our BMI statistic with the value from the table above. If our BMI is under the value that determines if we are underweight then our app will do one thing like print out relevant health information. Otherwise, the app will assume for now that the BMI is within normal range.

Relational And Equality Operators

To make this assessment we will need to evaluate the condition using ***relational and equality operators***. These operators are used to evaluate expressions and we include these in the parenthesis after the **if** keyword in an ***if statement***. Some of the expressions that you can evaluate are equal to, greater than, less than and not equal to. Here are the operators that you have available to you.

Operator	Meaning
==	Equal To
!=	Not Equal To
>	Greater Than
<	Less Than
>=	Greater Than Or Equal To
<=	Less Than Or Equal To

Since all we need to do for now is determine whether our BMI is underweight or not we can simply use the less than operator **<** in our if statement.

```
float heightInMeters = 1.8796;
float weightInKilograms = 117.934016;
float BMI;

BMI = weightInKilograms / (heightInMeters * heightInMeters);

if(BMI < 18.5){
     //execute if the BMI indicates the person is underweight
}
else{
     //execute if the BMI seems normal (not underweight)
}
```

If we were writing this app for real we would replace the comments in the if statement above with code that would communicate something of value to the user. The values I supplied above would cause the if statement to execute the code after the **else** keyword, but the real result would depend on what the user actually enters into the app when it is being used.

Scope

When you need to include more than one line of code as a consequence for an if statement then you must define the ***scope*** for a ***region of code***. In programming, scope refers to a region of code that is separated from the rest of the code in the program. This means that variables declared from within that region can only be used inside of that region. Scope is defined with curly braces: a left curly brace defines the beginning of a region of code, and a right curly brace defines the end of a region of code.

In the if statements that we have seen so far the code that executes is always contained in the area constrained by the curly braces to the right of the condition that we evaluate and also right after the **else** keyword where we put the code that evaluates when the condition is not met.

The thing to remember about scope is that variables that you declare within a region of code can only be used from within that region or from regions of code contained by the parent region. So if you declare a variable in the region of code like the block right after the else keyword you can only use that variable from within that region of code. If you try to use in further down in your program you will get an error warning.

Nested If Statements

So far our BMI example is incomplete since we only have one if statement that is testing whether we are underweight or not. If we were putting something like this into our own app our users would probably want to know what to do if they fell into one of the other categories as well. To get this done we will need to use more if statements.

This referred to as using nested if statements. Nested if statements are used when you need to include more if statements to further test the conditions present in the program. You can use nested if statements by including them in the regions of code after you evaluate your expressions. The general form that this takes looks like this:

```
BOOL condition = YES;
BOOL anotherCondition = NO;

if(condition){
     //take action when condition = YES
     if(anotherCondition){
          //take action when
          //anotherCondition = YES
     }
     else{
          //take action when
          //anotherCondition = NO
```

```
     }
}
else{
     //take action when condition = NO
     if(anotherCondition){
          //take action when
          //anotherCondition = YES
     }
     else{
          //take action when
          //anotherCondition = NO
     }
}
```

The example above has new if statements listed in both areas were we can include code. You can nest as many if statements as you need in code. In order to manage the complexity of your code though I would not recommend using more than three layers of nested if statements.

To apply these statements to our BMI example we will need to add a nested if statement to the region of code after the **else keyword**. What we want to do is test to see if BMI statistic indicates whether the person is normal weight.

```
float heightInMeters = 1.8796;
float weightInKilograms = 117.934016;
float BMI;

BMI = weightInKilograms / (heightInMeters * heightInMeters);

if(BMI < 18.5){
     //execute if the BMI indicates the person is underweight
}
else{
     //execute if the BMI is not underweight

     if(BMI >= 18.5 && BMI <= 24.9){
          //execute if the BMI indicates the person is normal weight
     }
     else{
          //execute if the BMI is not underweight
```

```
            //or normal weight
      }

}
```

If you take a look at the if statement condition in the nested if statement you will see that it is bit more complex that the first if statement. We need to determine if the BMI is in a range and to do that we need to employ the ***logical operator AND*** which uses the symbol **&&**. **&&** means ***AND*** so we use it when we want to evaluate two conditions both of which must be true in an if statement.

To finish our example we would continue to nest if statements. Our finished product becomes fairly complex when we attempt to include every possibility.

```
float heightInMeters = 1.8796;
float weightInKilograms = 117.934016;
float BMI;

BMI = weightInKilograms / (heightInMeters * heightInMeters);

if(BMI < 18.5){
      //execute if the BMI indicates the person is underweight
}
else{
      //execute if the BMI is not underweight
      if(BMI >= 18.5 && BMI <= 24.9){
            //execute if the BMI indicates the person is normal weight
      }
      else{
            //execute if the BMI is not underweight
            //or normal weight
            if(BMI >= 25 && BMI <= 29.9){
                  //execute if the BMI means the person is over weight
            }
            else{
                  //execute if the BMI is obese
            }
      }
}
```

Logical Operators

In the example above we used the logical operator **&&** because we wanted to make sure that both conditions that we were evaluating were true. We have two more logical operators that we can use in if statements. These two operators are **!** and ||which mean ***NOT*** and ***OR*** respectively.

The ***NOT*** operator is used to reverse the outcome of a condition. So, if your condition normally would evaluate to false but you wanted it to evaluate to true you could put **!**in front of the condition to achieve this. That may look something like this:

```
BOOL isTrue = YES;
if(!isTrue){
     //this would execute when isTrue == NO
}
else{
     //this would execute when isTrue == YES
}
```

The ***OR*** operator || is used when we want to evaluate a condition when at least one of the conditions is true. In this example I am imagining writing code to test to see if I can get my email or not. I know that to do that I would need to either have a cell coverage or access to a wifi network so this is a perfect time to use the ***OR*** operator.

```
BOOL hasWifi = NO;
BOOL hasCellCoverage = YES;

if(hasWifi || hasCellCoverage){
     //I can get my email since I have
     //either cell coverage or wifi
```

```
}
else{
     //no email since I am not connected
}
```

Hands-On Time

Expand on the idea of creating an app in the health niche by adding information about blood pressure and waist size. Use what you learned in the last chapter to first create a composite type definition that is composed of primitive types to described these attributes: BMI, diastolic blood pressure, systolic blood pressure and waist size. Also include a **BOOL** variable type called flag to help keep track of whether the person has any warning signs.

Look up the criteria for each of these attributes that may indicate an unusual measurement and use if statements to screen the data for people who may have warning signs for health problems. Use the **BOOL** variable flag to keep track of whether the data indicates that a person should consult a doctor about their health.

For a real challenge, add the dimension of gender and make sure to apply the appropriate criteria for each gender in your analysis.

Chapter Three: Switch Statements

Switch statements are used when we want to execute different bits of code based on the value of an integer variable. While the ***if statement*** from the last chapter gave us two options (either the condition was met or it wasn't), ***switch statements*** give us a way to conditionally execute code with more than two possibilities. Here is the general form that the switch statement takes.

```
int level = 0;

switch (level) {
     case 0:{
          //execute code when level == 0
          break;
     }
     case 1:{
          //execute code when level == 1
          break;
     }
     case 2:{
          //execute code when level == 2
          break;
     }
     default:{
          //execute code if level does not
          //meet any of the conditions above
          break;
     }
}
```

The ***switch statement*** above has three major components: the first is the expression that we are evaluating. In this case, we're checking the value of the integer variable**level**. The second component is a series of ***case statements*** that correspond to each possible value of the expression. Each case will include the

keyword **case**, followed by the expression that corresponds to the case, and a colon. Everything after the colon is code that will execute when the expression evaluates to the integer for that**case**. In the example above, the code after **case 0** and before the first **break** will execute, because the current value of **level** is 0. If you changed the value of level, a different **case** would be executed.

break statements mark the end of the code that will execute for a case. By putting a***break statement*** in a **case** you make sure that any remaining code in the ***switch statement*** will not execute.

The last part of the ***switch statement*** is a special type of **case** called **default**. This appears after the ***case statements***, and works in a similar way. The main difference is that **default** does not have a value associated with it-the code after default will only execute when the expression does not correspond to any of the values in the case statements.

Finding the Area of a Shape with switch

Let's see how a ***switch statement*** might work in a real application. Imagine that we had an app that figured out the area of a shape for us. We don't know beforehand what type of shape we're dealing with, and since there are different rules to compute the area of different shapes we'll need to use different code to compute the area for each type of shape we'll support.

This is a good time to use a ***switch statement***. In our real app we would ask the iPhone user what type of shape they wanted to compute the area for. Then we would use a ***switch statement*** to execute the appropriate code based on the user's choice. In this example, we'll just use default values for our shape and

other parameters since we have not yet learned how to ask the user for input. Of course, in a real program we would need to ask the user both for the type of shape as well as the relevant input parameters.

The first thing we will need is two variables: the first will be an integer that will represent the shape the user is interested in, and the second will be a float that will hold the shape's area.

```
int shape = 2;
//0 == Square, 1 == Parallelogram
//2 == Triangle, 3 == Circle

float area;
```

As you can see from the comments, we are going to support four types of shapes: squares, parallelograms, triangles, and circles. Next, let's fill in the skeleton of our***switch statement***. What we need now is to start the ***switch statement***. We'll start with the default **case**, which in this example means the user specified an unsupported shape. If this happens, we'll assign **-999** to the **area** variable:

```
int shape = 2;
//0 == Square, 1 == Parallelogram
//2 == Triangle, 3 == Circle

float area;

switch (shape) {
     default:{
          //Set to negative number if
          //no shape is specified
          area = -999;
          break;
     }
```

```
}
```

Now let's add the case for finding the area of a square, which will execute when **shape** is equal to **0**:

```
int shape = 2;
//0 == Square, 1 == Parallelogram
//2 == Triangle, 3 == Circle

float area;

switch (shape) {
     case 0:{
          //Square
          float length = 3;
          area = length * length;
          break;
     }
     default:{
          //Set to negative number if
          //no shape is specified
          area = -999;
          break;
     }
}
```

Now our ***switch statement*** will compute the area of a square when the **shape**variable equals **0**, and for any other value of **shape** it will just assign the value of **-999**to the **area** variable.

Next we can fill in the case for a parallelogram, which uses different variables and a different equation:

```
int shape = 2;
//0 == Square, 1 == Parallelogram
```

```
//2 == Triangle, 3 == Circle

float area;

switch (shape) {
     case 0:{
          //Square
          float length = 3;
          area = length * length;
          break;
     }
     case 1:{
          //Parallelogram
          float base = 16;
          float height = 24;
          area = base * height;
          break;
     }
     default:{
          //Set to negative number if
          //no shape is specified
          area = -999;
          break;
     }
}
```

In the two ***case statements*** above, you can see how each case uses different variables and a different equation. You may add as many ***cases*** to a ***switch statement*** as you need to support the levels that you can expect in your program.

Here are the remaining two cases we'll support for now, a triangle and a circle:

```
int shape = 2;
//0 == Square, 1 == Parallelogram
//2 == Triangle, 3 == Circle

float area;

switch (shape) {
     case 0:{
          //Square
```

```
            float length = 3;
            area = length * length;
      break;
      }
      case 1:{
            //Parallelogram
            float base = 16;
            float height = 24;
            area = base * height;
            break;
      }
      case 2:{
            //Triangle
            float base = 16;
            float height = 24;
            area = .5 * base * height;
            break;
      }
      case 3:{
            //Circle
            float pi = 3.14159;
            float radius = 6;
            area = pi * radius * radius;
            break;
      }
      default:{
            //Set to negative number if
            //no shape is specified
            area = -999;
            break;
      }
}
```

Omitting break Statements

The example above is the most common way to use ***switch statements***, but it's possible to get your ***switch statements*** to behave in a different way. If you omit the***break statements*** at the end of each **case**, control of the program will not return to the next level of code. Instead, each successive ***case statement*** will continue to execute, until a **break** is reached or the ***switch statement*** ends.

What happens is that once a **case** is evaluated to be true then every bit of code remaining in the***switch statement*** is executed.

To illustrate this point, I created a ***switch statement*** below that doesn't have any***break statements***. What this is going to do is pretend to count so we are going to start with an **int** called count and an **int** called **addThisMany**. The idea is that when the ***switch statement*** has a **case** that matches the value of **addThisMany** then we will start adding values to the **int count**. This means that if we make the value of**addThisMany** equal to **3** then the ***switch statement*** will evaluate each **case** until it finds one that matches **3**. When it does, all the remaining code in the ***switch statement*** will execute. In our case, these means that **count** gets incremented in each ***case statement***. We will step through this in a second, but here is the code:

```
int addThisMany = 3;
int count = 0;
switch (addThisMany) {
     case 5:
          count++;
     case 4:
          count++;
     case 3:
          count++;
     case 2:
          count++;
     case 1:
          count++;
     default:
          break;
}
```

Let's step through each ***case statement*** to see how this is working. Again, we are going to assume that the variable value is **3** so that we will not start counting

until we find a case that is equal to **3**. In put each case in a table below to make it easier to follow. The first column is the **case**, the second indicates whether the case expression is true, the third column shows you the code that executes and the final column will show **count's** value as it increases through each case.

Case	Expression True	Executed Code	count's Value
case 5:	**NO**		0
case 4:	**NO**		0
case 3:	**YES**	**count++**	1
case 2:	**YES**	**count++**	2
case 1:	**YES**	**count++**	3

Hand's On Time

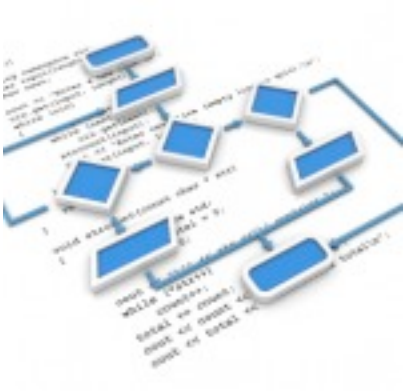
Imagine that you wanted to create a "secret decoder ring" app. A secret decoder ring is a toy that let's kids create secret messages by changing letters in the alphabet into numbers and this could be a fun iPhone app. In real life, the decoder ring would have a circle on it that matches up numbers and letters so that you could get a sequence of numbers to represent a phrase. So, instead of writing ABC you could write 123.

The first step to making an app like this would start with a ***switch statement***. Instead of a mechanical disk you would write a ***switch statement*** that had cases for each letter of the alphabet. The code in each **case** would assign a different

number to a variable depending on what letter of the alphabet the case represented.

For this exercise, create a **char** variable called **letterToEncode** and an **int** variable called **letterCodedAsInteger**. Next, write a ***switch statement*** that evaluates the **letterToEncode** variable and then assigns a number to the ***letterCodedAsInteger*** variable based on what letter in the alphabet **letterToEncode** represents. The only rule is that each case must assign a unique number.

Chapter Four: Loops



Loops are used in programming when we want to repeat a similar task many times. Instead of just writing out each line of code we can use a loop to repeat code for a set number of times or until a condition is met. There are three types of loops that we will be discussing here: ***for loop***, ***while loop*** and ***do loop***.

An example of something that you may want to do in programming that would be repetitive task is counting from 1 to 10. From what you learned in the past chapters you may try to count like this:

```
int count;
count = 0;
count = count + 1;
count = count + 1;
count = count + 1;
count = count + 1;
count = count + 1;
count = count + 1;
count = count + 1;
count = count + 1;
count = count + 1;
count = count + 1;
```

Clearly this task is very repetitive and with loops we can write it out in a much better way. Let's see how to do the simple example above using each type of loop that we have available so far starting with the ***for loop***.

For Loops

The ***for loop*** is used when we know how many times that we want to repeat a task. So, our counting problem from above is a good example of a task that a for loop could help with since we know that we simply want to add one to the count integer variable 10 times. Here is what the ***for loop*** will look like for our counting problem.

```
int count = 0;

for(int i=0;i<10;i++){
     count = count + 1;
}
```

Let's see how to count from one to 10 using a ***for loop***. The ***for loop*** starts with the **for** keyword. Next is the three bits of code in parenthesis that control how many times code in the loop will execute. The first thing in this part of the ***for loop*** is the integer variable **i**. This is what is used to control the loop. This variable is initially set to **0 (int i=0;)**. Next the ending condition is defined **(i<10;)**. Finally, each time the loop executes the variable **i** is incremented by **1 (i++)**.

What this all means is that when the code reaches the loop a temporary variable called **i** will be created with an initial value of zero. When we use a variable in a ***for loop*** like **i** we refer to it as the ***control variable*** because it controls the loop. Each time the code in the loop (that will appear after the **for** statement) executes the value of **i** will be increased by 1. The loop will keep on going until **i** is no longer less than 10.

Now that we have our loop we need to include the code that will be executed each line the loop cycles. This is sometimes called ***iterating*** through the loop. The first thing we did was to define a ***region of code*** with curly braces .

The next part is the code that we want to repeat 10 times. So to use the example that we started with we could replace some of those lines of code with **count = count +1;**.

The ***for loop*** is very powerful and you could probably do most of your repetitious programming with just this loop. But, there are two other ways to do loops in programming that may also use.

While Loops

While loops operate in the much the same way as ***for loops***. The only real difference other than the use of the **while** keyword is that placement of the ***control variable*** and the operations associated with the ***control variable***. Here is how we would count from one to ten using a ***while loop***.

```
int i = 0;
while  (i < 10){
     count = count + 1;
     i++;
}
```

To start coding a ***while loop*** we will need to first declare and set an initial value for an integer variable. This will serve as our ***control variable***. Next we will use the **while**keyword to specify the ***while loop***. We must also include the ending condition in parenthesis here.

Next we use curly braces **{}** to define our ***region of code***.

When using ***while loops*** be careful to make sure to keep incrementing the ***control variable***. In our example, we are using **i** to keep track of how many times we have gone through the loop. Each time the while loop executes, it checks on the value of i to see if it is still less than 10. If it is, then the loop will iterate one more time. If not, then the loop stops executing and the program moves on. If you forget to increment the control variable each time then your app may get stuck repeating this loop forever.

The last part of the ***while loop*** here is the same as what we have for the ***for loop***. It is the code we want to repeat which is **count = count + 1;**.

As you can see this loop works just like a for loop but the syntax is a little bit different. Both the ***for loop*** and the ***while loop*** could have a situation where the code in the loop never executes. This happens when the control variable was already equal to 10 (and not zero like we demonstrated above). If that were to happen then these loops would not execute any code at all. Try this variation of the example above yourself to see what I mean:

```
int i = 10;
while (i < 10){
     count = count + 1;
     i++;
}
```

If you want to make sure that the code in your loop is guaranteed to execute at least once you must use a ***do loop***.

Do Loop

The ***do loop*** is distinguished from both the ***for loop*** and the ***while loop*** in that a ***do loop*** will execute its code at least once even if the condition is already met. Here is what a do loop looks like.

```
int i = 0;
do{
     count = count + 1;
     i++;
}while (i < 10);
```

To start to code a ***do loop*** you start by declaring the ***control variable*** and setting its initial value. The next step is to simply type in the **do** keyword and the curly braces we will need.

In contrast to the previous two loops we will put the ending condition at the end of this loop. We need to be careful to increment the ***control variable*** here as well.

Now that we have the loop all set up we can again stick in our counting code to complete the example. Finally, we can add the code that we want to repeat.

The way that this ***do loop*** is set up right now is equivalent to the other two loops that we used to count with. However, there is a subtle difference here that will only be seen when the control variable would seem to make the loop not execute at all. In our example this would be when **i = 10**.

```
int i = 10;
do{
     count = count + 1;
```

```
    i++;
}while (i < 10);
```

The above code would execute the counting code once time. This is because the expression that controls whether the loop executes does not get evaluated until the end so the code gets at least one chance to execute.

Hands On Time

Code the factorial of 5 using a loop. In math, a factorial is the product of all positive numbers equal to or less than the number. This is noted with an exclamation point after the number. So, the factorial 5 would be noted as 5!. To figure out what the factorial of 5 is you can simple multiple all the numbers that are equal to or less than the number like this: 5! = 5 * 4 * 3 * 2 * 1 = 120. So, the factorial of 5 is 120.

Write code to solve the factorial of any number. Use loops to do this and make sure to write code three times using each loop.

Chapter Five: Functions

A function in programming is a discreet area of code that is independent from the rest of the program. Functions are also called subroutines and as that name implies functions are a bit like small programs. Generally, functions may accept variables as inputs and a function may produce a value as an output.

Take a look at figure 1 to see what all the pieces of a function look like before we discuss each piece separately.

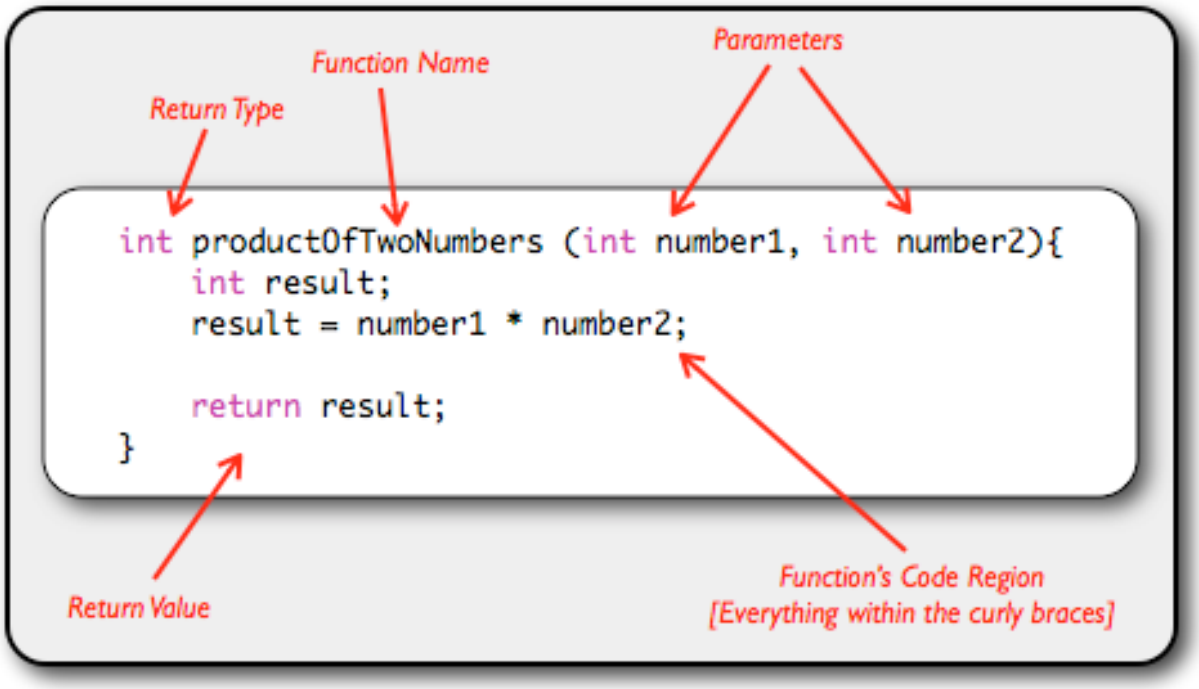


Figure 1: Anatomy of a Function

Return Type

Something that a function can do is produce a result based on information that you give to the function. A result can be a number like an integer or float. Functions can also return other primitive types like ***Boolean*** values like **YES** and **NO** as well as characters like **A** or **B**.

Take a look at figure 1 to see where the return type belongs in the function declaration. Our function returns an integer as a result so we use the **int** keyword to indicate this.

```
int productOfTwoNumbers (int number1, int number2){
     int result;
     result = number1 * number2;

     return result;
}
```

This works in the same way as it does for declaring variables so you can see return types specified with **int**, **BOOL**, **float**, **double** and **char**.

Functions can even have a return type of nothing at all! What? Why? Sometimes we want to separate part of our program but we do not need a result. In those situations we will would like to use a function but we do not need to get a result back.

To code a function that does not return a value you may use the **void** keyword. Another difference with functions that use **void** to specify that they are returning nothing is that they may omit the return value statement.

Function Name

The next part of the function declaration is the function name. The function name goes right after the ***return type*** which you can see clearly in figure 1.

```
int productOfTwoNumbers (int number1, int number2){
     int result;
     result = number1 * number2;

     return result;
}
```

We use functions to attack a big problem by first breaking the big problem up into manageable chunks. Now we can treat each chunk on its own. When we are satisfied that each chunk works the way that we expect we can then use those functions together to solve our bigger problem. Let's see how to use functions.

Parameters

Functions can accept inputs as well as return output. Inputs are information that will be used in the function. You can specify the inputs for a function in the parameter list. The parameter list is a list of variables enclosed in parenthesis that are placed right after the function name. In the figure 1 you can see that we specify two integers as parameters.

```
int productOfTwoNumbers (int number1, int number2){
     int result;
     result = number1 * number2;

     return result;
}
```

You declare parameters in the same way as you do variables. They are located right after the function name in parenthesis and each variable is separated by a comma.

Function's Code Region (Scope)

We talked about the notion of scope already. Functions get their own region of code with has it's own scope. This region is defined starting with the first curly brace after the parameter list and the region ends with the match curly brace after the function's last line of code.

Any variable that you code within these curly braces will only be in scope from within the function's curly braces. This gives us a great way of keeping the code that only belongs in the function separate from the rest of the program. In function displayed in figure 1 all the code that falls between the curly braces is in scope for the function.

```
int productOfTwoNumbers (int number1, int number2){
     int result;
     result = number1 * number2;
     return result;
}
```

Return Value

For functions that will return a result we need to use the return keyword to specify what variable value will be returned to the code that is using the function. Return values serve as the output of the function. Return values must match the

type declaration of the function. So, if you declare a function with the return type as an integer they you must be sure to return an integer in your return statement.

In figure 1 we use the integer variable we called **result** to temporarily store the result of our calculation. At the end of the function's region of code we use the return statement to return the **result** value back to the program.

```
int productOfTwoNumbers (int number1, int number2){
     int result;
     result = number1 * number2;

     return result;
}
```

We use functions to attack a big problem by first breaking the big problem up into manageable chunks. Now we can treat each chunk on its own. When we are satisfied that each chunk works the way that we expect we can then use those functions together to solve our bigger problem. Let's see how to use functions.

Calling Functions

So that is how you declare a function. To use a function you use do must call the function from the main program. So, if we were to assume that the function from figure 1 was declared in a place that allowed our main program to get access to it we could use **productOfTwoNumbers**.

The first thing we would need from the main program is an integer to hold the output value that the function is going to return to us.

```
int resultOfFunction;
```

Now that we have something to hold the value we can call the function and assign the result to our **resultOfFunction** integer.

```
int resultOfFunction;
resultOfFunction = productOfTwoNumbers(5, 4);
```

Calling the function works by including the function name with parameter values in parenthesis after the function name. The parameter values that you provide here in the function will be assigned to the parameters declared in the function. These may then be used by code in the function. In the code above **resultOfFunction** will now contain the value returned from the function which in this case is 20.

Where To Declare Functions

Now that you have an idea of what a function looks like it's time to go through the process of adding a function to your app. For the first example we can simply add the **resultOfFunction** function to an XCode project.

Add Header And Code Files

Functions are used to split big problems into smaller problems that are easier to solve, but they are also a way to re-use our coding work. The idea is that if you create a function that solves a math problem (like our example in figure 1) then

you may want to use that function in many parts of your program. You may even want to use this function in other programs and apps. If you code a really great function then you may even want to share this with other programmers.

To make sure that we can do this we must code our functions in their own code files. Our files can have more than one function in them, but you should take care to only include related functions in the same file.

Functions must be coded in two files. The first file is called the ***header file*** (this is also called the ***interface file***). The header file has a file extension of **.h**. This second file is an ***implementation file*** and ends in **.m**. Let's add these two files to our XCode project before we talk about them further.

To add a new file to your XCode project, select XCode's main menu and then select**File > New > New File…**. You will see a dialog box that pops up and looks like figure 2. Choose **iOS > Cocoa-Touch > Objective-C Class**.

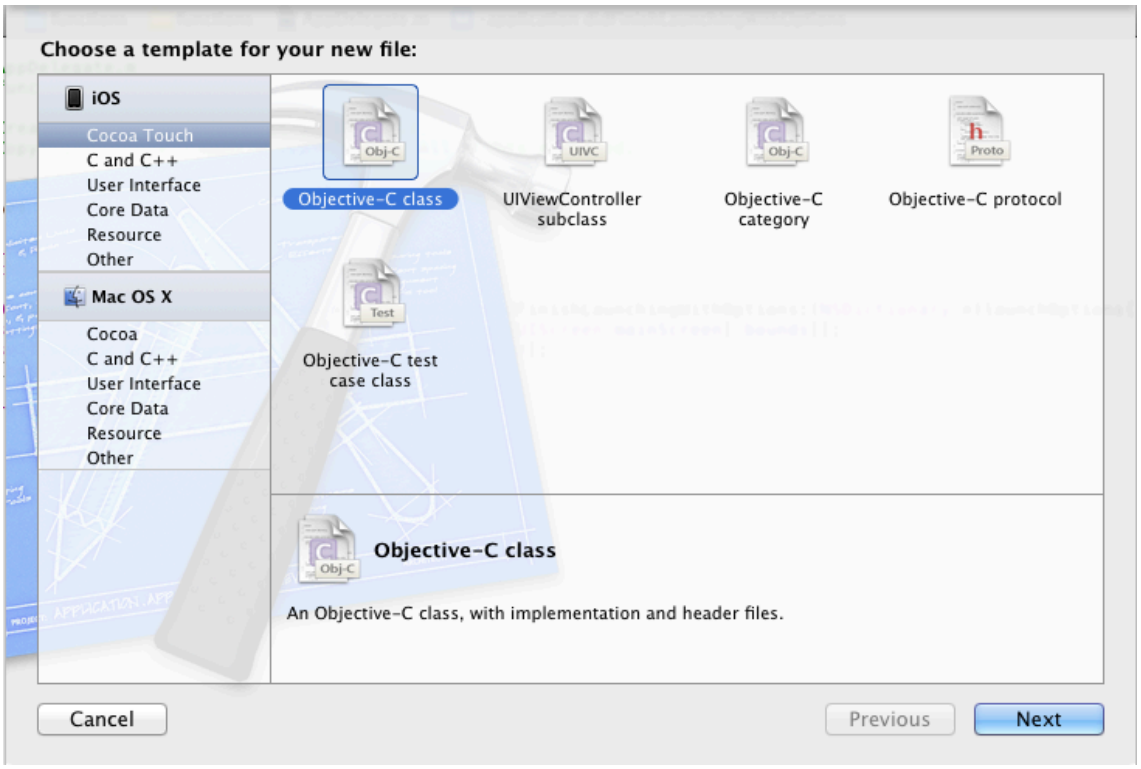


Figure 2: Add New File Dialog

Now click **Next** and name the file ***Functions***. You should now see two new files appear in the XCode project's group folder called **Functions.h** and **Functions.m**.

Let's talk about what goes into each of these files next.

Header Files

The header file is used to specify what is called a ***forward declaration***. A forward declaration is a declaration of a variable, function or type when the complete definition has not yet been given. What you are doing when you use a forward declaration is telling the program that some will be defined in the future that matches the specifications that we outline in the header file.

Our header file will contain a forward declaration for the function we are looking at in figure 1. We can code this by simply typing out the very first line of the

function from figure 1 and replacing the curly brace that would start the function's region of code with a semi-colon.

Click on the **Functions.h** file that was created for you in the previous step. You will see a file that has some comments written in.

Note: comments are the green text that start with **/*** and end with ***/**

Your file will look something like this except you will have your own personal information include here in place of mine.

```
//
//  Functions.h
//  functions
//
//  Created by Matthew Campbell on 10/25/11.
//  Copyright (c) 2011 Mobile App Mastery. All rights reserved.
//

#import <Foundation/Foundation.h>

@interface Functions : NSObject

@end
```

The first thing we need to do is to erase all the code that is in this file except for the **#import** statement since we do not need the boilerplate code that XCode provides for us right now. Your revised Functions.h file should look at this:

```
//
//  Functions.h
//  functions
//
//  Created by Matthew Campbell on 10/25/11.
//  Copyright (c) 2011 Mobile App Mastery. All rights reserved.
```

```
//

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

Now that we go that out of the way we can add the forward declaration for our function just type into this file after the comments.

```
//
//  Functions.h
//  functions
//
//  Created by Matthew Campbell on 10/25/11.
//  Copyright (c) 2011 Mobile App Mastery. All rights reserved.
//

#import <Foundation/Foundation.h>

int productOfTwoNumbers (int number1, int number2);
```

Now it is time to move on to writing the code to make this function work in the implementation file.

Implementation Files

Implementation files are where we put the code that was declared in our forward declaration. This is where all the work of the function gets done. Click on the **functions.m** to see the contents of the implementation file that XCode created for you.

```
//
//  Functions.m
//  functions
//
```

```
//  Created by Matthew Campbell on 10/25/11.
//  Copyright (c) 2011 Mobile App Mastery. All rights reserved.
//

#import "Functions.h"

@implementation Functions

@end
```

This looks like the header file except that we have this **#import** statement added at the top of the file. This #import imports the forward declaration into this file for us. Like the header file we have some extra code in the implementation file that we need to get ride of for our example.

Get ride of the extra code XCode put in here for by deleting the lines starting with **@implementation** and **@end**. When you are finished your file should look like this:

```
//
//  Functions.m
//  functions
//
//  Created by Matthew Campbell on 10/25/11.
//  Copyright (c) 2011 Mobile App Mastery. All rights reserved.
//

#import "Functions.h"
```

Now what we need to do in this implementation file is to add the function.

```
//
//  Functions.m
//  functions
```

```
//
//  Created by Matthew Campbell on 10/25/11.
//  Copyright (c) 2011 Mobile App Mastery. All rights reserved.
//

#import "Functions.h"

int productOfTwoNumbers (int number1, int number2){
     int result;
     result = number1 * number2;

     return result;
}
```

That is it – we now can use this function from anywhere in our program by taking a few key steps. Let's do that now.

Calling Functions

Let's use our function in our app. Select the file in your XCode project named **AppDelegate.m**. Your app delegate file will look something like this:

```
//  AppDelegate.m
//  functions
//
//  Created by Matthew Campbell on 10/25/11.
//  Copyright (c) 2011 Mobile App Mastery. All rights reserved.
//

#import "AppDelegate.h"

@implementation AppDelegate
@synthesize window = _window;

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];

    return YES;
}
```

```
@end
```

The first step that we need to take to call our function is to import the header file into our app delegate. You can do this at the very top of the file after the automatically generated comments.

```
//
//  AppDelegate.m
//  functions
//
//  Created by Matthew Campbell on 10/25/11.
//  Copyright (c) 2011 Mobile App Mastery. All rights reserved.
//

#import “Functions.h”
#import "AppDelegate.h"

@implementation AppDelegate
@synthesize window = _window;

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];

    return YES;
}

@end
```

Importing the header file gives us access to that function so now we can use the function in the same way we learned before. Declare an integer variable named **result** and use the function to assign a value to result. You should put this code in the **didFinishLaunchingWithOptions** method.

```
//
//  AppDelegate.m
//  functions
//
//  Created by Matthew Campbell on 10/25/11.
//  Copyright (c) 2011 Mobile App Mastery. All rights reserved.
//

#import "AppDelegate.h"

@implementation AppDelegate
@synthesize window = _window;

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];

    int result = productOfTwoNumbers(3, 6);

    return YES;
}

@end
```

Now the value of result will be 18 because we called the function **productOfTwoNumbers** with the parameters 3 and 6.

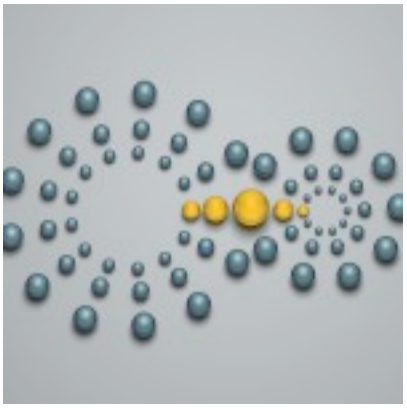
Hands-On Time

In the last chapter, you wrote code help a user generate secret codes. Essentially, what you did was create a switch statement that used the value of a variable to transform a letter into a character and vice versa. For this exercise, create two functions based on that code.

The first function should be called **encryptThisLetter**. **encryptThisLetter** will have one parameter of type **char** and it will have a return type of **int**.

The second function will be used to decode and the function should be called **decryptThisNumber**. This function will have a parameter of type **int** and it will return a type of **char**.

Chapter Six: Object-Oriented Programming



Object oriented programming is a way of thinking about programming in terms of objects. An object in programming is like an object in the real world. That is, an object is something that you can describe and does things. Objects in the real world also have relationships to other objects. That is, an object in the real world may be made up or other objects, use other objects or it may be a variation of another object.

To help understand objects in programming, consider the example of a real world object like a car. You can describe a car as something that has four wheels, a color, model, make and so on. A car also does things like go forward, turn left and stop. Finally, a car is related to other objects in the world. A car is a type of vehicle (which itself is a kind of object), a car is made up on other objects (engine, wheels, stereo system) and a car is used by a person.

This way of describing and organizing things is natural for most of us. We tend to think of the world in terms of objects. In programming, objects are like the car example: an object's attributes can be described and they can demonstrate behavior. Sometimes objects in programming actually correspond to objects in the real world. Programming objects also have relationships to other programming objects.

Programming languages that support objects are called object oriented programming languages. The object oriented programming language that is used with iOS is called Objective-C.

In order to really understand object oriented programming and how to implement objects in your app we need to get comfortable with some of the vocabulary and high-level concepts.

Object

In programming, an object is an entity that contains both data and behavior. Objects are described by many data points and may have many behaviors.

Objects in programming can correspond to objects in the real world like cars and people. In these cases, an object in our computer program acts as a model (a simplified version) of the real world object. An object in our program that represents a person may have data that represents things like a person's eye color, height and weight and the object will also have a way to represent a person's behaviors like walking, talking and eating.

Objects also are used to represent more abstract things like decision-making processes and software components like menus and buttons and Internet connections.

Attributes And Properties

The data that is contained in an object is referred as attributes or properties. An attribute is a data point that describes an aspect of an object. So a person object may include a name property as well as a height, age and weight property.

Objects are described by this collection of property values. Properties describe what the object ***IS***.

Behavior And Methods

Objects contain both data (properties) and behavior. Behavior is what the object can do. A person object may have the behavior of walking and talking. More software-centric objects may have behavior like opening an Internet connection or writing to the file system. Each behavior that an object can do is called a method. Methods describe what the object can ***DO***.

Messages

Objects communicate to one another using messages. So if a person object wanted a car object to slow down the person object would need to send a **slowDown**message to the car object. A message will correspond to a method in a given object. That is, in the example above the person object sent the message to the car object. This works only if the car object has a method defined in its class (defined next) that corresponds to the message. Here the car would need to have a **slowDown**message defined in its class definition.

Class Definition

Class definitions are used to define an object in programming. Just like we need to define our own composite type before we can use them we need to define our own class type before we can use objects. The class is what is used to define all the properties and methods that will make up the object that we will be using.

Classes are related to objects like blueprints are related to widgets in a factory. You create a blueprint once and then use this blueprint as a guide to create any

number of widgets. Class definitions are used to define what makes up an object and are used to create any number of classes.

Instantiation And Constructors

The process of creating an object from a class definition is called ***instantiation***. Objects may be instantiated from a class as often as the program needs more objects. Instantiation requires a special method called a ***constructor***. A constructor is a method that returns an object based on the object's class definition. In Objective-C you can tell what when a method is a constructor when it has the prefix**init**.

Here is an example of an object being instantiated from a constructor that you see often in iOS:

```
UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Alert View"
                                                message:@"Hello World"
                                               delegate:self
                                      cancelButtonTitle:@"Ok", nil
                                      otherButtonTitles:nil];
```

The code you see above is an example of creating an object using a constructor. Here we are using a class that has already been defined for us. The class name is**UIAlertView**. The object here is called **alert**. You may notice that alert has an asterisk ***** in front of it. This is something that is required for object declarations that you do not need for primitive or composite types. The constructer method is all the stuff that comes after the part that starts with **initWithTitle:**.

Object-Oriented Design Concepts

OOP is a way of thinking about code that helps programmers keep complex systems organized. OOP also helps programmers conceptualize systems by creating models and systems of interaction based on the what people expect in the real world. The first part of this has to do with the idea of objects. But, there are a few more design considerations unique to OOP that go beyond the idea of grouping methods and data together in objects.

You can think of the following design considerations as guideposts on how to think of the objects you create. Knowing the design ideas below will also help you to consume objects that other programmers create.

Responsibility

Responsibility has to do with the design idea that an object should be ultimately responsible for all the information and behavior that belongs to its domain. That is, if you have created a person object then that object will be responsible for everything that a person is and does.

You would not attach methods or properties that logically belong to the person object with other objects. This is something that you may find tempting when quickly coding something in your app. For example, it may not be out of the range of possibility that I would carelessly define a car method that had a rouge property on it called something like **driversEyeColor** even though this is not something that logically a car object would be responsible for. The problem with defining behaviors and attributes in a class were they don't belong is that it becomes quite painful to attempt to debug these things when needed later on.

Encapsulation

Encapsulation means that the internal workings of an object are hidden from the rest of the program. This means that any data and procedures required for the object to work are not directly accessible by other objects. Really other objects do not care or need to know how your object gets things done, they just need to know that certain things will get done.

You as the program must choose what will remain privately encapsulated in your object and what will become publicly available to other objects. Some behavior and data are not really important to the rest of the program so you may decide to just keep them to yourself. There will be some data and behaviors that you do want exposed to other objects and that is ok but you will have to make a judgment about what is private and stays encapsulated in your object and what is public and available for use by other objects.

These two ideas of responsibility and encapsulation with objects is what makes object oriented programming much easier to create an organized system. If you stick to following the notion of responsibility then you can be pretty sure that methods and data that you need to maintain in the future can be located in the appropriate class definition. Encapsulation ensures that we don't inappropriately meddle in the inner workings of other objects and keeps us from wasting unnecessary time on code that our other objects can't touch.

Inheritance (Is-A Relationship)

Objects in programs can have relationships just like objects in the real world. The inheritance relationship refers to kind of relationship where one object is a specific type of another object. Imagine that you have a computer object where you encapsulate everything that you know about what a computer is and does.

This computer object is fine most of the time but if you think about it there are other types of computers in the world: Windows PC, Macs, iPhones, onboard car computers and so on. Each of these things ***is-a*** computer but they are more than that as well.

In programming we call this relationship ***inheritance*** because if you were to design a class definition for **iPhone** your first step would be to inherit everything that already exists for **Computer**. This way we do not have to re-code everything that we have already coded for Computer. The class that you are inheriting from is called the parent class and when you inherit a parent class you automatically get all the parent classes attributes and behaviors as if you coded them again in your new class.

The inheritance relationship is one way in which object oriented programmers can efficiently deal with the problem of code entities that are mostly the same but have variations.

Composition (Has-A Relationship)

Composition is another relationship that objects can have and you will see this one all the time when you are looking at class definitions. What composition means is that objects may be composed of other objects. So maybe you have an office object. That object may have as part of its definition other objects that make it up. You will probably have a fax object, a secretary object, a desk object and objects for everything that makes up an office. We would say that an office object is composed of a desk object, a fax object and a secretary object.

This notion of composition allows us to still use the behaviors and attributes that we need for other objects while maintaining the idea of responsibility and encapsulation. So, in our office example we have a fax object that will be responsible for everything fax. The office object is not responsible for the behavior of the fax object even though the office object can use the fax object.

The Person Object Example

To make things a bit more concrete I am going to go through the process of describing a person in object oriented programming terms. If this is starting to feel a bit abstract don't worry too much about that – in the next chapter you will start to see how these object oriented concepts are applied to iOS programming with Objective-C.

When framing things in object oriented terms it helps to think of the object itself and just write down what the object is and what the object does. When I think of a person as an object some things that come to mind are that a person can be described by their name, age, eye color and height. Some things that a person can do is walk, talk and eat.

If I were to diagram this definition of a person that will eventually turn into a class definition for a person in my program it would look something like this:

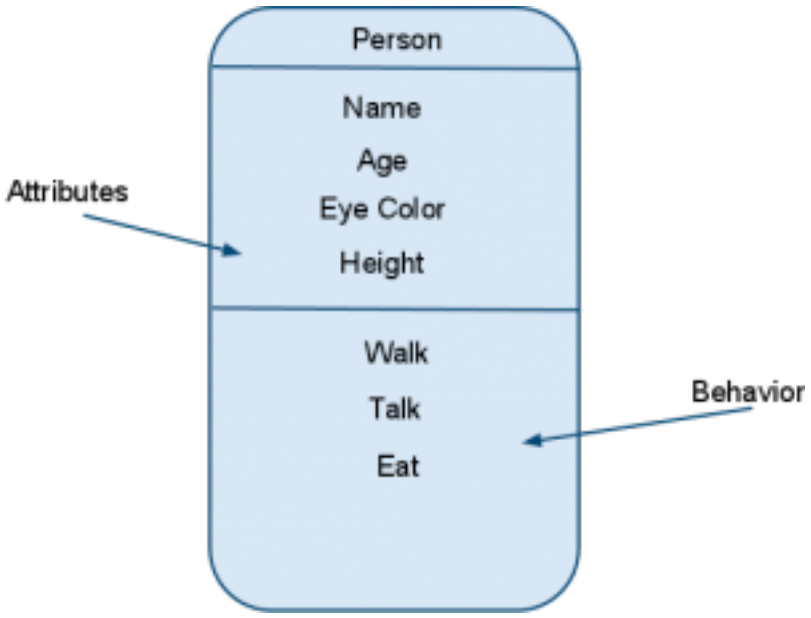


Figure 1 – Class Definition for a person

Now if I wanted to use this class definition of a person I would need to create objects by instantiating person objects using this class definition. Each person object would have the same attributes but the attribute values would be different. For example,

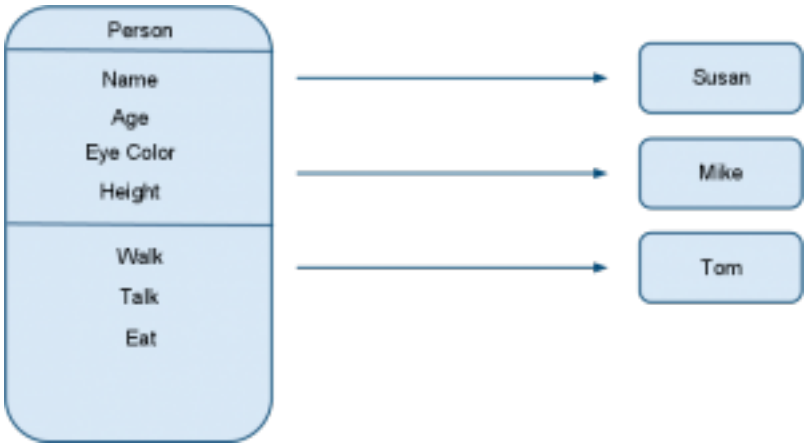


Figure 2 – Person Objects Instantiated From The Person Class Definition

My program may also need specific types of persons to deal with cases when a person has a special abilities or roles. So if I needed something to deal with artists, police officers and programmers I might have something like this:

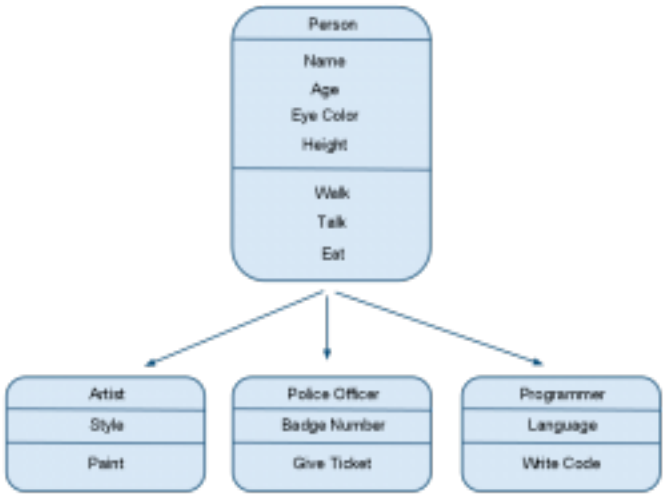


Figure 3 – Inheritance Is-A Relationships

Artist, **Police Officer** and **Programmer** are all a kind of **Person.** Each of these have their own attributes and behaviors but they all inherit things like name, eye color and the ability to walk and talk based on their relationship to the Parent class definition.

So this wraps up the high level object oriented stuff. In the next chapter, you will see these how these ideas are used in the object-oriented frameworks Foundation and**UIKit** that are used in iPhone development.

Chapter Seven: Introduction To Foundation & UIKit

iOS developers use two key object oriented frameworks called ***Foundation*** and ***UIKit***to do many of the most important programming tasks needed to make apps. Frameworks are collections of code that are shared by programmers that solve common problems. Foundation was built by engineers to solve common computing problems like using strings, managing collections of objects, connecting to the Internet, storing data, writing to the file system and much more. UIKit is the framework that is responsible for all the visual elements that you see on iOS apps like buttons, tables and tab bars.

I want to go over these frameworks with you in this chapter for two reasons. The first is that the best way to start learning object oriented programming is to see how objects are used in code. You need to learn how to instantiate objects from classes, manage memory with objects and how to use objects. Most of modern programming focuses on using code from frameworks like Foundation to develop software.

The NS And UI Prefixes

Let's take a look at a Hello World example first so we can see where some of these frameworks come into play.

```
-(BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions {

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Alert View"
                                                    message:@"Hello World"
                                                   delegate:self
                                          cancelButtonTitle:@"Ok"
                                          otherButtonTitles:nil];
    [alert show];
```

```
    [window makeKeyAndVisible];

    return YES;
}
```

I highlighted the code that I wanted to point out above. **UIAlert** is the class definition that we used above to instantiate our **alert** object. **UIAlert** objects are the dialog boxes that pop out and send the user a simple message. But, why does this class definition have the **UI** in front? Why not use the more understandable word Alert for a name of the class definition? Even odder you will find that there are many classes like this such as **UITableView**, **UITableViewController**, **UIView**, **UIViewController**,**UIButton** and so on. All of these classes are part of the framework called **UIKit**.

The **UI** and other two letter prefixes are there to help us distinguish classes that belong to a certain framework. So, all the **UI** classes belong to **UIKit**. It is a matter of convention to either make this prefix a hint about what the framework is supposed to do or to use the first two letters of the company or person that created the framework. In **UIKit** here the **UI** stands for User Interface which indicates to us that all the classes in UIKit with be responsible for user interface components like buttons and other things that appear on the screen.

NS is the prefix used for all the Foundation classes. This prefix is not as obvious, but most people believe that this refers to the company that originally created Foundation (NextStep). In the examples below you will see that the Foundation classes that we are using all have the **NS** prefix.

NSObject

One of the most important classes in Foundation is called **NSObject**. **NSObject** is a class that defines generally what an object is in Objective-C. **nNSObject** includes code that is useful for any object that we will use in programming. For this reason, most other classes are in an inheritance relationship with **NSObject**. **NSObject** is considered the root (or first) class.

One of the things that **NSObject** does is provide methods to allocate memory for objects and set the initial properties for objects. Most classes that inherit from**NSObject** use these methods to instantiate objects. **NSObject** also has a method called **description** that returns a text description that describes the class.

Sending Messages

In Objective-C, we get things done by sending messages to classes and objects. When you want something to happen you must send a message to the object or class responsible for the action.

Sending messages to objects is a bit like calling functions in procedural programming. When you send a message to an object or class the message will match a method definition (which is coded in a similar way to a function) that is part of the class. You will find out how to code your own methods in the next chapter. Messages can have parameters and they may perform some action or return a value. You can send messages to classes or to objects. You can send a message to a class at any time but you can only send messages to objects after the object has been instantiated.

Sending Messages To Classes

You can send a message to a class by enclosing the class definition name in square brackets and writing in the name of the message that you would like to send. For example, if I wanted to send a message to the **NSObject** class asking for a description of **NSObject** I would send the **description** message to **NSObject**:

```
[NSObject description];
```

n the message about we did not do anything with the return value (the description message returns a string). A more practical way to use the description message is by writing its output to the log like this:

```
NSLog(@"%@", [NSObject description]);
```

This would print out **NSObject**'s description to the log which is simply ***NSObject***. Notice too that the function, **NSLog**, that we have been using has the **NS** prefix which marks NSLog as part of Foundation.

The messages that you send to classes are defined in the class as ***class methods***. Class methods are methods that are only used with the class itself. You can find out what class methods are available to you by looking up the class definition in the documentation that Apple provides with iOS SDK. Each class

has a special section that lists all the class methods under the title ***Class Methods***.

Sending Messages To Objects

You can also send messages to objects. The only difference here is that you must have an object in place that has already been ***instantiated***. Messages sent to objects correspond to what are called ***instance methods*** defined in the class definition. You can find out what instance methods are available by looking up the class in the Apple documentation and checking under the heading ***Instance Methods***.

You will see some real examples of messages sent to objects below. But, they will all follow this pattern, which looks like the pattern we used to send messages to classes:

```
[object doSomething];
```

You can send messages to objects and classes that have parameters but before I show you how to do that I want to introduce you to some more Foundation classes so I can give you more concrete examples. First though, we need to see how to instantiate objects.

How To Create Objects (Instantiation)

To create objects we follow a process that starts off a bit like declaring variables in procedural programming. To jog your memory, to declare an integer variable you would simply type this:

```
int myNumber;
```

The first step to creating an object looks very similar, but for objects we do not use the primitives type definitions like **int** or **float**. Objects require a class definition in place of the type. Objects also require us to use the * in front of the object's variable name. Here is an example of how to declare an object using the **NSObject** class definition:

```
NSObject *object1;
```

Next we need to do two things: instantiate an instance of the class (the object) and initialize the object. To instantiate an instance of a class we send the alloc message to the class. This will allocate memory for the object and return the object, which we can then assign to our object variable.

```
object1 = [NSObject alloc];
```

The next part of the process is to initialize the new object. **NSObject** gives us the **init**instance method to do this for us, but other others will have other initialization methods. By convention, initialization methods begin with init.

You send the **init** message and assign the return value from **init** to the **object1**variable.

```
object1 = [object1 init];
```

That is all there is to instantiating an object in Objective-C. But, there is one more thing. Even though I showed you how to do this in three steps with three different lines of code, most programmers will do this all in one line of code by declaring the object and then using nested messages. Nested messages are messages that are enclosed in other messages. So usually you will see objects declared, instantiated and initialized all on the same line like this:

```
NSObject *object1 = [[NSObject alloc] init];
```

In the line of code above the **[NSObject alloc]** message is nested in inside the **init**message. How it works is that the innermost message gets sent first to do its thing and then the next nested message will be sent. So above you first send **[NSObject alloc]** which returns an instance of **NSObject**. Next we send the **init** message to the object that was just returned.

Creating Objects Without Alloc and Init

There is another way to create objects, and that is done through what I like to call convenience methods. These methods also return objects like the **alloc** and **init** methods, but they do things a little bit differently in the background.

Here is an example of a convenience method that returns an **NSString** object.

```
NSString *aString = [NSString stringWithString:@"My New String"];
```

NSString is a commonly used and important class that we will cover next. You can see from the code above that we are still creating an object, but we are not using**alloc** to allocate memory for the object nor are we using an **init** message to initialize the object. This is how you can tell that you are using a convenience method to create an object.

NSString

A string is a list of characters like “See spot run”. If you have working with strings using what you know of programming so far you have probably been a bit frustrated. C has support for individual characters with the **char** primitive type but if you need to use more than one letter you are would be in for more work if it wasn’t for NSString.

NSString is used very often in iOS apps so it is a great class to get to know well. For the first example we are going to create an object using the **NSString** class definition. Then we are going to use this object as a parameter in the **UIAlert** object alert to make **alert** present a different string to the user.

The easiest way to use **NSString** is to declare a **NSString** object and assign it a string value on the same line. Let’s create a **NSString** for “See spot run”:

```
NSString *spotString = [NSString stringWithString:@"See spot run"];
```

The first part of this line of code is **NSString *spotString**. This is how we declare our**NSString** object here called **spotString**. Since this is an object declaration we need to remember to use the * before the object name. In the next part we use the assignment operator to assign the string “See Spot Run” to the **spotString** object variable using the **NSString** convenience method **stringWithString:**.

The @ symbol here is a called a compiler directive. Compiler directives are used to let the compiler know that there is something special about the code following the directive. Here the @ indicates that the text in the parenthesis is a **NSString** object and not a regular C string. The code we are using here is a shorthand version of a convenience method that we covered earlier, **stringWithString:**.

Once you have a **NSString** object you may use it throughout your program when you need a string. Since we want to use our string immediately we can just use the convenience method to create the string object.

Here is an example of how you could use the string object. If we want our alert object to say “See spot run” instead of “Hello World” we can use **spotString** as a parameter in place of the “Hello World” string that we already have in there.

Before

```
-(BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions {

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Alert View"
                                                    message:@"Hello World"
                                                   delegate:self
                                          cancelButtonTitle: @"Ok"
                                          otherButtonTitles:nil];
```

```
    [alert show];

    [window makeKeyAndVisible];

    return YES;
}
```

After

```
-(BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions {

    NSString *spotString = @"See spot run";

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Alert View"
                                                    message:spotString
                                                   delegate:self
                                          cancelButtonTitle: @"Ok"
                                          otherButtonTitles:nil];
    [alert show];

    [window makeKeyAndVisible];

    return YES;
}
```

Now when you use the code to run your app you will see this instead of the Hello World message:

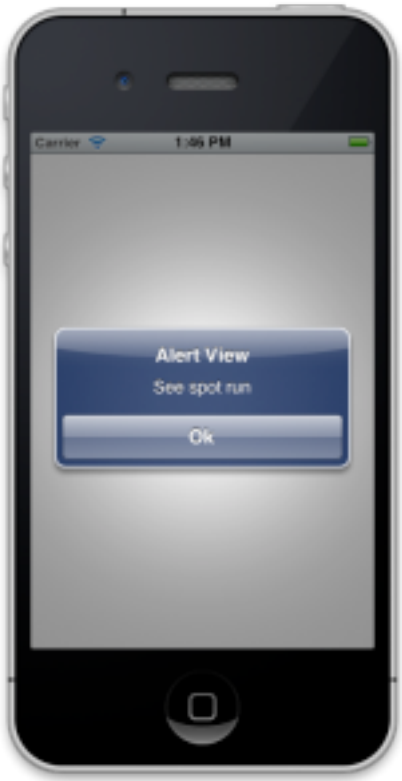


Figure 7.1: spotString object presented in alert

Since **spotString** is an object created from the **NSString** class definition you can do a lot more with **spotString** than simply use it in other objects. Remember that objects in object oriented programming have ***attributes*** and ***behaviors***. We usually call ***attributes properties*** when we talk about them in the context of code. **NSString**objects have two properties: **description** and **length**.

Description is the characters that make up the string. To use this property you can use ***dot notation***, **spotString.length** or as a shortcut you can simply use the object itself like you did in the example above. The length property is an integer that tells us how long the string is. So as an example here is how you would write out the length of **spotString** to the console screen:

```
NSLog(@"spotString.length = %i", spotString.length);
```

NSString objects also have some other methods that are very useful. For instance, if you would like see in your string in all capital letters you could use the**uppercaseString method:**.

```
NSLog(@"spotString all caps = %@", [spotString uppercaseString]);
```

The line above would print out this message to the console:

```
spotString all caps = SEE SPOT RUN
```

Similar methods exist to display lowercase letters, to compare strings, search for substrings and even to capitalize the first letter in the string.

Hands On Time

Foundation is a very rich framework so I am going to spend some more time showing you what you can do with Foundation classes in the next chapter. But, for now start getting a feel for how to use object oriented programming with Foundation by experimenting with the **NSString** class.

Change the Hello World code to have the **alert** object display an all lowercase version of your “Hello World” message. To do this you must create your own**NSString** object to use in the alert.

For a challenge, see if you can figure out how to use your **NSString** object with the string “Hello World” to display only the word “Hello”. Check out the documentation for some methods that will help you do this. The easiest way to check out the documentation for **NSString** is to hold down the ***command key*** and **click** on the word **NSString** while you are in XCode. You may also simply look up **NSString** on the Apple Developer website. While you are there take a look at all the features included with **NSString**.

Hint: the method that you are looking for as the word ***substring*** in it.

Chapter Eight: Essential Foundation Classes

Now that we have a better handle on how object oriented programming works on iOS we should go over some of the key Foundation classes that you will use in your own apps. Like the name suggests, Foundation supports the key frameworks used in iOS development. Most of what you will be doing when building iOS apps comes from Foundation and will use Foundation classes.

Foundation Inheritance Hierarchy

Remember that classes can have an ***inheritance relationship*** with other classes. Although we did not mention it specifically we have already seen classes in Foundation that are in an inheritance relationship. The two classes that I am talking about are **NSObject** and **NSString**. **NSString** inherits from **NSObject**. You might also say that **NSString** is a kind of **NSObject**. Because of this relationship, **NSString** has all the properties and methods it needs to be and do what an object is.

If you look at all the object's inheritance relationships as a whole you have what is called an inheritance hierarchy. It really helps to see the relationships between classes when you draw them out in a diagram. Below in Figure 1 is a diagram of the inheritance relationships that we have already covered in Foundation.

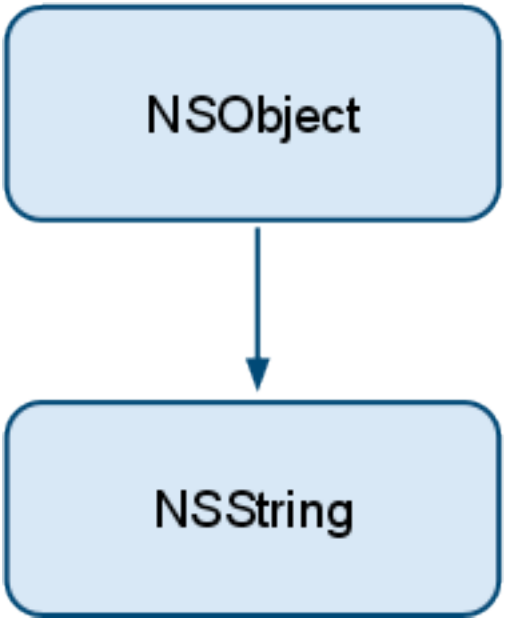


Figure 8.1: Inheritance

Let's fill out more of this inheritance hierarchy with some key Foundation classes.

NSMutableString

In you look closely at the documentation and examples for **NSString** you will not find any way to change the content of the string itself. That is because **NSString** cannot be changed so there is no way to add characters or to edit **NSString** objects in any way. If you need to use a string that has values that change you must use**NSMutableString**.

NSMutableString inherits from **NSString**, but **NSString** has additional behavior that you can use to change the content of the string. If you were to view**NSMutableString** in the context of our inheritance hierarchy it would look like this:

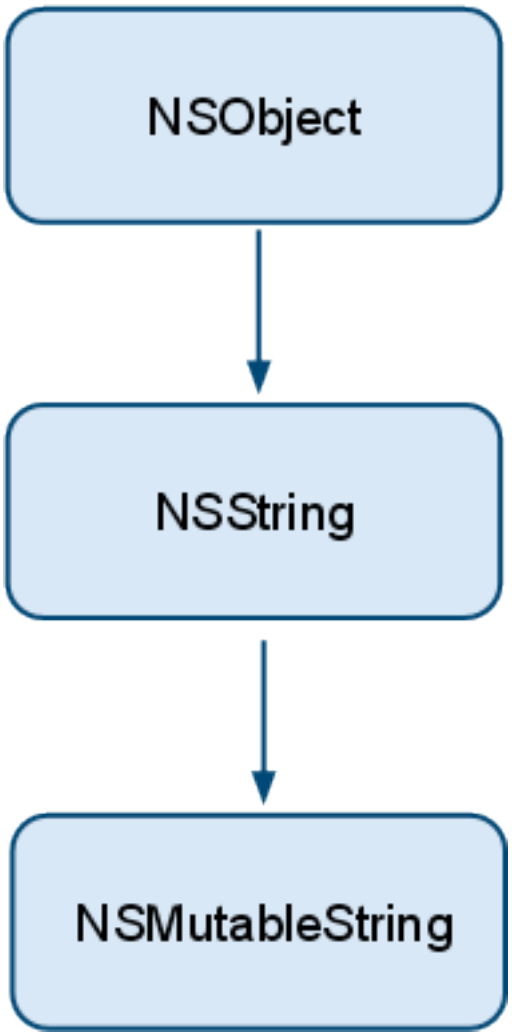


Figure 8.2: Inheritance

The first thing that I want to do to demonstrate **NSMutableString** is change the**UIAlertView** code that I used as the Hello World example. What I will have now is two **UIAlertView** objects so when the user uses the app two separate boxes will pop up with messages. Each time the **UIAlertView** object appears it will display the content in our **NSMutableString**. Here is what the code should look like:

```
UIAlertView *alert1 = [[UIAlertView alloc] initWithTitle:@"Alert View"
                                                 message:@"Hello World"
                                                delegate:self
                                       cancelButtonTitle:@"Ok", nil
                                       otherButtonTitles:nil];
[alert1 show];

UIAlertView *alert2 = [[UIAlertView alloc] initWithTitle:@"Alert View"
                                                 message:@"Hello World"
                                                delegate:self
                                       cancelButtonTitle:@"Ok", nil
                                       otherButtonTitles:nil];
[alert2 show];
```

When you build and run this project you will see two message boxes pop up, one right after another. Right now they will both still just say “Hello World” but next we will replace them with the contents of a **NSMutableString** object.

The first thing that we need to do is to instantiate a **NSMutableString** object. We can use a convenience method to do this for now.

```
NSMutableString *messageString = [NSMutableString
stringWithString:@"Hello"];
```

Make sure to locate this object before the code used to create the two **UIAlertView**objects. Now let’s replace the text in the two **UIAlertView** objects with the**NSMutableString** object that we just created:

```
NSMutableString *messageString = [NSMutableString stringWithString:@"Hello"];

UIAlertView *alert1 = [[UIAlertView alloc] initWithTitle:@"Alert View"
                                                 message:messageString
                                                delegate:self
                                       cancelButtonTitle:@"Ok", nil
                                       otherButtonTitles:nil];
[alert1 show];

UIAlertView *alert2 = [[UIAlertView alloc] initWithTitle:@"Alert View"
                                                 message:messageString
```

```
                                                delegate:self
                                        cancelButtonTitle:@"Ok", nil
                                        otherButtonTitles:nil];
[alert2 show];
```

If you run this right now you will simply get two **UIAlertViews** that say “Hello”. What I want to do now is use **appendString** method to add “ World” to the hello world message. To do this we can send the **appendString** message to the **messageString**object right in between two **UIAlertView** objects.

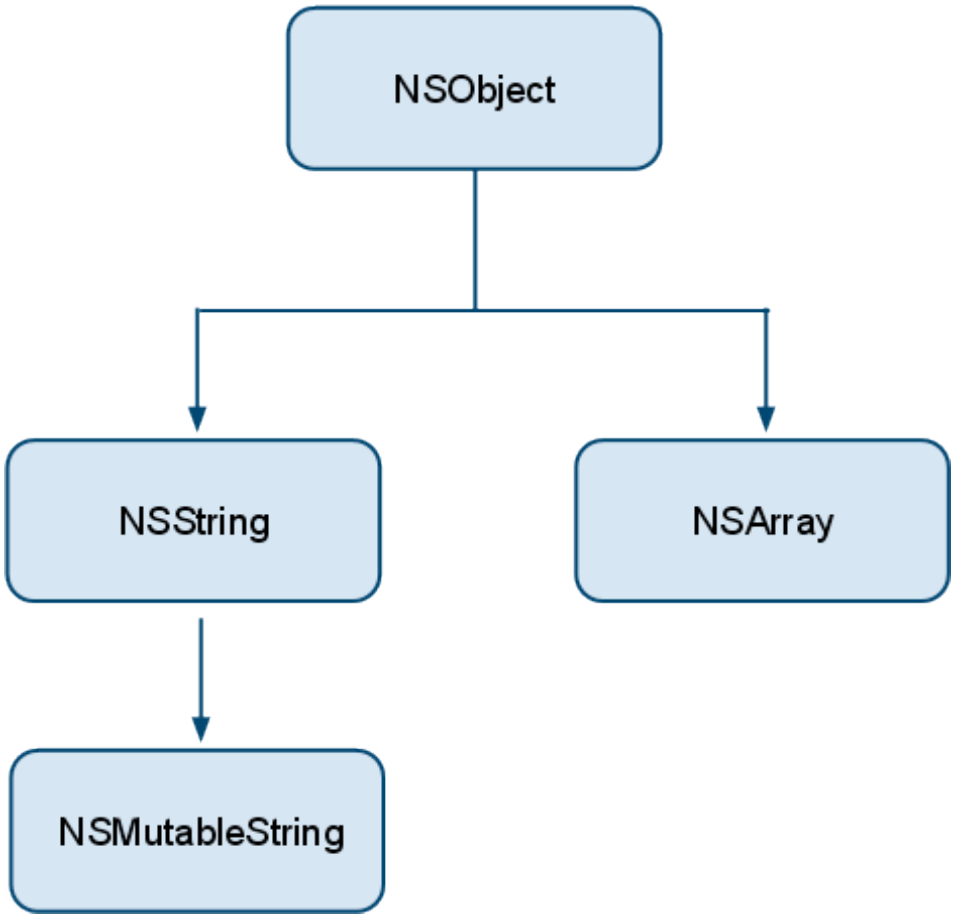
```
NSMutableString *messageString = [NSMutableString stringWithString:@"Hello"];

UIAlertView *alert1 = [[UIAlertView alloc] initWithTitle:@"Alert View"
                                                 message:messageString
                                                delegate:self
                                        cancelButtonTitle:@"Ok", nil
                                        otherButtonTitles:nil];
[alert1 show];

[messageString appendString:@" World"];

UIAlertView *alert2 = [[UIAlertView alloc] initWithTitle:@"Alert View"
                                                 message:messageString
                                                delegate:self
                                        cancelButtonTitle:@"Ok", nil
                                        otherButtonTitles:nil];
[alert2 show];
```

Now when you build and run the project the first **UIAlertView** object will pop up and say “Hello” and the second will say “Hello World”.

NSMutableString has more methods that you can use to alter the contents of the string: you can insert characters, delete characters and more.

NSArray

NSArray is a class that you can use when you want to work with a list of other objects. This is the object oriented version of the arrays that you were introduced to in the chapter on variables and arrays. The image below shows you how **NSArray** fits into the inheritance hierarchy. **NSArray** can only be used for objects (no primitive types like **int** or **float** allowed). Unlike the arrays you already have seen, **NSArray** has behaviors that can help you manage the objects contained in the array.



To create a **NSArray** object you must send the **alloc** and **initWithObjects** messages.**initWithObjects** requires a comma-separated list of objects and the last one must be**nil**. Let's create an **NSArray** object right now:

```
NSArray *listOfStrings = [[NSArray alloc] initWithObjects:@"One", @"Two", @"Three", nil];
```

If you are following along you can put this code right after the line of code where you created your **NSMutableString** object. Each of the strings in the code above (@"One", @"Two", @"Three") are all **NSString** objects here. Using the **@** compiler directive along with double quotes is a shortcut that you can use anytime that you need a temporary **NSString** object. The initWithObjects method requires a list of objects and the last object must be nil. We have not discussed nil yet, essentially nil is a reference to an empty object.

Now that you have this list of objects called **listOfStrings** you can access each of the members of this list using the **objectAtIndex** message. **objectAtIndex** requires us to send along a parameter to tell the **NSArray** object what object we want to reference. This parameter will be an integer that corresponds to the member of the array that we want. Let's say we want to use the first and second members of **listOfStrings** in our **UIAlertView** objects. We could replace our **NSMutableString**object with an **objectAtIndex** message sent to **listOfStrings**.

```
NSMutableString *messageString = [NSMutableString stringWithString:@"Hello"];

NSArray *listOfStrings = [[NSArray alloc] initWithObjects:@"One", @"Two", @"Three", nil];

UIAlertView *alert1 = [[UIAlertView alloc] initWithTitle:@"Alert View"
                                                 message:[listOfStrings objectAtIndex:0]
                                                delegate:self
```

```
                                          cancelButtonTitle:@"Ok", nil
                                          otherButtonTitles:nil];
[alert1 show];

UIAlertView *alert2 = [[UIAlertView alloc] initWithTitle:@"Alert View"
                                                  message:[listOfStrings objectAtIndex:1]
                                                 delegate:self
                                          cancelButtonTitle:@"Ok", nil
                                          otherButtonTitles:nil];
[alert2 show];
```

When you ***Build and Run*** your project now the two **UIAlertView** objects will display the first and second members of your list. See the image below as an example:

For Each Loop

A really nice feature of **NSArray** is that you can use the for each loop. We already have three types of loops that we use in programming that we learned about in previous chapters: for, do and while loops. These loops all require used to either know how many actions we needed to take beforehand, how many elements were in the arrays were using or they required us to define some other condition to end the loop. For each loops don't require any of these things and they are very easy to use.

The idea behind the for each loop is that the loop will do something for each member of the list. All you really need to do is set up a for statement that specifies that class definition of the objects in the list. For example, if I want to quickly write out each string in our **listOfStrings** object I could simply use a for each loop to do this in two lines of code:

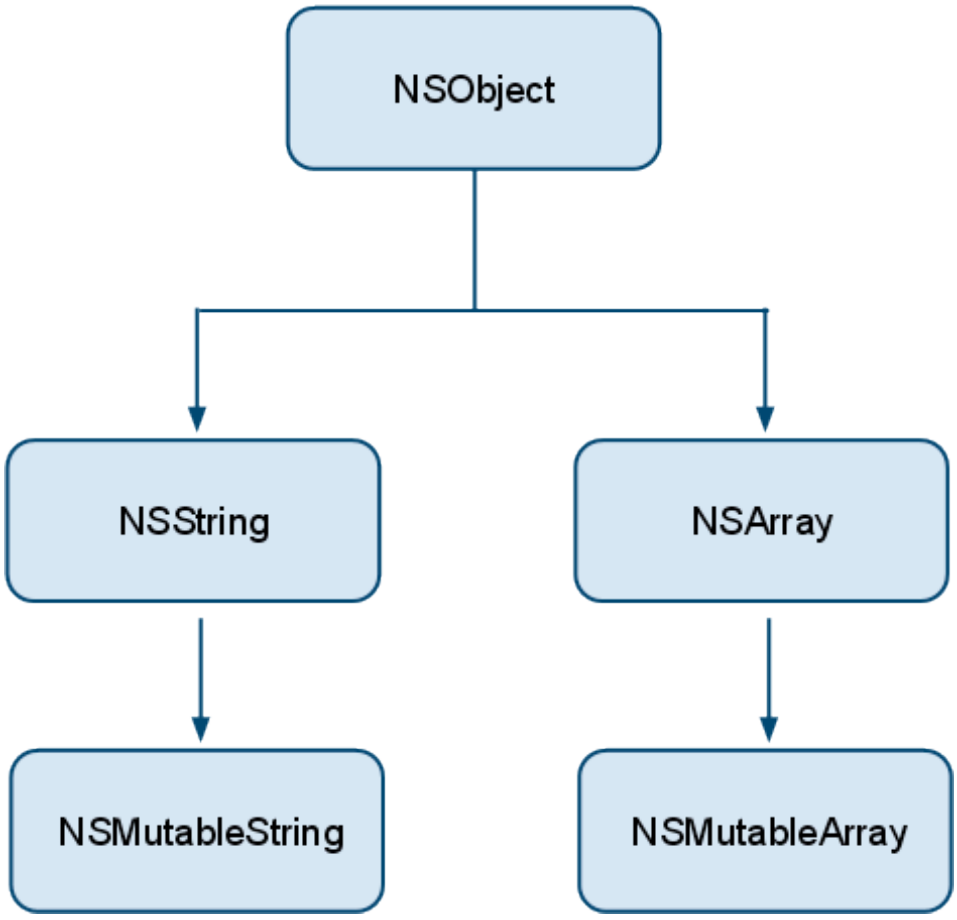
```
for(NSString *s in listOfStrings)
          NSLog(@"%@", s);
```

We need to start out our for each loop with the keyword for and then put an expression in parenthesis that includes the class definition (**NSString** here), a temporary object variable ***s**, the keyword **in** and finally the **NSArray** object that we are looking at. The statement above will print something like this out to your console:

```
One
Two
Three
```

NSMutableArray

NSArray is great, but like **NSString** you cannot make any changes to **NSArray**objects once you create them. This means that you cannot add or remove objects from your lists. If you need a list that has the ability to add and remove objects then you must use **NSMutableArray**. **NSMutableArray** is a kind of **NSArray** and so has all the features of **NSArray** but with the additional of methods that allow adding and removing objects. See the image below to find out where **NSMutableArray** belongs in the Foundation inheritance hierarchy.



You can instantiate a **NSMutableArray** object in the same way as a **NSArray** object. There are actually a few ways to create objects like this, but the easiest way to create a **NSMutableArray** object is to simply use the **alloc** and **init** messages.

```
NSMutableArray *mutableListOfStrings = [[NSMutableArray alloc] init];
```

You can now start adding objects to this list. Just to demonstrate I am going to add some strings to **mutableListOfStrings** objects by using the **addObject** message.

```
NSMutableArray *mutableListOfStrings = [[NSMutableArray alloc] init];
[mutableListOfStrings addObject:@"One"];
[mutableListOfStrings addObject:@"Two"];
[mutableListOfStrings addObject:@"Three"];
```

You can use **NSMutableArray** objects like **NSArray** objects as well since**NSMutableArray** is an kind of **NSArray**. That means that you can use the for each loop with **NSMutableArray** so this would work in the same way as it did for the previous example.

```
NSMutableArray *mutableListOfStrings = [[NSMutableArray alloc] init];
[mutableListOfStrings addObject:@"One"];
[mutableListOfStrings addObject:@"Two"];
[mutableListOfStrings addObject:@"Three"];

for(NSString *s in mutableListOfStrings)
     NSLog(@"%@", s);
```

You can also remove objects from **NSMutableArray** lists by using **removeObjectAtIndex** message. This message requires the position of the

member object that you want to remove. So if you want to remove the second object in **mutableListOfStrings** then you need to pass the integer 1 as a parameter with this message.

NOTE: Arrays are indexed starting with the integer 0 so the first member in an array would be member 0, the second would be member 1 and so on.

Here is an example of how to remove an object from an **NSMutableArray** list:

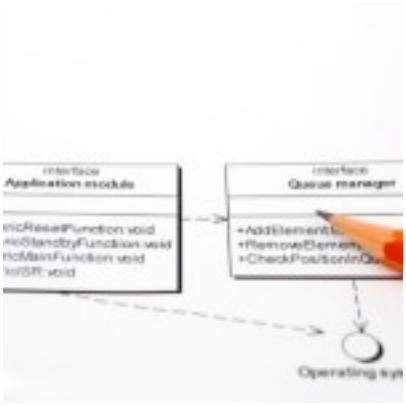
```
[mutableListOfStrings removeObjectAtIndex:1];
```

You have some variations of this method that you can use as well as methods to insert objects as well. As always, it is a good idea to consult the documentation to get a full appreciation of what you can do with **NSMutableArray**.

Hands On Time

Create a **NSMutableArray** and add the names of five people that you know to the array. Next, look up the **insertObject:AtIndex** method to find out how to add your own name to the middle of the array. Finally, use a for each loop to go through your entire array. Add code to your for each loop to have a **UIAlertView** present each name.

Chapter Nine: How To Create Your Own Classes



At some point you will want to define your own classes to use in your app. So now let's expand on the notion of object oriented programming by showing you how to implement your own custom classes. At this point you should be familiar with using existing objects in code; but you also need to be able to create your own types of objects.

Sub-Classes

One of the most common things that you will be doing with classes is something called sub-classing. As you can guess the iPhone has classes already in place to do things like present data tables, present windows and so on. Usually you will be creating your own object that inherits one of these preexisting objects and adding your own custom code to it. This is sub-classing.

Adding a New Class

XCode will help you do the initial set up work when you are creating a new class. To add a class you can simply **control-click** on a group folder in XCode and select **New File…** . In the dialog box that pops up select **iOS > Cocoa Touch > Objective-C class** and click **Next** . Then name your class and choose **NSObject** as the subclass.

Let's do this now and call our class **myClass**. XCode created both the interface and the implementation files for **myClass** and added the code we need to get started. The interface file is the one that ends in **.h**. If you open that file you will see the code that XCode automatically filled in for us.

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

@interface myClass : NSObject{

}

@end
```

Interface File

This interface file uses an import directive to import the Foundation classes (which we always need to use). It also uses the interface keyword with the name of the class (**myClass** here) to indicate that we are defining a class.

The colon and the name **NSObject** means that our class will be a subclass of**NSObject**. Sub-classing and inheritance mean the same thing. You can say that**myClass** inherits from **NSObject** or you can say **myClass** is a subclass of **NSObject**.

We end the interface declaration with the end keyword **@end**.

Implementation File

XCode also created an implementation file, in case you did not notice XCode will create a separate file for the interface (header file that ends in .h) and the

implementation file (ends in **.m**). Click on the implementation file to see the code that XCode added on our behalf.

```
#import "myClass.h"

@implementation myClass

@end
```

Our implementation file simply imports the **myClass** interface and includes the**@implementation** and **@end** keywords along with the name of our class.

Simple enough – we have created our first class. Right now it behaves exactly like**NSObject**, but soon we will be adding our own properties and methods to the class.

Adding Properties

When you add a property in a class you end up using a little black magic. That is, the system takes care of some of the implementation details for you if you do things in the typical way. You need to do two things to get a working property into your class: declare the property and then use **@synthesize**.

Here is what this looks like in the code:

Interface File (myClass.h):

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

@interface myClass : NSObject

@property (strong) NSString *name;
```

```
@end
```

The property is defined by using the **@property** directive. The key thing to note here is **strong**.

Back in the implementation file we use **@synthesize** to evoke all the black magic that is required to create the property based on our definition in the interface file.

```
#import "myClass.h"

@implementation myClass
@synthesize name;

@end
```

One thing that is significant though is that when you use primitive types as properties the syntax is a little different. For example, I am going to add a number property to my class that is an integer. Notice that it is done in a slightly different way:

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

@interface myClass : NSObject

@property (strong) NSString *name;
@property (assign) int number;

@end
```

The first thing is that there is no asterisk * since this is not an object variable. The other thing is that instead of using the **strong** keyword it is using **assign** in the property declaration.

Dot Notation

What all this does for you is give you the ability to assign and retrieve the property values using dot notation. You have already encountered this in the examples. Here is how you use dot notation with a **myClass** object.

```
//Instantiate an object from myClass:
myClass *object = [[myClass alloc] init];

//assign properties:
object.name = @"Obie";
object.number = 3;

//Access and use the properties
//with dot notation:
NSLog(@"My object's name is %@", object.name);
NSLog(@"My Object's number is %i", object.number);
```

Note that we use an **alloc** and **init** message even though we never created these in our class definition. Since our class is a subclass of **NSObject** we can simply use the original **NSObject** constructer to create objects. We can also code additional constructors if we want to set properties before we return our object to the system.

Class Methods

As you remember from chapter ten , objects have both properties and methods. Properties describe the object (the object's attributes) while methods are what the object does (the object's behaviors).

Furthermore, methods can be either class methods or instance methods. Class methods do not require an instance of a class (an object) to be used, you simply send the message to the class itself.

Here is how to declare a class method:

Interface File:

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

@interface myClass : NSObject

@property (strong) NSString *name;
@property (assign) int number;

+(void)writeSomething;

@end
Implementation File:
#import "myClass.h"

@implementation myClass

@synthesize name, number;

+(void)writeSomething{
      NSLog(@"I'm writing something");
}

@end
```

Here, the plus sign indicates that this is a class method and the void in between the parenthesis means that the method will not return a value. This is followed by

the name of the method and curly braces. Whatever we want the function to do we put in between the curly braces as code.

Since this is a class method we do not even need an object to use this method. All we need to do is send a message to the class name itself:

```
[myClass writeSomething];
```

Instance Methods

Instance methods also are used to code behavior, but these are different than class methods because they may only be used by sending a message to an object. You code them the same way inside the class definition but you prefix the method with a minus sign instead of a plus sign:

Interface File:

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

@interface myClass : NSObject

@property (strong) NSString *name;
@property (assign) int number;

+(void)writeSomething;
-(void)writeSomethingForAnObject;

@end
```

Implementation File:

```
#import "myClass.h"
```

```
@implementation myClass

@synthesize name, number;

+(void)writeSomething{
      NSLog(@"I'm writing something");
}

-(void)writeSomethingForAnObject{
      NSLog(@"I'm writing something, but only as an object");
}

@end
```

To use an instance method we must have an object available.

```
myClass *object = [[myClass alloc] init];
[object writeSomethingForAnObject];
```

Method Parameters

Like functions in C you can pass parameters to methods in Objective-C. The syntax looks different. To put a parameter into the method declaration you must add a colon after the method name and then declare the object type and name the parameter:

Interface File:

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

@interface myClass : NSObject

@property (strong) NSString *name;
@property (assign) int number;
```

```
+(void)writeSomething;
-(void)writeSomethingForAnObject;
-(void)aMethodWithThisParameter:(NSString *)param;

@end
```

Implementation File:

```
#import "myClass.h"

@implementation myClass

@synthesize name, number;

+(void)writeSomething{
      NSLog(@"I'm writing something");
}

-(void)writeSomethingForAnObject{
      NSLog(@"I'm writing something, but only as an object");
}

-(void)aMethodWithThisParameter:(NSString *)param{
      NSLog(@"%@", param);
}

@end
```

To use this method you would do this (this will be familiar to you):

```
myClass *object = [[myClass alloc] init];
[object aMethodWithThisParameter:@"Say What?"];
```

Multiple Parameters

You may have your methods take more than one parameter. Objective-C has a unique way of doing this where you may include a descriptive phrase in the method declaration.

Interface File:

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

@interface myClass : NSObject

@property (strong) NSString *name;
@property (assign) int number;

+(void)writeSomething;
-(void)writeSomethingForAnObject;
-(void)aMethodWithThisParameter:(NSString *)param;
-(void)aMethodWithThisParameter:(NSString *)param
               andThisParameter:(int)num;

@end
```

Implementation File:

```
#import "myClass.h"

@implementation myClass
@synthesize name, number;

+(void)writeSomething{
      NSLog(@"I'm writing something");
}

-(void)writeSomethingForAnObject{
      NSLog(@"I'm writing something, but only as an object");
}

-(void)aMethodWithThisParameter:(NSString *)param{
      NSLog(@"%@", param);
}

-(void)aMethodWithThisParameter:(NSString *)param
               andThisParameter:(int)num{
      NSLog(@"%@ + %i", param, num);
}
```

```
@end
```

The key difference between methods with one parameter and methods with two is that starting with the second parameter each gets its own descriptive text prefix. Above it is **andThisParameter:** which is in front of the parameter **num**.

Your probably already guessed that you send a message to an object using two parameters like this:

```
[object aMethodWithThisParameter:@"One" andThisParameter:2];
```

Constructors

We know that constructers are special methods that return an instance of an object back to the system. We are already using the constructer that we inherited from**NSObject**, **init** to instantiate an object from **myClass**. We can create a custom constructor if it is needed.

In general, we always prefix our constructor with **init**. This is a matter of convention. Let's start by defining our constructor in the **myClass** implementation file:

```
#import

@interface myClass : NSObject

@property (strong) NSString *name;
```

```
@property (assign) int number;

+(void)writeSomething;
-(void)writeSomethingForAnObject;
-(void)aMethodWithThisParameter:(NSString *)param;
-(void)aMethodWithThisParameter:(NSString *)param
                        andThisParameter:(int)num;
-(id)initWithName:(NSString *) aName;

@end
```

Instead of being declared as a **void** type as our previous methods our constructor will be declared as an **id**. id means that our method will be returning a value that not yet defined. Whenever you replace the void with another type you are saying that this method will be returning a value (like a function in C).

Our constructor will also be taking a parameter which we will use to assign a name to the object's name property. Here is how we implement this constructor:

```
#import "myClass.h"

@implementation myClass
@synthesize name, number;

+(void)writeSomething{
      NSLog(@"I'm writing something");
}

-(void)writeSomethingForAnObject{
      NSLog(@"I'm writing something, but only as an object");
}

-(void)aMethodWithThisParameter:(NSString *)param{
      NSLog(@"%@", param);
}

-(void)aMethodWithThisParameter:(NSString *)param
                        andThisParameter:(int)num{
      NSLog(@"%@ + %i", param, num);
}
```

```
-(id)initWithName:(NSString *) aName{
      if (self = [super init]){
            self.name = aName;
      }
      return self;
}

@end
```

In the if-then statement above we assign the object returned from the constructor that we inherited from **NSObject** to the current object (notice that there is only one equals sign). The self keyword always refers to the object in which the **self** keyword was placed while the **super** keyword refers to the class it is being inherited from (sometimes called the superclass).

Once we have successfully retrieved the object from the **super init** we can assign values to our properties.

Here is how you would use your new constructor to create an object:

```
myClass *object = [[myClass alloc] initWithName:@"MyObject"];
```

Hands On Time

Practice creating your class by first thinking all some attributes and behaviors could represent a person in your code. Keep it simple but meaningful. Use the techniques that you learned in this chapter to create a class definition for a person. Instantiate and use objects created from this class definition. Try using a **NSMutableArray** to hold a list of your person objects.

Chapter 10: Extending Classes with Categories

Much of the day to day work in object-oriented programming involves using classes that are already written. Sometimes we want our classes to do just a *little bit more*. That means they need more properties and methods and adding these entities changes the nature of the class. This is called sub-classing and we already covered how to do that.

There are times, however, when we want to extend the abilities of class but we don't want to define a completely new class. You will encounter these situations more and more as you plunge into Objective-C programming. Perhaps more importantly, you will find examples of category use throughout the Mac and iOS programming examples written by Apple and other developers.

So, to extend a class without sub-classing you have the option of using ***categories***. Let's talk about how to do this.

Defining a Category

In the last chapter we worked with a class called **myClass** so let's continue to use this class as an example. Let's assume that we are working with **myClass** but we need one more method that **myClass** does not currently support. Only our application delegate needs this method and so it we do not want every component in our system to have access to this very specialized method.

This is a great opportunity to use a ***category***. A category definition looks like a class definition in that it has an interface and an implementation definition. To keep things simple I just put it into the file I am working with at the time before the implementation. However, you could also put them into an external file and use an import directive if you want to share categories.

For now I am just going to add this to the application delegate and I will put code right after the import directives and before the application delegate's own implementation:

```
#import "ObjCExamplesAppDelegate.h"
#import "myClass.h"

@interface myClass (AddToMyClass)
- (int) numOfStuff;
@end

@implementation myClass (AddToMyClass)
- (int) numOfStuff{
     return 5;
}
@end

@implementation ObjCExamplesAppDelegate
@synthesize window;

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions {
    [window makeKeyAndVisible];
}

@end
```

The two highlighted regions above are the interface and implementation for the **AddToMyClass** category. As you can see these look like class definitions except that these do not use the colon indicating that they are a subclass but they have

the category name in parenthesis instead. Defining the methods works the same way as it does for classes.

Now that we have included this category definition we can use the **numOfStuff** function from within the application delegate. You use these methods like any other:

```
#import "ObjCExamplesAppDelegate.h"
#import "myClass.h"

@interface myClass (AddToMyClass)
- (int) numOfStuff;
@end

@implementation myClass (AddToMyClass)
- (int) numOfStuff{
     return 5;
}
@end

@implementation ObjCExamplesAppDelegate
@synthesize window;

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions {

     myClass *object = [[myClass alloc] initWithName:@"MyObject"];
     NSLog(@"Numbers of Stuffs: %i", [object numOfStuff]);

    [window makeKeyAndVisible];
}

@end
```

Using Categories with Standard Objects

One clever thing that you may find yourself doing with categories is using them with the regular foundation classes. It is a good way to quickly extend these

classes when especially when it would be confusing to use your own subclasses of common classes like **NSString**.

Extend NSString

Here is an example of how and why you may want to do this. I commonly need to enclose strings into HTML tags like <p> and <h3> in my blogging application. I could simply subclass **NSString** and maybe call it **NSHTMLString** or something and give that class new methods to make adding tags a little easier.

I could simply add a category as an alternative. So, here is how I would do that in my application delegate:

```
#import "ObjCExamplesAppDelegate.h"
#import "myClass.h"

@interface NSString (HTMLTags)
- (NSString *) encloseWithParagraphTags;
@end

@implementation NSString (HTMLTags)
- (NSString *) encloseWithParagraphTags{
      return [NSString stringWithFormat:@"<p>%@</p>",self];
}
@end

@implementation ObjCExamplesAppDelegate
@synthesize window;

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary
*)launchOptions {
      NSString *text =@"Have you ever read a blog post?";
      NSLog(@"%@", [text encloseWithParagraphTags]);
      [window makeKeyAndVisible];
}

@end
```

All I need to do to turn my string into a correctly tagged paragraph is send the **encloseWithParagraphTags** message. Here is the output:

```
ObjCExamples[644:20b] <p>Have you ever read a blog post?</p>
```

Summary

In this chapter you learned an alternative to subclassing when you want to extend class functionality. You may not need to use this technique in the beginning, but using categories is a simple trick that could make your life easier down the road.

Hands On Time

Experiment with categories by adding methods to the **NSString** class that you think could be useful. Try to use this method by putting the category definition in a separate file and using an import directive to share the category definition among at least two other classes in a project.

Chapter 11: Protocols In Objective-C

A protocol is a way to require classes to implement a set of methods. We define a protocol much like a class or a category, it is essentially a list of methods. Protocols do not do anything themselves, they are a definition of a type of contract that we can require other classes to implement.

We call that adopting the protocol and we've seen this before. You may recall that the application delegate file that XCode created for use included this as part of the application delegate interface:

```
<UIApplicationDelegate>
```

This was required because we needed our class to act as an application delegate. This delegation pattern is implemented using protocols because we need a way to require a class to implement a set of methods in order for it to act as a delegate. The code above is how you indicate to the system that a class is adopting a protocol.

How to Define a Protocol

To define a protocol you need a separate header file. You can get one of these by right clicking your group folder and selecting **New > New File... > C and C ++ >**

Header File. Then you simply start the protocol definition using the **@protocol** keyword:

```
#import <Foundation/Foundation.h>
@protocol ImplementIt
@end
```

As you can see it works like a class definition. All you need to do now is fill in what methods you want to be part of the protocol. I want my ImplementIt protocol to require my classes to return three strings that would be used in HTML documents:

```
#import <Foundation/Foundation.h>
@protocol ImplementIt
- (NSString *) header;
- (NSString *) title;
- (NSString *) paragraph;
@end
```

That is all there is too it and this looks just like an interface definition. If I want to have a class adopt my protocol all I need to do is add <ImplementIt> to the interface definition and import the header file. Of course, now I will be required to implement the three methods above or XCode will report an error.

Here is how I did this for **myClass**:

Interface:

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

```
#import "ImplementIt.h"
@interface myClass : NSObject<ImplementIt>
[CODE OMITTED HERE]
@end
```

Implementation:

```
#import "myClass.h"
@implementation myClass
@synthesize name, number;
[CODE OMITTED]
//Implementing ImplementIt protocol
- (NSString *) header{
      return @"<h1>Header Text</h1>";
}
- (NSString *) title{
      return @"<title>Title Text</title>";
}
- (NSString *) paragraph{
      return @"<p>Paragraph Text</p>";
}
@end
```

That is all there is to it. If you have not done much object oriented programming and are confused about why you would need to use protocols don't worry about it right now. This technique solves a problem that you probably just have not dealt with yet. The key thing to remember for now is that protocols are what is required to make delegation work and delegation is used frequently in iPhone programming.

Chapter 12: Key-Value Coding (KVC)

Another advanced technique is key value coding or KVC. This involves using a set of **NSObject** methods that can set or get property values based on their string representation. So, if a class has a **name** property you can get or set that value by using the dot notation in code or you can query the object with the string **name** to get or set the value.

This is an alternative to the typical way of working with properties that you can use in situations where you may not know the name of the property you will need at the time of writing the code. You may also use KVC to query arrays of objects or even arrays that are properties of an object.

How to Use KVC

Let's assume that I have an object of **myClass** available to me. If I wanted to find the name I could just use dot notation as we have been discussing:

```
NSLog(@"name: %@", object.name);
```

Alternatively I could use KVC:

```
NSLog(@"name: %@", [object valueForKey:@"name"]);
```

The less than obvious benefit to KVC is that you do not really need to know the name of the property beforehand to use the property value. Again, if you do not see why this is a big deal then this is just solving a problem that you have not yet run into. When you do remember to come back to this section and apply what you learn.

You can also set properties using KVC:

```
[object setValue:@"New Object Name" forKey:@"name"];
```

You can even use KVC on hierarchies of objects. So, if I wanted to get the uppercase representation of my string name I could use the **valueForKeypath** message to do this:

```
NSLog([object valueForKeyPath:@"name.uppercaseString"]);
```

The code above hints a bit more at why this technique is powerful. KVC really shines when we have an array of objects that we want to query.

How to Query an Array of Objects

We will end with this very powerful trick. Imagine that you have a list of objects and each object is composed of many properties. There may be some times where you would like to know what property value is in each object in your list.

For instance, if you had a list of contacts you may want to just extract every email address that is in the list.

You can do this with KVC very easily. All you need to do is send the **valueForKey** message to the array that is holding your objects and you will get a new array filled with each value that you asked for. In the example below I will first add three **myClass** objects to an array:

```
myClass *object = [[myClass alloc] init];
NSMutableArray *listOfObjects = [[NSMutableArray alloc] init];
object = [[myClass alloc] initWithName:@"FirstName"];
[listOfObjects addObject:object];
object = [[myClass alloc] initWithName:@"AnotherName"];
[listOfObjects addObject:object];
object = [[myClass alloc] initWithName:@"LastName"];
[listOfObjects addObject:object];
```

Now, to retrieve the value of each name property I will use the **valueForKey** message to populate a new array with the name values:

```
NSArray *results = [listOfObjects valueForKey:@"name"];
for(NSString *s in results)
      NSLog(@"name: %@", s);
```

The code above puts all the name values into the results array and prints them out to the log:

```
ObjCExamples[4646:20b] name: FirstName
ObjCExamples[4646:20b] name: AnotherName
ObjCExamples[4646:20b] name: LastName
```

Hands-On Time

Use the class definitions you created in the previous lesson to experiment with key-value coding. Put your objects into an array and find different ways to retrieve property values in your objects. Think of a way to incorporate the idea of protocols into your car and driver system. Implement your idea.