
Become an Xcoder

Start Programming
the Mac Using Objective-C



By Bert Altenburg, Alex Clarke and Philippe Mouglin

Copyright © 2006 by Bert Altenburg, Alex Clarke and Philippe Mougin. Version 1.2.
Attribution: The licensors, Bert Altenburg, Alex Clarke and Philippe Mougin, permit others to copy, modify and distribute the work. In return, the licensees must give the original authors credit.

Non-commercial: The licensors permit others to copy, modify and distribute the work and use the work in paid-for and free courses. In return, licensees may not sell the work itself, although it may accompany other work that is sold.

Contents

INTRODUCTION	4
CHAPTER 0 - BEFORE WE START	5
CHAPTER 1 - A PROGRAM IS A SERIES OF INSTRUCTIONS	6
CHAPTER 2 - NO COMMENT? UNACCEPTABLE!	11
CHAPTER 3 - FUNCTIONS	12
CHAPTER 4 - PRINTING ON SCREEN	17
CHAPTER 5 - COMPILING AND RUNING A PROGRAM	21
CHAPTER 6 - CONDITIONAL STATEMENTS	28
CHAPTER 7 - REPEATING STATEMENTS FOR A WHILE	30
CHAPTER 8 - A PROGRAM WITH A GUI	32
CHAPTER 9 - FINDING METHODS	46
CHAPTER 10 - AWAKEFROMNIB	49
CHAPTER 11 - POINTERS	51
CHAPTER 12 - STRINGS	53
CHAPTER 13 - ARRAYS	58
CHAPTER 14 - MEMORY MANAGEMENT	61
CHAPTER 15 - SOURCES OF INFORMATION	63

INTRODUCTION

Apple provides you with all the tools you need to create great Cocoa applications, for free. This set of tools, known under the name Xcode, comes with Mac OS X, or you can download it from the developer section on Apple's website.

Several good books on programming for the Mac exist, but they assume that you already have some programming experience. This book doesn't. It teaches you the basics of programming, in particular Objective-C programming, using Xcode. After some 5 chapters, you will be able to create a basic program without a Graphical User Interface (GUI). After a few more chapters, you will know how to create simple programs with a GUI. When you have finished this booklet, you will be ready for the above-mentioned more advanced books. You will have to study those too, because there is a **lot** to learn. For now though, don't worry because this book takes it easy.

How to use this book

As you will see, some paragraphs are displayed in a box like this:

Some tidbits

We suggest you read each chapter (at least) twice. The first time, skip the boxed sections. The second time you read the chapters, include the boxed text. You will in effect rehearse what you have learned, but learn some interesting tidbits which would have been distracting the first time. By using the book in this way, you will level the inevitable learning curve to a gentler slope.

This book contains dozens of examples, consisting of one or more lines of programming code. To make sure you associate an explanation to the proper example, every example is labeled by a number placed between square brackets, like this: [4]. Most examples have two or more lines of code. At times, a second number is used to refer to a particular line. For example, [4.3] refers to the third line of example [4]. In long code snippets, we put the reference after a line of code, like this:

```
volume = baseArea * height;           // [4.3]
```

Programming is not a simple job. For your part, it requires some perseverance and actually trying all the stuff taught in this book yourself. You cannot learn how to play the piano or drive a car solely by reading books. The same goes for learning how to program. This book is in an electronic format, so you do not have any excuse not to switch to Xcode frequently. Therefore, as of chapter 5, we suggest you go through each chapter three times. The second time, try the examples for real, and then make small modifications to the code to explore how things work.

CHAPTER 0

BEFORE WE START

We wrote this book for you. As it is free, please allow me to say a couple of words on promoting the Mac in return. Every Macintosh user can help to promote their favorite computer platform with little effort. Here is how.

1. The more efficient with your Mac you are, the easier it is to get other people to consider a Mac. So, try to stay up to date by visiting Mac-oriented websites and reading Mac magazines. Of course, learning Objective-C or AppleScript and putting those to use is great too. For businesses, the use of AppleScript can save tons of money and time. Check out my free booklet *AppleScript for Absolute Starters*, available from:

<http://www.macscripter.net/books>

2. Show the world that not everybody is using a PC by making the Macintosh more visible. Wearing a neat Mac T-shirt in public is one way, but there are even ways you can make the Mac more visible from within your home. If you run Activity Monitor (in the Utilities folder which you find in the Applications folder on your Mac), you will notice that your Mac uses its full processing power only occasionally.

Scientists have initiated several distributed computing (DC) projects, such as Folding@home or SETI@home, that harness this unused processing power, usually for the common good. You download a small, free program, called a DC client, and start processing work units. These DC clients run with the lowest level of priority. If you are using a program on your Mac and that program needs full processing power, the DC client immediately takes a back seat. So, you will not notice it is running. How does this help the Mac? Well, most DC projects keep rankings on their websites of work units processed. If you join a Mac team (you'll recognize their names in the rankings), you can help the Mac team of your choice to move up the rankings. So, users of other computer platforms will see how well Macs are doing. There are DC clients for many topics, such as math, curing diseases and more. To choose a DC project you like, check out:

<http://distributedcomputing.info/projects.html>

One problem with this suggestion: It may become addictive!

3. Make sure the Macintosh platform has the best software. No, not just by creating cool programs yourself. Make it a habit to give (polite) feedback to the developers of programs you use. Even if you tried a piece of software and didn't like it, tell the developer why. Report bugs by providing an accurate description as possible of the actions you performed when you experienced the bug.

4. Pay for the software you use. As long as the Macintosh software market is viable, developers will continue to provide great software.

5. Please contact at least 3 Macintosh users who could be interested in this programming, tell them about this book and where to find it. Or advise them about the above 4 points.

OK, while you download a DC client in the background, let's get started!

CHAPTER 1

A PROGRAM IS A SERIES OF INSTRUCTIONS

If you learn how to drive a car, you have to learn to handle several things in one go. You must know both about the clutch, and the gas and the brake pedals. Programming also requires you to keep a lot of things in mind, or your program will crash. While you were familiar with the interior of a car before you started how to learn to drive, you don't have that advantage when learning how to program using Xcode. In order not to overwhelm you, we leave the actual programming environment for a later chapter. First, we are going to make you comfortable with some Objective-C code, by starting with some basic math you are very familiar with.

In primary school you had to do calculations, filling in the dots:

$2 + 6 = \dots$
 $\dots = 3 * 4$ (the star $*$ is the standard way to represent multiplication on computer keyboards)

In secondary school, dots were out of fashion and **variables** called x and y (and a new fancy word, "algebra") were all the hype. Looking back, you may wonder why people felt so intimidated by this very small change in notation.

$2 + 6 = x$
 $y = 3 * 4$

Objective-C uses variables too. Variables are nothing more than convenient names to refer to a specific piece of data, such as a number. Here is an Objective-C **statement**, i.e. a line of code, where a variable is given a particular value.

```
[1]  
x = 4;
```

The variable named x is given a value of 4. You will note there is a semi-colon at the end of the statement. That is because the semi-colon is required at the end of every statement. Why? Well, the code snippet of example [1] may look geeky to you, but a computer does not know what to do with it at all. A special program, called a **compiler**, is necessary to convert the text you typed into the necessary zeros and ones your Mac understands. Reading and understanding the text a human typed is very hard for a compiler, so you need to give it certain clues, for example where a particular statement ends. Which is what you do by using the semi-colon.

If you forget a single semi-colon in your code, the code cannot be compiled, that is, it cannot be turned into a program your Mac can execute. Don't worry too much about that, because the compiler will complain if it can't compile your code. As we will see in a future chapter, it will try to help you find out what is wrong.

While variable names themselves have no special meaning to the compiler, descriptive variable names can make a program much easier to read and hence easier to understand. That is a big bonus if you need to track down an error in your code.

Errors in programs are traditionally called *bugs*. Finding and fixing them is called *debugging*.

Hence, in real code we avoid using non-descriptive variable names like *x*. For example, the variable name for the width of a picture could be called *pictureWidth* [2].

```
[2]
pictureWidth = 8;
```

From the big issue a compiler makes out of forgetting a semi-colon, you will understand that programming is all about details. One of those details to pay attention to is the fact that code is case-sensitive. That is, it matters whether you use capitals or not. The variable name *pictureWidth* is not the same as *pictureWIDTH*, or *PictureWidth*. In accordance with general conventions, I make my variable names up by fusing several words, the first without capital, and all other words making up the variable name starting with a capital, just as you can see in example [2]. By sticking to this scheme, I reduce the chance on programming mistakes due to case-sensitivity tremendously.

Please note that a variable name always consists of a single word (or single character, at a pinch).

While you have plenty freedom choosing variable names, there are several rules which a variable name has to conform with. While I could spell them all out, that would be boring at this point. The prime rule you must obey is that your variable name may not be a reserved word of Objective-C (i.e., a word that have a special meaning to Objective-C). By composing a variable name as contracted words, like *pictureWidth*, you are always safe. To keep the variable name readable, the use of capitals within the variable name is recommended. If you stick to this scheme, you'll have fewer bugs in your programs.

If you insist on learning a couple of rules, finish this paragraph. Apart from letters, the use of digits is allowed, but a variable name is not allowed to start with a digit. Also allowed is the underscore character: "_".

Here are a few examples of variable names.

Fine variable names: *door8k*, *do8or*, *do_or*

Not allowed: *door 8* (contains a space), *8door* (starts with digit)

Not recommended: *Door8* (starts with capital)

Now we know how to give a variable a value, we can perform calculations. Let's take a look at the code for the calculation of the surface area of a picture. Here is the code [3] that does just that.

```
[3]
pictureWidth=8;
pictureHeight=6;
pictureSurfaceArea=pictureWidth*pictureHeight;
```

Surprisingly, the compiler doesn't nitpick about spaces (except within variable names, keywords etc.). To make the code easier on the eyes, we can use spaces.

```
[4]
pictureWidth = 8;
pictureHeight = 6;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

Now, take a look at example [5], and in particular the first two statements.

```
[5]
pictureWidth = 8;
pictureHeight = 4.5;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

Numbers in general can be distinguished into two types: **integers** (whole numbers) and fractional numbers. You can see an example of each in the statements [5.1] and [5.2], respectively. Integers are used for counting, which is something we will do when we have to repeat a series of instructions a specified number of times (see chapter 7). You know fractional or **floating-point** numbers, for example, from baseball hitting averages.

The code of example [5] will not work. The problem is that the compiler wants you to tell it in advance what variable names you are going to use in your program, and what type of data they are referring to, i.e. integers or floating point numbers. In geek-speak, this is called "to declare a variable".

```
[6]
int pictureWidth;
float pictureHeight, pictureSurfaceArea;
pictureWidth = 8;
pictureHeight = 4.5;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

In line [6.1], **int** indicates that the variable *pictureWidth* is an integer. In the next line, we declare two variables in one go, by separating the variable names with a comma. More specifically, statement [6.2] says that both variables are of type **float**, i.e. numbers that contain fractional parts. In this case it is a bit silly that *pictureWidth* is of a different type than the other two variables. But what you can see is that if you multiply an int with a float, the result of the calculation is a float, which is why you should declare the variable *pictureSurfaceArea* as a float [6.2].

Why does the compiler want to know whether a variable represents an integer or a number with a fractional part? Well, a computer program needs part of the computer's memory. The compiler reserves memory (bytes) for each variable it encounters. Because different types of data, in this case int and float, require different amounts of memory and a different representation, the compiler needs to reserve the correct amount of memory and to use the correct representation.

What if we are working with very big numbers or very high precision decimal numbers? They wouldn't fit in the few bytes reserved by the compiler, would they? That's right. There are two answers to this: first, both int and float have counterparts that can store bigger numbers (or numbers with higher precision). On most systems they are *long long* and *double*, respectively. But even these can fill up, which bring us to the second answer: as a programmer, it will be your job to be on the watch for problems. In any case, it is not a problem to be discussed in the first chapter of an introductory book.

By the way, both integers and decimal numbers can be negative, as you know for example from your bank account. If you know that the value of a variable is never negative, you can stretch the range of values that fit in the bytes available.

```
[7]
unsigned int chocolateBarsInStock;
```

There is no such thing as a negative number of chocolate bars, so an unsigned int could be used here. The unsigned int type represents numbers greater than or equal to zero.

It is possible to declare a variable and assign it a value in one go [8].

```
[8]
int x = 10;
float y= 3.5, z = 42;
```

It does save you some typing.

In the previous examples, we performed a multiplication operation. Use the following symbols, officially known as **operators**, for doing basic mathematical calculations.

- + for addition
- for subtraction
- / for division
- * for multiplication

Using the operators, we can perform a wide range of calculations. If you take a look at the code of professional Objective-C programmers, you will come across a couple of peculiarities, probably because they're lazy typists.

Instead of writing `x = x + 1;` programmers often resort to something else like [9] or [10]

```
[9]
x++;
```

```
[10]
++x;
```

In either case this means: increase x by one. Under some circumstances it is important whether the ++ is before or after the variable name. Check out the following examples [11] and [12].

```
[11]
x = 10;
y = 2 * (x++);
```

```
[12]
x = 10;
y = 2 * (++x);
```

In example [11], when all is said and done, y equals 20 and x equals 11. In contrast, in statement [12.2], x is incremented by one before the multiplication by 2 takes place. So, in the end, x equals 11 and y equals 22. The code of example [12] is equivalent with example [13].

```
[13]
x = 10;
x++;
y = 2 * x;
```

So, the programmer has actually merged two statements into one. Personally, I think this makes a program harder to read. If you take the shortcut that is fine but be aware that a bug may be lurking there.

It will be old hat for you if you managed to pass high school, but parentheses can be used to determine the order in which operations are performed. Ordinarily `*` and `/` take precedence over `+` and `-`. So `2 * 3 + 4` equals 10. By using parenthesis, you can force the lowly addition to be performed first: `2 * (3 + 4)` equals 14.

The division operator deserves some special attention, because it makes quite a difference whether it is used with integers or floats. Take a look at the following examples [14, 15].

```
[14]
int x = 5, y = 12, ratio;
ratio = y / x;
```

```
[15]
float x = 5, y = 12, ratio;
ratio = y / x;
```

In the first case [14], the result is 2. Only in the second case [15], the result is what you'd probably expect: 2.4.

An operator you're probably unfamiliar with is `%`. The result of the `%` operator is the remainder from the integer division of the first operand by the second (if the value of the second operand is zero, the behavior of `%` is undefined).

```
[16]
int x = 13, y = 5, remainder;
remainder = x % y;
```

Now the result is that *remainder* is equal to 3, because `x` is equal to `2*y + 3`.

Here are a few more example:

21 % 7 is equal to 0	30 % 2 is equal to 0	50 % 9 is equal to 5
22 % 7 is equal to 1	31 % 2 is equal to 1	60 % 29 is equal to 2
23 % 7 is equal to 2	32 % 2 is equal to 0	
24 % 7 is equal to 3	33 % 2 is equal to 1	
27 % 7 is equal to 6	34 % 2 is equal to 0	

It can come in handy at times, but note that it only works with integers.

CHAPTER 2

NO COMMENT? UNACCEPTABLE!

By using sensible variable names, we can make our code more readable and understandable [1].

```
[1]
float pictureWidth, pictureHeight, pictureSurfaceArea;
pictureWidth = 8.0;
pictureHeight = 4.5;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

While we will currently stick to examples only a couple of statements long, real code you may write will get longer and longer. Of prime importance when writing a program is not just worrying that it actually does what you want it to do, but also that it is properly documented. Later, if you haven't seen the code for a while and want to change it, you really need comments to help you understand what a particular part of your code does and why that part is there in the first place. You are strongly advised to take some time commenting your code. We can assure you that you will gain back manifold the time spent in the future. Also, if you share your code with someone else, that person will be able to adapt it to his/her own needs quickly if you have provided comments. To create a comment, start the comment with two forward slashes.

```
// This is a comment
```

In Xcode comments are shown in green. If a comment is long, and spans multiple lines, put it between `/*` and `*/`

```
/* This is a comment
   extending over two lines */
```

We will discuss debugging a program shortly, as Xcode has great facilities for that. One way to debug the old-fashioned way is called outcommenting. By placing part of your code between `/* */`, you can temporarily disable ("outcomment") that part of the code, to see if the rest works as expected. This allows you to hunt down a bug. If the outcommented part should result in, for example, a value for a particular variable, you can include a temporary line where you set the variable to a value suitable for testing the remainder of your code.

The importance of comments cannot be overstated. It is often useful to add an explanation in plain English about what goes on in a long series of statements. That is because you don't have to deduce what the code does, and you can immediately see if the problem you are experiencing is in that part of the code. You should also use comments to express things that are difficult, or impossible to deduce from the code. For instance, if you program a mathematical function using a specific model described in details somewhere in a book, you would put a bibliographical reference in a comment associated with your actual code. Sometimes it is useful to write some comments before writing the actual code. It will help you to structure your thoughts and programming will be easier as a result.

The code examples in this book do not contain as many comments as we would ordinarily have written in, because they are already surrounded by explanations.

CHAPTER 3

FUNCTIONS

The longest piece of code that we have seen so far had only five statements. Programs of many thousands of lines may seem a long way off, but because of the nature of Objective-C, we have to discuss the way programs are organized at an early stage.

If a program were to consist of a long, continuous succession of statements, it would be hard to find and fix a bug. Beside, a particular series of statements may appear in your program in several places. If there is a bug there, you must fix the same bug at several places. A nightmare, because it is easy to forget one (or two)! So, people have thought of a way to organize the code, making it easier to fix bugs.

The solution to this problem is to group the statements depending on their function. For example, you may have a set of statements that allows you to calculate the surface area of a circle. Once you've checked that this set of statements works reliably, you will never have to go through that code again to see if the bug is there. The set of statements, called a **function**, has a name, and you can call that set of statements by this name to have its code executed. This concept of using functions is so fundamental, that there is always at least one function in a program: the **main()** function. This main() function is what the compiler looks for, so it will know where execution of the code at runtime must start.

Let's take a look at the main() function in more detail. [1]

```
[1]
main()
{
    // Body of the main() function. Put your code here.
}
```

Statement [1.1] shows the name of the function, i.e. "main", followed by opening and closing parentheses. While "main" is a reserved word, and the main() function is required to be present, when you define your own functions, you can call them just about anything you like. The parentheses are there for a good reason, but we won't discuss that until later in this chapter. In the following lines [1.2,1.5], there are curly braces. We must put our code between those curly braces. Anything between the curly braces is called the body of the function. I took some code from the first chapter and put it where it belongs [2].

```
[2]
main()
{
    // Variables are declared below
    float pictureWidth, pictureHeight, pictureSurfaceArea;

    // We initialize the variables (we give the variables a value)
    pictureWidth = 8.0;
    pictureHeight = 4.5;

    // Here the actual calculation is performed
    pictureSurfaceArea = pictureWidth * pictureHeight;
}
```

If we were to continue to add code to the body of the main() function, we would end up with the difficult to debug, unstructured code we said we wanted to avoid. Let's write

another program, now with some structure. Apart from the obligatory `main()` function, we will create a `circleArea()` function [3].

```
[3]
main()
{
    float pictureWidth, pictureHeight, pictureSurfaceArea;
    pictureWidth = 8.0;
    pictureHeight = 4.5;
    pictureSurfaceArea = pictureWidth * pictureHeight;
}

circleArea()    // [3.9]
{
}

```

That was easy, but our custom function starting at statement [3.9] doesn't do anything yet. Note that the function specification is outside the body of the `main()` function. In other words, functions are not nested.

Our new `circleArea()` function must be called from the `main()` function. Let's see how we can do that [4].

```
[4]
main()
{
    float pictureWidth, pictureHeight, pictureSurfaceArea,
          circleRadius, circleSurfaceArea;    // [4.4]
    pictureWidth = 8.0;
    pictureHeight = 4.5;
    circleRadius = 5.0;    // [4.7]
    pictureSurfaceArea = pictureWidth * pictureHeight;

    // Here we call our function!
    circleSurfaceArea = circleArea(circleRadius);    // [4.11]
}

```

Note: the remainder of the program is not shown (see [3]).

We added a pair of variable names of type `float` [4.4], and we initialized the variable `circleRadius`, i.e. gave it a value [4.7]. Of most interest is line [4.11], where the `circleArea()` function is called. As you can see, the name of the variable `circleRadius` has been put between the parentheses. It is an **argument** of the `circleArea()` function. The value of the variable `circleRadius` is going to be passed to the function `circleArea()`. When the function `circleArea()` has done its job of performing the actual calculation, it must return the result. Let's modify the `circleArea()` function of [3] to reflect this [5].

Note: only the `circleArea()` function is shown.

```
[5]
circleArea(float theRadius)    // [5.1]
{
    float theArea;
    theArea = 3.1416 * theRadius * theRadius;    // pi times r square    [5.4]
    return theArea;
}

```

In [5.1] we define that for the `circleArea()` function a value of type `float` is required as input. When received, this value is stored in a variable named *theRadius*. We use a second variable, i.e. *theArea* to store the result of the calculation [5.4] in, so we must declare it [5.3], in the same way we declared variables in the main function [4.4]. You will note that the declaration of the variable *theRadius* is done within the parentheses [5.1]. Line [5.5] returns the result to the part of the program from which the function was called. As a result, in line [4.11], the variable *circleSurfaceArea* is set to this value.

The function in example [5] is complete, except for one thing. We have not specified the type of data that the function will return. The compiler requires us to do that, so we have no choice but to obey and indicate it is of type `float` [6.1].

```
[6]
float circleArea(float theRadius)
{
    float theArea;
    theArea = 3.1416 * theRadius * theRadius;
    return theArea;
}
```

As the first word of line [6.1] indicates, the data returned by this function (i.e., the value of variable *theArea*) is of type `float`. As a programmer, you will have ensured that the variable *circleSurfaceArea* in the `main()` function [4.8] is of that data type too, so the compiler has no reason to nag us on this one.

Not all functions require an argument. If there is none, the parentheses are still required, even though they are empty

```
[7]
int throwDice()
{
    int noOfEyes;

    // Code to generate a random value from 1 to 6

    return noOfEyes;
}
```

Not all functions return a value. If a function does not return a value, it is of type “void”. The “return” statement is then optional. If you use it, the return keyword must not be followed by a value/variable name.

```
[8]
void beepXTimes(int x);
{
    // Code to beep x times.
    return;
}
```

If a function has more than one argument, like the `pictureSurfaceArea()` function below, the arguments are separated by a comma.

```
[9]
float pictureSurfaceArea(float theWidth, float theHeight)
{
    // Code here
}
```

The `main()` function should, by convention, return an integer, and so yes, it does have a return statement too. It should return 0 (zero, [10.9]), to indicate that the function was executed without problems. As the `main()` function returns an integer, we must write "int" before `main()` [10.1]. Let's put all the code we have in one list.

```
[10]
int main()
{
    float pictureWidth, pictureHeight, pictureSurfaceArea,
        circleRadius, circleSurfaceArea;
    pictureWidth = 8;
    pictureHeight = 4.5;
    circleRadius = 5.0;
    pictureSurfaceArea = pictureWidth * pictureHeight;
    circleSurfaceArea = circleArea(circleRadius);    // [10.9]
    return 0;
}

float circleArea(float theRadius)                // [10.13]
{
    float theArea;
    theArea = 3.1416 * theRadius * theRadius;
    return theArea;
}
```

As you can see [10], we have a `main()` function [10.1] and another function we defined ourselves [10.13]. If we were to compile this code, the compiler would still balk. In line [10.9] it would claim not to know any such function named `circleArea()`. Why? Apparently, the compiler starts reading the `main()` function and suddenly it encounters something it doesn't know. It doesn't look any further and gives you a warning. To satisfy the compiler, just add a **function declaration** before the statement containing `int main()` [11.1]. There is nothing hard about it, as it is the same as line [10.13], except that it ends with a semi-colon.

```
[11]
float circleArea(float theRadius); // function declaration

int main()
{
```

Note: the remainder of the program is not shown (see [10]).

We will soon compile this program for real. First a couple of odds and ends.

When writing programs, it is advisable to keep future reuse of code in mind. Our program could have a `rectangleArea()` function, as shown below [12], and this function could be called in our `main()` function. This is useful even if the code we put in a function is used only once. The `main()` function becomes easier to read. If you have to debug your code, it will be easier to find where the bug might be in your program. You might find that it is in a function. Instead of having to go through a long sequence of statements, you just have to check the statements of the function, which are easy to find, thanks to the opening and closing curly braces.

```
[12]
float rectangleArea(float length, float width)
{
    return (length * width);
}
```

As you can see, in a simple case like this, it is possible to have a single statement [12.3] for both the calculation and returning the result. I used the superfluous variable *theArea* in [10.15] just to show you how to declare the variable in a function.

While the functions we defined ourselves in this chapter are rather trivial, it is important to realize that you can modify a function without impact on the code that calls the function as long as you do not change the declaration of the function (i.e., its first line).

For example, you can change the variable names in a function, and the function still works (and this will not disrupt the rest of the program as well). Someone else could write the function, and you could use it without knowing what goes on inside the function. All you need to know is how to use the function. That means knowing:

- the function's name
- the number, order and type of the function's arguments
- what the function returns (the value of the surface area of the rectangle), and the type of the result

In the example [12], these answers are, respectively:

- `rectangleArea`
- Two arguments, both floats, where the first represents the length, the second the width.
- The function returns something, and the result is of type float (as can be learned from the first word of statement [12.1]).

The code inside the function is shielded from the main program, and from other functions, for that matter. This is a most essential feature of Objective-C. In Chapter 5, we will discuss this behavior again. But first, we are going to start with Xcode and run the above program [11].

CHAPTER 4

PRINTING ON SCREEN

We made good progress with our program, but we did not discuss how to display the results of our calculations. There are various options for displaying a result on screen. In this book, we'll use a function provided by Cocoa: the `NSLog()` function. That is nice, because now you don't have to worry (nor have to program anything) to get your results "printed" on screen.

The `NSLog()` function is primarily designed to display error messages, not to output application results. However it is so easy to use that we adopt it in this book to display our results. Once you have some mastery of Cocoa, you'll be able to use more sophisticated techniques.

Let's look into how `NSLog()` function is used.

```
[1]
int main()
{
    NSLog(@"Julia is a pretty actress.");
    return 0;
}
```

Upon execution, the statement of example [1] would result in the text "Julia is a pretty actress." being displayed. Such a text between `@` and `"` is called a string.

In addition to the string itself, the `NSLog()` function prints various additional information, like the current date and the name of the application. For example, the complete output of the program [1] on my system is:

```
2005-12-22 17:39:23.084 test[399] Julia is a pretty actress.
```

A string can have a length of zero or more characters.

Note: In the following examples only the interesting statements of the `main()` function are shown.

```
[2]
NSLog(@"");
NSLog(@" ");
```

Statement [2.1] contains zero characters and is called an empty string (i.e., it has a length equal to zero). Statement [2.2] is not an empty string, despite how it looks. It contains a single space, so the length of that string is 1.

Several special character sequences have a special meaning in a string. For instance, to force the last word of our sentence to begin printing on a new line, a special code must be included in statement [3.1]. This code is `\n`, short for a new line character.

```
[3]
NSLog(@"Julia is a pretty \nactress.");
```

Now the output looks like this (only the relevant output is shown):

```
Julia is a pretty
actress.
```

The backslash in [3.1] is called an **escape** character, as it indicates to the NSLog() function that the next character is not an ordinary character to be printed to the screen, but a character that has a special meaning: in this case the "n" means "start a new line".

In the rare event that you want to print a backslash to the screen, it may seem you have a problem. If a character after a backslash has a special meaning, how is it possible to print a backslash? Well, we just put another backslash before (or indeed after) the backslash. This tells the NSLog() function that the (second) backslash, i.e. the one more to the right, is to be printed and that any special meaning should be ignored). Here is an example:

```
[4]
NSLog(@"Julia is a pretty actress.\n");
```

Statement [4.1] would result, upon execution, in

```
Julia is a pretty actress.\n
```

So far, we have displayed static strings only. Let's print the value obtained from a calculation to the screen.

```
[5]
int x, integerToDisplay;
x = 1;
integerToDisplay = 5 + x;
NSLog(@"The value of the integer is %d.", integerToDisplay);
```

Please note that, between parentheses, we have a string, a comma and a variable name. The string contains something funny: %d. Like the backslash, the percentage character % has a special meaning. If followed by a d (short for decimal number), upon execution, at the position of %d the output value of what is after the comma, i.e. the value of the variable *integerToDisplay*, will be inserted. Running example [5] results in

```
The value of the integer is 6.
```

To display a float, you have to use %f instead of %d.

```
[6]
float x, floatToDisplay;
x = 12345.09876;
floatToDisplay = x/3.1416;
NSLog(@"The value of the float is %f.", floatToDisplay);
```

It is up to you how many significant digits (the ones after the period) are displayed. To display two significant digits, you put .2 between % and f, like this:

```
[7]
float x, floatToDisplay;
x = 12345.09876;
floatToDisplay = x/3.1416;
NSLog(@"The value of the float is %.2f.", floatToDisplay);
```

Later, when you know how to repeat calculations, you may want to create a table of values. Imagine a conversion table of Fahrenheit to Celsius. If you want to display the values nicely, you want the values in the two columns of data to have a fixed width. You can specify this width with an integer value between % and f (or % and d, for that matter). However, if the width you specify is less than the width of the number, the width of the number takes prevalence.

```
[8]
int x = 123456;
NSLog(@"%2d", x);
NSLog(@"%4d", x);
NSLog(@"%6d", x);
NSLog(@"%8d", x);
```

Example [8] has the following output:

```
123456
123456
123456
    123456
```

In the first two statements [8.1, 8.2] we actually claim too little space for the number to be displayed in full, but the space is taken anyway. Only statement [8.4] specifies a width wider than the value, so now we see the appearance of additional spaces, indicative of the width of the space reserved for the number.

It is also possible to combine the specification of width and the number of decimal numbers to be displayed.

```
[9]
float x=1234.5678
NSLog(@"Reserve a space of 10, and show 2 significant digits.");
NSLog(@"%10.2d", x);
```

Of course, it is possible to display more than one value, or any mix of values [10.3]. You do have to make sure that you properly indicate the type (int, float), using %d and %f.

```
[10]
int x = 8;
float pi = 3.1416;
NSLog(@"The integer value is %d, whereas the float value is %f.", x, pi);
```

It is important that you use the right symbol for the variable type. If you fool up the first one, the second one may not be displayed correctly either! For example,

```
[10b]
int x = 8;
float pi = 3.1416;
NSLog(@"The integer value is %f, whereas the float value is %f.", x, pi);
```

gave the following output:

```
The integer value is 0.000000, whereas the float value is 0.000000.
```

We are only one question and one answer away from executing our first program. How does our program know about this function NSLog()? Well, it doesn't unless we tell it to.

To do that, our program has to tell the compiler to import a library of goodies, including the function `NSLog()`, using the statement:

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

This statement must be the first statement of our program. When we put together all that we have learned in this chapter, we get the following code, which we are going to run in the next chapter.

```
[11]
#import <Foundation/Foundation.h>
float circleArea(float theRadius);
float rectangleArea(float width, float height);

int main()
{
    float pictureWidth, pictureHeight, pictureSurfaceArea,
        circleRadius, circleSurfaceArea;
    pictureWidth = 8.0;
    pictureHeight = 4.5;
    circleRadius = 5.0;
    pictureSurfaceArea = rectangleArea(pictureWidth, pictureHeight);
    circleSurfaceArea = circleArea(circleRadius);
    NSLog(@"Area of circle: %10.2f.", circleSurfaceArea);
    NSLog(@"Area of picture: %f. ", pictureSurfaceArea);
    return 0;
}

float circleArea(float theRadius) // first custom function
{
    float theArea;
    theArea = 3.1416 * theRadius * theRadius;
    return theArea;
}

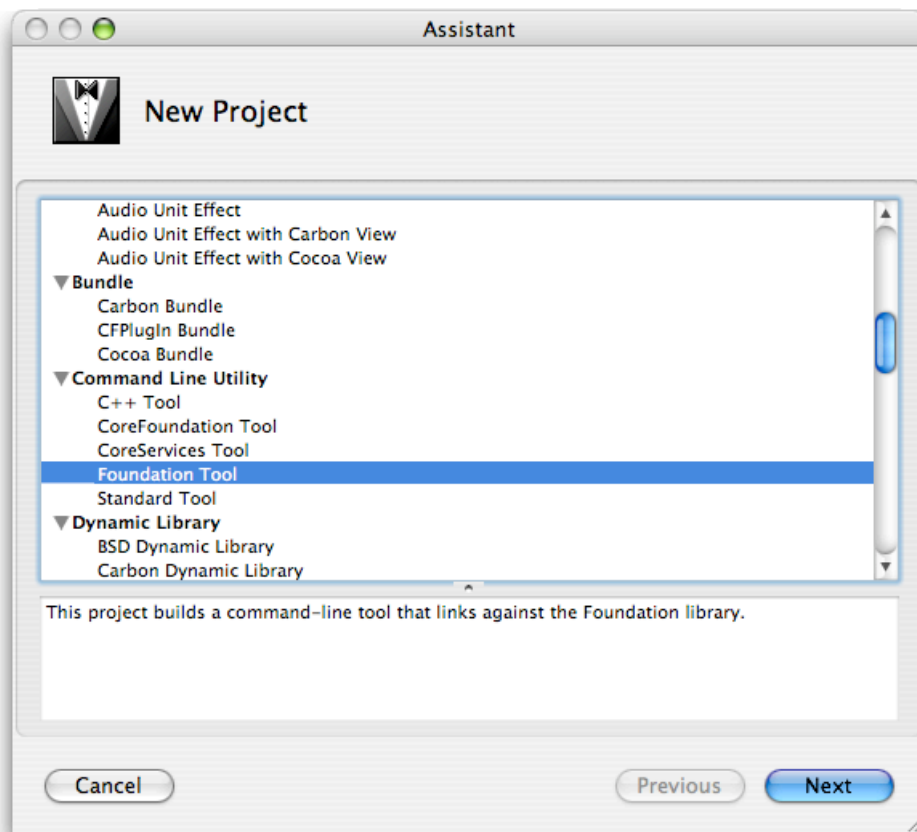
float rectangleArea(float width, float height) // second custom function
{
    return width*height;
}
```

CHAPTER 5

COMPILING AND RUNNING A PROGRAM

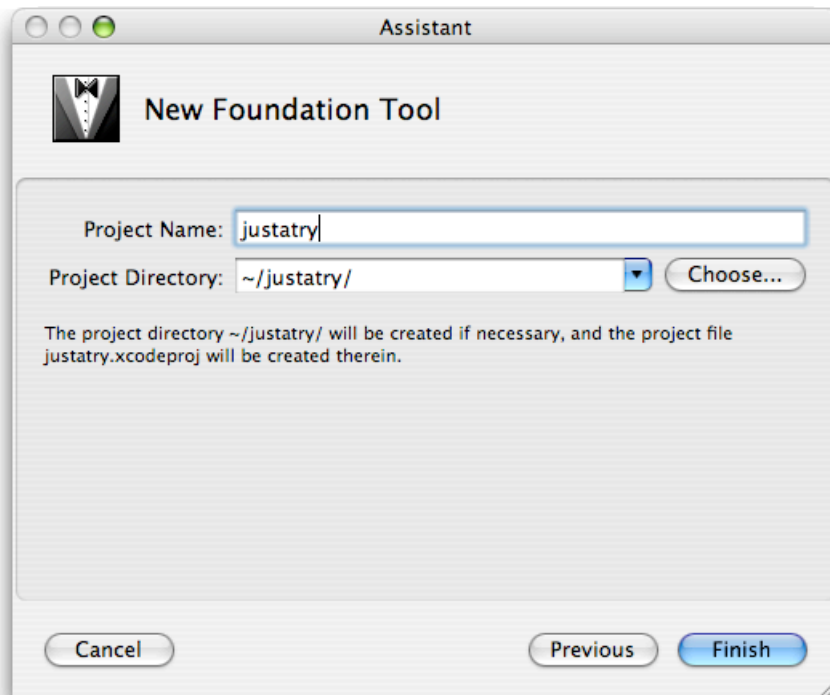
The code we have produced so far is nothing more than a lot of text we human beings can read. Although it is not exactly prose to us, it is even worse for your Mac. It can't do anything with it at all! A special program, called a compiler, is necessary to convert your programming code into runtime code that can be executed by your Mac. It is part of Apple's free Xcode programming environment. You should have installed Xcode using the disk that came with your copy of Mac OS X. In any case, verify that you have the latest version, which you can download from the developer section at <http://developer.apple.com> (free registration required).

Now, start Xcode, which you find in the Applications folder of the Developer folder. When you do that for the first time, it will ask you a couple of questions. Agree with the default suggestions, they are fine, and you can always change them in the Preferences later, should you want to. To really get started, select New Project from the File menu. A dialog window appears containing a list of possible project types.



The Xcode assistant lets you create new projects.

We want to create a very simple program in Objective-C, without a GUI (Graphical User Interface), so scroll down and select *Foundation Tool* under the *Command Line Utility* section.



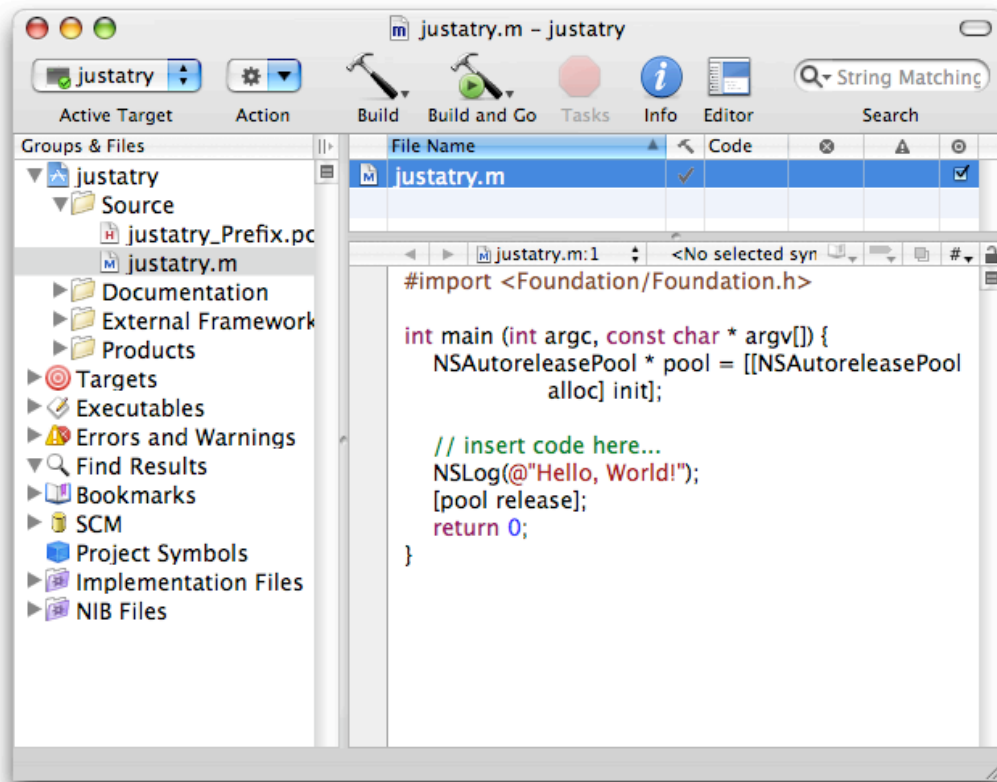
Setting the name and location of the new project.

Enter a name for your application, such as *justatry*. Choose a location where you want to save your project, and click *Finish*.

The project we are about to create can be run from the Terminal. If you want to be able to do that, and want to avoid some hassle, make sure the name of your project is just one word. Also, it is customary not to start the names of programs run from the Terminal with a capital letter. On the other hand, names of programs with a graphical user interface should start with a capital.

Now you are presented with a window that you as a programmer will see a lot. The window has two frames. At the left is the "Groups & Files" frame for accessing all the files that your program is made up of. Currently there aren't too many, but later when you are creating multilingual GUI programs, this is where the files for your GUI and for the various languages can be found. The files are grouped and kept within folders, but you will search for these folders on your hard disk in vain. Xcode offers these virtual folders ("Groups") for the purpose of organizing your stuff.

In the frame at the left named *Groups & Files*, open the group *justatry* to go to the group that reads *Source*. In it is a file named *justatry.m* [1]. Remember that every program must contain a function named *main()*? Well, this is the file that contains this *main()* function. Later in this chapter we are going to modify it to include the code of our program. If you open *justatry.m* by double-clicking its icon, you are in for a pleasant surprise. Apple has already created the *main()* function for you.



Xcode displaying the main() function.

```
[1]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[]) // [1.3]
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init]; // [1.5]

    // insert code here...
    NSLog(@"Hello, World!");
    [pool release]; // [1.9]
    return 0;
}
```

Take a look at the program and look for things you recognize. You will see:

- The import statement required for functions such as NSLog(), starting with a pound sign.
- The main() function.
- The curly braces which are to contain the body of our program.
- A comment, which invites us to put our code there.
- An NSLog statement for printing a string to the screen.
- The return 0; statement.

There are also a couple of things you will not recognize

- Two funny-looking arguments between the parentheses of the main() function [1.3]
- A statement starting with NSAutoreleasePool [1.5]
- Another statement containing the words pool and release [1.9]

Personally I'm not exactly happy when book authors present me, the reader, with code full of unfamiliar statements and promises that it will all become clear later. Yeah, sure. That is why I went out of my way to familiarize you with the concept of "functions" so you wouldn't be confronted with too many new concepts. You already do know that functions are a way to organize a program, that every program has a `main()` function, and what a function looks like. However, I have to admit that I can't fully explain everything you see in example [1] right now. I'm really sorry that I have to ask you to ignore these statements (i.e., [1.3, 1.5 and 1.9]) for the time being. There are other things about the Objective-C language that you need to become familiar with first, allowing you to write simple programs. The good thing is, that you have already made it past two difficult chapters, and the upcoming three chapters are pretty easy before we have to deal with some harder stuff again.

If you really don't want to be left without any explanation, here is the executive summary. The arguments in the `main` function are required for running the program from the Terminal. Your program takes up memory. Memory that other programs would like to use when you're done with it. As a programmer, it is your job to reserve the memory that you need. Of equal importance, you have to give the memory back when you're done. This is what the two statements with the word "pool" in them are for

Let's run the program provided by Apple [1]. Press the second hammer icon labeled *Build and Go* to build (compile) and run the application.



The *Build and Go* button.

The program is executed and the results are printed in the *Run Log* window, together with some additional information. The last sentence says that the program has exited (stopped) with return 0. There you see the value of zero that is returned by the `main()` function, as discussed in Chapter 3 [7.9]. So, our program made it to the last line and didn't stop prematurely. So far so good!


Let's go back to example [1] and see what happens if there is a bug in the program. For example, I've replaced the `NSLog` statement with another one, but I "forgot" the semi-colon indicating the end of the statement.

```
[2]
#import <Foundation/Foundation.h>

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

    // insert code here...
    NSLog(@"Julia is a pretty actress")    //Whoops, forgot the semicolon!
    [pool release]; //[2.9]
    return 0;
}
```

To build the application, press the build icon in the tool bar. A red circle appears before statement [2.9].



```
// insert code here...
NSLog(@"Julia is a pretty actress")
[pool release];
return 0;
}
```

Xcode signals a compilation error.

If you click it, the line below the toolbar shows a brief description of the complaint:

error: parse error before "release".

Parsing is one of the first things a compiler does: It walks through the code and checks whether it can understand each and every line. To help it understand the meaning of various parts, it's up to you to provide clues. So, for the import statement [2.1], you have to provide a pound sign (#). To indicate the end of a statement [2.8], you have to provide a semi-colon. By the time the compiler is at line [2.9], it notices something is wrong. However, it doesn't realize that the problem occurred not in this line, but in the previous line where the semi-colon is missing. The important lesson here is that, while the compiler tries to give sensible feedback, that feedback is not necessarily an accurate description of the actual problem, nor is the position in the program necessarily the actual position of the error (although it will probably very close).

Fix the program by adding the semi-colon and run the program again to make sure it works fine.

Now let's take the code from the last chapter, and weave it into the code Apple provided [1], resulting in example [3].

```
[3]
#import <Foundation/Foundation.h>

float circleArea(float theRadius);    // [3.3]

int main (int argc, const char * argv[])    // [3.5]
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int pictureWidth;
    float pictureHeight, pictureSurfaceArea,
          circleRadius, circleSurfaceArea;
    pictureWidth = 8;
    pictureHeight = 4.5;
    circleRadius = 5.0;
    pictureSurfaceArea = pictureWidth * pictureHeight;
    circleSurfaceArea = circleArea(circleRadius);
    NSLog(@"Area of picture: %f. Area of circle: %10.2f.",
          pictureSurfaceArea, circleSurfaceArea);
    [pool release];
    return 0;
}

float circleArea(float theRadius)        // [3.22]
{
    float theArea;
    theArea = 3.1416 * theRadius * theRadius;
    return theArea;
}

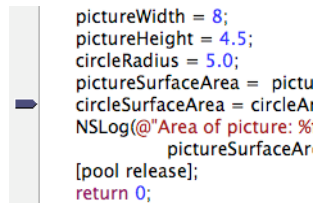
float rectangleArea(float width, float height)    // [3.29]
{
    return width*height;
}
```

Take your time to make sure you understand the structure of the program. We have the function headers [3.3, 3.4] of our custom functions `circleArea()` [3.22] and `rectangleArea()` [3.29] before the `main()` function [3.5], as they should be. Our custom functions are outside the curly braces of the `main()` function [3.5]. We put the code of the body of the `main()` function where Apple has told us to put it.

When the code is executed, we get the following output:

```
Area of picture: 36.000000. Area of circle:      78.54.
justatry has exited with status 0.
```

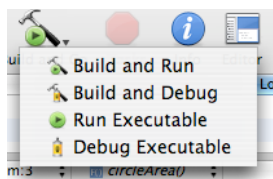
When a program gets more complicated, it gets harder to debug. So sometimes you want to find out what is going on inside the program while it is running. Xcode makes this easy to do. Just click in the grey margin before the statements for which you want to know the values of the variables. Xcode will insert a "breakpoint" represented by a blue grey arrow icon.



Setting a breakpoint in your code

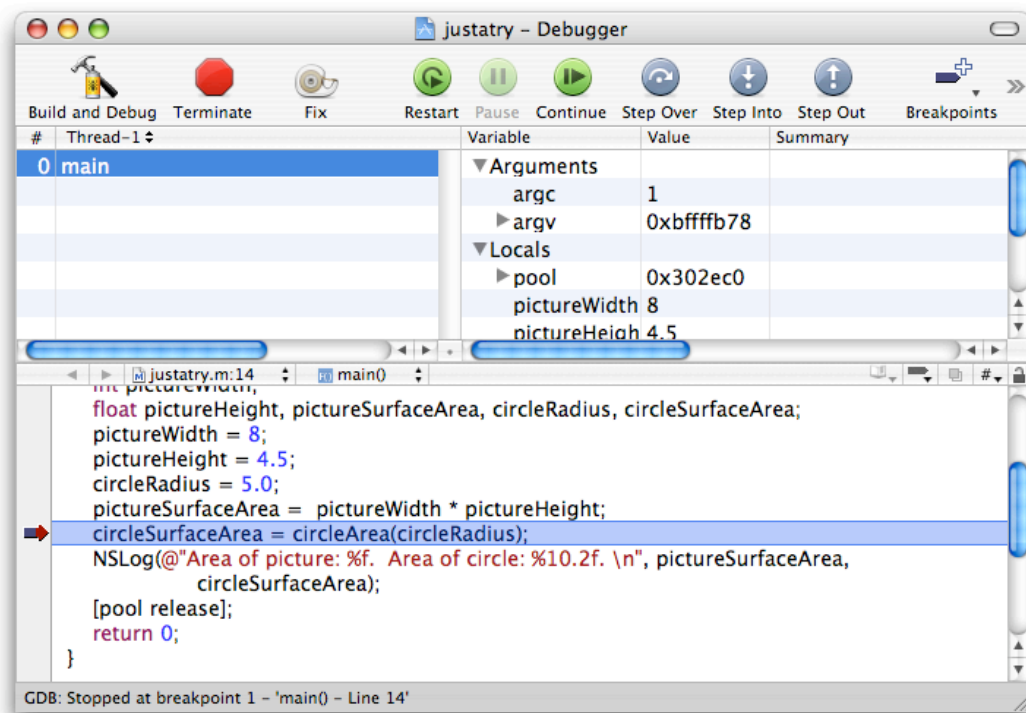
Please note that you will see the values of the variables before that particular statement is executed, so often you'll need to put the breakpoint at the statement after the one you are interested in.

Now, keep the mouse down while clicking the second hammer button in the toolbar, and a menu will pop-up.



The *Build and Go* popup menu.

Select *Build and Debug*. You will see the following window.



The Xcode debugger lets you execute the program step by step and look at variables.

The program will run until it reaches the first breakpoint. If you check the top right pane, you will be able to see the values of the various variables. Any values set or changed since the last breakpoint are displayed in red. To continue executing, use the Continue button. The debugger is a powerful tool. Play with it for a while to familiarize with it.

We have now all that is needed to write, debug and run simple programs for Mac OS X.

If you do not wish to make Graphical User Interface programs, the only thing you have to do now is increase your knowledge of Objective-C to enable you to develop more sophisticated non-graphical programs. In the next few chapters we're going to do exactly that. After that, we will dive into GUI-based applications. Read on!

CHAPTER 6

CONDITIONAL STATEMENTS

At times, you will want your code to perform a series of actions only if a particular condition is met. Special keywords are provided to achieve this [1.2].

```
[1]
int age = 42;
if (age > 30)    // The > symbol means "greater than"
{
    NSLog(@"age is older than thirty."); // [1.4]
}
NSLog(@"Finished."); // [1.6]
```

Line [1.2] shows the *if...* instruction, also known as a conditional instruction. You will recognize the curly braces, which will contain all the code you want to execute provided the logical expression between parentheses evaluates to true. Here, if the condition *age > 30* is met then the string [1.4] will be printed. Whether the condition is met or not, the string of line [1.6] will be printed, because it is outside the curly braces of the *if* clause.

We may also provide an alternative set of instructions if the condition is not met, using an *if...else* statement [2].

```
[2]
int age = 42;
if (age > 30)
{
    NSLog(@"age is older than thirty."); // [2.4]
}
else
{
    NSLog(@"age is not older thirty."); // [2.7]
}
NSLog(@"Finished."); // [1.6]
```

The string in statement [2.7] would only be printed if the condition were not met, which is not the case here [2].

Apart from the greater than sign in statement [2.2], the following comparison operators for numbers are at your disposal.

==	equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
!=	not equal to

Take particular note of the equality operator – two equals signs. It is all too easy to forget this and use merely one equal sign. Unfortunately that is the assignment operator, and would set the variable to a particular value. This is a common cause of confusion, and buggy code, for beginners. Now say it out loud: I will not forget to use two equal signs when testing for equality!

Comparison operators are quite useful when you want to repeat a series of statements several times. That will be the topic of the next chapter. First, we will discuss some other aspects of if statements that may come in handy.

Let's take a closer look at performing a comparison. A comparison operation results in one of only two possible outcomes: The result is either true or false.

In Objective-C, true and false are represented as either 1 or 0 respectively. There even is a special data type, named `BOOL` that you can use to represent such values. To denote the "true" value, you can write either 1 or `YES`. To denote the "false" value, you can write either 0 or `NO`.

```
[3]
int x = 3;
BOOL y;
y = (x == 4); // y will be 0.
```

It is possible to check for more conditions. If more than one condition must be met, use a logical AND, represented by two ampersands: `&&`. If at least one of the conditions must be met, use a logical OR represented by two pipes: `||`.

```
[4]
if ( (age >= 18) && (age < 65) )
{
    NSLog(@"Probably has to work for a living.");
}
```

It is also possible to nest conditional statements. This is simply a matter of putting one conditional statement inside the curly brackets of another conditional statement. First the outermost condition will be evaluated, then, if it is met, the next statement inside, and so on:

```
[5]
if (age >= 18)
{
    if (age < 65)
    {
        NSLog(@"Probably has to work for a living.");
    }
}
```

CHAPTER 7

REPEATING STATEMENTS FOR A WHILE

In all the code we have discussed so far, each statement was executed just once. We could always repeat code in functions by calling them repeatedly [1].

```
[1]
NSLog(@"Julia is a pretty actress.");
NSLog(@"Julia is a pretty actress.");
NSLog(@"Julia is a pretty actress.");
```

But even then, that would require repeating the call. At times, you will need to execute a one or more of statements several times. Like all programming languages, Objective-C offers several ways to achieve that.

If you know the number of times the statement (or group of statements) has to be repeated, you may specify that by including that number in the **for** statement of example [2]. The number must be an integer, because you cannot repeat an operation, say, 2.7 times.

```
[2]
int x;
for (x = 1; x <= 10; x++)
{
    NSLog(@"Julia is a pretty actress.");
}
NSLog(@"The value of x is %d", x);
```

In example [2], the string [1.4] would be printed 10 times. First, *x* is assigned the value of one. The computer then evaluates the condition with the formula we have put in place: *x* <= 10. This condition is met (since *x* is equal to 1), so the statement(s) between the curly braces are performed. Then, the value of *x* is increased, here by one, due to the expression *x++*. Subsequently, the resulting value of *x*, now 2, is compared with 10. As it is still smaller than and not equal to 10, the statements between the curly braces are executed again. As soon as *x* is 11, the condition *x* <= 10 is no longer met. The last statement [2.6] was included to prove to you that *x* is 11, not 10, after the loop has finished.

At times, you will need to make larger steps than just a simple increment using *x++*. All you need to do is substitute the expression you need. The following example [2] converts degrees Fahrenheit to degrees Celsius.

```
[3]
float celsius, tempInFahrenheit;
for (tempInFahrenheit = 0; tempInFahrenheit <= 200; tempInFahrenheit =
tempInFahrenheit + 20)
{
    celsius = (tempInFahrenheit - 32.0) * 5.0 / 9.0;
    NSLog(@"%10.2f -> %10.2f", tempInFahrenheit, celsius);
}
```

The output of this program is:

```

0.00 ->    -17.78
20.00 ->    -6.67
40.00 ->     4.44
60.00 ->    15.56
80.00 ->    26.67
100.00 ->   37.78
120.00 ->   48.89
140.00 ->   60.00
160.00 ->   71.11
180.00 ->   82.22
200.00 ->   93.33

```

Objective-C has two other ways to repeat a set of statements:

```
while () { }
```

and

```
do {} while ()
```

The former is basically identical to the for-loop we discussed above. It starts by performing a condition evaluation. If the result of that evaluation is false, the statements of the loop are not executed.

```

[4]
int counter = 1;
while (counter <= 10)
{
    NSLog(@"Julia is a pretty actress.\n");
    counter = counter + 1;
}
NSLog(@"The value of counter is %d", counter);

```

In this case, the value of counter is 11, should you need it later in your program!

With the `do {} while ()` instruction, the statements between the curly braces are executed at least once.

```

[5]
int counter = 1;
do
{
    NSLog(@"Julia is a pretty actress.\n");
    counter = counter + 1;
}
while (counter <= 10);

NSLog(@"The value of counter is %d", counter);

```

The counter's value at the end is 11.

You have gained some more programming skills, so now to tackle a harder subject. In the next chapter, we are going to build our first program with a Graphical User Interface (GUI).

CHAPTER 8

A PROGRAM WITH A GUI

Having increased our knowledge of Objective-C, we are ready to discuss how to create a program with a Graphical User Interface (GUI). I have to confess something here. Objective-C is actually an extension of a programming language called C. Most of what we have discussed so far is just plain C. So how does Objective-C differ from plain C? It is in the "Objective" part. Objective-C deals with abstract notions known as objects.

Up until now we have mainly dealt with numbers. As you learned, Objective-C natively supports the concept of numbers. That is, it let you create numbers in memory and manipulate them using math operators and mathematical functions. This is great when your application deal with numbers (e.g. a calculator). But what if your application is, say, a music jukebox that deals with things like songs, playlists, artists etc.? Or what if your application is an air traffic control system that deals with planes, flights, airports etc.? Wouldn't it be nice to be able to manipulate such things with Objective-C as easily as you manipulate numbers? This is where objects kick in. With Objective-C, you can define the kind of objects you are interested to deal with, and then write applications that manipulate them.

As an example, let's take a look at how windows are handled within a program written in Objective-C, such as Safari. Take a look at an open Safari window of your Mac. At the top left, there are three buttons. The red one is the close button. So what happens if you close a window by clicking that red button? A message is sent to the window. In response to this message, the window executes some code in order to close itself.



A *close* message is sent to the window

The window is an object. You can drag it around. The three buttons are objects. You can click them. These objects have a visual representation on screen, but this is not true of all objects. For instance, the object that represents a connection between Safari and a given web site does not have a visual representation on screen.



An object (e.g. the window) can contain other objects (e.g. the buttons)

You can have as many Safari windows as you want. Do you think that Apple's programmers:

- a. programmed each of those windows in advance, using their massive brainpower to anticipate how many windows you might want to have, or
- b. made a kind of template and let Safari create the window objects from it on the fly?

Of course, the answer is b. They created some code, called a class, which defines what a window is and how it should look and behave. When you create a new window, it is actually this class that creates the window for you. This class represents the concept of a window, and any particular window is actually an instance of that concept (in the same way that 76 is an instance of the concept of a number).

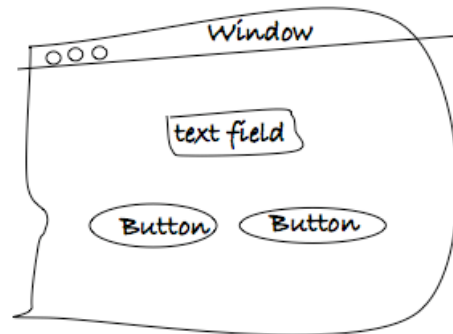
The window you created is present at a certain location on the screen of your Mac. If you minimize the window to the Dock, and make it reappear, it will take exactly the same position on the screen that it had before. How does this work? The class defines variables suitable for remembering the position of the window on the screen. The instance of the class, i.e. the object, contains the actual values for these variables. So, each window object contains the values of certain variables, and different window objects will in general contain different values for those variables.

The class not only created the window object, but also gave it access to a series of actions it can perform. One of those actions is *close*. When you click the "close" button of a window, the button sends the message *close*, to that window object. The actions that can be performed by an object are called **methods**. As you will see, they resemble functions very closely, so you will not have much trouble learning to use them if you have followed us so far.

When the class creates a window object for you, it reserves memory (RAM) to store the position of the window and some other information. However, it does not make a copy of the code to close the window. That would be a waste of computer memory because this code is the same for every window. The code to close a window needs to be present only once, but every window object has access to that code belonging to the window class.

As before, the code you are about to see in this chapter contains some lines for reserving memory and releasing it back to the system. As indicated earlier, we will not discuss this advanced subject until much later. Sorry.

We are going to create an application with two buttons and a text field. If you press one button, a value is entered into the text field. If you press the other button, another value is put into the text field. Think of it as a two-button calculator that can't do calculations. Of course, once you learn more you can figure out how to create a calculator for real, but I like small steps.

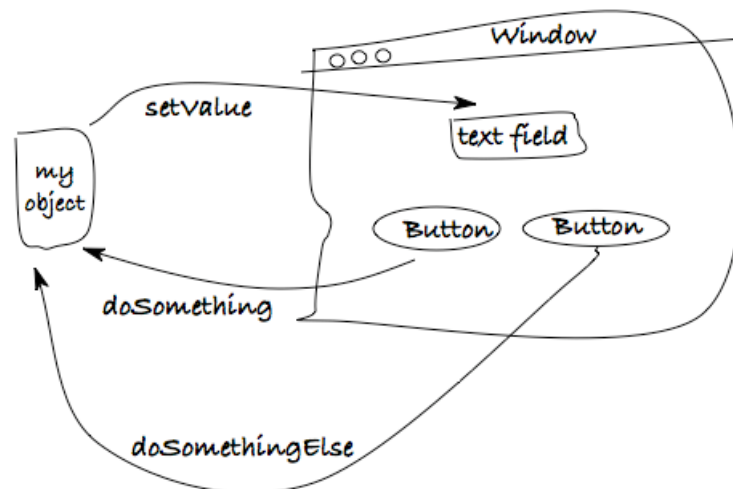


A sketch of the application we want to create.

If one of the buttons of our app is pressed, it will send a message. This message contains the name of a method to be executed. This message is sent to, well, to what? In case of the window, the *close* message was sent to that window object, which was an instance of the window class. What we need now is an object that is capable of receiving a message from each of the two buttons, and can tell the text field object to display a value.

So, we first have to create our own class, and then create an instance of it. That object will be the receiver of the message from the buttons (Please refer to the sketch below). Like a window object, our instance is an object, but in contrast to a window object, we can't see our instance on the screen when we run the program. It is just something in the memory of the Mac.

When our instance receives a message sent by one of the (two) buttons, the appropriate method is executed. The code of that method is in the class (not in the instance itself). Upon execution, this method will send a message to the text field object. Apart from the name of the text field object, the message sending expression contains, as always, the name of a method (of the text field class). Upon receipt of the message, the text field object will execute the method. We want the text field object to display a value depending on the button clicked. So, the message sending expression not only contains the name of the object and the method name, but also an argument (value) to be used by the method of the text field object.



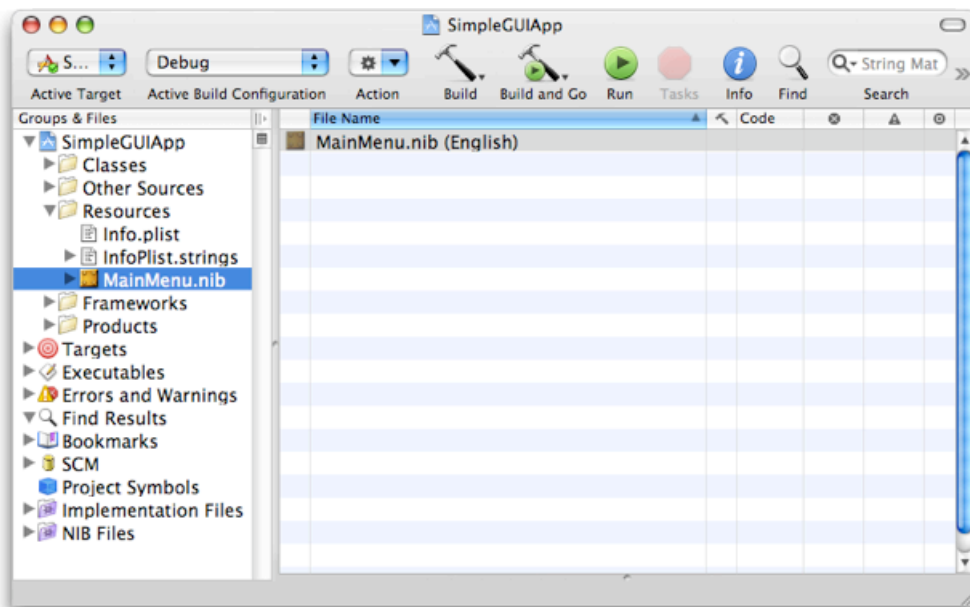
A sketch of message exchanges between the objects in our application.

Here is the general format of how to send messages in Objective-C, without [1.1] and with [1.2] an argument:

```
[1]
[receiver message];
[receiver message:argument];
```

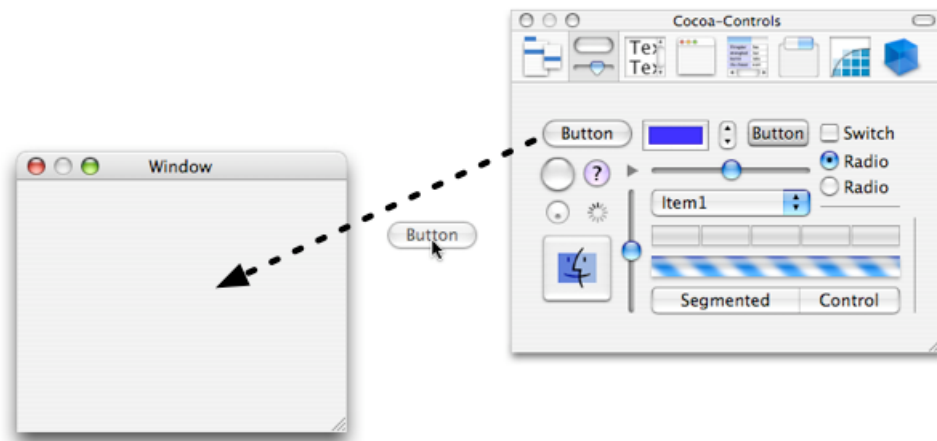
As you can see in each of these statements, the whole shebang is placed between square brackets and the eternal semi-colon is present as the finishing touch. Between the brackets, the receiving object is mentioned first, followed by the name of one of its methods. If the method being called requires one or more values, they must be provided as well [1.2].

Let's see how this works for real. Start up Xcode to create a new project. Select Cocoa Application under the Application heading. Give your project a name (by convention, the name of your GUI application should start with a capital). In the Groups & Files frame of the Xcode window that appears, open the Resources folder. Double-click on *MainMenu.nib*.



Double click on the MainMenu.nib file in Xcode.

Another program, Interface Builder, will start-up. As a lot of windows appear, you may want to choose Hide Others from the File menu. You will see three windows. The one named "Window" is the window that the users of your application will see. It is a bit big, so you may want to resize it. To the right of the window "Window", there is a window whose name starts with "Cocoa-". It is a kind of repository for all kinds of objects you can have in your GUI and is known as the "palettes window". Click the second icon in the toolbar of this window and drag two buttons onto the GUI window "Window". Click the third button in the palettes window toolbar and similarly drag one text field having the text "System Font Text" onto the GUI window.



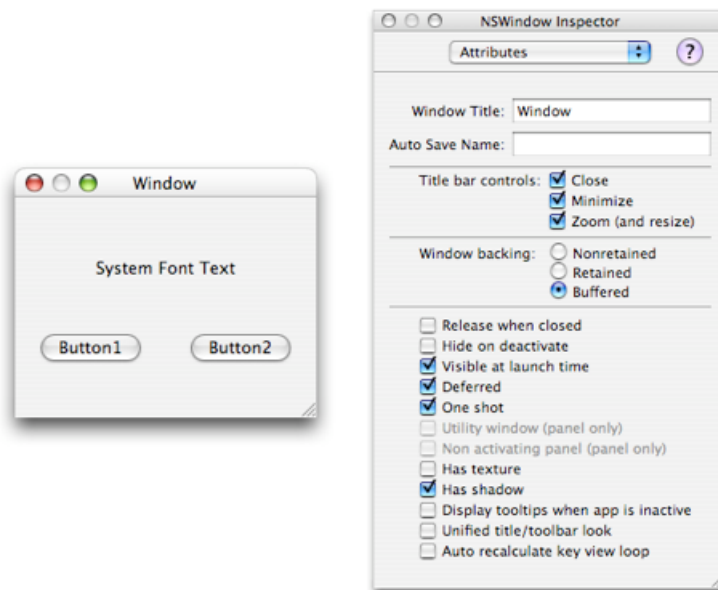
Dragging GUI objects from the palettes window to your application's window.

Behind the scenes, the action of dragging a button from the palettes window to your application's window creates a new button object and puts it in your window. The same goes on for the text field and any other objects you might drag to your window from the palettes window.

Note that if you hold your cursor over an icon in the palettes window, a name will be displayed, such as *NSButton* or *NSTextView*. These are the names of classes provided by Apple. Later in this chapter we will see how we can find the methods of these classes, which we need to perform the necessary actions in our program.

Arrange the objects you dragged onto the window "Window" nicely. Resize them as you see fit. Change the text of the button objects by double-clicking them, one at a time. I invite you to explore the palettes window after we have finished this project to get a grip on how to add other objects to your window.

To change the properties of an object, select it and press command-shift-i. Explore this as well. For example, select the window "Window" (you can see it gets selected under the Instances tab of the bottom-left window) and press command-shift-i. If the pop-up menu near the top reads *Attributes*, you can set a tick for *Textured Window*, and this gives your window a brushed-steel look. This means that you can customize the look of your application to a large extent without writing a single line of code!



Our window in Interface Builder, along with its inspector.

As promised above, we are going to create a class. But before we do that, let's look a bit deeper into how classes work.

To save a lot of programming effort, it would be nice if you could build on what others have already built, instead of writing everything from scratch. If you, for example, wanted to create a window with special properties (capabilities), you would need to add just the code for these properties. You wouldn't need to write code for all other behavior, such as minimizing or closing a window. By building on what other programmers have done, you would inherit all these kinds of behavior for free. And this is what makes Objective-C so different from plain C.

How is this done? Well, there is a window class (NSWindow), and you could write a class that inherits from that class. Suppose you add some behavior to your own window class. What happens if your special window receives a "close" message? You didn't write any code for that, and didn't copy such code into your class either. Simple, if the class of the special window doesn't contain the code for a particular method, the message is transferred automatically to the class from which the special window class inherits (its "superclass"). And if necessary, this goes on until the method is found (or it reaches the top of the hierarchy of inheritance.)

If the method cannot be found, you sent a message that can't be handled. It is like requesting a garage to change the tires of your sleigh. Even the boss of the garage can't help you. In such cases Objective-C will signal an error.

What if you want to implement your own behavior for a method already inherited from your superclass? That is easy, you can override particular methods. For example, you could write code that, on clicking the close button, would move the window out of view before actually closing it. Your special window class would use the same method name for closing a window as the one defined by Apple. So, when your special window receives a *close* message, the method executed is yours, and not Apple's. So, now the window would move out of sight, before being closed for real.

Hey, closing a window for real was already programmed by Apple. From inside our own *close* method, we can still invoke the *close* method implemented by our superclass, although it requires a slightly different call to make sure our own *close* method is not called recursively.

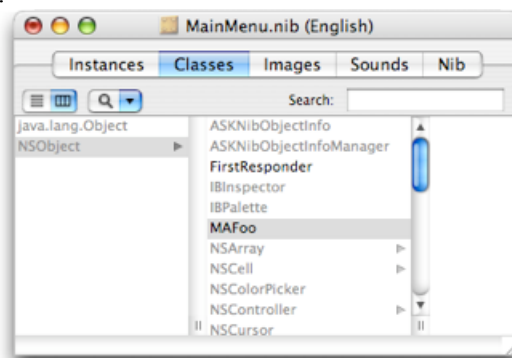
```
[2]
// Code to move the window out of sight here.
[super close]; // Use the close method of the superclass.
```

This stuff is way too advanced for this introductory booklet and we don't expect you to "get" it from these few lines.

King of the hill, among classes, is the class named "NSObject". Nearly all the classes you will ever create or use will be subclasses of NSObject, directly or indirectly. For example the `NSWindow` class is a subclass of the `NSResponder` class, which is itself a subclass of `NSObject`. The `NSObject` class defines the methods common to all objects (e.g. generating a textual description of the object, asking the object whether it is able to understand a given message etc.)

Before I bore you with too much theory, let's see how to create a class.

Go to the *MainMenu.nib* window, and select the *Classes* tab. In the first column, you will see the `NSObject` class. Select it and go to the *Classes* menu in the menubar. There, select *Subclass NSObject*. Go back to the *MainMenu.nib* and give your class a nice name. I named mine "MAFoo".



Creating the MAFoo class.

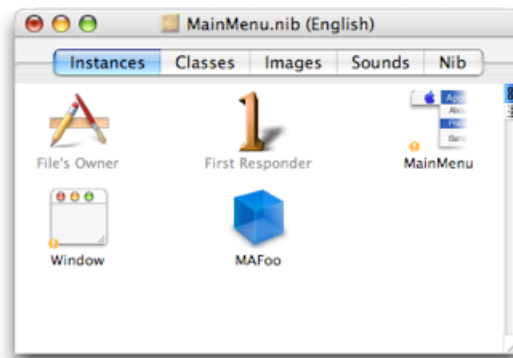
The first two capitals of MAFoo stand for **My Application**. You can invent class names as you like. Once you start writing your own applications, we recommend that you do something similar (i.e., choosing a two or three letter prefix that you'll use for all your classes in order to avoid clashes with existing class names). However, don't use `NS`, as this may confuse you later on. `NS` is used for Apple's classes. It stands for `NextStep`, `NextStep` being the operating system `Mac OS X` was based on when Apple bought `Next, Inc.`, and got `Steve Jobs` back as a bonus.

The `CocoaDev` wiki contains a list of other prefixes to avoid. You should check it when choosing your own prefix: <http://www.cocoadev.com/index.pl?ChooseYourOwnPrefix>

When creating a new class you should give it a name that conveys useful information about that class. For instance, we've already seen that in `Cocoa` the class used to represent windows is named `NSWindow`. Another example is the class that is used to represent

colors, which is named `NSColor`. In our case, the `MAFoo` class we are creating is just here to illustrate the way objects communicate together in an application. This is why we gave it a generic name with no special meaning.

Back in Interface Builder select *Instantiate MAFoo* from the *Classes* menu. As you can see under the *Instances* tab, you now have a new icon named `MAFoo`. This icon stands for the new instance we've just created.



Creating an instance of `MAFoo`

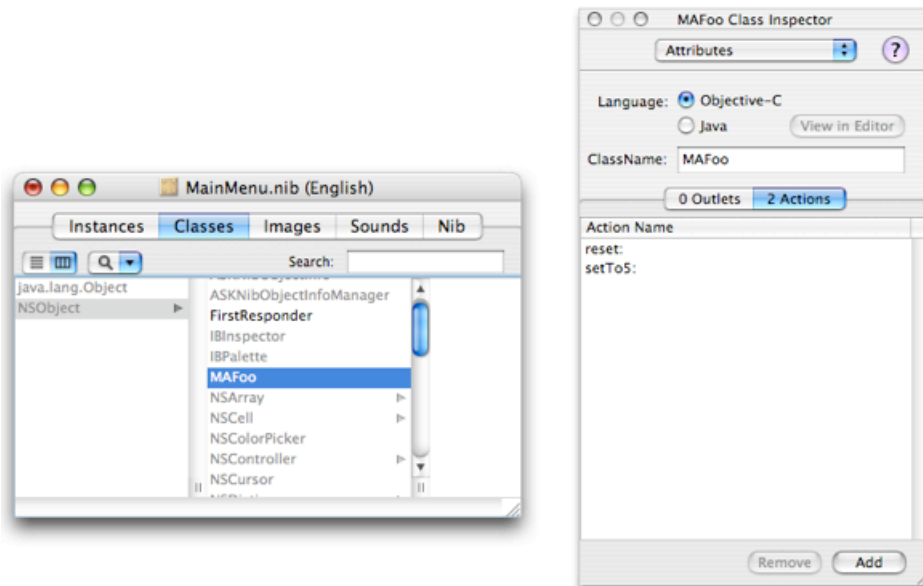
Our next step is to create connections between the buttons (from which messages are sent) to our `MAFoo` object. In addition, we are going to create a connection back from the `MAFoo` object to the text field, because a message will be sent to the text field object. An object has no way of sending a message to another object if it doesn't have a reference to the other object. By making a connection between a button and our `MAFoo` object, we are providing that button with a reference to our `MAFoo` object. Using this reference, the button will be able to send messages to our `MAFoo` object. Likewise, establishing a connection from our object to the text field will allow the former to message the latter.

Let us again go through what the application has to do. Each of the buttons can send, when clicked, a message corresponding to a particular action. This message contains the name of the method of the class `MAFoo` that has to be executed. The message is sent to the instance of the `MAFoo` class we've just created, the `MAFoo` object. The `MAFoo` object itself doesn't contain the code to perform the action, but the `MAFoo` class does. So, this message sent to the `MAFoo` object triggers a method of the class `MAFoo` to do something: in this case, sending a message to the text field object. Like every message, this one consists of the name of a method. In this case, the method of the text field object has the task of displaying a value. The value is sent as part of the message, along with the name of the method to invoke on the text field.

Our class needs two actions (methods), which will be called by the (two) button objects. Our class needs one outlet, a variable for remembering which object (i.e., the text field object) is to be sent a message.

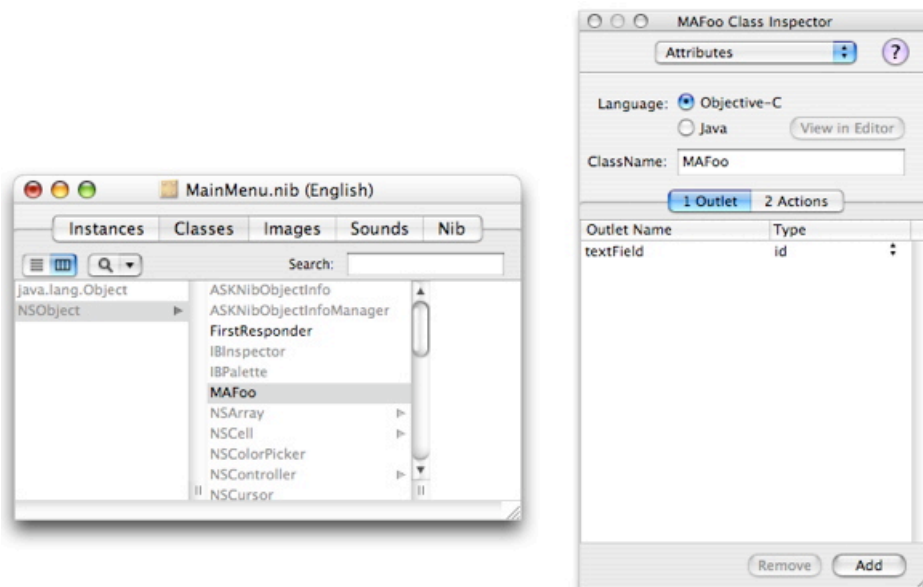
Make sure `MAFoo` is selected under the *Classes* tab in the *MainFile.nib* window. On your keyboard, press command-shift-i in order to bring on screen the inspector for the class. In the inspector window, select the *Action* tab and click the *Add* button to add an action (i.e., a method) to the `MAFoo` class. Replace the default name provided by Interface Builder by a more meaningful name (for example, you can enter "setTo5:" because we will program this method to display the number 5 in the text field). Add another method, and give it a name

(for example "reset:", because we will program it to display the number 0 in the text field). Note that our method names both end with a colon (":"). More about this later.



Adding action methods to the MAFoo class

Now, in the inspector window, select the *Outlet* tab, add an outlet and give it a name (for example "textField").



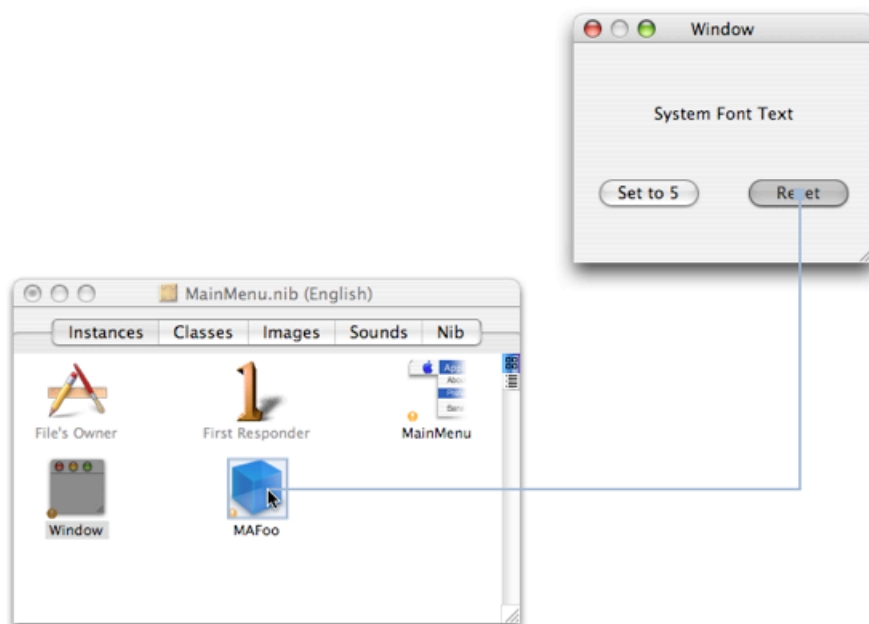
Adding an outlet to the MAFoo class

Before establishing connections between objects, we are going to give meaningful names to our two buttons. Since the first one is going to ask our MAFoo instance to display the number 5 in the text field, we name it "Set to 5" (we already learned how to change the name of a button: double click on its name on-screen, and then enter the new name). Likewise, we name the second one "Reset". Note that this step of giving this button a particular name is not required for our program to work correctly. It is just that we want our user interface to be as descriptive as possible to the end-user.

Now we are ready to create the actual connections between

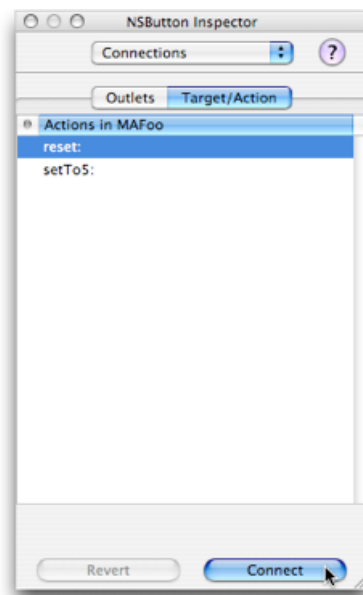
- a) the button "Reset" and the MAFoo instance.
- b) the button "Set to 5" and the MAFoo instance.
- c) the MAFoo instance and the text field.

To create the connections, click the *Instances* tab of the *MainFile.nib* window. Then, press the Control key on you keyboard and use the mouse to drag from the *Reset* button to the MAFoo instance (don't do it the other way around!). A line representing the connection will appear on screen. Make the line go from the button to the MAFoo instance and then release the mouse button.



Connecting a button to the MAFoo object.

When you release the mouse, the button inspector displays a connection panel that lists the action methods available on the MAFoo object. Select the appropriate action (i.e., "reset:") and click on the *Connect* button to complete the connection process.



Establishing the connection in the inspector.

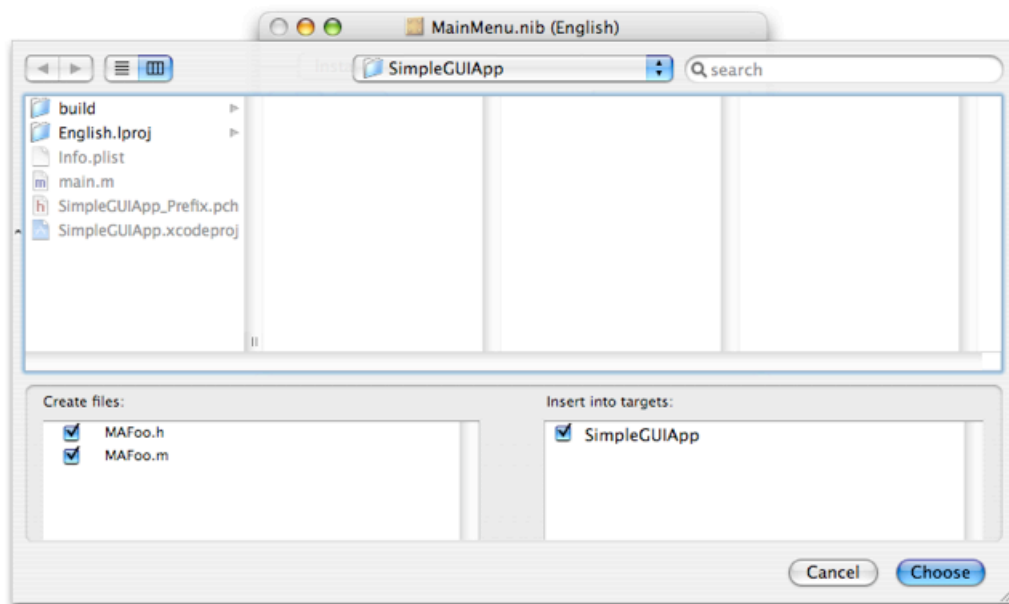
Now the button holds a reference to our MAFoo object, and will send it the *reset:* message whenever it is pressed.

You can now connect the *Set to 5* button to the MAFoo object by applying the same process.

To create the connection between the MAFoo object and the text field, start with MAFoo and control-drag to the text field object. Click *Connect*.

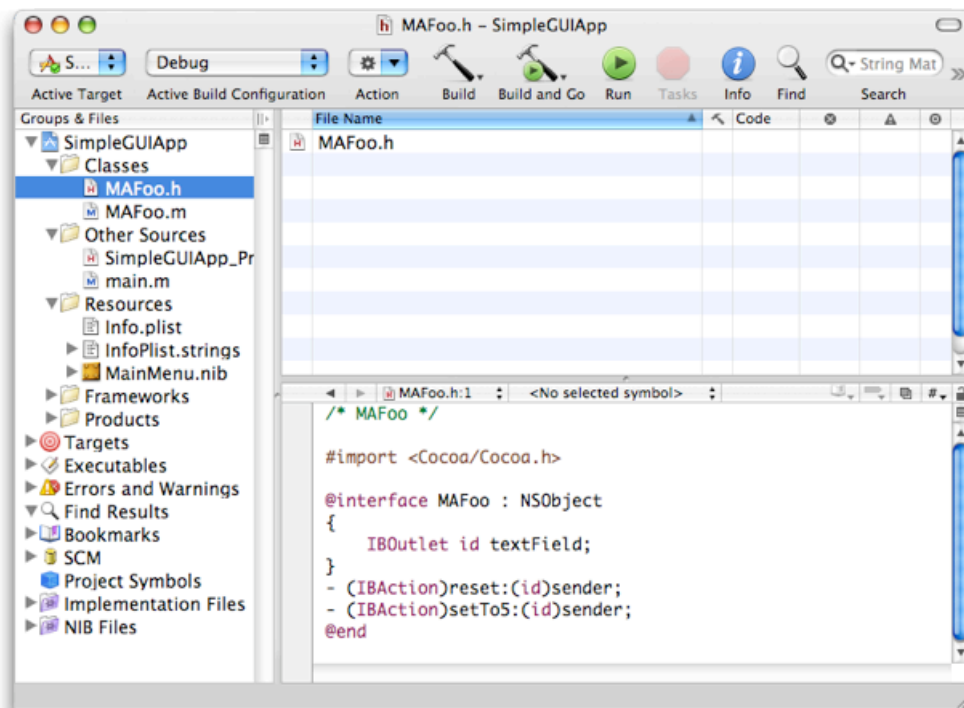
What was this all about? Well, as you will see in a minute, you have just created the equivalent of a lot of code without writing a single line of it.

Make sure the MAFoo instance is selected in the *MainMenu.nib* window and then switch to the *Classes* tab. You should see a list of classes within which the MAFoo class is selected. In the *Classes* menu, select *Create Files for MAFoo*. Interface Builder then asks you where you want your generated file to be put on disk. By default, the files will go in the project folder of our application, which is exactly what we want.



Generating the code for the MAFoo class.

Now, if you switch back to Xcode, you'll see the generated files in your project window, inside the *Other Sources* group. You can drag them in the *Classes* group if you want, as the generated files actually represent a class.



The generated files appear in our Xcode project.

Let's go back for a moment to Chapter 4, where we discussed functions. Do you remember the function header [11.1]? It was a kind of warning for the compiler to tell it what it could expect. One of the two files is named MAFoo.h, and it is a header file: it contains info

about our class. For example, you'll recognize that there is a line [3.5] containing `NSObject`, which line means that our class inherits from the `NSObject` class.

```
[3]
/* MAFoo */

#import <Cocoa/Cocoa.h> // [3.3]

@interface MAFoo : NSObject
{
    IBOutlet id textField; // [3.7]
}
- (IBAction)reset:(id)sender;
- (IBAction)setTo5:(id)sender;
@end
```

You will see that there is one outlet [3.7] to the text field object. "id" indicates object. "IB" stands for Interface Builder, the program you used to create this code.

IBAction [3.9, 3.10] is equivalent to *void*. Nothing is returned to the object that sends the message: the buttons in our program do not get a reply from the `MAFoo` object in response to their message.

You can also see there are two Interface Builder Actions.

Earlier we have seen `#import <Foundation/Foundation.h>` instead of line [3.3], The former is for non-GUI apps, the latter for GUI-apps.

Now let's check out the second file, `MAFoo.m`. Again we get a lot of code for free.

```
[4]
#import "MAFoo.h"

@implementation MAFoo

- (IBAction)reset:(id)sender // [4.5]
{
}

- (IBAction)setTo5:(id)sender
{
}

@end
```

First of all, the `MAFoo.h` header file is imported, so the compiler knows what to expect. Two methods can be recognized: *reset:* and *setTo5:*. These are the methods of our class. They are similar to functions in that you need to put your code between the curly braces. In our application, when a button is pressed, it sends a message to your `MAFoo` object, requesting the execution of one of the methods. We do not have to write any code for that. Making the connections between the buttons and the `MAFoo` object in Interface Builder is all what is required. However, we do have to write the code that sends a message from our `MAFoo` object to the text field object, so we provide the statements [5.7, 5.12].

```
[5]
#import "MAFoo.h"

@implementation MAFoo

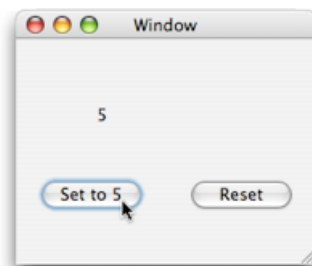
- (IBAction)reset:(id)sender
{
    [textField setIntValue:0];    // [5.7]
}

- (IBAction)setTo5:(id)sender
{
    [textField setIntValue:5];    // [5.12]
}

@end
```

As you can see, we send a message to the object referenced by the *textField* outlet. Since we connected this outlet to the actual text field, using Interface Builder, our message will be sent to the correct object. The message is the name of a method, *setIntValue:*, together with an integer value. The *setIntValue:* method is capable of displaying an integer value in a text field object. In the next chapter we will tell you how we found out about this method.

You are now ready to compile your application and launch it. As usual press the *Build and Go* button in the Xcode toolbar. It will take a few second for Xcode to build the application and to launch it. Eventually, the application will appear on screen and you'll be able to test it.



Our application running.

In short, you have just created a (very basic) application, for which you had to write just two lines of code yourself!

CHAPTER 9

FINDING METHODS

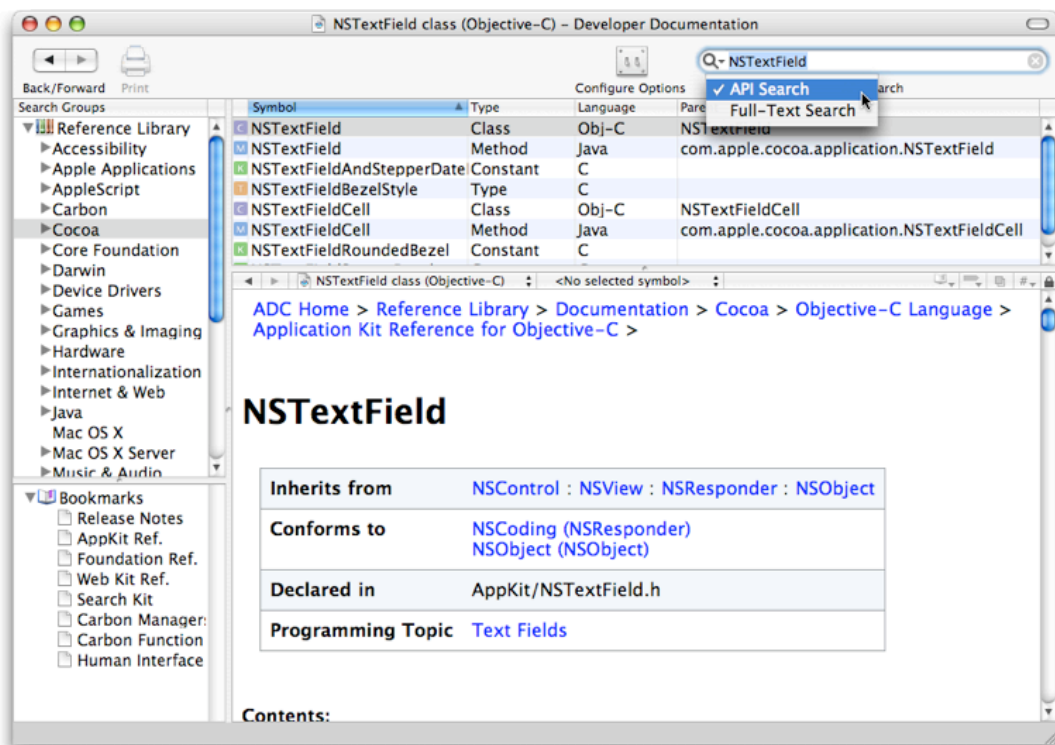
In the previous chapter, we learned about methods. We wrote (the body of) two methods ourselves, but we also used one provided by Apple. *setIntValue:* was the method for displaying the value of an integer in the text field object. How do you find out about the available methods?

Remember, for every method you use that was created by Apple, you don't have to write any code yourself. Plus, it is more likely to be bug-free. So, it's always worth taking the time to check whether suitable methods are available before programming your own.

In Interface Builder, if you hold the cursor over an object in the palettes window, a small label pops up. If you hold your cursor above the button icon, you'll see "NSButton". If you hold it above the text field reading "System Font Text", you will see "NSTextField". Each of these names is a class name. Let's check out the NSTextField class to see what methods are available.

Go to Xcode, and, in the menu, select *Help->Documentation*. In the frame on the left, select *Cocoa* and then enter "NSTextField" in the search field (make sure the API-Search mode is selected; see the screen-shot below). While you are typing, the list of possible hits is reduced significantly and soon you will see NSTextField appearing on top.

Click the line that reads NSTextField (of type *Class*) to get information on the class NSTextField displayed in the lower frame.



Navigating in the Cocoa documentation with Xcode.

The first thing you should notice is that this class inherits from a series of other classes. The last one in the list is top-dog, king of the hill `NSObject`. A little lower (please scroll) there is the heading:

Method Types

That is where we are going to start our search. A quick glance over the subheadings will tell us that we are not going to find the method that we need to display a value in the text field object here. Because of the principle of inheritance, we need to visit the immediate superclass of the `NSTextField` class, which is `NSControl` (and if we fail, we have to scrutinize its superclass `NSView`, etc.). Because all the documentation is in HTML, all we have to do is click the word [NSControl](#) (as shown above in the *Inherits from* list). This brings us to the information on the `NSControl` class:

NSControl

Inherits from	NSView : NSResponder : NSObject
----------------------	---

You can see we moved one class up. In the method list, we notice a subheading:

Setting the control's value

That is what we want, we want to set a value. Below this subheading we find:

- `setIntValue:`

Sounds promising, so we check out the description of this method by clicking the *setIntValue:* link.

setIntValue:

- `(void)setIntValue:(int)anInt`

Sets the value of the receiver's cell (or selected cell) to the integer `anInt`. If the cell is being edited, it aborts all editing before setting the value; if the cell doesn't inherit from `NSActionCell`, it marks the cell's interior as needing to be redisplayed (`NSActionCell` performs its own updating of cells).

In our application, our `NSTextField` object is the receiver and we have to feed it an integer. We can also see this from the **signature** of the method:

- `(void)setIntValue:(int)anInt`

In Objective-C, the minus sign marks the beginning of an instance method declaration (as opposed to a class method declaration, which we'll talk about later). `void` indicates that nothing is returned to the invoker of the method. That is, when we send a *setIntValue:* message to *textField*, our `MAFood` object does not receive a value back from the text field object. That is ok. After the colon, *(int)* indicates that the variable *anInt* must be an integer. In our example, we send it a value of 5 or 0, which are integers, so we are fine.

Sometimes it is a bit more difficult to find out which is the appropriate method to use. You will get better at this when you'll be more familiar with the documentation, so keep practicing.

What if you want to read the value from our text field object *textField*? Remember the great thing about functions being that all variables inside were shielded? The same is true for methods. Often however, objects have a pair of related methods, called "Accessors", one for reading the value, and one for setting the value. We already know the last one, this is the *setIntValue:* method. The first one looks like this:

```
[1]
- (int) intValue
```

As you can see, this method returns an integer. So, if we want to read the integer value associated with our *textField* object we have to send it a message like this:

```
[2]
resultReceived = [textField intValue];
```

Again, in functions (and methods) all variables are shielded. That is great for variable names, because you do not have to fear that setting a variable in one part of your program will affect a variable with the same name in your function. However, function names must still be unique. Objective-C takes shielding one step further: method names have to be unique within a class only, but different classes may have method names in common. This is a great feature for large programs, because programmers can write classes independent of each other, without having to fear conflicts in method names.

But there is more. The fact that different methods in different classes can have the same name is called "polymorphism" in geek speak, and is one thing that makes object-oriented programming so special. It allows you to write chunks of code without having to know in advance what are the classes of the objects you are manipulating. All that is required is that, at run-time, the actual objects understand the messages you send them. Taking advantage of this, you can write applications that are flexible and extensible by design. For instance, in the GUI application we created, if we replace the text field by an object of a different class that is able to understand the *setIntValue:* message, our application will still work without requiring us to modify our code, or even to recompile it. We are even able to vary the object at run-time without breaking anything. Therein lies the power of object-oriented programming.

CHAPTER 10

awakeFromNib

Apple has done a lot of work for you, making it easier to create your programs. In your little application, you didn't have to worry about drawing a window and buttons to a screen, amongst many other things. Most of this work is made available through two frameworks. The Foundation Kit framework that we imported in example [12] of Chapter 4, provides most services not associated with a graphical user interface. The other framework, called the Application Kit, deals with objects you see on screen and user-interaction mechanisms. Both frameworks are well documented.

Let's go back to our GUI application. Suppose we want our application to display a particular value in the text field object immediately when the application is launched and the window is initially shown.

All the information for the window is stored in a **nib file** (*nib* stands for Next Interface Builder). This is an indication that the method we need may be part of the Application Kit. Let's find out how to get information about this framework. In Xcode, go to the *Help* menu and select Documentation.

In the documentation window make sure *Full-Text Search* is enabled (to do that, click on the little lens in the search field and select *Full-Text Search* in the associated menu). Then type *Application Kit* in the search field and press Return. Xcode provides you with multiples results. Among them is a document named *Application Kit Reference for Objective-C*. Inside it you'll find a list of services provided by this framework. Under the headings Protocols there is a link called *NSNibAwakening*.

NSNibAwaking

Declared in	AppKit/NSNibLoading.h
Programming Topic	Loading Resources

Contents:

[Protocol Description](#)

[Method Types](#)

[Instance Methods](#)

Protocol Description

This informal protocol consists of a single method, [awakeFromNib](#). Classes can implement this method to perform final initialization of state after objects have been loaded from an Interface Builder archive.

If we implement this method, it will be called when our object is loaded from its nib file. Thus we can use it to achieve our goal: displaying a value in the text field at launch time.

By no means do I want to suggest that it is always trivial to find the correct method. Often, it will require quite a bit of browsing and the creative use of keywords for searches, to find a promising method. For that reason, it is highly important that you

familiarize yourself with the documentation of both frameworks, so you will know what classes and methods are available to you. You may not need it at that time, but it will help you to figure out how to get your program to do what you want.

Ok, now we have found our method, all we need to do is to add the method to our implementation file `MAFoo.m` [1.15].

```
[1]
#import "MAFoo.h"

@implementation MAFoo

- (IBAction)reset:(id)sender
{
    [textField setIntValue:0];
}

- (IBAction)setTo5:(id)sender
{
    [textField setIntValue:5];
}

- (void)awakeFromNib // [1.15]
{
    [textField setIntValue:0];
}

@end
```

When the window is opened, the *awakeFromNib* method is called automatically. As a result, the text field displays zero when you lay your eyes on the newly opened window.

CHAPTER 11

POINTERS

Warning! This chapter contains advanced concepts and deals with underlying C language concepts that beginners may find intimidating. If you don't understand it all now, don't worry. Thankfully, in general - although understanding how pointers work is useful - it is not essential to start programming in Objective-C.

When you define a variable your Mac associates this variable with some space in its memory in order to store the value of the variable.

For instance, examine the following instruction:

```
[1]
int x = 4;
```

In order to execute it, your Mac finds some space in its memory that is not already in use and then note that this space is where the value of the variable *x* is to be stored (of course we could and should have used any other name here for our variable). Look at the instruction [1] again. Indicating the type of the variable (here *int*) lets your computer know how much space in memory is needed to store the value of *x*. If the value were of type *long* or *double*, more memory would have to be reserved.

The assignment instruction "*x* = 4" stores the number 4 in this reserved space. Of course, your computer remembers where the value of the variable named *x* is stored in its memory, or, in other words, what the **address** of *x* is. That way, each time you use *x* in your program, your computer can look in the right place (at the right address) and find the actual value of *x*.

A **pointer** is a variable that contains the address of another variable.

Given a variable, you can get its address by writing `&` before the variable. For example, to get the address of *x*, you write `&x`.

When the computer evaluates the expression *x* it returns the value of the variable *x* (in our example, it will return 4). By contrast, when the computer evaluates the expression `&x`, it returns the address of the variable *x*, not the value stored there. The address is a number that denotes a specific place in the memory of the computer (like a room number denotes a specific room in a hotel).

You declare a pointer like this:

```
[2]
int *y;
```

This instruction defines a variable named *y* that will contain the address of a variable of type *int*. To store in a variable *y* what the address of variable *x* is (in official geek speak: assign the address of *x* to *y*), you do:

```
[3]
y = &x;
```

Given a pointer, you can get at the variable it points to by writing an asterisk before the pointer. For instance, evaluating the expression

```
*y
```

will return 4. This is equivalent to evaluating the expression "x". Executing the instruction

```
*y = 5
```

is equivalent to executing the instruction "x = 5".

Pointers are useful because sometimes you don't want to refer to the value of a variable, but to the address of that variable. For instance, if you want to program a function that adds 1 to a variable, you need pass it the address of that variable. This is because you want to modify what is stored in this variable, not just use the current value. Thus, you use a pointer as argument:

```
[4]
void increment(int *y)
{
    *y = *y + 1;
}
```

You can then call it like this:

```
[5]
int x = 4;
increment(&x);
// now x is equal to 5
```

CHAPTER 12

STRINGS

So far, we have seen several data types: integer, long, float, double, BOOL. Plus in the last chapter we introduced pointers. While we touched on the subject of strings, we have only discussed it in relation to the NSLog() function. This function allowed us to print a string to the screen, replacing codes starting with a %-sign, such as %d, with a value.

```
[1]
float piValue = 3.1416;
NSLog(@"Here are three examples of strings printed to the screen.\n");
NSLog(@"Pi approximates %10.4f.\n", piValue);
NSLog(@"The number of eyes of a dice is %d.\n", 6);
```

We did not discuss strings as data types before, for good reason. They are objects, created using the class NSString or the class NSMutableString. Let's discuss these classes, beginning with NSString.

```
[2]
NSString *favoriteComputer;
favoriteComputer = @"Mac!";
NSLog(favoriteComputer);
```

You'll probably find the second statement comprehensible, but the first one [2.1] deserves a bit of explanation. Remember that, when we declared a pointer variable, we had to tell what type of data the pointer was pointing to? Here is a statement from chapter 11 [2].

```
[3]
int *y;
```

We tell the compiler that the pointer variable *y* contains the address of a memory location where an integer can be found. In [2.1] we tell the compiler that the pointer variable *favoriteComputer* contains the address of a memory location where an object of type NSString can be found. We use a pointer to hold our string because in Objective-C, objects are never manipulated directly, but always through pointers to them.

Ok, why does this funny @ sign show up all the time? Well, Objective-C is an extension of the C-language, which has its own ways to deal with strings. To differentiate the new type of strings, which are fully-fledged objects, Objective-C uses an @ sign.

How does Objective-C improve on strings of the C language? Well, Objective-C strings are Unicode strings instead of ASCII strings. Unicode-strings can display characters of just about any language, such as Chinese, as well as the Roman alphabet.

Of course, it is possible to declare and initialize the pointer variable for a string in one go [4].

```
[4]
NSString *favoriteActress = @"Julia";
```

The pointer variable *favoriteActress* points to a location in memory where the object representing the string "Julia" is stored.

Once you have initialized the variable, i.e. *favoriteComputer*, you may give the variable another value, but you cannot change the string itself [3] because it is an instance of class `NSString`. More on this in a minute.

```
[5]
#import <Foundation/Foundation.h>

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

    NSString *x;
    x = @"iBook"; // [5.7]
    x = @"MacBook Pro Intel"; // Hey, I'm just trying to make
                             // this book look up to date!

    NSLog(x);
    [pool release];
    return 0;
}
```

When executed, the program prints:

```
MacBook Pro Intel
```

A string of class `NSString` is called immutable, because it cannot be modified.

What good is a string you can't modify? Well, strings that can't be modified are easier for the operating system to handle, so your program can be faster. In fact when you use Objective-C to write your own programs, you'll find that most times you don't need to modify your strings.

Of course, at times you will need strings that you can modify. So, there is another class, and the string objects you create with it are modifiable. The class to use is the class `NSMutableString`. We'll discuss it later in this chapter. First, let's make quite sure that you understand that strings are objects. As they are objects, we can send messages to them. For example, we can send the message *length* to a string object [6].

```
[6]
#import <Foundation/Foundation.h>

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

    foo = @"Julia!";
    theLength = [foo length]; // [6.10]
    NSLog(@"The length is %d.", theLength);

    [pool release];
    return 0;
}
```

When executed, the program prints:

```
The length is 6.
```

Programmers often use *foo* and *bar* as variable names when explaining things. Actually, they're bad names, because they are not descriptive, just like *x*. We expose you to them here, so you will not be puzzled when you see them in discussions on the Internet.

In line [6.10] we send the object *foo*, the message *length*. The method *length* is defined in the `NSString` class as follow:

```
- (unsigned int)length
```

Returns the number of Unicode characters in the receiver.

You may also change the characters of the string to uppercase [7]. To that end, send the string object the appropriate message, i.e. *uppercaseString*, which you should be able to find in the documentation yourself (check the methods available in the `NSString` class). Upon reception of this message, the string object creates and returns a new string object containing the same content, with each character changed to its corresponding uppercase value.

```
[7]
#import <Foundation/Foundation.h>

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

    NSString *foo, *bar;
    foo = @"Julia!";
    bar = [foo uppercaseString];
    NSLog(@"%@ is converted into %@.", foo, bar);

    [pool release];
    return 0;
}
```

When executed, the program prints:

```
Julia! is converted into JULIA!
```

Sometimes you might want to modify the content of an existing string instead of creating a new one. In such case you'll have to use an object of class `NSMutableString` to represent your string. `NSMutableString` provides several methods that allow you to modify the content of a string. For instance, the method *appendString:* appends the string passed as argument to the end of the receiver.

```

[8]
#import <Foundation/Foundation.h>

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

    NSMutableString *foo;                                // [8.7]
    foo = [@"Julia!" mutableCopy];                        // [8.8]
    [foo appendString:@" I am happy"];
    NSLog(@"Here is the result: %@.", foo);

    [pool release];
    return 0;
}

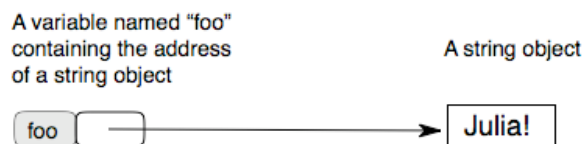
```

When executed, the program prints:

Here is the result: Julia! I am happy.

In line [8.8], the method *mutableCopy* (which is provided by the *NSString* class) creates and returns a mutable string with the same content as the receiver. That is, after the execution of the line [8.8], *foo* points to a mutable string object which contains the string "Julia!".

Earlier in this chapter we stated that, in Objective-C, objects are never manipulated directly, but always through pointers to them. This is why, for instance, we use the pointer notation in line [8.7] above. Actually, when we use the word "object" in Objective-C, what we usually mean is "pointer to an object". But since we always use objects through pointers, we use the word "object" as a shortcut. The fact that objects are always used through pointers has an important implication that you must understand: several variables can reference the same object at the same time. For instance, after the execution of line [8.7], the variable *foo* references an object representing the string "Julia!", something we can represent with the following picture:

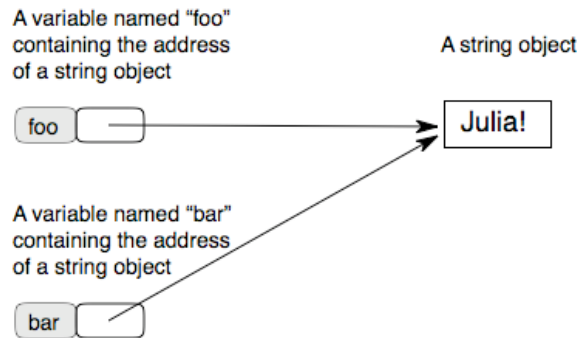


Objects are always manipulated through pointers.

Now suppose we assign the value of *foo* to the variable *bar* like this:

```
bar = foo;
```


The result of this operation is that both `foo` and `bar` now point to the same object:



Multiples variables can reference the same object.

In such a situation, sending a message to the object using `foo` as the receiver (e.g. `[foo dosomething];`) has the same effect as sending the message using `bar` (e.g. `[bar dosomething];`), as shown in this example:

```
[9]
#import <Foundation/Foundation.h>

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

    NSMutableString *foo = [@"Julia!" mutableCopy];
    NSMutableString *bar = foo;

    NSLog(@"foo points to the string: %@", foo);
    NSLog(@"bar points to the string: %@", bar);
    NSLog(@"-----");

    [foo appendString:@" I am happy"];

    NSLog(@"foo points to the string: %@", foo);
    NSLog(@"bar points to the string: %@", bar);

    [pool release];
    return 0;
}
```

When executed, the program prints:

```
foo points to the string: Julia!
bar points to the string: Julia!
-----
foo points to the string: Julia! I am happy
bar points to the string: Julia! I am happy
```

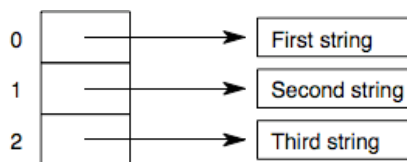
Being able to have references to the same object from different places at the same time is an essential feature of object-oriented languages. Actually we've already used it in previous chapters. For instance, in chapter 8, we referenced our `MAFoo` object from two different button objects.

CHAPTER 13

ARRAYS

At times you will need to hold collections of data. For example, you might need to maintain a list of strings. It would be quite cumbersome to use a variable for each of those strings. Of course, there is a more convenient solution: the array. An array is an ordered list of objects (or, more exactly, a list of pointers to objects). You can add objects in an array, remove them or ask the array to let you know which object is stored at a given index (i.e., at a given position). You can also ask the array to let you know how many elements it contains.

When you count items, you usually start with 1. In arrays however, the first item is at index zero, the second at index 1 and so on.



Example: an array containing three strings

We will give you some example of code later in this chapter, allowing you to see the effect of counting starting with zero.

Arrays are provided by two classes: `NSArray` and `NSMutableArray`. As with strings, there is an immutable and a mutable version. In this chapter, we'll consider the mutable version.

One way to create an array is to execute an expression like this:

```
[NSMutableArray array]
```

When evaluated, this code creates and returns an empty array. But... wait a minute... this code seems odd, doesn't it? Indeed, in this case we have used the name of the `NSMutableArray` class for specifying the receiver of a message. But we are supposed to send messages to instances, not to classes, right?

Well, we have just learned something new: the fact that, in Objective-C, we can also send messages to classes (and the reason is that classes are also objects, instances of what we call *meta-classes*, but we won't explore that idea further in this introductory article). In the Cocoa documentation, the methods we can call on classes are denoted by a "+" symbol, instead of the "-" symbol we usually see before the name of methods (for example Chapter 8 [4.5]). For instance, in the documentation we see this description for the `array` method:

array

+ (id)array

Creates and returns an empty array. This method is used by mutable subclasses of `NSArray`.

See Also: + [arrayWithObject:](#), + [arrayWithObjects:](#)

Let's go back to coding. The following program creates an empty array, stores three strings in it, and then prints the number of elements in the array.

```
[1]
#import <Foundation/Foundation.h>

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

    NSMutableArray *myArray = [NSMutableArray array];

    [myArray addObject:@"first string"];
    [myArray addObject:@"second string"];
    [myArray addObject:@"third string"];

    int count = [myArray count];

    NSLog(@"There are %d elements in my array", count);

    [pool release];
    return 0;
}
```

When executed, the program prints:

```
There are 3 elements in my array
```

The following program is the same as the previous one except that it prints the string stored at index 0 in the array. To get at this string, it uses the *objectAtIndex:* method [2.13].

```
[2]
#import <Foundation/Foundation.h>

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

    NSMutableArray *myArray = [NSMutableArray array];

    [myArray addObject:@"first string"];
    [myArray addObject:@"second string"];
    [myArray addObject:@"third string"];

    NSString *element = [myArray objectAtIndex:0]; // [2.13]

    NSLog(@"The element at index 0 in the array is: %@", element);

    [pool release];
    return 0;
}
```

When executed, the program prints:

```
The element at index 0 in the array is: first string
```

You'll often have to step through an array in order to do something with each element of an array. To do that, you can use a loop construct like in the following program which prints each element of the array along with its index:

```
[3]
#import <Foundation/Foundation.h>

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

    NSMutableArray *myArray = [NSMutableArray array];

    [myArray addObject:@"first string"];
    [myArray addObject:@"second string"];
    [myArray addObject:@"third string"];

    int i;
    int count;

    for (i = 0, count = [myArray count]; i < count; i = i + 1)
    {
        NSString *element = [myArray objectAtIndex:i];
        NSLog(@"The element at index %d in the array is: %@", i, element);
    }

    [pool release];
    return 0;
}
```

When executed, the program prints:

```
The element at index 0 in the array is: first string
The element at index 1 in the array is: second string
The element at index 2 in the array is: third string
```

Note that arrays are not limited to contain strings. They can contain any object you want.

The `NSArray` and `NSMutableArray` classes provide many other methods, and you are encouraged to look at the documentation for these classes in order to learn more about arrays. We'll end this chapter by talking about the method that allows you to replace an object at a given index in by another object. This method is named *replaceObjectAtIndex:withObject:*. Up until now we have only dealt with methods that take at most one argument. This one is different, and this is why we are looking at it here: it takes two arguments. You can tell because its name contains two colons. In Objective-C methods can have any number of arguments. Here is how you can use this method:

```
[4]
[myArray replaceObjectAtIndex:1 withObject:@"Hello"];
```

After executing this method, the object at index 1 is the string `@ "Hello"`. Of course, this method should only be invoked with a valid index. That is, there must already be an object stored at the index we give to the method, in order for the method to be able to replace it in the array by the object we pass. As you can see, method names in Objective-C are like sentences with holes in them (prefixed with colons). When you invoke a method you fill the holes with actual values, creating a meaningful "sentence". This way of denoting method names and method invocation comes from Smalltalk and is one of Objective-C's greatest strengths, as it makes the code very expressive. When you create your own methods, you should strive to name them in a way that they form expressive sentences when called. This helps make Objective-C code readable, which is very important in keeping your programs easily maintainable.

CHAPTER 14

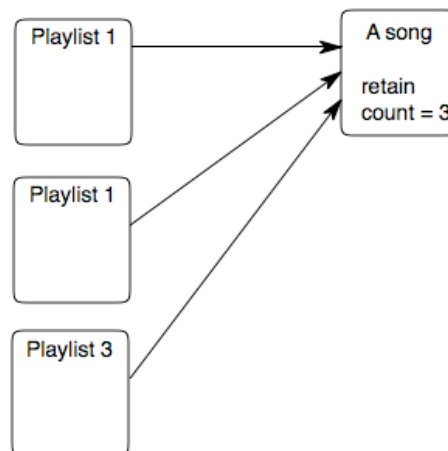
MEMORY MANAGEMENT

For many a chapter I've apologized that I did not explain a couple of statements in the examples. These statements dealt with memory. Your program is not the only program on your Mac, and RAM is a valuable commodity. So, if your program doesn't need a part of memory anymore, you should give it back to the system. When your mom told you that you have to be polite and live in harmony with the community, she was teaching you how to program! Even if your program were the only one running, memory not freed would eventually paint your program into a corner and your computer would slow down to a crawl.

When your program creates an object, the object occupies some space in memory and you have to free that space when your object is no longer used. That is, when your object is no longer used, you should destroy it. However, determining when an object is finished being used may not be easy to do. For instance, during the execution of the program, your object may be referenced by many other objects, and therefore must not be destroyed as long as there is a possibility that it may be used by some other objects (trying to use an object that has been destroyed can cause your program to crash or behave in unpredictable ways).

In order to help you destroy objects when they are no longer needed, Cocoa associates a counter with each object, which represents what is called the "retain count" of the object. In your program, when you store a reference to an object, you have to let the object know about that by increasing its retain count by one. When you remove a reference to an object, you let the object know about that by decreasing its retain count by one. When the retain count of an object becomes equal to zero, the object knows that it is no longer referenced anywhere and that it can be destroyed safely. The object then destroys itself, freeing the associated memory.

For instance, suppose your application is a digital jukebox and you have objects representing songs and playlists. Suppose that a given song object is referenced by three playlists objects. If it is not referenced elsewhere, your song object will have a retain count of three.



An object knows how many times it is referenced, thanks to its retain count.

In order to increase the retain count of an object, all you have to do is to send the object a *retain* message. In order to decrease the retain count of an object, all you have to do is to send the object a *release* message. Cocoa also offers a mechanism called "autorelease pool" which lets you send a delayed *release* message to an object - not immediately, but at a later time. To use it, you just have to register the object with what is called an autorelease pool, by sending your object an *autorelease* message. The autorelease pool will take care of sending the delayed *release* message to your object. The statements dealing with autorelease pools that we have seen previously in our programs are instructions we give to the system in order to correctly set up the autorelease pool machinery.

The memory management technique used by Cocoa and introduced in this chapter is commonly known as *reference counting*. You will find complete explanations of the Cocoa memory management system in more advanced books or articles (see Chapter 15). Some Apple engineers have recently hinted that Apple is working on a new memory management model for Cocoa, known as *automatic garbage collection*. This model is much more powerful than the current one, much easier to use and less error prone. At the time of this writing, however, there is no guarantee when or if this new technology will be completed and made available by Apple.

CHAPTER 15

SOURCES OF INFORMATION

The modest goal of this book was to teach you the basics of Objective-C in the Xcode environment. If you have been through this book twice, and tried the examples with your own variations thereof yourself, you are ready to learn how to write the killer applications you are looking to create. This book gave you sufficient knowledge to run into problems quickly. As you made it to this chapter, you're ready to exploit other resources, and the ones mentioned below should have your attention.

An important advice before you start writing your code: Don't start right away! Do check the frameworks, because Apple may have already done the work for you, or provided classes that require little work to get at what you need. Also, somebody else may already have done what you need, and made the source code available. So, save yourself time by looking through the documentation and searching the Internet. Your first visit should be Apple's developer site at: <http://www.apple.com/developer>

We strongly recommend that you also bookmark:

<http://osx.hyperjeff.net/reference/CocoaArticles.php>
<http://www.cocoadev.com>
<http://www.cocoabuilder.com>
<http://www.stepwise.com>

The above sites have a large number of links to other sites and sources of information. You should also subscribe to the cocoa-dev mailing list at <http://lists.apple.com/mailman/listinfo/cocoa-dev>. This is a place where you can post your questions. Helpful members will do their best to help you. In return, be polite and do check whether you can find the answer to your question in the archive (<http://www.cocoabuilder.com>) first. For some advice about posting questions on mailing lists, see "How To Ask Questions The Smart Way" at <http://www.catb.org/~esr/faqs/smart-questions.html>

There are several very good books out there about Cocoa development. *Programming in Objective-C*, by Stephen Kochan is targeted at beginners. Some books assume you have at least some of the knowledge you gained from this book. We enjoyed *Cocoa Programming for Mac OS X* by Aaron Hillegass of the Big Nerd Ranch, where he teaches Xcode for a living. We also enjoyed *Cocoa with Objective-C* by James Duncan Davidson and Apple, published by O'Reilly.

Finally, a kind word of caution. You are programming for the Mac. Before you release your program, make sure not only that it is bug free, but also that it looks great and adheres to the Apple human interface guidelines (which are described at <http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIGuidelines/index.html>). Once you have done that, don't be shy to make your program available! Comments from others will help you to refine and expand your program and maintain a focus on quality.

We hope you enjoyed this book and that you will pursue programming with Xcode. Oh, and, please, do not forget chapter 0.

Bert, Alex, Philippe.