OBJECTIVE-C GUIDE FOR DEVELOPERS

Matteo Manferdini

1. The basics **5**

Comments 5

Variables and basic types 6

Operators 8

Object variables 10

A note on prefixes and namespaces 11

2. Branching and decisions **12**

If-else statement 12

Switch statement 13

Ternary operator 14

Relational and logical operators 14

3. Loops **16**

For loop 16

While loop 17

Do-while loop 18

Break and continue 18

4. Object oriented programming **20**

Method calling 20

Class methods and object creation 24

A note about messaging 26

Working with nil 26

Values and literals 27

5. Collections **30**

Arrays 30

Dictionaries 32

Sets 32

Nil values in collections 33

Enumerating colletions 34

6. Creating classes **36**

Headers, importing and forward declarations 37
Properties and instance variables 39
Accessors methods 41
Initialization 44
Designated initializers 46

7. Extending classes 48
Categories 48
Interface extensions 51
Protocols 51
ARC and memory management 54

8. Custom types 60
Defining custom types 60
Constants and enumerations 62
Bitwise operators and bitmasks 65
Structures 67

9. Blocks 70
What blocks are useful for 70
Declaring and using blocks 71
Passing blocks as parameters to methods 74
Capturing the context 75
Blocks and memory management 76

If you know someone that can benefit from this free guide, please feel free to share it with them. Just sent them this link:

http://matteomanferdini.com/objective-c-guide-for-developers

Now, let's dive in…

1. THE BASICS

When, as a developer, you come from another language to Objective-C, you usually want to map the general concepts about programming you already know to this new language that at first might seems obscure. I remember myself being confused by the Objective-C syntax when I started learning it. What at first look might not make sense actually does a lot when you get a grasp of it and in my opinion (and the opinion of many other developers) makes the code much more readable. What at first sight might seem complicated is just something a bit different from what you are used to, thus feeling unfamiliar.

This guide is intended for developers who already know object oriented programming and goes over all the concepts you need to understand in order to use the language and learn programming for iOS and OS X.

It does not assume, though, that you know already C as other guides do. Many times, when people approach the language the first time, they get said to study C first. Although Objective-C is an object oriented extension built on top of C and inherits its the syntax, primitive types, and flow control statements, there are things you probably won't ever use while writing Objective-C code or do it in very rare and specific occasions. So I believe that Objective-C and the parts you need from C can be learned along to each other. This guide integrates all the parts of C you need to know to start using the language, while ignoring the rest.

COMMENTS

Comments in Objective-C come directly from C and are similar to what you find in other languages. One line comments are written like this:

```
// This is a single line comment.
```

Everything after the // is ignored by the compiler until the end of the line. They also work after instructions:

```
int counter = 0; // This is a counter.
```

C allows also multiline comments, which some languages don't have

```
/*
This is a multiline comment
which spans more than one line.
*/
```

VARIABLES AND BASIC TYPES

As in many other languages, you need to store your values and object inside variables. In some languages you can just use variables freely when you need them. Moreover you can change the content of the variable to anything you want: an integer, a decimal number, a string or an arbitrary object.

This does not happen in C. Variables are typed, which means that you have to declare the type of data a variable will contain and you won't be able to change it later. Variable declarations in C look like this:

```
int variable1 = 7;
float variable2 = 3.14;
char variable3 = 'm';
```

Variable names can contain digits but cannot start with them and cannot include spaces, but can include capital letters (used for camel case) or the underscore character _

As you might have noticed in the example above, each instruction in C ends with a semicolon, which is required. Many languages do to, but some don't, so this might be new for you. You will learn fast to end instructions with semicolons.

Take the habit of always initializing your variables to a specific value, even 0. In some languages variables are always initialized to 0 if no other value is specified. This does not happen in an Objective-C method, where non initialized variables will contain garbage and cause strange behaviours in your programs.

These are the most common types of C used in Objective-C programs.

Type	Description	Examples
`int`	integer numbers, including negatives	`0, 105, -12`
`unsigned int`	positive integers	`0, 27, 315`
`float, double`	floating point decimal numbers	`1.61, -17.375`
`bool`	boolean values	`true, false`
`char`	single text character	`'a', 'F', '?'`

If you need bigger numbers that the **int** type can contain, there are the **long** and the **long long** types, with the related unsigned types.

Now, the fact is that when you look at Objective-C code, you rarely find these basic types, which some developers and Apple SDKs still use, while you find others more. Here are some common ones you will find often:

```
NSInteger, for integer numbers
NSUInteger, for positive integers
CGFloat, for floating point decimal numbers, used for graphical values like
drawing coordinates or sizes of graphical objects
BOOL, for booleans, which uses YES and NO instead of true and false
```

The reason these types where created was to make transition from 32 bits to 64 bits architectures easier. C types map to a specific sizes in memory, while the types above change regarding to the architecture the code is compiled for. Objective-C also has its own **BOOL** type (the reason for it is a bit more complicated).

Try to use the latter types when you can and pay attention to which types are used and returned by APIs, or you might end with weird results if values are truncated when switching between the two types.

C allows also to define other kind of simple types, a practice that is used extensively in Objective-C. We will see these types later.

OPERATORS

Like any other language, C has its own arithmetic operators to work with the numeric values stored in the variables (which are probably the same as the ones you already know).

The arithmetic operators of C are the following:

Operator	Description	Syntax
`+`	addition	`a + b`
`-`	subtraction	`a - b`
`*`	multiplication	`a * b`
`/`	division	`a / b`
`%`	module (integer remainder of division)	`a % b`

Arithmetic operators follow standard math operator precedence, so multiplication, division and modulo have precedence over addition and subtraction.

C has also increment and decrement operators:

Operator	Description	Syntax
`++`	increment	`a++ or ++a`
`--`	decrement	`a-- or --a`

Increment and decrement operators have precedence over the arithmetic operators. Pay particular attention to these operators. When used in isolation, they are equivalent:

```
someVariable++;
```

produces the same result as

```
++someVariable;
```

The difference comes out when you use them inside another expression. When put before the variable, they increment or decrement the value *before* it is used. When put after, they increments or decrement the value *after* using it.

This example explains how this works:

```
int  x;
int  y;

// Increment operators
x = 1;
y = ++x; // x is now 2, y is also 2
y = x++; // x is now 3, y is 2

// Decrement operators
x = 3;
y = x--; // x is now 2, y is 3
y = --x; // x is now 1, y is also 1
```

We have already seen the assignment operator **=** which might be different in the languages you are used to. That is not the only assignment operator. The arithmetic operators (with the exception of the modulo) attach to the assignment and produce another four assignment operators, which are used as a shorthand for incrementing, decrementing, multiplying and dividing the content of a variable:

```
Operator   Example    Shorthand for

+=         a += 10;   a = a + 10;
-=         a -= 7;    a = a - 7;
*=         a *= 3;    a = a * 3;
/=         a /= 5;    a = a / 5;
```

Unlike in other languages, in Objective-C there is no operator overloading. This means that these operators (and the other operators we will see later) only work with basic types and they do not work with objects (to concatenate strings, for example).

OBJECT VARIABLES

Let's have a first look at objects. Objective-C is an object oriented language after all, so we need variables to contain objects. The generic type for an object variable is **id**:

```
id myObject = nil;
```

The **id** type means that the variable can contain any type of object. The **nil** value is equivalent to what is called **null** in some other languages (and in C) and means the variable is empty and does not contain any object. Objective-C is a dynamically typed language, so we can declare variables of type **id** and use them for any object. We can then call any method we want on that object, regardless of the fact that that method might exist or not. This might be useful in some cases, but the majority of the time we want the compiler to check for our mistakes as soon as possible and give variables a defined type.

Object variables with a specific type are declared in a slightly different way:

```
NSString *aString = nil;
```

Now what does that * mean? Well, strictly speaking that means that this variable is a pointer, a thing that makes many programmers recoil in horror. Pointers are one of those things coming from C you don't want to deal with. Many modern languages don't have pointers, so this might be a concept you are not familiar with. Although in the future, when you will be more familiar with the language, it might be useful to know what a pointer is, this is something you don't have to worry about for now. You can write Objective-C without knowing this concept yet.

When programming I rarely think in terms of pointers. Just remember this simple rule: you need a * when you declare an object variable and you don't when you use it.

A NOTE ON PREFIXES AND NAMESPACES

As you probably noticed, all types in Objective-C have some prefix at the beginning of the name (**NSString**, **CGFloat**, etc). The reason is that Objective-C does not have namespaces. To avoid name collisions (different objects ending up having the same name) Apple decided to prepend a two letters prefix to its types and classes. The two letter are generally coming from the framework declaring the type: UIKit classes and types have a UI prefix (**UIView**, **UILabel**, etc.), Core Graphics uses the CG prefix (**CGFloat**, **CGRect**, etc.) and the Foundation and AppKit frameworks (the latter present only on the Mac) use NS (**NSString**, **NSArray**, etc.). Why NS? Because these libraries were first developed at NeXT for the NeXTSTEP operating system.

It is a good practice to you add a prefix to your classes and types too. Even if you are not developing a framework but a normal app (which is often the case), just use letters from the app name to be sure to avoid collisions to code you might add later from a different source (like some open source code you might use in your project). There is no strict check from the compiler, so you can omit it, but it's better not to.

2. BRANCHING AND DECISIONS

Let's now see how flow control works in Objective-C. Flow control is another part that the language inherits directly from C, so the structures are the same. Decisions and branching happen, in Objective-C, through three branching statements.

IF-ELSE STATEMENT

The main statement to perform decisions in Objective-C is the **if-else** statement

```
if (expression) {
    if-body
} else {
    else-body
}
```

The ***if-body*** is executed only if the expression to **YES**, **true** or any non zero value. Otherwise the ***else-body*** is executed. The **else** clause can be omitted like in many other languages. In the case of just one instruction in the ***if-body*** or ***else-body***, the respective curly braces can be omitted too. If you come from a language that has no curly braces, there are many different styles that state where to place them.

I personally use the K&R style now, that in my experience seems to be quite widespread in the Objective-C community, but I've moved through many different styles during my programming career. The only solid advice I can give you regarding this topic is to be consistent: pick one style and always use it everywhere.

If you want to use more than one condition, the if-else statement allows multiple branches. This is usually the case in many languages, although it's not possible in a few.

```
if (expression1) {
    if-body
} else if (expression2) {
    else-if-body
} else {
    else-body
}
```

The **else if** clause can be repeated as many times as needed.

SWITCH STATEMENT

When you have multiple choices you can use, as we saw, the if-else statement with as many branches as needed. If the expression evaluates to an integer, though, C offers another branching facility, the switch statement

```
switch (expression) {
    case value1:
        case-body-1
        break;
    case value2:
        case-body-2
        break;
    ...
    default:
        default-body
        break;
}
```

The values for the different cases must be constant integers, which means that you can either write a number or a constant in them, but not a variable.

The **break** statements are actually used to exit immediately from the switch statement. They are not mandatory, but pay attention not to forget them, otherwise the execution will continue into the next **case**.

The **default** statement is optional, and is executed when the value of the expression does not match any case. The **break** in the **default** statement is not mandatory either and not necessary unless you put another case after the **default**, which I strongly advise against, since it would be a poor coding practice.

TERNARY OPERATOR

Another branching facility that Objective-C inherits from C is the ternary operator, also known as conditional operator. This is an operator that in some languages might not be available, and it's a shorthand of the **if-else** statement and a **return** statement (which we will see later, but you are probably already familiar with). The ternary operator takes this form:

```
condition ? expression1 : expression2;
```

If the condition evaluates to **YES**, **true**, or any non zero value, ***expression1*** is executed and it's value is returned, otherwise the same happens for ***expression2***. Thus, it is equivalent to the following:

```
if (condition) {
    return expression1;
} else {
    return expression2;
}
```

The ternary operator has the advantage over the **if-else** statement that it can be written inside another expression, like an assignment. Thus this is possible and sometimes used:

```
a = condition ? expression1 : expression2;
```

RELATIONAL AND LOGICAL OPERATORS

To write the conditions for the branching statement and the loops we will see in a short while, C and Objective-C offer the following relational operators:

Operator	Description	Syntax
==	equal to	a == b
!=	not equal to	a != b
>	greater than	a > b
>=	greater than or equal to	a >= b
<	less than	a < b
<=	less than or equal to	a <= b

Pay attention to the **==** operator. A very common mistake, especially when coming from other languages where this would not be allowed, is to use **=** in comparisons instead, which is the assignment operator. This is perfectly legal in C, as any assignment also returns a value that is checked by the **if-else** statement, which checks for 0 and non zero values. So, the program will compile and run perfectly, but probably not behave as you expect. This is a mistake that costs many people a lot of time in debugging. Luckily the latest versions of Xcode might warn you of this common mistake when you build your project, if the proper settings are enabled.

To combine conditions and form compound ones, C offers the common logical operators you find in many other languages:

Operator	Description	Syntax
!	negation (not)	!a
&&	and	a && b
||	or	a || b

Pay attention to the double symbols used in *and* and *or*: C has also the single character versions, which are bitwise operators and not logical ones. We will have a look at these later.

Many languages have a strict boolean notion of true and false. As we have seen, C, and Objective-C, have a more loose notion where zero and non zero values are checked instead. Some old versions of C don't even have the **bool** type. It's certainly beneficial to think in terms of truthiness of the conditions when writing them. It's also useful nonetheless to know a bit of what happens behind the curtains. Objective-C NO and YES values are also mapped respectively to 0 and 1.

This is useful to know for a very common idiom in Objective-C that comes from C. The **nil** value is also mapped to 0, which means that it is equivalent to NO. So, instead of checking if a variable (or value) is equal to **nil**, it's common in Objective-C to check it's negation, which is logically equivalent. So instead of writing

```
if (someObject == nil)
```

we write

```
 if (!someObject)
```

Many times, though, checking for **nil** values is not necessary as it is in other languages. We will see later why.

3. LOOPS

Objective-C loops also come from C. In addition to these Objective-C also introduces a statement to enumerate collections that we will see later when we will talk about those. Looping over collections is much more common in an Objective-C program, but the basic loops are still useful nonetheless.

FOR LOOP

The first loop statement, present in many modern languages as well is the for loop. In C it has the following syntax:

```
for (initialization-expression; loop-condition; update-expression) {
    loop-body
}
```

Please notice that the separators between ***initialization-expression***, ***loop-condition*** and ***update-expression*** are semicolons and not commas.

The for loop is quite flexible and can accept different kind of expressions in it. In it's most common form, it's used like this:

- The ***initialization-expression*** is an expression executed before entering the loop. It's most commonly used to declare and/or initialize an integer variable used to count through the loop.

- The ***loop-condition*** is a boolean condition used to check when to terminate the loop. It is evaluated before each pass through the loop. When the loop condition is false, the loop is terminated. This is usually used to check if the counter declared in the ***initialization-expression*** is over or under a certain value.

- The ***update-expression*** is an expression that is evaluated at the end of each pass through the loop, after the loop body has been executed, and just before looping back to evaluate the ***condition-expression*** again. It's usually used to update the value of the counter variable declared in the ***initialization-expression***.

In it's most common incarnation, the for loop takes the following form:

```
for (NSUInteger index = 0; index < someValue; index++) {
    loop-body
}
```

Where ***someValue*** is a variable or a constant with some positive integer value. In this way the loop iterates exactly ***someValue*** times.

WHILE LOOP

The **for-loop** is normally used to iterate a fixed number of times. If we want to loop until a certain condition is met, then the while loop is normally used.

```
while (condition) {
    loop-body
}
```

The ***condition*** is checked at the beginning of every iteration and the loop ends when the condition evaluates to a false value. If it evaluates to false on the first iteration (before executing the ***loop-body*** once), then the ***loop-body*** is not executed.

DO-WHILE LOOP

If we want the loop to iterate at least one time, we have the **do-while** loop.

```
do {
    loop-body
} while (condition);
```

In this case the order of ***loop-body*** execution and ***condition*** evaluation is reversed. First the loop goes through one iteration and then checks the ***condition***. If it evaluates to true, the loop continues, until it evaluates to false.

As for the branching statement, the curly braces can be omitted in any loop if the loop body only has one instruction.

BREAK AND CONTINUE

There are two statements that are used to alter the flow of the loop structures: **break** and **continue**. They can be used in any loop.

The **break** statement is used when you want to terminate the loop prematurely, for reasons often different from the loop condition and at any point during an iteration, so you only execute part of it. This illustrates how usually it's used:

```
while (loop-condition) {
    some-instructions
    if (exit-condition) {
        break;
    }
    some-other-instructions
}
```

If the ***exit-condition*** evaluates to true during any iteration, the loop exits without executing ***some-other-instructions***, and execution continues after the loop. You can of course use more than one break checks in the loop body.

Sometimes you want to just skip the rest of the current iteration like with the **break** statement, but without exiting the loop. The **continue** statement is used for this specific reason.

```
while (loop-condition) {
    some-instructions
    if (skip-condition) {
        continue;
    }
    some-other-instructions
}
```

When the ***skip-condition*** is true in any iteration, ***some-other-instructions*** are not executed, but the loop continues to the next iteration, checking again the ***loop-condition*** and executing the ***loop-body*** again.

The **break** and **continue** statements can also be mixed inside the same loop, to have a more complex control over the flow structure.

4. OBJECT ORIENTED PROGRAMMING

METHOD CALLING

Let's start with method calling, which is what confuses people the most when they approach the language. The reason for this confusion is the syntax Objective-C uses for method calling, which is different from what you usually find in the majority of languages out there.

Let's first have a look at a method without parameters. The most common way to call a method on an object in other languages, such as Java, Ruby, C#, Javascript, etc. is to use the dot notation:

```
object.method();
```

Where object is an object variable and method is the name of the method we are calling. Objective-C wraps a method call inside square brackets instead:

```
[object method];
```

To find out what methods a class provides we usually look at its declaration or at the documentation. This method is declared in this way:

```
- (void)method;
```

This method does not return any value because it's return type is **void**, and does not accept any parameter, since none is declared. Please notice the **-** sign at the beginning of the line, which is important and we will see why later.

An example of such a method declaration is the following method of **NSMutableArray** (which we will see in more detail when we will talk about collections).

```
- (void)removeAllObjects;
```

Which we call in this way:

```
[mutableArray removeAllObjects];
```

Where mutableArray is a variable of type **NSMutableArray**. Methods can, of course, also return values, in which case the method declaration usually indicates the type of the returned value in place of **void**. For example, this is the method to get the length of a string:

```
- (NSUInteger)length;
```

and it's called like this:

```
NSUInteger stringLength = [aString length];
```

Where Objective-C becomes a bit more confusing is when we add parameters to method calls. In other object oriented languages, parameters of methods or functions are enclosed in parentheses:

```
object.method(parameter);
```

Objective-C uses colons instead of parentheses:

```
[object method:parameter];
```

The declaration of such a method is the following:

```
- (return-type)method:(parameter-type)parameter;
```

For example, an **NSString** object has a method that returns a new string obtained by appending another string to it:

```
- (NSString *)stringByAppendingString:(NSString *)aString;
```

Which would be called in this way:

```
NSString *concatenatedString = [aString stringByAppendingString:anotherString];
```

Now the tricky part. How do we call methods with multiple parameters? In other languages this is usually done by adding the second parameter in the parentheses and separating parameters with commas:

```
object.method(parameter1, parameter2);
```

This is where Objective-C becomes the most confusing to newcomers. For methods with multiple parameters, the name of the method is split in multiple parts and the parameters are put in between these parts, using colons. So a multiParameter method call looks as follows:

```
[object methodPart1:parameter1 methodPart2:parameter2];
```

And this is how the declaration looks like:

```
-(return-type)methodPart1:(parameter-type1)parameter1 methodPart2:(parameter-
type2)parameter2;
```

Let's look at an example to make it more clear. **NSString** has a method to replace a substring with another string. This is its declaration:

```
- (NSString *)stringByReplacingOccurrencesOfString:(NSString *)target
withString:(NSString *)replacement
```

and this is how you call it:

```
NSString *newString = [aString stringByReplacingOccurrencesOfString:target
withString:replacement];
```

The reason why methods in Objective-C get split into pieces is because Objective-C values clarity and readability of code a lot. In this way each method incorporates also an explanation of what parameters it takes, so every time you see a method call, you know exactly what the method needs without looking at the documentation. For this reason Objective-C does not have a feature that other object oriented languages have, which is method overloading where the same method can take different types of parameters. The reason is that when a method takes different parameters in Objective-C, it also gets a different name, so method overload is not needed.

There is a dirty trick used by some developers to make method calls look more like what you find in other languages. The compiler allows you to declare a method by omitting all the parts of its name except the first one, as follows:

```
- (return-type)method:(parameter-type-1)parameter1 :(parameter-
type-2)parameter2;
```

Which yields this method call:

```
[object method:parameter1 :parameter2];
```

I am showing you this only to prepare you to what you might find in some code bases that are not well written. I strongly advise you to never use this trick, since you'll be fighting against the language, which was designed in this way specifically, and the entire developer community that uses the language every day.

CLASS METHODS AND OBJECT CREATION

We've seen how to call methods on objects, but we still didn't talk about how we create those objects in the first place. This is done through a feature which is not usually present in other programming languages: in Objective-C classes are objects too. This means that you can call methods on classes as you do on objects.

So, in Objective-C there is no *new* operator to instantiate objects. You create instances by calling method on classes. The typical use of class methods is to declare factory methods. Factory methods are used to create objects as an alternative of allocating objects directly. So, for example, you can create a new string like this:

```
NSString *myString = [NSString string];
```

Calling a method on classes, as you can see, works the same way as calling methods on objects. What is slightly different is the declaration of the method:

```
+ (NSString *)string;
```

where the only difference is that the line starts with a **+** instead of a **-** as we saw for object method declarations.

Factory methods are a way to create objects, but not the only way. The other one is to allocate directly the object by calling the **alloc** class method. This is a method of the class **NSObject**, which is the root class from which all the Objective-C classes inherit. This method takes care to allocate the memory to contain the object, and it also cleans the memory so it won't contain garbage.

After allocating the object you have to initialize it so that the object has all it's variables and properties set to proper initial values. You usually do that by calling one of the the initializers on the newly created instance. The common idiom to create and initialize an object is the following:

```
NSString *myString = [[NSString alloc] init];
```

As you can see in this example, method calls can be nested, as in any other language and this is always done when creating an object. This idiom for creating objects is so common that actually **NSObject** has a shorthand class method for it:

```
NSString *myString = [NSString new];
```

Pay attention, though, that the **init** method might return a different object than created by the **alloc**. For this reason it's best practice to nest the calls or to use the **new** method instead. So, do not do this:

```
// Never do this!
NSString *myString = [NSString alloc];
[myString init];
```

Initialization methods often have parameters, like any other method. In this case you have to call **alloc** and cannot use **new**:

```
NSNumber *aNumber = [[NSNumber alloc] initWithFloat:7.0];
```

A NOTE ABOUT MESSAGING

Now that we've seen how method calling works in Objective-C, we can have a little look at how this works behind the scenes to grasp a bit more the power of the language.

Method calling in Objective-C is different from other languages, where calls are decided by the compiler when the application binary is built. Objective-C method resolution is completely dynamic and it's done at runtime. For this reason in Objective-C we speak about sending messages to objects instead of calling methods (or if we use the latter, we are aware of the difference).

When a message is sent to an object, the runtime checks if that object responds to that message, resolves dynamically its address and executes it. The advantage is that we get the capability to check if a feature exists at runtime and we can decide wether to use it or not. Moreover we can even change our classes at runtime, adding or removing methods and so on. These are very powerful and advanced concepts, so it's not needed to know or use them when learning the language. Just use message passing as method calling in other languages. But it's good to spend a couple of words about it now, so that you'll keep in mind in the future.

WORKING WITH NIL

We have already seen the **nil** value used to represent empty object variables. There is one special thing about the **nil** value in Objective-C which is different from many other languages. You can safely call any method on a **nil** object. The result will just be no operation at all. If you expect a return value from the method, that will be **nil** for methods that return objects, **0** for methods that return numbers and **NO** for methods who return booleans.

This is different from other languages where trying to call a method on a null value results in an exception or a crash. This relieves the developer from many checks for **nil** values and makes the code more cleaner.

There are still some cases where you actually want to check wether an object is **nil** or not. You can, of course, use the relational operators. This check, though, is commonly done in Objective-C with a much more common idiom where you actually only provide the variable to the **if** clause or negate it. Thus:

```
if (someObject != nil) {
    ...
}
```

becomes

```
if (someObject) {
    ...
}
```

and

```
if (someObject == nil) {
    ...
}
```

becomes

```
if (!someObject) {
    ...
}
```

VALUES AND LITERALS

C offers already basic types for values you need in your code, like numbers, characters and booleans. Objective-C adds some value objects on top of that.

We have already seen one type of value object, **NSString**, which is used for strings manipulation instead of using the standard arrays of characters offered by C, which are more complicated to use. We already saw how we can allocate a **NSString** using either one of the factory methods or the direct allocation with initializers. Objective-C offers a third way to create new values: *literals*.

Literals in Objective-C are obtained by prepending the @ symbol to a C value and are used to create value objects and collections in a more concise way than the ones we already saw. String literals use double quotes, so to create a new string, we can write:

```
NSString *myString = @"This is a string";
```

Unlike their counterparts in other languages, many objects in Objective-C are immutable, which means that they cannot be changed once you create them. This is a powerful concept which avoids many bugs introduced by code that changes the content of shared objects. Having immutable objects ensures that you can safely pass them around as parameters or return values without worrying for their content to change. It's a concept that is advisable to introduce in your classes as well, when possible. **NSString** is one of the classes of which the instances are immutable and the class offers many methods to derive new strings without changing the content of the original ones. If you really need to change the content of a string for some reason, there is the **NSMutableString** subclass of **NSString** that actually allows this.

Objective-C offers also a class for booleans and other basic numeric values: **NSNumber**. This is needed for collections, for example, which can contain only objects and not basic types. **NSNumber** has a list of factory methods and initializers to create instances from any type of C number. Moreover, Objective-C literals also work with **NSNumber**, thus making the creation of numbers much more concise and readable. These are examples on how you do it:

```
NSNumber *integerNumber = @42;
NSNumber *unsignedNumber = @42u;
NSNumber *longNumber = @42L;
NSNumber *floatNumber = @3.14f;
NSNumber *doubleNumber = @3.1415926535;
NSNumber *charNumber = @'M';
NSNumber *boolNumber = @YES;
```

NSNumber instances are immutable too and there is no mutable subclass. Just create new objects when needed.

Literals work also on boxed expressions, so you can create an **NSNumber** from the result of any expression. Boxed expressions are enclosed in parentheses:

```
NSNumber *expressionNumber = @(3 * 7);
```

While some other languages offer unboxing of values, in Objective-C there is none. So, if you need to calculate expressions between **NSNumbers**, you have to retrieve their basic values first using the appropriate methods.

5. COLLECTIONS

The C language offers arrays to hold collections of scalar values and objects, but those are usually cumbersome and complicated to use, especially if you want dynamic collections that change in size. For this reason C arrays are almost never used in Objective-C, which offers instead more advanced types of collections.

ARRAYS

The first collection we visit is **NSArray**, which is used for ordered collections of objects. The literal for creating arrays uses square brackets:

```
NSArray *someArray = @[firstObject, secondObject, thirdObject];
```

Before the introduction of literals, arrays were commonly created using factory or init methods and that is still possible. Literals, though, are a more concise and preferred way. This also prevents some errors: if you use the **NSArray** creation methods that take a variable number of arguments, you have to terminate that list with a **nil** value. The problem is that it's possible to truncate the list of items unintentionally if one of the provided values is accidentally **nil**. The new literal syntax does not have this problem.

The objects contained in an array don't have to be all of the same type. Accessing array objects at specific indexes is done through subscripting, like this:

```
id firstArrayItem = someArray[0];
```

Pay attention that if you request an invalid index out of the array boundaries you will get an exception at runtime which will crash your app.

Collections are another set of objects in Objective-C that are immutable. If you need to perform an operation that actually requires the collection content to change, this is sometimes done through methods that actually return new copies of the collection instead of modifying the original one.

For example, if we want to add an object at the end, the **arrayByAddingObject:** method returns a new array with the new item appended at the end:

```
NSArray *newArray = [someArray arrayByAddingObject:newObject];
```

There are cases where you actually need to change the content of a collection. In this case you can create an instance of it's mutable subclass. You cannot create an instance of a mutable class using literals though, so you have to do it using the other object creation methods. But you can still use subscripting to change the content:

```
NSMutableArray *mutableArray = [NSMutableArray arrayWithObjects:firstObject,
secondObject, thirdObject, nil];
mutableArray[0] = newFirstObject;
```

Be aware that you cannot extend the array using subscripting. An invalid index will still throw an exception. If you need to make the array bigger use the **addObject:** method.

You can also get a mutable array from an immutable one with the **mutableCopy** method:

```
NSMutableArray *mutableArray = [anArray mutableCopy];
```

And you can also switch back to an immutable instance when you need to pass a copy around:

```
NSArray *immutableArray = [NSArray arrayWithArray:mutableArray];
```

DICTIONARIES

Another very common collection class is **NSDictionary**. Dictionaries store key-value pairs and are usually called hash maps in other languages. It's best practice to use a string as a key in a dictionary, but it can actually be any object that supports the **NSCopying** protocol (we will see what protocols are later).

Dictionaries also have factory methods to create them, but again, the preferred way is by using literals. Dictionary literals use braces:

```
NSDictionary *dictionary = @{@"anObject" : someObject,
                              @"aString" : @"This is a string",
                              @"aNumber" : @7,
};
```

You can query dictionaries using subscripting too, passing a key as a subscript:

```
NSNumber *storedNumber = dictionary[@"aNumber"];
```

NSDictionary also has a mutable subclass called **NSMutableDictionary**. Unlike **NSMutableArray**, you can use subscripting not only to replace items but also to add new objects to the mutable dictionary. Similarly to arrays, you can switch back and forth between immutable and mutable dictionaries using the **mutableCopy** and **dictionaryWithDictionary:** methods.

SETS

A less used but still useful collection is **NSSet**. Sets are unordered collections of objects. Their main advantage is that they provide performance improvements over arrays when testing for membership. So if you need to know if an object is in a collection, **NSSet** might be a good choice. Sets have no literal for creation, so they need to be created with factory methods or initializers:

```
NSSet *aSet = [NSSet setWithObjects:@"This is a string in a set", @7, anObject, 
nil];
```

As for other collections, the objects contained in a set don't have to be of the same type. **NSSet** instances are also immutable and there is a mutable subclass called **NSMutableSet**. Objects in sets can be present only once. If you need to put an object in a set more than once, there is the **NSCountedSet** subclass of **NSMutableSet**.

NIL VALUES IN COLLECTIONS

It's not possible to insert **nil** values inside of collections. If you try to do that, you'll get an exception which will, usually, crash your app (more on exceptions later). So you have to check wether an object is **nil** before trying to insert it into a collection.

There are cases, though, where you actually might want to do this on purpose. For example, you might want to do it so signal that a position in an array is empty. Since you cannot insert a **nil** in a collection, Objective-C offers the **NSNull** class:

```
NSArray *array = @[anObject, @7, [NSNull null]];
```

Beware that **NSNull** does not behave like **nil**, since it's actually a real object. If you try to call a method that does not exist on a **NSNull** object, you will get an exception, so you have to check before calling methods on objects coming from collections that contain **NSNull** values. **NSNull** is a singleton class, which means that its **null** method will always return the same instance. This means that you can check whether an object in a collection is equal to the shared **NSNull** instance using the equality operators:

```
id objectFromArray = [anArray objectAtIndex:0];
if (objectFromArray != [NSNull null]) {
    ...
}
```

ENUMERATING COLLETIONS

When you need to go through all the elements of a collection, Objective-C offers many different ways to do so. C loops work fine of course, but they are prone to programming errors, so it's advised not to use them. The preferred method to go through a collection is using fast enumeration which has this form:

```
for (Type variable in collection) {
    ...
}
```

So, to enumerate all the object in an array we would write:

```
for (id object in array) {
    ...
}
```

Fast enumeration works with dictionaries too. In this case the loop iterates through the keys and you have to retrieve the values yourself:

```
for (NSString *key in dictionary) {
    id object = dictionary[key];
    ...
}
```

You can use **break** and **continue** in fast enumeration loops as you would use them in a normal loop. If you need to go through a collection in reverse order, you can use an **NSEnumerator** object:

```
for (id object in [array reverseObjectEnumerator]) {
   ...
}
```

Beware that you cannot change the content of a mutable collection (adding, replacing or removing objects) while you are enumerating it, or you will get an exception. A quick fix to this is to enumerate through a copy of it:

```
for (id object in [aMutableArray copy]) {
    ...
}
```

6. CREATING CLASSES

The declaration and the implementation of a class are separated in Objective-C. To declare a new class we first declare the interface using the **@interface** directive:

```
@interface ClassName : SuperClass

@end
```

By convention, class (and type) names start with a capital letter (and don't forget prefixes as we discussed before. I will omit them in the examples here for clarity, but I advise you to follow the conventions when writing your code).

After the colon the superclass from which our class inherits must be specified. The root class in Objective-C is **NSObject**. When one object encounters another object, it expects to be able to interact using at least the basic behavior defined by the **NSObject** class. **NSObject** offers a lot of basic behaviors like the **+alloc** and **-init** methods we have seen in the paragraph about creating objects.

The implementation of the class goes, intuitively enough, in the **@implementation** part:

```
@implementation ClassName

@end
```

Methods are declared in the **@interface** of the class:

```
@interface MyClass : NSObject

- (void)doSomething;

@end
```

and get implemented in the **@implementation** part. The implementation of the method goes inside curly braces:

```
@implementation MyClass

- (void)doSomething {
    ...
}

@end
```

When an object needs to send a message to itself, you can do so from within a method by using the **self** keyword.

```
- (void)someMethod {
    ...
    [self someOtherMethod];
    ...
}
```

To instead access the implementation of a method in the superclass (when you, for example, override one of its methods) you use the **super** keyword:

```
- (void)someMethod {
    ...
    [super someMethod];
    ...
}
```

HEADERS, IMPORTING AND FORWARD DECLARATIONS

In Objective-C the interface and the implementation of a class are usually kept in two different files, unlike other languages where everything is kept in a single file. The interface goes in a header file with .h extension, while the implementation goes in a file with .m extension.

It is possible to declare interface and implementation of classes in only the .m file, which can contain more than one class. This is used if you want to declare an internal class that is used only in a single class implementation and not anywhere else. While some other languages have special constructs for this, in Objective-C you accomplish it by just putting the new class in the implementation file.

To reference class header files, Objective-C uses the **#import** directive. Always use this instead of the **#include** coming from C, also if you are importing C files, because **#import** will check automatically for double inclusion of headers so you will never have problems with recursive inclusion. **#import** uses angular brackets **< >** for global inclusions and double quotation marks ” “ for local ones. So if you are including a framework you do it like this:

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

while for your own classes in the project:

```
#import "MyClass.h"
```

Sometimes you need to have circular dependencies in some classes, where a class A needs a class B, and B needs A.

```
@interface ClassA : NSObject

- (ClassB *)methodThatReturnsAnInstanceOfClassB;

@end

@interface ClassB : NSObject

- (ClassA *)methodThatReturnsAnInstanceOfClassA;

@end
```

This code cannot compile, because **ClassA** needs to know about **ClassB**, which at that point still has not been declared. This happens also with Objective-C protocols, as we will see later in this guide. You can use a forward declaration to avoid this problem, with the **@class** keyword:

```
@class ClassB;

@interface ClassA : NSObject

- (ClassB *)methodThatReturnsAnInstanceOfClassB;

@end

@interface ClassB : NSObject

- (ClassA *)methodThatReturnsAnInstanceOfClassA;

@end
```

Forward declarations are often used in header files also to avoid importing the header of another class, which would import all the declarations present in the latter (classes, protocols, categories, types and constants) which, in turn, would also spread in all the files that import the header that includes it.

PROPERTIES AND INSTANCE VARIABLES

We now need to know how to store the internal state of an object. It is a best practice in Objective-C to use properties for this purpose, which are declared in the interface of the class.

```
@property Type propertyName
```

Let's use the typical example for an class: the Person class. If we want to declare a class to stores the first and last name of people, as well as their age, this would be its declaration:

```
@interface Person : NSObject

@property NSString *firstName;
@property NSString *lastName;
@property NSUInteger age;

@end
```

We can now use the dot syntax present in many languages to read or assign values to properties:

```
Person *person = [Person new];
person.name = @"Matteo";
```

When you declare a property, there are many things happening behind the scene. In the first place, a corresponding instance variable is automatically synthesized by the compiler for each property you declare to store the value of the property. Although it's best practice to access the properties through the dot syntax, you might want to access these instance variables directly (for example, as we will see, to change a read-only property, in initializers, deallocation or custom accessors, or to avoid triggering key-value coding notifications). The name of the synthesized instance variable is the name of the corresponding property prefixed with an underscore.

```
(void)someMethod {
    _firstName = @"Some other name";
}
```

You can change the name of the instance variable synthesized for a property if you want, declaring it explicitly with the **@synthesize** keyword in the class **@implementation** section:

```
@implementation ClassName

@synthesize propertyName = differentInstanceVariableName;

@end
```

For example, we can rename the instance variables in our **Person** class

```
@implementation Person

@synthesize firstName = ivar_firstName;

@end
```

You can also declare your own instance variables without a relative property. You can do so either in the interface of a class, to make these variables visible to subclasses:

```
@interface ClassName : SuperClass {

    Type _myInstanceVariable;
}

...

@end
```

or in the implementation of the class:

```
@implementation ClassName {
    Type *_myInstanceVariable;
}

...

@end
```

ACCESSORS METHODS

Another thing that happens behind the curtains when declaring properties is that the compiler synthesizes automatically corresponding accessors methods. Objective-C is not like other languages where properties access directly the memory where the value for a property is stored. What happens instead is that every time you use a property with the dot syntax, that is translated into the corresponding method and a message is sent to the object, in the exact same way as if you were calling a method.

The method used to access the value, called the *getter* method, has the same name as the property. The method used to set the value, called the *setter* method, instead starts with the word "set" and then uses the capitalized property name.

So, in our **Person** class, when we declare the **firstName** property, these two methods are added to the class:

```
- (NSString *)firstName;
- (void)setFirstName:(NSString *)firstName;
```

You can use these methods yourself to read and write the value of the property, which is what happens anyway when you use the dot notation.

Because of this automatic translation I have seen some developers mistake some method call for a property, and also Xcode (the IDE used to develop for iOS and Mac OS X) will autocomplete it anyway. I've seen this a lot, for example, with the count method of arrays.

```
NSUInteger numberOfItemsInTheArray = someArray.count;
```

The **NSArray** class has no property called **count**, but only a method. But this line of code will work anyway because of how properties work in Objective-C. The correct syntax would be:

```
NSUInteger numberOfItemsInTheArray = [someArray count];
```

Which is more consistent because the former implies the existence of a property that does not exists.

The accessor methods are synthesized automatically by the compiler, but you can provide your own ones if you want. This is one of the cases I mentioned where you actually need to access the instance variables behind properties directly.

For getter methods a common implementation is:

```
- (Type)property {
    return _property;
}
```

And for setter methods:

```
(void)setProperty:(Type)property {
    _property = property;
}
```

These are basic implementation that only read and write values of a property, but you might want to start from here to add further behavior to the accessors.

Pay attention that if you implement both the accessor methods for a property yourself, the compiler will assume that you are taking control of the property and won't synthesize the instance variable for you. If you still need it, you need to declare it yourself in the class implementation:

```
@synthesize property = _property;
```

Sometimes you might want to declare a property as **readonly**. This happens when you create a property that is derived from other values and cannot be set, or simply when you want to have immutable objects or properties not changeable from an external object. If, for example we want to add a **fullName** property to our **Person** class, composed by the first name followed by the last name, we will declare this property as **readonly**:

```
@property (readonly) NSString *fullName;
```

We then need to provide a custom accessor method for it:

```
- (NSString *)fullName {
    return [NSString stringWithFormat:@"%@ %@", self.firstName, self.lastName];
}
```

For readonly properties, providing the getter method is enough to prevent the compiler from synthesizing a corresponding instance variable.

The opposite of **readonly** is **readwrite**. Usually there's no need to specify the **readwrite** attribute explicitly, because it is the default, but there is one case in which you might want to do it anyway. Sometimes you might want to declare a property as **readonly** for external objects only, but still be able to change its value from inside the object itself. You can do so by accessing the instance variable directly, but you might prefer to still use the property internally, for example to trigger Key-Value Coding notifications (which we will see later). In that case you can declare the property as **readonly** in the interface of the class and then redeclare it as **readwrite** in the interface extension, which we will see later.

INITIALIZATION

As we have seen previously, while other languages have constructs to create and initialize object, in Objective-C an object is created with the **+alloc** method and initialized though an initializer, often the **-init** method or another initializer that takes some parameters needed at the object creation moment.

All initializers in Objective-C start with the init word and have a return type **id**. As a rule an initializer should always be called in a nested call with the **+alloc** method:

```
MyClass *object = [[MyClass alloc] init];
```

This is done because the initializer might not return the same object that was created with the **+alloc** method. There are different reasons for this to happen: a class might be a singleton, thus allowing only one instance of it to exist at any time. In this case an init method will return that instance, if it already exists, discarding the one coming from **+alloc.**

If an initializer takes some unique identifier as a parameter, the init method might retrieve that object if it exists and return it, again discarding the one created with **+alloc**. This is not a singleton, since many instances of the class are allowed, but still specific instances are unique.

When you are writing your own classes it's very likely that you need to implement your own initializers to setup your objects properly. If you don't need any parameter when initializing an instance, you simply override the **-init** method inherited from **NSObject**:

```
- (id)init {
    if (!self = [super init]) {
        return nil;
    }
    ... // Instance variables are set here
    return self;
}
```

The first thing you need to do when implementing an initializer, is to call the initializer of the superclass first. If the result is **nil**, it means that something went wrong with the initialization and you have to return **nil** yourself.

You might have notice that in the condition of the **if** the result of the initializer of the superclass is assigned to **self**. This is again for the same reason: that method might return a different instance, substituting the current one.

The assignment in the **if** condition is actually a syntax that is allowed by C. In C the assignment operator also returns the value that gets assigned, so it can be tested in an condition. I personally consider it a bad programming practice, because it does two things at the same time, hiding one of them (the assignment) and making it a side effect. This is the source of many programming errors, where some developer that wants to check for equality uses the **=** operator by mistake instead of the **==** operator. If you actually make yours the practice of testing assignments directly this kind of errors will be even harder to spot, since they will become invisible to your eyes.

This said, it is idiomatic to do this in an initializer in Objective-C, so I consider this case, and only this case, acceptable. My advice is to avoid it in all other cases.

After the call to **super**, you can and usually initialize the instance variables to their initial value and return **self** at the end of the method. You should always access the instance variables directly from within an initialization method instead of using properties. This is because at the time a property is set, the rest of the object may not yet be completely initialized, creating undefined behavior. Another reason is that properties might trigger Key-Value Coding notifications and cause side effects. Even if you don't provide custom accessor methods or know of any side effects from within your own class, a future subclass may override the behavior and create them.

Keep in mind that this form of initializer where you return **nil** on failure of the initializer of the superclass or return **self** at the end is the most common type but not the only one. As I said before, you might perform additional checks to see wether the initialization can proceed and return **nil** at any point for other reasons, or retrieve some other instance of the class and return that one instead of **self**.

DESIGNATED INITIALIZERS

Sometimes a class provides multiple initializers that take data in different forms. In this case one of the initializers should be chosen as the designated initializer for the class. This initializer has to ensure that all the inherited instance variables are initialized by invoking the designated initializer of the superclass. The designated initializer is typically the one with the most parameters and which does most of the initialization work. The secondary initializers should call this initializer instead of calling the designated initializer of the superclass themselves.

To take our **Person** class as an example again, we can implement an initializer which takes the first name, the last name and the age as parameters, to make sure that we create objects that are immediately populated correctly:

```
- (id)initWithFirstName:(NSString *)firstName lastName:(NSString *)lastName
age:(NSUInteger)age {
    if (!self = [super init]) {
        return nil;
    }
    _firstName = firstName;
    _lastName = lastName;
    _age = age;
    return self;
}
```

Let's suppose that we then want a convenience initializer that takes the full name as a single string where first name and last name are separated by a space, since it might come in this form from some source, like a web service or database. The designated initializer for the **Person** class would then be the one we already implemented and this new convenience initializer would call it:

```
- (id)initWithFullName:(NSString *)fullName age:(NSUInteger)age {
    NSArray *fullNameComponents = [fullName componentsSeparatedByString:@" "];
    return [self initWithFirstName:fullNameComponents[0]
                          lastName:fullNameComponents[1]
                               age:age];
}
```

Please do not take this as a real world example since it lacks all the input validation and makes too many assumptions about people names.

7. EXTENDING CLASSES

CATEGORIES

Objective-C has a very powerful and useful feature that many other languages miss: categories. It's a good programming practice to keep the inheritance hierarchy as shallow as possible, since inheritance it introduces complexity when overriding methods. The common way to do this is to use composition (objects that use other objects) and leave inheritance to cases where it's necessary. For example, instead of subclassing **NSArray**, it's better to write a class that uses a **NSArray** instance internally. Objective-C though offers another alternative to composition through categories, that allow you to add methods to existing classes.

This includes any class, so it means you can also add methods to classes from external frameworks, including the ones provided by Apple. This is very powerful because it does not only mean that you don't need to subclass a class to add behavior to it, but also that the methods you add will be available to its subclasses, which you would not be able to do through subclassing. Moreover you can alter instances used internally by other classes. For example, **UIViewController** objects create their own **UIView** instance if you don't provide one, or **UILabel** objects have their own instance of the **UIFont** class. With categories, your methods will also be available to these instances created by classes you don't own.

The category declaration uses the **@interface** keyword like the class declaration but does not indicate any inheritance. Instead, it specifies the name of the category in parentheses:

```
@interface ClassName (CategoryName)

@end
```

A category then has an **@implementation** section like a normal class, where you put the additional method implementations:

```
@implementation ClassName (CategoryName)

@end
```

A category usually has a .h and .m files like normal classes. The file names are created with by the name of the class and the name of the category separated by a +, in the form **ClassName+Category.h** (or .m).

At runtime, there's no difference between a method added by a category and one that is implemented by the original class.

Let's say for example that we want to add a method to **NSString** to know if a string starts with a capital letter. The declaration of the category would be as follows:

```
@interface NSString (Capitals)

- (BOOL)startsWithACapitalLetter;

@end
```

And this the implementation:

```
#import "NSString+Capitals.h"

@interface NSString (Capitals)

- (BOOL)startsWithACapitalLetter {
    unichar firstCharacter = [self characterAtIndex:0];
    return [[NSCharacterSet uppercaseLetterCharacterSet]
        characterIsMember:firstCharacter];
}

@end
```

You can then call this method on any **NSString** instance, even those coming from literals:

```
NSString *myCapitalizedString = @"This string starts with a capital letter";
if ([myCapitalizedString startsWithACapitalLetter]) {
    ...
}
```

Categories can add methods to classes, but not instance variables. So if you need to add functionality to a class that requires storing some value, the only option you have is to create a new subclass.

In the case of properties we have a partial behavior: as we have seen in Objective-C properties add new accessors methods to a class. This works in categories too, so a category can declare new properties for a class. But a category cannot add new instance variables to a class and this still holds true for properties. This means that properties added through a category are not backed up by instance variables like normal properties are. So, when you add properties through a category, you always have to provide your accessors since the compiler will not synthesize them for you. Moreover they can only reference existing instance variables.

Pay attention not to override existing methods in categories. Although I've seen some developers declare that this is fine, it's not. As per Apple documentation, if there is a name clash with a method in a category, which one will be chosen at runtime is undefined, so you are never sure if your implementation is the one that is going to win. Categories are not a valid way to override methods.

To avoid name collisions when you declare a method on a class you don't own, it's best practice to prepend a prefix to the method name. In this case the prefix is lower case to respect conventions for method names and is usually separated with an underscore.

INTERFACE EXTENSIONS

A special type of category is the class interface extension, also known as anonymous category. The interface extension can only be added to your own classes and the methods it declares are usually implemented in the class own **@implementation** block instead of a separate category implementation. An interface extension is declared without specifying the category name in the parentheses.

```
@interface ClassName ()

@end
```

What is special about it is that, unlike other categories, an interface extension can declare new instance variables and the properties it declares behave like properties declared directly in the class interface (they are backed up by instance variables and the accessor methods are automatically synthesized by the compiler).

Interface extensions are used to declare private information for a class. While other languages have a special keyword for this, Objective-C solves this problem with interface extensions. This allows to have partially private methods and properties for selected classes, by declaring the interface extension in a separate header file which is imported only by those classes. This is how Apple declares its own private API which is not available to other developers.

PROTOCOLS

Sometimes you need to declare a minimum interface that a class needs to implement to interact with another class. A class interface or a category declare methods that are specific to a class, while a protocol declares properties and methods that are independent and can be implemented in many different classes. Other languages have a similar feature to protocols (Java calls them interfaces, which might generate some confusion at first if you are a Java developer, since interfaces are a different thing in Objective-C). When a class conforms to a protocol, it must implement the required methods declared by it.

A very common example is the **UITableView** class. **UITableView** is a class found in iOS to display a vertically scrollable list of items. You stumbled upon one already if you use an iPhone or an iPad, since it's omnipresent. Since **UITableView** is a generic class that is used to display many different kinds of lists, with diverse visualizations for the items, all this information needs to be provided to the table view by some other objects.

A **UITableView** defines two protocols that declare what methods it expects two other classes, called the data source and the delegate, to implement to be able to retrieve the information it needs. Since the protocols are separated, the two classes can also be separated, but they are generally the same class.

The data source implements methods that tell the table view how many items and sections there should be and provides them when the table view asks for them to display them on screen. The delegate provides instead information on the visualization of these items, like the kind and size of views used to represent items (called cells).

Any class (usually implemented by you) can be the data source or delegate of a table view and to do so it needs to conform to these two protocols.

A protocol is declared with the **@protocol** keyword:

```
@protocol ProtocolName

@end
```

Inside the protocol interface you declare the methods that a conforming class needs to implement. It is possible to declare optional methods in a protocol that a conforming class can implement only if it needs to. You do so using the **@optional** directive in the protocol declaration:

```
@protocol ProtocolName

// list of required methods

@optional

// list of optional methods

@end
```

There is also a **@required** directive to switch back to declaring required methods, but it's better not to switch back and forth for the readability of the protocol. If you mark some method as optional, you will have to check if the receiving object implements the method before calling it, or you will get an exception. You check this by using the **respondsToSelector:** method of **NSObject** (so it's available to every class). This method takes a selector as a parameter, which you can obtain with the **@selector**() directive around a method name, in this way:

```
if (object respondsToSelector:@selector(someMethod)) {
    [object someMethod];
}
```

To indicate that a class conforms to a protocol, the protocol name is indicated in angular brackets in the class interface:

```
@interface ClassName : Superclass <Protocol>

@end
```

A class can conform to multiple protocols, which are then comma separated inside of the angular brackets:

```
@interface ClassName : Superclass <Protocol, SecondProtocol, ThirdProtocol>

@end
```

The same syntax is used to declare that a variable or a property contains an object that must conform to one or more protocols:

```
id <protocol-list> variableName;
```

or:

```
@property id<protocol-list> propertyName;
```

In this way the compiler will check that the object stored in the variable or property conforms to the protocol, helping to avoid programming errors.

Protocols can conform to other protocols, to include the methods declared in the latter. You specify this conformance in the same way you do for a class:

```
@protocol ProtocolName <protocol-list>

@end
```

ARC AND MEMORY MANAGEMENT

The approaches to memory management you find in other languages are usually two: either memory management is completely left to the developer (like in C or C++) or is handled by a garbage collector (like in Java, C#, Python or Ruby).

In the first case developer has to know when to allocate and especially release memory "manually", while avoiding to address memory that does not exist yet or releasing still used memory too soon. Both these tasks are tedious and error prone and might lead to crashes, unexpected behavior, or leaks that eventually fill up all the available memory.

In the case of the garbage collector, the developer abdicates the memory management to a process that periodically scans the memory and releases the one that is not used anymore. This relieves a lot of pain, so it has become the preferred way in modern languages, but still has some pitfalls. Since to know what parts of memory are used this the garbage collector looks at all the references in the object that are in memory at a given time, the developer has to pay attention not to create reference cycles between object, where two objects reference each other and the memory is not released even if those two object are not referenced anywhere else.

Objective-C comes from a history of semi automatic memory management. Apple used for a long time an in between approach, called reference counting. Reference counting works this way: whenever some objects needs to keep a reference to another object, it retains it. Retaining an object increases a count of references to the object by one. When the object is not needed anymore, object that retained it have to release it. Releasing decrements the count by one. When an objects reaches a retain count of 0, it gets removed from memory by the runtime.

Retaining and releasing are still responsibility of the developer and, if done wrong, they still lead to accessing deallocated memory (which usually causes crashes) or memory leaks. The benefit is that reference counting allows the developer to think about memory locally, asking when an object needs to retain another, instead of globally. This alleviates a lot the pain of manual memory management and paired with some common programming patterns was much safer and easier than manual memory management.

For a brief period Apple adopted garbage collection on Mac OS 10.6. But when the iPhone came out, the resources on the device were too limited to run a garbage collection process. One downside is that the garbage collector needs to be run periodically, while the program execution is halted to avoid problems with changing references. This is usually not perceived on a normal computer, but on a phone with limited resources it freezes apps for some time, leading to a bad user experience. Another downside is that allocated memory of the program keeps accumulating until the garbage collector is activated, which is again a problem on a device with very limited amount of memory. For this reason, when the iOS SDK was released, Apple switched back to reference counting.

In modern Objective-C, memory management is done through what is called ARC. Reference counting is still supported for old legacy code, but since ARC works back to iOS 4 and Mac OS 10.7, reference counting should not be needed anymore and we will not have a look to how it works.

So, what is ARC? As I said, reference counting is led by common patterns and best practices on when retain and release should be performed and how to name methods that involve reference counting. For this reason Apple saw an opportunity to automate it and introduced Automatic Reference Counting, or ARC.

ARC removes reference counting from the developer hands and automates it in the compiler. The benefit is that, in addition to taking away responsibility for tedious memory management from the programmer, ARC is done at compile time, when the binary of the app is created, thus removing any runtime process that might slow down the device. What the compiler actually does is to add the proper memory retain and release calls in the code where they are needed.

ARC has been highly optimized, so it works generally faster than the memory management done manually. Moreover it forces some memory checks into the compiler, which then signals problems to the developer to be fixed, or the app won't compile, removing many memory management errors. Since at this point in time ARC is supported on the vast majority of machines and devices, it is advise to migrate all code bases, so probably you will never have to learn manual reference counting. Xcode has a tool to automate this transition as much as possible.

ARC still suffers from one pitfall though, as garbage collection does. If an object circular references exists, a retain cycle is created and the memory used by the object will never be released, exactly how it happens in garbage collected languages.

To avoid retain cycles, Objective-C has some lifetime qualifiers. For properties, two qualifiers exist: **strong** and **weak**. The default qualifier is **strong**, which signals that the reference object needs to be kept in memory until that reference exists. Thus, the standard declaration we saw for properties

```
@property Class *propertyName;
```

is the same as

```
@property (strong) Class *propertyName;
```

If you need to create a reference cycle to make two (or more) objects communicate with each other in a circular manner, one of the two needs to have **weak** reference to the other one. This is used a lot, for example, in the delegate design pattern, a very common pattern in Objective-C. To avoid a retain cycle, one of the classes still uses a strong property:

```
@interface ClassA : NSObject

@property ClassB *objectB;

@end
```

while the other uses a weak one:

```
@interface ClassB : NSObject

@property (weak) ClassA *objectA;

@end
```

When nothing references the **objectA** anymore, it is removed from memory because the weak reference does not count when counting references. So **objectA** is not retained by **objectB**, making it possible to release **objectA** when it's not referenced anymore by other objects. When **objectA** gone, the strong reference to **objectB** is removed, thus removing **objectB** too (if it's not referenced strongly from anywhere else). When an object referenced weakly is removed from memory, all the weak references pointing to it are automatically set to **nil**, making it safe for the referencing objects to still call methods on it.

When using normal variables or instance variables, the corresponding lifetime qualifiers are **__strong** and **__weak**. As per Apple documentation, the qualifier needs to be specified after the * in the declaration, with this syntax:

```
NSArray * __weak array;
```

Although the documentation says that the compiler "forgives" other variants, it's better not to use them since you never know when in the future the compiler won't be so kind anymore.

Pay attention to **__weak** variables. When there is no other reference to the object they store they get immediately allocated leading to this common problem:

```
NSMutableArray * __weak strings = [NSMutableArray new];
[strings addObject:@"Hello"]; // On this line strings is already nil
```

On the second line, the **strings** array does not exist anymore and the variable will be **nil** already. This is because, even if the second line references it, there is no other strong reference to the array when it is created, therefore the compiler deallocates it immediately. Other subtle cases might be not so easy to spot, so pay attention when using **__weak** variables.

There are two more qualifiers for variables: **__unsafe_unretained** and **__autoreleasing**. The first one works like **__weak**, but the variable is not set to **nil** when the object it references is deallocated. For this reason it's unsafe (as the name implies) because it leaves a "dangling" reference to deallocated memory. This leads to crashes if you try to call a method on it, unlike **nil**, which is safe. This identifier exists only to support ARC in iOS 4 and Mac OS 10.6, so you probably will never need it. If you inherit old codebases, pay attention also to the **assign** qualifier for properties, if you find any, because that's equivalent to **__unsafe_unretained**. Change it to **weak**.

The **__autoreleasing** qualifier is used in parameters of methods passed by reference, which we will see later.

8. CUSTOM TYPES

DEFINING CUSTOM TYPES

We have already seen in the previous parts of this guide that there are two kinds of types a variable can have in Objective-C: basic types, as we have seen in the first part of this guide or object types, coming from classes either defined in Apple frameworks or defined by us. We have spent the last issues seeing how to create and extend the latter, but what about basic types? Can we define them too? If so, why would this be useful since we already have classes?

It turns out, of course, that we can indeed define new basic types, and actually we have already met some of these newly defined types, although I didn't mention what they were. The C language basically offers only basic types for integers and floating point values. As we have seen, even the bool and char types are in the end just integers used in a different manner (and with a different byte size in memory, to be precise).

C, unlikely other languages like Java, allows also the definition of new basic types, the only condition being that these new types must be derived from the ones already provided by the language (or by composing them, as we will see later). In fact, the Objective-C basic types we encountered, like **NSInteger** or **CGFloat**, are just redefinitions of basic C types, namely **long** and **double**, respectively.

Why this redefinition? Could we not just use **long** and **double** instead? Of course we could, but there are some advantages in defining new types. The first one is clarity and legibility. **CGFloat** is not the only redefinition of the **double** type, there are also others, like **NSTimeInterval**. Despite being both the same type of value in memory, in our code we can clearly see that a variable of the first type represents graphical values, like coordinates on screen or sizes of graphical objects, while a variable of the second type represent an amount of time (in seconds, as defined by the documentation).

The other reason is one we already encountered, which is to differentiate the actual memory representation of a type based on the architecture in a way that is transparent to the developer. I said that a **CGFloat** is just a **double**, but actually that is not entirely correct: it's a **double** on 64 bits architectures, while it's a **float** on 32 bits ones. This redefinition allows the redefinition of the underlying representation of the types when needed, without requiring to change all the code that has been written up to that point (and this is exactly what happened to **NSInteger** and **CGFloat** in the transition from 32 to 64 bits architectures).

Now that we know the reasons to define new types, here is how to do it:

```
typedef existingType newType;
```

So, for example, if I want to define a type to represent people's age as a positive integer, I can write:

```
typedef unsigned int Age;
```

or alternatively:

```
typedef NSUInteger Age;
```

I can then use this new type normally to define variables or properties in objects. Let's rewrite our previous example of the Person class to include an age property:

```
typedef unsigned int Age;

@interface Person : NSObject

@property NSString *firstName;
@property NSString *lastName;
@property Age age;

@end
```

CONSTANTS AND ENUMERATIONS

Sometimes we need variables to be constant and to not change their value over time. This is easily done with the **const** keyword in front of a variable declaration:

```
const double Pi = 3.14159;
```

The compiler will make sure that the content of this variable is never changed and will emit an error if you try to assign a new value to it after this declaration.

(By the way, if you need pi it's already defined as the **M_PI** constant, together with other common mathematical constants in the header file **math.h**, which is always available in C, so you don't have to include it directly in your source files).

Sometimes though we need to define more constants to enumerate different options we might have and we don't really care about the values these constants might have as long as they are different from each other (sometimes we might need these values to be ordered).

Let's say we want to add a sex property to our Person class. The only values we want to allow are Male, Female and Undefined when sex is not specified (for simplicity of the example I will not include the various transgender identities, but when you deal with such kind of data in real like it's helpful to not make the assumption that people only identify themselves with the two canonical sexes).

We could already use the **const** keyword for this and define each sex as an integer constant with values 0, 1 and 2. This would work but would be impractical for larger sets of constants, because when we needed to introduce a new value inside the order, we will have to change all the other values manually. Luckily C has a special construct made for this, called **enum**, which stands for enumeration:

```
enum {
    FirstConstant,
    SecondConstant,
    ...
    LastConstant
};
```

In this way all the constants get consecutive values starting from 0. Notice that the last constant does not have a comma after it and there is a semicolon after the closing brace (these are two common mistakes that cause syntax errors). It's also possible to start from a value different than 0:

```
enum {
    FirstConstant = number,
    SecondConstant,
    ...
    LastConstant
};
```

The consecutive constants will get incremental values starting from the selected number. It is even possible to assign an arbitrary value to each constant, if needed.

It does not stop here. It's possible to use an enumeration to define a new type:

```
typedef enum {
    FirstConstant,
    SecondConstant,
    ...
    LastConstant
} TypeName;
```

Where the name of the type goes at the end, after the closing brace. Since iOS 6 and Mac OS 10.8, Apple introduces a new **NS_ENUM** macro, which is now the preferred way to define enumerations. I included the other ways in this guide for completeness, since you might still find them in some code bases or in Apple documentation, but you should use **NS_ENUM** for your enumerations:

```
typedef NS_ENUM(baseType, newTypeName) {
    FirstConstant,
    SecondConstant,
    ...
    LastConstant
};
```

This macro adds a *baseType* to ensure a fixed size for the newly defined type. It also provides hints to the compiler for type-checking and switch statement completeness. It's also a bit more readable since the new type name is at the top and not at the bottom.

We can now expand our Person class to include sex:

```
typedef unsigned int Age;

typedef NS_ENUM(NSUInteger, Sex) {
    Undefined,
    Male,
    Female
};

@interface Person : NSObject

@property NSString *firstName;
@property NSString *lastName;
@property Age age;
@property Sex sex;

@end
```

I put the *Undefined* constant at the beginning, so it will have a value of 0 which is the default value properties have when an object is created. We can then use the constants when creating our object:

```
Person *person = [Person new];
person.firstName = @"Matteo";
person.lastName = @"Manferdini";
person.age = 33;
person.sex = Male;
```

Pay attention though that the compiler does not enforce the values that we can assign to a variable with an enumeration type. So this will still compile and run perfectly fine:

```
person.sex = 7;
```

Apple makes extensive uses of enumerations in its classes. For example, to create a new button in iOS 7, you would use the +buttonWithType: factory method with a UIButtonTypeSystem type:

```
UIButton *button = [UIButton buttonWithType:UIButtonTypeSystem];
```

BITWISE OPERATORS AND BITMASKS

There is still a class of operators I didn't mention over when I spoke about operators in Objective-C: bitwise operators. They are used for bitwise operations as the name implies, which are usually needed for some low level implementations (like network protocols), but can be used also for other purposes as we will now see. Like the other operators, these ones come directly from C. This is a table summing them up:

Operator	Description	Syntax
&	bitwise and	a & b
|	bitwise or	a | b
^	bitwise xor	a ^ b
<<	left shift	a << b
>>	right shift	a >> b
~	bitwise not, or one's complement	~a

Let's see how they are usually used in Objective-C. With enumerations we have seen how we can create types that allow us to name different constants in a more readable way. The limitation with enumerations, though, is that a variable can only have one of the defined values at a time. What if we want to have a predefined set of options that are not mutually exclusive but can be set at the same time?

One way of doing this would be again to create a class with a boolean property for each option we want to enable simultaneously, but this would be an overkill and we would need to then assign or check every property singularly, making it very tedious to check all the options.

This special case can be handled easily with bitwise operators. We can make each constant in the enumeration have only one bit set to 1 and all others set to 0 (remember that we can assign an arbitrary value to an enumeration). Then, to specify multiple options at the same time we can use the bitwise **or** operator to group them in the same variable. Lets see how this works with an example.

Let's say we want to have 4 non mutually exclusive options. We can declare an enumeration with bitmasks using the left shift operator. As we had the **NS_ENUM** macro for normal enumerations, we have the **NS_OPTIONS** macro for bitmask enumerations (but a simple **typedef enum** would still work, of course):

```
typedef NS_OPTIONS(NSUInteger, Options) {
    NoOptions = 0,
    Option1 =   1 << 0,
    Option2 =   1 << 1,
    Option3 =   1 << 2,
    Option4 =   1 << 3,
} Options;
```

The options will have the following bit representations in memory (I will omit leading zeros for clarity and only show the last 4 bits):

```
Option      Value

NoOptions   0000
Option1     0001
Option2     0010
Option3     0100
Option4     1000
```

When we want to group multiple options in the same variable, we can use the bitwise **or** operator:

```
Options enabledOptions = Option1 | Option3;
```

This groups the multiple options together in this way:

```
Option1         0001
Option3         0100
                ----
enabledOptions  0101
```

As you can see, both the first and the third bit (from the right) are set in the **enabledOptions** variable, thus containing both values. To check wether any option is enabled we can use the bitwise **and** operator:

```
if (enabledOptions & Option1) {
    ...
}
```

If an option is enabled, operating a bitwise **and** on the value will produce a non zero value, that can be used as a true value in a condition. A bitwise **and** with a non enabled option produces a value of zero, which is equivalent to false.

You can use this in your code and Apple indeed does use it in different places. If we want to specify, for example, an autoresizing mask for a view with both flexible width and height, we can do it this way:

```
UIView *view = [[UIView alloc] initWithFrame:CGRectZero];
view.autoresizingMask = UIViewAutoresizingFlexibleWidth |
                        UIViewAutoresizingFlexibleHeight;
```

STRUCTURES

Now, what if we want a variable to hold more than one value? One possibility is of course to create an object for that, as we have already seen, and in many languages this is the only option to achieve this purpose. But Objective-C inherits a feature from C that allows variables to contain more than one value, without being an object: structures.

You rarely need to create structures in your apps, but if you need something more lightweight than an object to contain simple values and with faster access a struct might be your tool. A structure is just a some space in memory that contains some values exactly as variables do, so it does not incur in the additional overhead of objects. A structure variable is declared with the keyword **struct** and can contain as many values as you want:

```
struct variableName {
    type memberName;
    ...
};
```

The syntax is similar to that of enumerations, but notice that here there is a semicolon after each member of the structure, because they act like variable declarations inside of the structure.

Let's say, for example, that you want to store the value of a point on screen, which is identified by x and y coordinates. You can do so with a structure:

```
struct point {
    CGFloat x;
    CGFloat y;
};
```

You can then access the members with dot notation:

```
point.x = 100.0;
point.y = 200.0;
```

Or you can initialize the whole structure in one line using the following syntax with curly braces:

```
point = {100.0, 200.0};
```

As happens for enumerations, you can turn structures into basic types to be able to reuse them:

```
typedef struct {
    type memberName;
    ...
} typeName;
```

This is exactly how the **CGPoint** type is declared, for example. **CG-Point** is used to store coordinates of graphical objects in iOS apps.

A disadvantage of structures over objects is that functionality needs to be defined externally. After all structures are only a basic type and cannot have methods attached. This requires you to write C functions to operate on them, which you usually don't want to do. For example, **CGPoint** has a function to initialize a new structure:

```
CGPoint point = CGPointMake(100.0, 200.0);
```

or a function to compare two points:

```
if (CGPointEqualToPoint(point1, point2) {
    ...
}
```

Structures can also contain other structures, as in **CGFrame**, a type that indicated rectangles of graphical objects. A **CGFrame** has a origin, which is a **CGPoint**, and a size expressed in width and height, which is a **CGSize**. Under ARC, though, a structure cannot contain objects, so a structure can only be composed from other basic types.

These are structures you will encounter quite often, with other structures (like **CGAffineTransform**, for example, if you want to rotate, scale or translate graphical objects). There are a lot of functions in Apple frameworks to deal with these structures, but as you can see, this separates the functionality from the data, which defeats the Object Oriented Paradigm. For this reason it's usually better to create a new class instead of a structure for your values, unless you have specific reasons like compatibility with C or C++ code, or to optimize execution speed.

9. BLOCKS

WHAT BLOCKS ARE USEFUL FOR

Many (but not all) modern languages have what is usually called *lambdas* or *closures* (although these two terms are not equivalent, as we will see below, people sometimes use them interchangeably). For a long time Objective-C lacked such feature, but luckily in recent years blocks were introduced to fill the gap. I would say that they are not as elegant as in other languages because of they awkward syntax, but they are still extremely useful.

For those who don't know the concept yet, lambdas (and blocks) are nameless functions that can, in the code, be treated also as data. What this means is that you can take a piece of functionality and store it in a variable to use it later, or pass it as a parameter to a method. This comes handy in different ways .

The first useful use case for blocks is when you need to provide asynchronous callbacks. Examples for this are tasks that take some time to produce a result, like downloading data from a network connection or knowing when an animation ends. When these tasks are completed, we usually need to have some specific code to be executed. The way this was done before the introduction of blocks was to create a delegate protocol, declaring methods for the callback or callbacks needed. The object that needs to be notified then conforms to this protocol, setting itself as a delegate on the notifying object, before the call that starts the asynchronous task. When the task is completed, the performing object calls the callback or callbacks defined by the protocol on the delegate. As you can see, this usually requires a number of steps, while It would be nicer to just say indicate what code to execute at the end of the task, without having to deal with protocol and delegates. One way could be to pass a selector as a parameter (a selector is the signature of a method), but this is a limited approach we are not going to explore, because delegate protocols have always been the standard way of solving this problem. Blocks allow us to skip creating a new protocol altogether, because, as said, they allow us to a function directly as a parameter of a method, providing the code to execute at the end of a task.

Another use for blocks is to enable methods to be more customizable than they are through normal parameters. To provide a common simple example, let's say we have to sort an array of objects. The sorting algorithm is going to be fixed, but the way in which we compare objects between themselves to sort them is going to be different for different types. One way to solve this is to implement a comparison method inside the objects themselves (which is a common approach. This is done in Objective-C by overriding the **-isEqual:** method). Another way to do accomplish this using blocks is to pass to the sorting algorithm the actual function it should use to compare elements.

DECLARING AND USING BLOCKS

Blocks are functions, so when we declare them we have to indicate a return type and the parameters that the block might need. Blocks are declared using the ^ operator. This is the syntax to declare a block variable:

```
returnType (^variableName)(parameterTypes);
```

As you can see, the name of the variable needs to be indicated inside parentheses after a ^ symbol, which is what makes the syntax a bit awkward. Like in a normal function or methods, the parameters of the block are separated by a comma and can be omitted for blocks without parameter (using the **void** keyword), and the return type of the block can be **void** for blocks that don't return any value. The parameters do not need to be named in the declaration (but can) and only their type needs to be provided. This is how a block is declared. To create one, the ^ operator is again used, to indicate the beginning of a block literal. After that the parameters of the block are declared between parentheses (this time with names), followed by the block body placed between curly braces.

```
blockVariable = ^(parameters) {
    ...
};
```

Pay attention to the semicolon after the closing brace, which you are probably not used to see. Since this is an assignment, it needs a semicolon at the end like any other statement. As with any other variable, declaration and assignment can be composed in a single statement:

```
returnType (^variableName)(parameterTypes) = ^(parameters) {
    ...
};
```

You use a block the same way you would use a function, calling it by its name (which is, in this case, the variable's name, since a block is by definition an anonymous function) followed by parameters inside parentheses. There is one caveat to pay attention to, though. Unlike what happens with objects, where it's possible to call methods on **nil** values with no adverse effect, calling a **nil** block will result in an exception and a crash. For this reason, you often need to guard a block call with an if statement:

```
if (block) {
    block(parameters);
}
```

Let's see an example. We will declare and use a block that triples a number passed as a parameter. It's not a very useful or common block by itself, but it allows us to have a look at a simple case, since the syntax for blocks is a bit uncommon.

```
NSInteger (^triple)(NSInteger) = ^(NSInteger number){
    return number * 3;
};
NSInteger six = triple(2);
```

Notice that, as stressed, **triple** is not the name of the block itself, but just the name of the variable that contains it. This means we can assign a completely different block to the variable and the compiler will not complain, provided it has the same return type and parameters.

```
triple = ^(NSInteger number){
    return number - 4;
};
NSLog(@"%i", triple(10)); // Prints 6
```

If you want to store a block in an object property to use it later, the property can be declared using the same syntax used for block variables:

```
@property returnType (^propertyName)(parameterTypes);
```

Since the blocks syntax is a bit hard to remember and error prone, Apple recommends in the documentation to define a new type when we need a block, to improve readability.

```
typedef returnType (^typeName)(parameterTypes);
typeName variableName;
```

The previous example becomes:

```
typedef NSInteger (^Block)(NSInteger);
Block triple = ^(NSInteger number){
    return number * 3;
};
```

Just pick a better type name than the one I put in the example.

PASSING BLOCKS AS PARAMETERS TO METHODS

Blocks become really useful when you can pass them as parameters to methods, to provide callbacks or customization. The syntax to declare a block parameter in a method is as follows:

```
- (void)methodName:(returnType (^)(parameterTypes))blockParameterName;
```

The syntax gets a bit more awkward and harder to remember because the ^ operator in this case is alone in the parentheses, because the variable name has moved to the usual place for parameter names in a method. If a method that uses blocks has more than one parameter, it is good practice to declare the block parameters last, to make the method call more readable. When calling a method with a block parameter, a block variable can, of course, be passed as a parameter, but a block literal can be directly passed inline as well. When using autocompletion in Xcode you can hit the return key on the parameter placeholder to make Xcode expand it with a block literal, that you will just need to fill with instructions. This is how to call a method with an inline block:

```
[methodName:^(parameters) {
    ...
}];
```

One good example of use of blocks as method parameters, for both customizing a method and as callbacks, is the **UIView** animation API. One of the methods in the API has this declaration:

```
+ (void)animateWithDuration:(NSTimeInterval)duration animations:(void (^)
(void))animations completion:(void (^)(BOOL finished))completion;
```

The second parameter is a block with no return type and no parameters, which is used to provide the changes to the view hierarchy that will be animated. This is an example of method customization: the animation routine always performs the same steps in preparing the environment to perform the animation and in tearing it down after the animations have been performed. What is always different are the animations themselves, which could not be passes simply as data (or at least, not without complicate constructs). The last parameter of the animation method is the callback that will be executed when the animation stops (it has a parameter to know if the animation was interrupted or not). This is how you call this method:

```
[UIView animateWithDuration:0.2 animations:^{
    ...
} completion:^(BOOL finished) {
    ...
}];
```

CAPTURING THE CONTEXT

One of the useful features of blocks is that they can capture values from the context that surrounds them when they are created, and be able to reference them when the block is actually executed. This is called "closing over the environment" and it's why lambdas are sometimes called closures (the technical difference is that the former are only anonymous functions, without closing). What I mean here with "context" or "environment" are the variables that defined in the lexical scope enclosing the block, outside the block itself. These variables are copied and stored inside of the block. This means that the block has at its disposal a snapshot of the context in which it was created that he can reference at the moment of its execution, even if that context is long gone or has changed. Let's see a simple example.

```
NSInteger value = 3;
    NSInteger (^doubleValue)(void) = ^{
    return value * 2;
};

value = 5;
NSLog(@"%i", doubleValue()); // Prints 6
```

As you can see, when the block is executed, it uses the value **3**, which is the old value of the **value** variable, although in the meantime it has change to **5**. Local variables captured in a block are copied, which means that if we change them inside of the block, their value won't be changed outside of it.

```
NSInteger value = 3;
NSInteger (^assign)(void) = ^{
    value = 5;
};

assign();
NSLog(@"%i", value); // Prints 3
```

In case of object variables, what is copied is only the reference to the object, which means we are still able to change the object properties from inside the block. Sometimes though, we might want to be able to change an external variable from inside of a block. This can be done by using the **__block** keyword:

```
__block NSInteger value = 3;
NSInteger (^assign)(void) = ^{
    value = 5;
};

assign();
NSLog(@"%i", value); // Prints 5
```

BLOCKS AND MEMORY MANAGEMENT

Regarding memory, blocks behave like objects. When a new block is created, it is allocated on the heap like any other object and handled by ARC in the same way. If a block captures a reference to an object, that reference is going to be a strong one, since the object referenced needs to be kept in memory until the block is in memory too and can be executed. This, as it happens for object, can lead to retain cycles, causing memory leaks. One particularly tricky case is when the block captures the **self** reference of an object. This often goes unnoticed by the developer, because we are usually using the **self** keyword inside the block not by itself, but to access some property. An example explains this better than words:

```
@interface Logger : NSObject

@property NSString message;
@property void (^loggingBlock)(void);

@end

@implementation

- (void)setLoggingBlock {
    void (^block)(void) = ^{
        NSLog([self.message]);
    }
    self.loggingBlock = block;
}

- (void)printLogMessage {
    if (self.loggingBlock) {
        self.loggingBlock();
    }
}

@end
```

This might look innocuous at a first glance, and that's why it's a tricky case. In the **-setLogMessage:** we are creating a block that logs the **message** property. Since we need the block later in the **-print-LogMessage** method, we store it in another property. What we might think the block does is capture the reference to the **message** property (an thus to an instance of **NSString**). What actually happens though is that the block captures a reference to **self** instead. Since the block is then retained in a property inside the object itself, this creates a strong reference cycle. How do we break this? The solution is not very elegant, but does the job:

```
- (void)setLoggingBlock {
    __weak id weakSelf = self;
    void (^block)(void) = ^{
        Logger *strongSelf = weakSelf;
        NSLog([strongSelf.message]);
    }
    self.loggingBlock = block;
}
```

As I said, this mistake is hard to spot and you need to be constantly vigilant not to introduce strong reference cycles in your code. Luckily there is a warning in Xcode that tells you when you are making this mistake, but it is off by default, so you have to go and enable it explicitly in the project build settings.