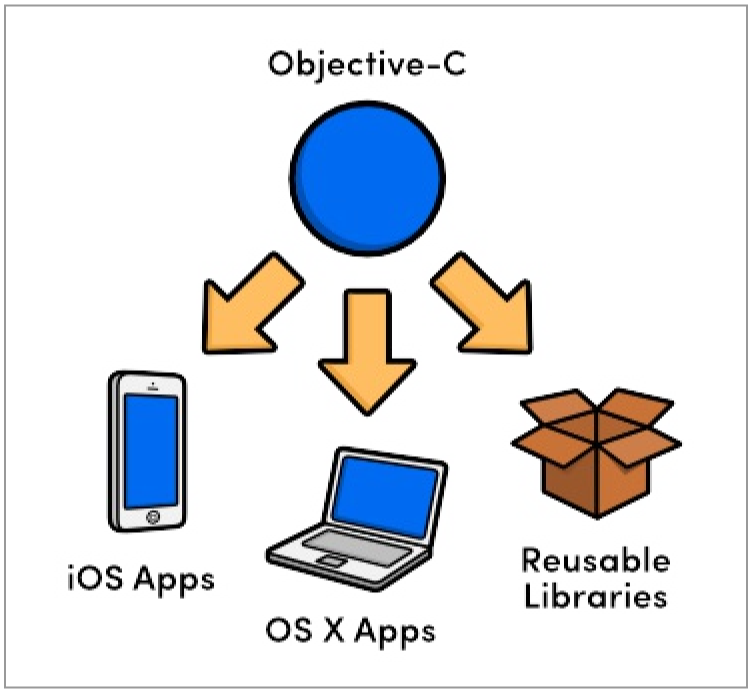
Table of Contents

Guide

Introduction

Objective-C is the native programming language for Apple's iOS and OS X operating systems. It's a compiled, general-purpose language capable of building everything from command line utilities to animated GUIs to domain-specific libraries. It also provides many tools for maintaining large, scalable frameworks.



Types of programs written in Objective-C

Like C++, Objective-C was designed to add object-oriented features to C, but the two languages accomplished this using fundamentally distinct philosophies. Objective-C is decidedly more dynamic, deferring most of its decisions to run-time rather than compile-time. This is reflected in many of the design patterns underlying iOS and OS X development.

Objective-C is also known for its verbose naming conventions. The resulting code is so descriptive that it's virtually impossible to misunderstand or misuse it. For example, the following snippet shows a C++ method call with its Objective-C equivalent.

```
// C++
john->drive("Corvette", "Mary's House")
// Objective-C
[john driveCar:@"Corvette" toDestination:@"Mary's House"]
```

As you can see, Objective-C methods read more like a human language than a computer one. Once you get used to this, it becomes very easy to orient yourself in new projects and to work with third-party code. If you're a little bit disarmed by the square-brackets, don't worry. You'll be quite comfortable with them by the end of the tutorial.

Frameworks

As with most programming languages, Objective-C is a relatively simple syntax backed by an extensive standard library. This tutorial focuses mostly on the language itself, but it helps to have at least some idea of the tools that you'll be interacting with in the real world.

There are a few different "standard libraries" out there, but Apple's Cocoa and Cocoa Touch frameworks are by far the most popular. These define the API for building OS X and iOS apps, respectively. The table below highlights some of the key frameworks in Cocoa and Cocoa Touch. For a more detailed discussion, please visit the Mac Technology Overview or iOS Technology Overview.

Framework	Description
Foundation	Defines core object-oriented data types like strings, arrays, dictionaries, etc. We'll explore the essential aspects of this framework in the Data Types module.
UIKit	Provides dozens of classes for creating and controlling the user interface on iOS devices.
AppKit	Same as UIKit, but for OS X devices.
CoreData	Provides a convenient API for managing object relationships, supporting undo/redo functionality, and interacting with persistent storage.
MediaPlayer	Defines a high-level API for playing music, presenting videos, and accessing the user's iTunes library.
AVFoundation	Provides lower-level support for playing, recording, and integrating audio/video into custom applications.
QuartzCore	Contains two sub-frameworks for manipulating images. The `CoreAnimation` framework lets you animate UI components, and `CoreImage` provides image and video processing capabilities (e.g., filters).
CoreGraphics	Provides low-level 2D drawing support. Handles path-based drawing, transformations, image creation, etc.

After you're comfortable with Objective-C, these are some of the tools that you'll be leveraging to build iOS and OS X applications. But again, this tutorial is not meant to be a comprehensive guide to app development—it's designed to *prepare you* to use the above frameworks. With the exception of the Foundation Framework, we won't actually be working with any of these libraries.

If you're interested in Mac App development, you should take a look at Ry's Cocoa Tutorial after you have a solid grasp on Objective-C. It shows you how to build OS X apps using the same hands-on methodology as this tutorial.

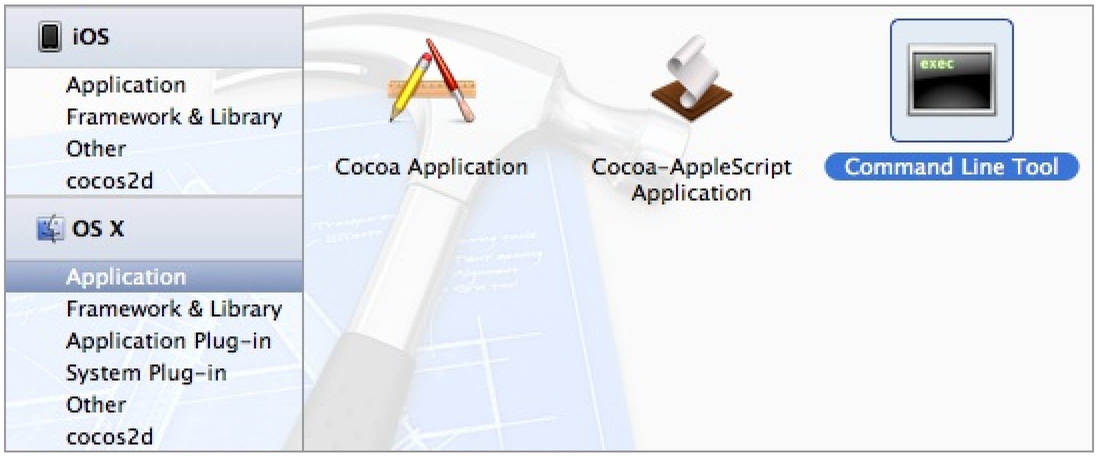
Xcode

Xcode is Apple's integrated development environment (IDE) for Mac, iPhone, and iPad app development. It includes not only a source code editor, but also an interface builder, a device simulator, a comprehensive testing and debugging suite, the frameworks discussed in the previous section, and everything else you need to make apps.

While there are other ways to compile Objective-C code, Xcode is definitely the easiest. We strongly recommended that you install Xcode now so you can follow along with the examples in this tutorial. It is freely available through the Mac App Store.

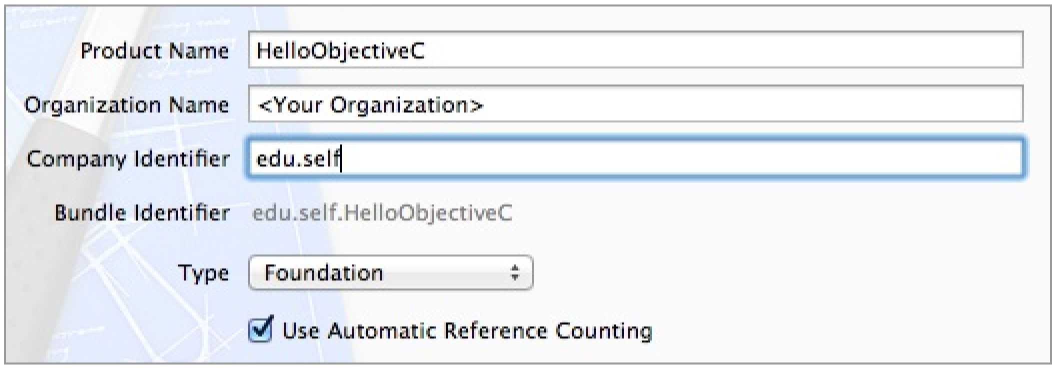
Creating an Application

Xcode provides several templates for various types of iOS and OS X applications. All of them can be found by navigating to *File > New > Project...* or using the *Cmd+Shift+N* shortcut. This will open a dialog window asking you to select a template:



Creating a command line application

For this tutorial, we'll be using the *Command Line Tool* template found under *OS X > Application*, highlighted in the above screenshot. This lets us strip away all of the elements specific to iOS/OS X and focus on Objective-C as a language. Go ahead and create a new *Command Line Tool* now. This opens another dialog asking you to configure the project:

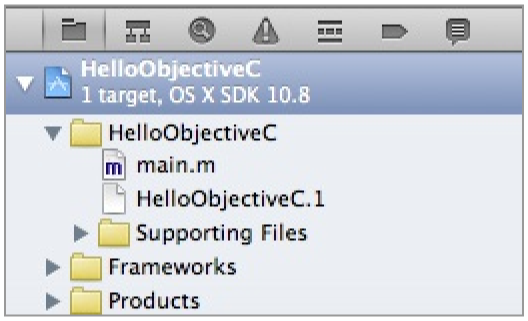

Product Name HelloObjectiveC
Organization Name <Your Organization>
Company Identifier edu.self
Bundle Identifier edu.self.HelloObjectiveC
Type Foundation
☑ Use Automatic Reference Counting

Configuring a command line application

You can use whatever you like for the *Product Name* and *Organization Name* fields. For the *Company Identifier* use `edu.self`, which is the canonical private use identifier. For production applications, you'll need to get a real company ID from Apple by registering as a developer.

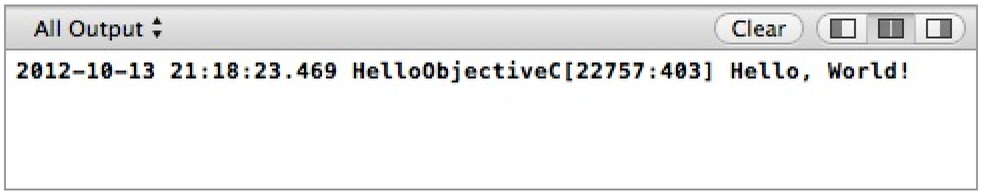
This tutorial utilizes several classes defined in the Foundation Framework, so be sure to select *Foundation* for the *Type* field. Finally, the *Use Automatic Reference Counting* checkbox should always be selected for new projects.

Clicking *Next* prompts you for a file path to store the project (save it anywhere you like), and you should now have a brand new Xcode project to play with. In the left-hand column of the Xcode IDE, you'll find a file called `main.m` (along with some other files and folders). At the moment, this file contains the entirety of your application. Note that the `.m` extension is used for Objective-C source files.



`main.m` in the Project Navigator

To compile the project, click the *Run* button in the upper-left corner of the IDE or use the *Cmd+R* shortcut. This should display `Hello, World!` in the *Output Panel* located at the bottom of the IDE:



```
2012-10-13 21:18:23.469 HelloObjectiveC[22757:403] Hello, World!
```

Xcode's Output Panel

The main() Function

As with plain old C programs, the `main()` function serves as the root of an Objective-C application. Most of the built-in Xcode templates create a file called `main.m` that defines a default `main()` function. Selecting our `main.m` in Xcode's *Project Navigator* panel should open the editor window and display the following.

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // insert code here...
        NSLog(@"Hello, World!");
    }
    return 0;
}
```

Inside of the `@autoreleasepool` block is where you can write code and experiment with the snippets from this tutorial. The above `main()` function simply calls the global `NSLog()` function defined by the Foundation Framework. This is Objective-C's general-purpose tool for outputting messages to the console. Also note that Objective-C strings are always prefixed with an at (`@`) symbol.

Throughout this tutorial, we'll directly edit the above `main.m` to see how new language features work, but in the real world, you'll probably never have to alter the `main()` function provided by the template. For most applications, the only thing `main()` needs to do is pass control of the program to the "application delegate." For example, the default `main()` function for a Mac App project looks like the following.

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

int main(int argc, const char * argv[]) {
    return NSApplicationMain(argc, argv);
}
```

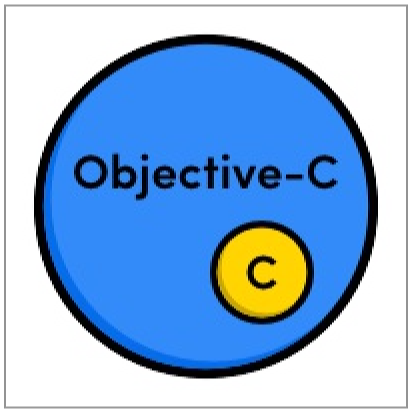
But, since we'll be sticking with command line tools, this is somewhat outside the scope of this tutorial. Application delegates are, however, an integral part of OS X and iOS development. The first several chapters of Ry's Cocoa Tutorial explain them in full detail.

Get Ready!

The next two modules explore the basic C syntax. After that, we’ll be ready to dive into classes, methods, protocols, and other object-oriented constructs. This tutorial is chock-full of hands-on examples, and we encourage you to paste them into the template project we just created, mess with some parameters, and see what happens.

C Basics

Objective-C is a strict superset of C, which means that it's possible to seamlessly combine both languages in the same source file. In fact, Objective-C *relies* on C for most of its core language constructs, so it's important to have at least a basic foundation in C before tackling the higher-level aspects of the language.



The relationship between Objective-C and C

This module provides a concise overview the C programming language. We'll talk about comments, variables, mathematical operators, control flow, simple data structures, and the integral role of pointers in Objective-C programs. This will give us the necessary background to discuss Objective-C's object-oriented features.

Comments

There are two ways to include commentary text in a C program. **Inline comments** begin with a double slash and terminate at the end of the current line. **Block comments** can span multiple lines, but they must be enclosed in `/*` and `*/` characters. For example:

```
// This is an inline comment


/* This is a block comment.
   It can span multiple lines. */
```

Since comments are completely ignored by the compiler, they let you include extra information alongside your code. This can be useful for explaining confusing snippets; however, Objective-C is designed to be very self-documenting, so you shouldn't really need to include a lot comments in your iOS or OS X applications.

Variables

Variables are containers that can store different values. In C, variables are statically typed, which means that you must explicitly state what kind of value they will hold. To **declare** a variable, you use the `<type> <name>` syntax, and to assign a value to it you use the `=` operator. If you need to interpret a variable as a different type, you can **cast** it by prefixing it with the new type in parentheses.

All of this is demonstrated in the following example. It declares a variable called `odometer` that stores a value of type `double`. The `(int)odometer` statement casts the value stored in `odometer` to an integer. If you paste this code into your `main.m` file in Xcode and run the program, you should see the `NSLog()` messages displayed in your *Output* panel.

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        double odometer = 9200.8;
        int odometerAsInteger = (int)odometer;

        NSLog(@"You've driven %.1f miles", odometer);        // 9200.8
        NSLog(@"You've driven %d miles", odometerAsInteger); // 9200
    }
    return 0;
}
```

Along with `double` and `int`, C defines a plethora of primitive data types. A comprehensive list can be found in the Primitives module, as well as an explanation of the `%.1f` and `%d` format specifiers used above.

Constants

The `const` variable modifier can be used to tell the compiler that a variable is never allowed to change. For example, defining a constant called `pi` and then trying to alter it will result in a compiler error:

```
double const pi = 3.14159;
pi = 42001.0;               // Compiler error
```

This is often used in function parameters to inform the caller that they can safely assume whatever value they pass will not be altered by the function.

Arithmetic

The familiar `+`, `-`, `*`, `/` symbols are used for basic arithmetic operations, and the modulo operator (`%`) can be used to return the remainder of an integer division. These are all demonstrated below.

```
NSLog(@"6 + 2 = %d",  6 + 2);    // 8
NSLog(@"6 - 2 = %d",  6 - 2);    // 4
NSLog(@"6 * 2 = %d",  6 * 2);    // 12
NSLog(@"6 / 2 = %d",  6 / 2);    // 3
NSLog(@"6 %% 2 = %d", 6 % 2);    // 0
```

Special care must be taken when performing operations that involve both floating-point and integer types. Please see Integer Division for details.

You'll also frequently encounter the increment (`++`) and decrement (`--`) operators when working with loops. These are convenience operators for adding or subtracting `1` from a variable. For example:

```
int i = 0;
NSLog(@"%d", i);    // 0
i++;
NSLog(@"%d", i);    // 1
i++;
NSLog(@"%d", i);    // 2
```

Conditionals

C provides the standard `if` statement found in most programming languages. Its syntax, along with a table describing the most common relational/logical operators, is shown below.

```
int modelYear = 1990;
if (modelYear < 1967) {
    NSLog(@"That car is an antique!!!");
} else if (modelYear <= 1991) {
    NSLog(@"That car is a classic!");
} else if (modelYear == 2013) {
    NSLog(@"That's a brand new car!");
} else {
    NSLog(@"There's nothing special about that car.");
}
```

Operator	Description
`a == b`	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`	Logical negation
`a && b`	Logical and
`a || b`	Logical or

C also includes a `switch` statement, however it only works with integral types—not floating-point numbers, pointers, or Objective-C objects. This makes it rather inflexible when compared to the `if` conditionals discussed above.

```
// Switch statements (only work with integral types)
switch (modelYear) {
    case 1987:
        NSLog(@"Your car is from 1987.");
        break;
    case 1988:
        NSLog(@"Your car is from 1988.");
        break;
    case 1989:
    case 1990:
        NSLog(@"Your car is from 1989 or 1990.");
        break;
    default:
        NSLog(@"I have no idea when your car was made.");
        break;
}
```

Loops

The `while` and `for` loops can be used for iterating over values, and the related `break` and `continue` keywords let you exit a loop prematurely or skip an iteration, respectively.

```
int modelYear = 1990;
// While loops
```

```
int i = 0;
while (i<5) {
    if (i == 3) {
        NSLog(@"Aborting the while-loop");
        break;
    }
    NSLog(@"Current year: %d", modelYear + i);
    i++;
}
// For loops
for (int i=0; i<5; i++) {
    if (i == 3) {
        NSLog(@"Skipping a for-loop iteration");
        continue;
    }
    NSLog(@"Current year: %d", modelYear + i);
}
```

While it's technically not a part of the C programming language, this is an appropriate time to introduce the `for-in` loop. This is referred to as the **fast-enumeration** syntax because it's a more efficient way to iterate over Objective-C collections like `NSSet` and `NSArray` than the traditional `for` and `while` loops.

```
// For-in loops ("Fast-enumeration," specific to Objective-C)
NSArray *models = @[@"Ford", @"Honda", @"Nissan", @"Porsche"];
for (id model in models) {
    NSLog(@"%@", model);
}
```

Macros

Macros are a low-level way to define symbolic constants and space-saving abbreviations. The `#define` directive maps a macro name to an expansion, which is an arbitrary sequence of characters. Before the compiler tries to parse the code, the preprocessor replaces all occurrences of the macro name with its expansion. In other words, it's a straightforward search-and-replace:

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

```
#define PI 3.14159
#define RAD_TO_DEG(radians) (radians * (180.0 / PI))

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        double angle = PI / 2;              // 1.570795
        NSLog(@"%f", RAD_TO_DEG(angle));    // 90.0
    }
    return 0;
}
```

This code snippet demonstrates the two types of C macros: object-like macros (`PI`) and function-like macros (`RAD_TO_DEG(radians)`). The only difference is that the latter is smart enough to accept arguments and alter their expansions accordingly.

Typedef

The `typedef` keyword lets you create new data types or redefine existing ones. After typedef'ing an `unsigned char` in the following example, we can use `ColorComponent` just like we would use `char`, `int`, `double`, or any other built-in type:

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

typedef unsigned char ColorComponent;

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        ColorComponent red = 255;
        ColorComponent green = 160;
        ColorComponent blue = 0;
        NSLog(@"Your paint job is (R: %hhu, G: %hhu, B: %hhu)",
              red, green, blue);
    }
    return 0;
}
```

While this adds some meaningful semantics to our code, `typedef` is more commonly used to turn `struct`'s and `enum`'s into convenient data types. This is demonstrated in the next two sections.

Structs

A `struct` is like a simple, primitive C object. It lets you aggregate several variables into a more complex data structure, but doesn't provide any OOP features (e.g., methods). For example, the following snippet uses a `struct` to group the components of an RGB color. Also notice how we `typedef` the `struct` so that we can access it via a meaningful name.

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

typedef struct {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
} Color;

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Color carColor = {255, 160, 0};
        NSLog(@"Your paint job is (R: %hhu, G: %hhu, B: %hhu)",
              carColor.red, carColor.green, carColor.blue);
    }
    return 0;
}
```

To populate the new `carColor` structure, we used the `{255, 160, 0}` **initializer syntax**. This assigns values in the same order as they were declared in the `struct`. And, as you can see, each of its fields can be accessed via dot-syntax.

Enums

The `enum` keyword is used to create an **enumerated type**, which is a collection of related constants. Like `structs`, it's often convenient to `typedef` enumerated types with a descriptive name:

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

typedef enum {
    FORD,
    HONDA,
    NISSAN,
    PORSCHE
} CarModel;

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        CarModel myCar = NISSAN;
        switch (myCar) {
            case FORD:
            case PORSCHE:
                NSLog(@"You like Western cars?");
                break;
            case HONDA:
            case NISSAN:
                NSLog(@"You like Japanese cars?");
                break;
            default:
                break;
        }
    }
    return 0;
}
```

Since the `myCar` variable was declared with the `CarModel` type, it can only store the four **enumerators** defined by the enumerated type: `FORD`, `HONDA`, `NISSAN`, and `PORSCHE`. Defining these in an enumerated type is more reliable than representing the various `CarModel`'s with arbitrary strings, as it's impervious to spelling errors (the compiler will let you know when you mistype one of the above enumerators).

The Cocoa frameworks rely on enums for many of their constants. For example, `NSSearchPathDirectory` defines the standard directory paths used by OS X. The Persistent Data chapter of Ry's Cocoa Tutorial provides many examples of how this works.

Primitive Arrays

Since Objective-C is a superset of C, it has access to the primitive arrays found in C. Generally, the higher-level `NSArray` and `NSMutableArray` classes provided by the Foundation Framework are much more convenient than C arrays; however, primitive arrays can still prove useful for performance intensive environments. Their syntax is as follows:

```
int years[4] = {1968, 1970, 1989, 1999};
years[0] = 1967;
for (int i=0; i<4; i++) {
    NSLog(@"The year at index %d is: %d", i, years[i]);
}
```

The `int years[4]` statement allocates a contiguous block of memory large enough to store 4 `int` values. We then populate the array using the `{1968, ...}` initializer syntax and access its elements by passing offsets between square brackets (e.g., `years[i]`).

Pointers

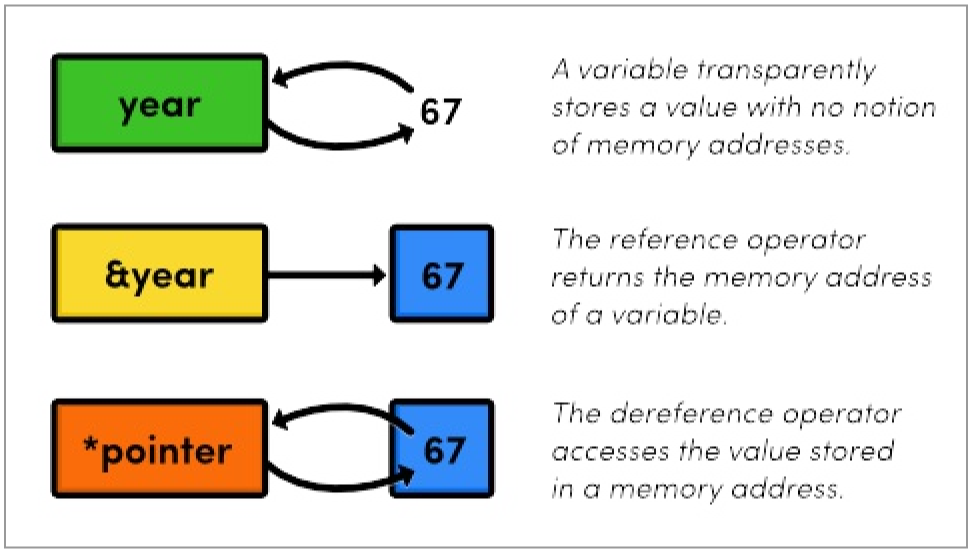
A pointer is a direct reference to a memory address. Whereas a variable acts as a transparent container for a value, pointers remove a layer of abstraction and let you see how that value is stored. This requires two new tools:

- The reference operator (`&`) returns the memory address of a normal variable. This is how you create pointers.
- The dereference operator (`*`) returns the contents of a pointer's memory address.

The following example demonstrates how to declare, create, and dereference pointers. Note that defining a pointer looks exactly like defining a normal variable, except it's prepended with an asterisk (`*`).

```
int year = 1967;          // Define a normal variable
int *pointer;             // Declare a pointer that points to an int
pointer = &year;          // Find the memory address of the variable
NSLog(@"%d", *pointer);   // Dereference the address to get its value
*pointer = 1990;          // Assign a new value to the memory address
NSLog(@"%d", year);       // Access the value via the variable
```

The behavior of these pointer operators can be visualized as follows:


year
67
A variable transparently stores a value with no notion of memory addresses.
&year
67
The reference operator returns the memory address of a variable.
*pointer
67
The dereference operator accesses the value stored in a memory address.

In the above example, pointers are merely an unnecessary abstraction for normal variables. Their real utility comes from the fact that you can *move* a pointer to the surrounding memory addresses. This is particularly useful for navigating arrays, which are just contiguous blocks of memory. For example, the code below uses a pointer to iterate through the elements of an array.

```
char model[5] = {'H', 'o', 'n', 'd', 'a'};
char *modelPointer = &model[0];
for (int i=0; i<5; i++) {
    NSLog(@"Value at memory address %p is %c",
          modelPointer, *modelPointer);
    modelPointer++;
}
NSLog(@"The first letter is %c", *(modelPointer - 5));
```

When used with a pointer, the `++` operator moves the it to the next memory address, which we can display through `NSLog()` with the `%p` specifier. Likewise, the `--` operator can be used to decrement the pointer to the previous address. And, as shown in the last line, you can access an arbitrary address relative to the current pointer position.

The Null Pointer

The null pointer is a special kind of pointer that doesn't point to anything. There is only one null pointer in C, and it is referenced through the `NULL` macro. This is useful for indicating empty variables—something that is not possible with a normal data type. For instance, the following snippet shows how `pointer` can be "emptied" using the null pointer.

```
int year = 1967;
int *pointer = &year;
NSLog(@"%d", *pointer);      // Do something with the value
```

```
pointer = NULL;             // Then invalidate it
```

The only way to represent an empty variable using `year` on its own is to set it to `0`. Of course, the problem is that `0` is still a perfectly valid value—it's not the *absence* of a value.

Void Pointers

A void pointer is a generic type that can point to *anything*. It's essentially a reference to an arbitrary memory address. Accordingly, more information is required to interpret the contents of a void pointer. The easiest way to do this is to simply cast it to a non-void pointer. For example, the `(int *)` statement in the following code interprets the contents of the void pointer as an `int` value.

```
int year = 1967;
void *genericPointer = &year;
int *intPointer = (int *)genericPointer;
NSLog(@"%d", *intPointer);
```

The generic nature of void pointers affords a lot flexibility. For example, the `NSString` class defines the following method for converting a C array into an Objective-C string object:

```
- (id)initWithBytes:(const void *)bytes
             length:(NSUInteger)length
           encoding:(NSStringEncoding)encoding
```

The `bytes` argument points to the first memory address of any kind of C array, the `length` argument defines how many bytes to read, and `encoding` determines how those bytes should be interpreted. Using a void pointer like this makes it possible to work with *any* type of character array. The alternative would be to define dedicated methods for single-byte, UTF-8, and UTF-16 characters, etc.

Pointers in Objective-C

This is all good background knowledge, but for your everyday Objective-C development, you probably won't need to use most of it. The only thing that you really have to understand is that *all* Objective-C objects are referenced as pointers. For instance, an `NSString` object must be stored as a pointer, not a normal variable:

```
NSString *model = @"Honda";
```

When it comes to null pointers, there is a slight difference between C and Objective-C. Whereas C uses `NULL`, Objective-C defines its own macro, `nil`, as its null object. A good rule of thumb is to use `nil` for variables that hold Objective-C objects and `NULL` when working with C pointers.

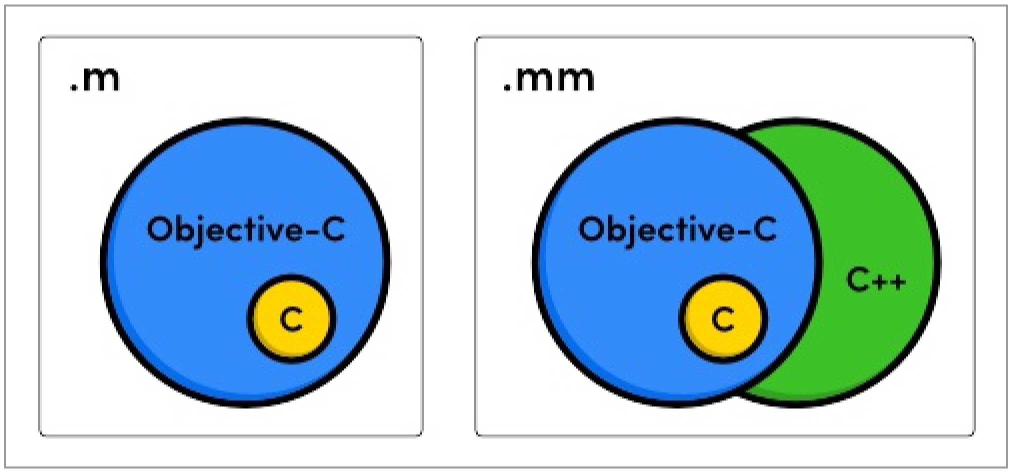
```
NSString *anObject;     // An Objective-C object
anObject = NULL;        // This will work
anObject = nil;         // But this is preferred
int *aPointer;          // A plain old C pointer
aPointer = nil;         // Don't do this
aPointer = NULL;        // Do this instead
```

Outside of variable declarations, the entire Objective-C syntax is designed to work with pointers. So, after defining an object pointer, you can basically forget about the fact that it's a pointer and interact with it as if it were a normal variable. This will be made abundantly clear from the examples throughout the rest of this tutorial.

Summary

This module introduced the fundamental aspects of the C programming language. While you're not expected to be a C expert just yet, we hope that you're feeling relatively comfortable with variables, conditionals, loops, structs, enums, and pointers. These tools form the foundation of any Objective-C program.

Objective-C *relies* on C for these basic constructs, but it also gives you the *option* of inserting C++ code directly into your source files. To tell the compiler to interpret your source code as either C, C++, or Objective-C, all you have to do is change the file extension to `.mm`.



Languages available to files with `.m` and `.mm` extensions

This unique language feature opens the door to the entire C/C++ ecosystem, which is a huge boon to Objective-C developers. For example, if you're building an iOS game and find yourself in need of a physics engine, you can leverage the well-known Box2D library (written in C++) with virtually no additional work.

The next module will complete our discussion of C with a brief look at functions. After that, we'll be more than ready to start working with Objective-C classes, methods, protocols, and the rest of its object-oriented goodness.

Functions

Along with variables, conditionals, and loops, functions are one of the fundamental components of any modern programming language. They let you reuse an arbitrary block of code throughout your application, which is necessary for organizing and maintaining all but the most trivial code bases. You'll find many examples of functions throughout the iOS and OS X frameworks.

Just like its other basic constructs, Objective-C relies entirely on the C programming language for functions. This module introduces the most important aspects of C functions, including basic syntax, the separation of declaration and implementation, common scope issues, and function library considerations.

Basic Syntax

There are four components to a C function: its return value, name, parameters, and associated code block. After you've defined these, you can **call** the function to execute its associated code by passing any necessary parameters between parentheses.

For example, the following snippet defines a function called `getRandomInteger()` that accepts two `int` values as parameters and returns another `int` value. Inside of the function, we access the inputs through the `minimum` and `maximum` parameters, and we return a calculated value via the `return` keyword. Then in `main()`, we call this new function and pass `-10` and `10` as arguments.

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


int getRandomInteger(int minimum, int maximum) {
    return arc4random_uniform((maximum - minimum) + 1) + minimum;
}


int main(int argc, const char * argv[]) {
    @autoreleasepool {
        int randomNumber = getRandomInteger(-10, 10);
        NSLog(@"Selected a random number between -10 and 10: %d",
              randomNumber);
    }
    return 0;
}
```

The built-in `arc4random_uniform()` function returns a random number between 0 and whatever argument you pass. (This is preferred over the older `rand()` and `random()` algorithms.)

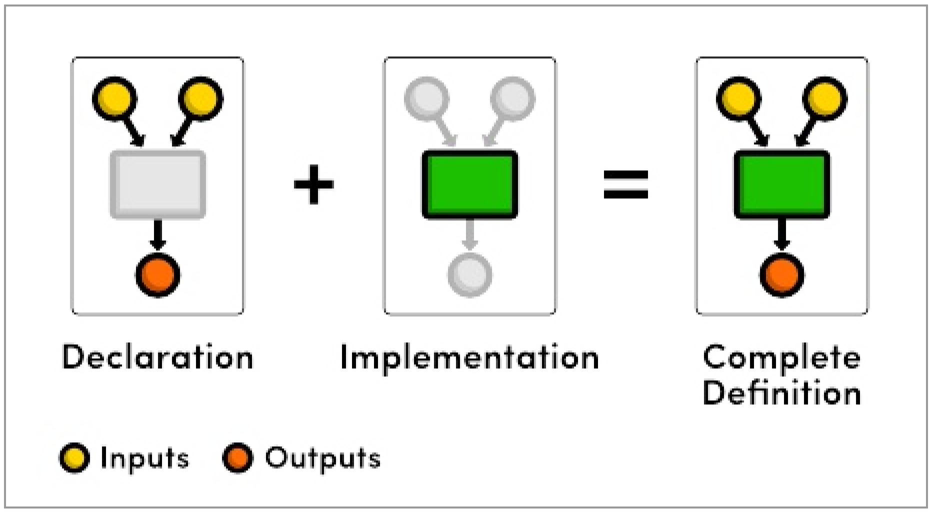
Functions let you use pointer references as return values or parameters, which means that they can be seamlessly integrated with Objective-C objects (remember that all objects are represented as pointers). For example, try changing `main.m` to the following.

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

NSString *getRandomMake(NSArray *makes) {
    int maximum = (int)[makes count];
    int randomIndex = arc4random_uniform(maximum);
    return makes[randomIndex];
}

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSArray *makes = @[@"Honda", @"Ford", @"Nissan", @"Porsche"];
        NSLog(@"Selected a %@", getRandomMake(makes));
    }
    return 0;
}
```

This `getRandomMake()` function accepts an `NSArray` object as an argument and returns an `NSString` object. Note that it uses the same asterisk syntax as pointer variable declarations.

Declarations vs. Implementations

Functions need to be defined *before* they are used. If you were to define the above `getRandomMake()` function *after* `main()`, the compiler wouldn't be able to find it when you try to call it in `main()`. This imposes a rather strict structure on developers and can make it hard to organize larger applications. To solve this problem, C lets you separate the declaration of a function from its implementation.



Function declarations vs. implementations

A function **declaration** tells the compiler what the function's inputs and outputs look like. By providing the data types for the return value and parameters of a function, the compiler can make sure that you're using it properly without knowing what it actually does. The corresponding **implementation** attaches a code block to the declared function. Together, these give you a complete function **definition**.

The following example declares the `getRandomMake()` function so that it can be used in `main()` before it gets implemented. Notice that the declaration only needs the data types of the parameters—their names can be omitted (if desired).

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

// Declaration
NSString *getRandomMake(NSArray *);

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSArray *makes = @[@"Honda", @"Ford", @"Nissan", @"Porsche"];
        NSLog(@"Selected a %@", getRandomMake(makes));
    }
    return 0;
}

// Implementation
NSString *getRandomMake(NSArray *makes) {
    int maximum = (int)[makes count];
    int randomIndex = arc4random_uniform(maximum);
    return makes[randomIndex];
}
```

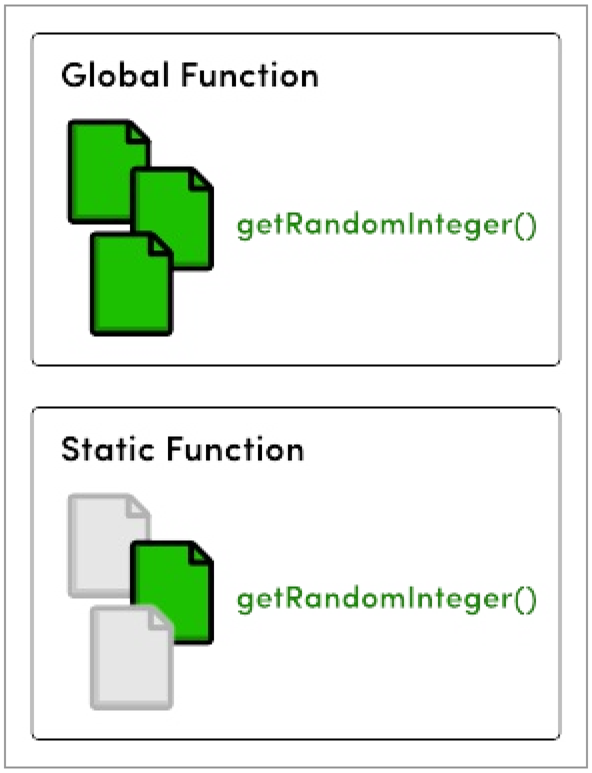
As we'll see in Function Libraries, separating function declarations from implementations is really more useful for organizing large frameworks.

The Static Keyword

The `static` keyword lets you alter the availability of a function or variable. Unfortunately, it has different effects depending on where you use it. This section explains two common use cases for the `static` keyword.

Static Functions

By default, all functions have a global scope. This means that as soon as you define a function in one file, it's immediately available everywhere else. The `static` specifier lets you limit the function's scope to the current file, which is useful for creating "private" functions and avoiding naming conflicts.



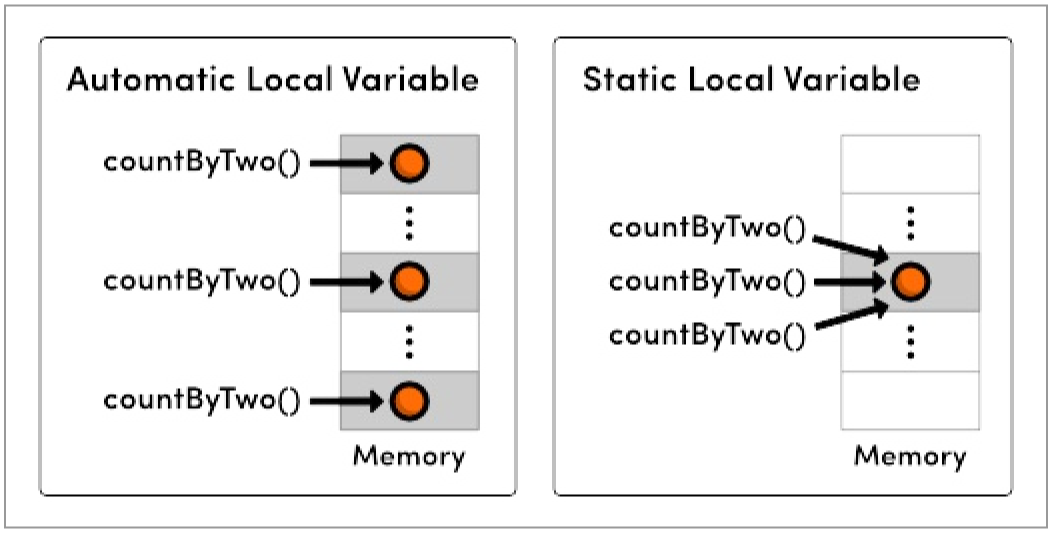
Globally-scoped functions vs. statically-scoped functions

The following example shows you how to create a static function. If you were to add this code to another file (e.g., a dedicated function library), you would not be able to access `getRandomInteger()` from `main.m`. Note that the `static` keyword should be used on both the function declaration and implementation.

```
// Static function declaration
static int getRandomInteger(int, int);


// Static function implementation
static int getRandomInteger(int minimum, int maximum) {
    return arc4random_uniform((maximum - minimum) + 1) + minimum;
}
```

Static Local Variables

Variables declared inside of a function (also called **automatic local variables**) are reset each time the function is called. This is an intuitive default behavior, as the function behaves consistently regardless of how many times you call it. However, when you use the `static` modifier on a local variable, the function "remembers" its value across invocations.



Independent automatic variables vs. shared static variables

For example, the `currentCount` variable in the following snippet never gets reset, so instead of storing the count in a variable inside of `main()`, we can let `countByTwo()` do the recording for us.

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


int countByTwo() {
    static int currentCount = 0;
    currentCount += 2;
    return currentCount;
}


int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSLog(@"%d", countByTwo());    // 2
        NSLog(@"%d", countByTwo());    // 4
        NSLog(@"%d", countByTwo());    // 6
    }
    return 0;
}
```

But, unlike the static functions discussed in the previous section, this use of the `static` keyword does *not* affect the scope of local variables. That is to say, local variables are still only accessible inside of the function itself.

Function Libraries

Objective-C doesn't support namespaces, so to prevent naming collisions with other global functions, large frameworks need to prefix their functions (and classes) with a unique identifier. This is why you see built-in functions like `NSMakeRange()` and `CGImageCreate()` instead of just `makeRange()` and `imageCreate()`.

When creating your own function libraries, you should declare functions in a dedicated header file and implement them in a separate implementation file (just like Objective-C classes). This lets files that use the library import the header without worrying about how its functions are implemented. For example, the header for a `CarUtilities` library might look something like the following:

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

NSString *CUGetRandomMake(NSArray *makes);
NSString *CUGetRandomModel(NSArray *models);
NSString *CUGetRandomMakeAndModel(NSDictionary *makesAndModels);
```

The corresponding implementation file defines what these functions actually do. Since other files are not supposed to import the implementation, you can use the `static` specifier to create "private" functions for internal use by the library.

```
// CarUtilities.m
#import "CarUtilities.h"

// Private function declaration
```

```
static id getRandomItemFromArray(NSArray *anArray);


// Public function implementations

NSString *CUGetRandomMake(NSArray *makes) {

    return getRandomItemFromArray(makes);

}

NSString *CUGetRandomModel(NSArray *models) {

    return getRandomItemFromArray(models);

}

NSString *CUGetRandomMakeAndModel(NSDictionary *makesAndModels) {

    NSArray *makes = [makesAndModels allKeys];

    NSString *randomMake = CUGetRandomMake(makes);

    NSArray *models = makesAndModels[randomMake];

    NSString *randomModel = CUGetRandomModel(models);

    return [randomMake stringByAppendingFormat:@" %@", randomModel];

}


// Private function implementation

static id getRandomItemFromArray(NSArray *anArray) {

    int maximum = (int)[anArray count];

    int randomIndex = arc4random_uniform(maximum);

    return anArray[randomIndex];

}
```

Now, `main.m` can import the header and call the functions as if they were defined in the same file. Also notice that trying to call the static `getRandomItemFromArray()` function from `main.m` results in a compiler error.

```
// main.m

#import <Foundation/Foundation.h>

#import "CarUtilities.h"


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

    @autoreleasepool {

        NSDictionary *makesAndModels = @{

            @"Ford": @[@"Explorer", @"F-150"],

            @"Honda": @[@"Accord", @"Civic", @"Pilot"],

            @"Nissan": @[@"370Z", @"Altima", @"Versa"],

            @"Porsche": @[@"911 Turbo", @"Boxster", @"Cayman S"]
        };

        NSString *randomCar = CUGetRandomMakeAndModel(makesAndModels);

        NSLog(@"Selected a %@", randomCar);

    }

    return 0;

}
```

Summary

This module finished up our introduction to the C programming language with an in-depth look at functions. We learned how to declare and implement functions, change their scope, make them remember local variables, and organize large function libraries.

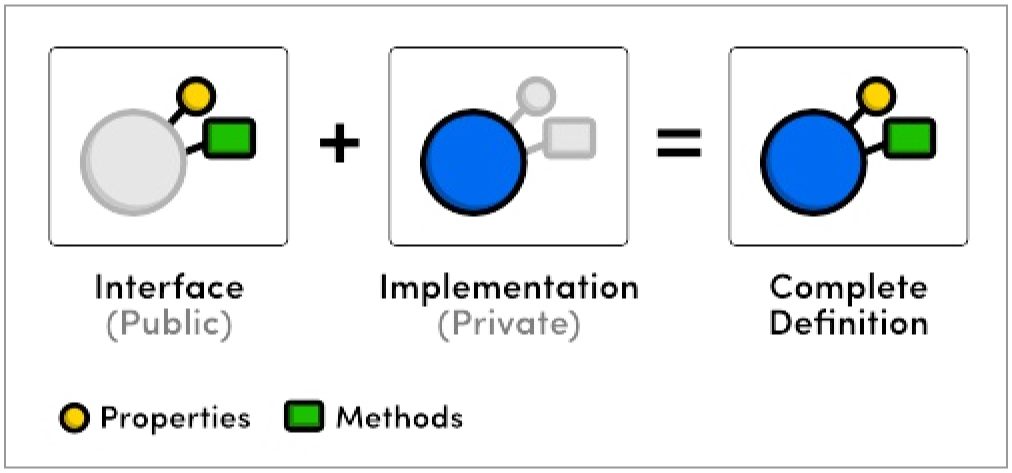
While most of the functionality behind the Cocoa and Cocoa Touch frameworks is encapsulated as Objective-C classes, there's no shortage of built-in functions. The ones you're most likely to encounter in the real world are utility functions like `NSLog()` and convenience functions like `NSMakeRect()` that create and configure complex objects using a friendlier API.

We're now ready to start tackling the object-oriented aspects of Objective-C. In the next module, we'll learn how to define classes, instantiate objects, set properties, and call methods.

Classes

As in many other object-oriented programming language, Objective-C classes provide the blueprint for creating objects. First, you define a reusable set of properties and behaviors inside of a class. Then, you instantiate objects from that class to interact with those properties and behaviors.

Objective-C is similar to C++ in that it abstracts a class's interface from its implementation. An **interface** declares the public properties and methods of a class, and the corresponding **implementation** defines the code that actually makes these properties and methods work. This is the same separation of concerns that we saw with functions.



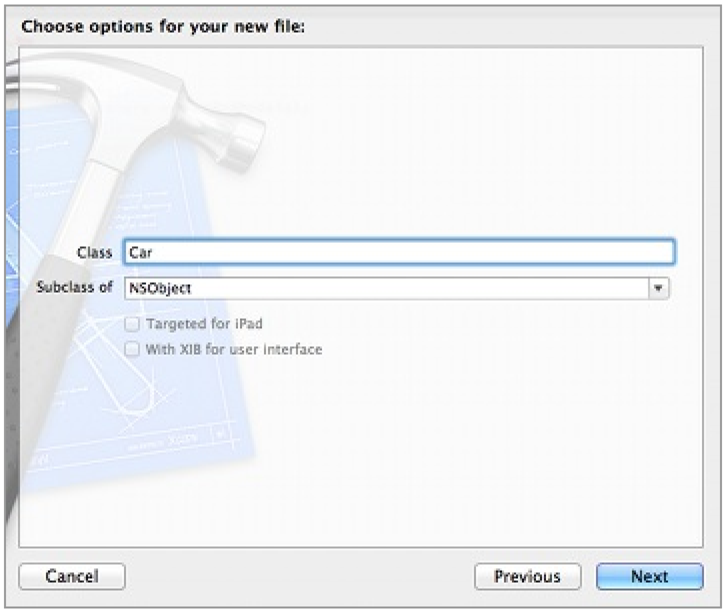
A class's interface and implementation

In this module, we'll explore the basic syntax for class interfaces, implementations, properties, and methods, as well as the canonical way to instantiate objects. We'll also introduce some of Objective-C's introspection and reflection capabilities.

Creating Classes

We'll be working with a class called `Car` whose interface resides in a file named `Car.h` (also called a "header") and whose implementation resides in `Car.m`. These are the standard file extensions for Objective-C classes. The class's header file is what other classes use when they need to interact with it, and its implementation file is used only by the compiler.

Xcode provides a convenient template for creating new classes. To create our `Car` class, navigate to *File > New > File…* or use the *Cmd+N* shortcut. For the template, choose *Objective-C class* under the *iOS > Cocoa Touch* category. After that, you'll be presented with some configuration options:


Choose options for your new file:

Class: Car

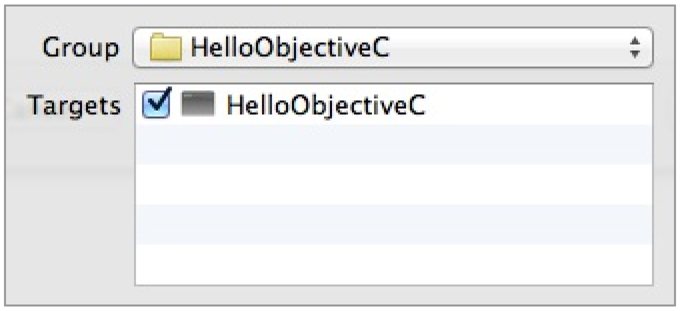
Subclass of: NSObject

☐ Targeted for iPad

☐ With XIB for user interface

Creating the `Car` class

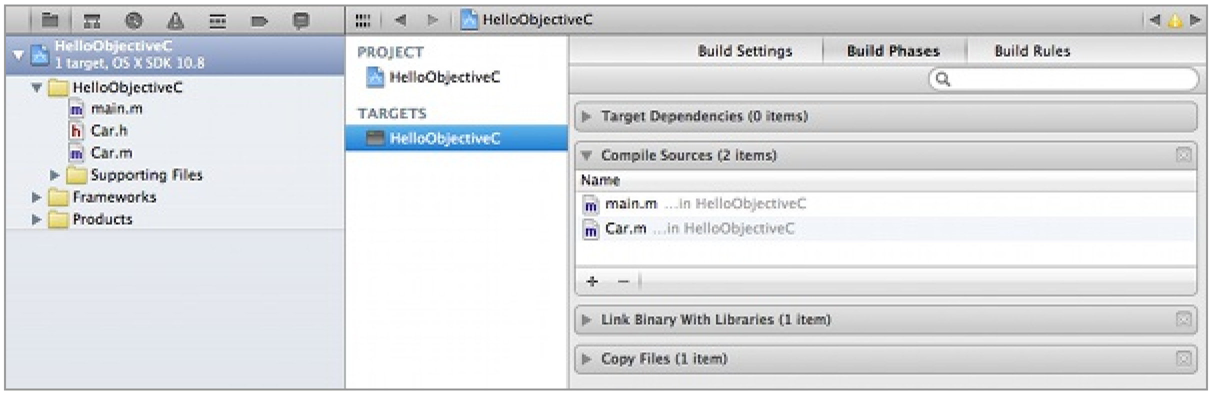
Use `Car` for the *Class* field and `NSObject` for the *Subclass of* field. `NSObject` is Objective-C's top-level class from which almost all others inherit. Clicking *Next* will prompt you to select a location for the file. Save it in the top-level of the project directory. At the bottom of that dialog, make sure that your project is checked in the *Targets* section. This is what actually adds the class to the list of compiled sources.


Group: HelloObjectiveC

Targets: ☑ HelloObjectiveC

Adding the class to the build target

After clicking *Next*, you should see new `Car.h` and `Car.m` files in Xcode's *Project Navigator*. If you select the project name in the navigator, you'll also find `Car.m` in the *Build Phases* tab under the *Compile Sources* section. Any files that you want the compiler to see must be in this list (if you didn't create your source files through Xcode, this is where you can manually add them).



The list of compiled source files

Interfaces

`Car.h` contains some template code, but let’s go ahead and change it to the following. This declares a property called `model` and a method called `drive`.

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

@interface Car : NSObject {
    // Protected instance variables (not recommended)
}

@property (copy) NSString *model;

- (void)drive;

@end
```

An interface is created with the `@interface` directive, after which come the class and the superclass name, separated by a colon. Protected variables can be defined inside of the curly braces, but most developers treat instance variables as implementation details and prefer to store them in the `.m` file instead of the interface.

The `@property` directive declares a public property, and the `(copy)` attribute defines its memory management behavior. In this case, the value assigned to `model` will be stored as a copy instead of a direct pointer. The Properties module discusses this in more detail. Next come the property’s data type and name, just like a normal variable declaration.

The `-(void)drive` line declares a method called `drive` that takes no parameters, and the `(void)` portion defines its return type. The minus sign prepended to the method marks it as an *instance* method (opposed to a *class* method).

Implementations

The first thing any class implementation needs to do is import its corresponding interface. The `@implementation` directive is similar to `@interface`, except you don’t need to include the super class. Private instance variables can be stored between curly braces after the class name:

```
// Car.m
#import "Car.h"

@implementation Car {
    // Private instance variables
    double _odometer;
}

@synthesize model = _model;    // Optional for Xcode 4.4+

- (void)drive {
    NSLog(@"Driving a %@. Vrooooom!", self.model);
}

@end
```

`@synthesize` is a convenience directive that automatically generates accessor methods for the property. By default, the getter is simply the property name (`model`), and the setter is the capitalized name with the `set` prefix (`setModel`). This is much easier than manually creating accessors for every property. The `_model` portion of the synthesize statement defines the private instance variable name to use for the property.

As of Xcode 4.4, properties declared with `@property` will be automatically synthesized, so it's safe to omit the `@synthesize` line if you're ok with the default instance variable naming conventions.

The `drive` implementation has the same signature as the interface, but it's followed by whatever code should be executed when the method is called. Note how we accessed the value via `self.model` instead of the `_model` instance variable. This is a best practice step because it utilizes the property's accessor methods. Typically, the only place you'll need to directly access instance variables is in `init` methods and the `dealloc` method.

The `self` keyword refers to the instance calling the method (like `this` in C++ and Java). In addition to accessing properties, this can be used to call other methods defined on the same class (e.g., `[self anotherMethod]`). We'll see many examples of this throughout the tutorial.

Instantiation and Usage

Any files that need access to a class must import its header file (`Car.h`)—they should never, ever try to access the implementation file directly. That would defeat the goal of separating the public API from its underlying implementation. So, to see our `Car` class in action, change `main.m` to the following.

```
// main.m
#import <Foundation/Foundation.h>
#import "Car.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Car *toyota = [[Car alloc] init];

        [toyota setModel:@"Toyota Corolla"];
        NSLog(@"Created a %@", [toyota model]);

        toyota.model = @"Toyota Camry";
        NSLog(@"Changed the car to a %@", toyota.model);

        [toyota drive];

    }
    return 0;
}
```

After the interface has been imported with the `#import` directive, you can instantiate objects with the `alloc/init` pattern shown above. As you can see, instantiation is a two-step process: first you must allocate some memory for the object by calling the `alloc` method, then you need to initialize it so it's ready to use. You should never use an uninitialized object.

It's worth repeating that *all* objects must be stored as pointers. This is why we used `Car *toyota` instead of `Car toyota` to declare the variable.

To call a method on an Objective-C object, you place the instance and the method in square brackets, separated by a space. Arguments are passed after the method name, preceded by a colon. So, if you're coming from a C++, Java, or Python background, the `[toyota setModel:@"Toyota Corolla"]` call would translate to:

```
toyota.setModel("Toyota Corolla");
```

This square-bracket syntax can be unsettling for newcomers to the language, but rest assured, you'll be more than comfortable with Objective-C's method conventions after reading through the Methods module.

This example also shows you both ways to work with an object's properties. You can either use the synthesized `model` and `setModel` accessor methods, or you can use the convenient dot-syntax, which should be more familiar to developers who have been using Simula-style languages.

Class Methods and Variables

The above snippets define instance-level properties and methods, but it's also possible to define class-level ones. These are commonly called "static" methods/properties in other programming languages (not to be confused with the `static` keyword).

Class method declarations look just like instance methods, except they are prefixed with a plus sign instead of a minus sign. For example, let's add the following class-level method to `Car.h`:

```
// Car.h
+ (void)setDefaultModel:(NSString *)aModel;
```

Similarly, a class method *implementation* is also preceded by a plus sign. While there is technically no such thing as a class-level variable in Objective-C, you can emulate one by declaring a `static` variable before defining the implementation:

```
// Car.m
#import "Car.h"


static NSString *_defaultModel;


@implementation Car {
...


+ (void)setDefaultModel:(NSString *)aModel {
    _defaultModel = [aModel copy];
}


@end
```

The `[aModel copy]` call creates a copy of the parameter instead of assigning it directly. This is what's going on under the hood when we used the `(copy)` attribute on the `model` property.

Class methods use the same square-bracket syntax as instance methods, but they must be called directly on the class, as shown below. They *cannot* be called on an instance of that class (`[toyota setDefaultModel:@"Model T"]` will throw an error).

```
// main.m
[Car setDefaultModel:@"Nissan Versa"];
```

“Constructor” Methods

There are no constructor methods in Objective-C. Instead, an object is **initialized** by calling the `init` method immediately after it’s allocated. This is why instantiation is always a two-step process: allocate, then initialize. There is also a class-level initialization method that will be discussed in a moment.

`init` is the default initialization method, but you can also define your own versions to accept configuration parameters. There’s nothing special about custom initialization methods—they’re just normal instance methods, except the method name should always begin with `init`. An exemplary “constructor” method is shown below.

```
// Car.h
- (id)initWithModel:(NSString *)aModel;
```

To implement this method, you should follow the canonical initialization pattern shown in `initWithModel:` below. The `super` keyword refers to the parent class, and again, the `self` keyword refers to the instance calling the method. Go ahead and add the following methods to `Car.m`.

```
// Car.m
- (id)initWithModel:(NSString *)aModel {
    self = [super init];
    if (self) {
        // Any custom setup work goes here
        _model = [aModel copy];
        _odometer = 0;
    }
    return self;
}

- (id)init {
    // Forward to the "designated" initialization method
    return [self initWithModel:_defaultModel];
}
```

Initialization methods should always return a reference to the object itself, and if it cannot be initialized, it should return `nil`. This is why we need to check if `self` exists before trying to use it. There should typically only be one initialization method that needs to do this, and the rest should forward calls to this **designated initializer**. This eliminates boilerplate code when you have several custom `init` methods.

Also notice how we directly assigned values to the `_model` and `_odometer` instance variables in `initWithModel:`. Remember that this is one of the only places you should do this—in the rest of your methods you should be using `self.model` and `self.odometer`.

Class-Level Initialization

The `initialize` method is the class-level equivalent of `init`. It gives you a chance to set up the class before anyone uses it. For example, we can use this to populate the `_defaultModel` variable with a valid value, like so:

```
// Car.m
+ (void)initialize {
    if (self == [Car class]) {
        // Makes sure this isn't executed more than once
        _defaultModel = @"Nissan Versa";
    }
}
```

The `initialize` class method is called once for every class before the class is used. This includes all subclasses of `Car`, which means that `Car` will get two `initialize` calls if one of its subclasses didn't re-implement it. As a result, it's good practice to use the `self == [Car class]` conditional to ensure that the initialization code is only run once. Also note that in class methods, the `self` keyword refers to the *class itself*, not an instance.

Objective-C doesn't force you to mark methods as overrides. Even though `init` and `initialize` are both defined by its superclass, `NSObject`, the compiler won't complain when you redefine them in `Car.m`.

The next iteration of `main.m` shows our custom initialization methods in action. Before the first time the class is used, `[Car initialize]` is called automatically, setting `_defaultModel` to `@"Nissan Versa"`. This can be seen in the first `NSLog()`. You can also see the result of the custom initialization method (`initWithModel:`) in the second log output.

```
// main.m
#import <Foundation/Foundation.h>
#import "Car.h"

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

        // Instantiating objects
        Car *nissan = [[Car alloc] init];
        NSLog(@"Created a %@", [nissan model]);

        Car *chevy = [[Car alloc] initWithModel:@"Chevy Corvette"];
        NSLog(@"Created a %@, too.", chevy.model);

    }
    return 0;
}
```

Dynamic Typing

Classes themselves are represented as objects, which makes it possible to query their properties (introspection), or even change their behavior on-the-fly (reflection). These are very powerful dynamic typing capabilities, as they let you call methods and set properties on objects even when you don't know what type of object they are.

The easiest way to get a class object is via the `class` class-level method (apologies for the redundant terminology). For example, `[Car class]` returns an object representing the `Car` class. You can pass this object around to methods like `isMemberOfClass:` and `isKindOfClass:` to get information about other instances. A comprehensive example is included below.

```
// main.m
#import <Foundation/Foundation.h>
#import "Car.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Car *delorean = [[Car alloc] initWithModel:@"DeLorean"];

        // Get the class of an object
        NSLog(@"%@ is an instance of the %@ class",
              [delorean model], [delorean class]);

        // Check an object against a class and all subclasses
        if ([delorean isKindOfClass:[NSObject class]]) {
            NSLog(@"%@ is an instance of NSObject or one "
                  "of its subclasses",
                  [delorean model]);
        } else {
            NSLog(@"%@ is not an instance of NSObject or "
                  "one of its subclasses",
                  [delorean model]);
        }

        // Check an object against a class, but not its subclasses
        if ([delorean isMemberOfClass:[NSObject class]]) {
            NSLog(@"%@ is a instance of NSObject",
                  [delorean model]);
        } else {
            NSLog(@"%@ is not an instance of NSObject",
                  [delorean model]);
        }

        // Convert between strings and classes
        if (NSClassFromString(@"Car") == [Car class]) {
            NSLog(@"I can convert between strings and classes!");
        }
    }
    return 0;
}
```

The `NSClassFromString()` function is an alternative way to get your hands on a class object. This is very flexible, as it lets you dynamically request class objects at runtime; however, it's also rather inefficient. For this reason, you should opt for the `class` method whenever possible.

If you're interested in dynamic typing, be sure to check out Selectors and The `id` Type.

Summary

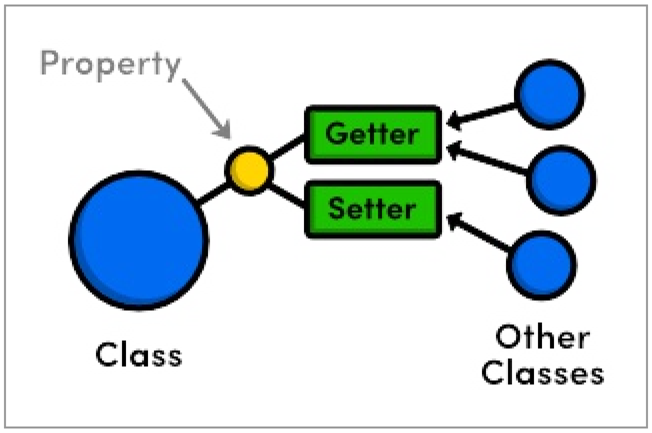
In this module, we learned how to create classes, instantiate objects, define initialization methods, and work with class-level methods and variables. We also took a brief look at dynamic typing.

The previous module mentioned that Objective-C doesn't support namespaces, which is why the Cocoa functions require prefixes like `NS`, `CA`, `AV`, etc to avoid naming collisions. This applies to classes, too. The recommended convention is to use a three-letter prefix for your application-specific classes (e.g., `XYZCar`).

While this is pretty much everything you need to know to start writing your own classes, we did skim over some important details, so don't worry if you're feeling not entirely comfortable with properties or methods. The next module will begin filling in these holes with a closer look at the `@property` directive and all of the attributes that affect its behavior.

Properties

An object’s properties let other objects inspect or change its state. But, in a well-designed object-oriented program, it’s not possible to directly access the internal state of an object. Instead, **accessor methods** (getters and setters) are used as an abstraction for interacting with the object’s underlying data.



Interacting with a property via accessor methods

The goal of the `@property` directive is to make it easy to create and configure properties by automatically generating these accessor methods. It allows you to specify the behavior of a public property on a semantic level, and it takes care of the implementation details for you.

This module surveys the various attributes that let you alter getter and setter behavior. Some of these attributes determine how properties handle their underlying memory, so this module also serves as a practical introduction to memory management in Objective-C. For a more detailed discussion, please refer to Memory Management.

The @property Directive

First, let’s take a look at what’s going on under the hood when we use the `@property` directive. Consider the following interface for a simple `Car` class and its corresponding implementation.

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

@interface Car : NSObject

@property BOOL running;

@end
```

```
// Car.m
#import "Car.h"

@implementation Car

@synthesize running = _running;    // Optional for Xcode 4.4+

@end
```

The compiler generates a getter and a setter for the `running` property. The default naming convention is to use the property itself as the getter, prefix it with `set` for the setter, and prefix it with an underscore for the instance variable, like so:

```
- (BOOL)running {
    return _running;
}
- (void)setRunning:(BOOL)newValue {
    _running = newValue;
}
```

After declaring the property with the `@property` directive, you can call these methods as if they were included in your class's interface and implementation files. You can also override them in `Car.m` to supply custom getter/setters, but this makes the `@synthesize` directive mandatory. However, you should rarely need custom accessors, since `@property` attributes let you do this on an abstract level.

Properties accessed via dot-notation get translated to the above accessor methods behind the scenes, so the following `honda.running` code actually calls `setRunning:` when you assign a value to it and the `running` method when you read a value from it:

```
// main.m
#import <Foundation/Foundation.h>
#import "Car.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Car *honda = [[Car alloc] init];
        honda.running = YES;                // [honda setRunning:YES]
        NSLog(@"%d", honda.running);        // [honda running]
    }
    return 0;
}
```

To change the behavior of the generated accessors, you can specify attributes in parentheses after the `@property` directive. The rest of this module introduces the available attributes.

The getter= and setter= Attributes

If you don't like `@property`'s default naming conventions, you can change the getter/setter method names with the `getter=` and `setter=` attributes. A common use case for this is Boolean properties, whose getters are conventionally prefixed with `is`. Try changing the property declaration in `Car.h` to the following.

```
@property (getter=isRunning) BOOL running;
```

The generated accessors are now called `isRunning` and `setRunning`. Note that the public property is still called `running`, and this is what you should use for dot-notation:

```
Car *honda = [[Car alloc] init];
honda.running = YES;                // [honda setRunning:YES]
NSLog(@"%d", honda.running);        // [honda isRunning]
NSLog(@"%d", [honda running]);      // Error: method no longer exists
```

These are the only attributes that take an argument (the accessor method name)—all of the others are Boolean flags.

The readonly Attribute

The `readonly` attribute is an easy way to make a property read-only. It omits the setter method and prevents assignment via dot-notation, but the getter is unaffected. As an example, let's change our `Car` interface to the following. Notice how you can specify multiple attributes by separating them with a comma.

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


@interface Car : NSObject


@property (getter=isRunning, readonly) BOOL running;
```

```
- (void)startEngine;
- (void)stopEngine;

@end
```

Instead of letting other objects change the `running` property, we'll set it internally via the `startEngine` and `stopEngine` methods. The corresponding implementation can be found below.

```
// Car.m
#import "Car.h"

@implementation Car

- (void)startEngine {
    _running = YES;
}
- (void)stopEngine {
    _running = NO;
}

@end
```

Remember that `@property` also generates an instance variable for us, which is why we can access `_running` without declaring it anywhere (we can't use `self.running` here because the property is read-only). Let's test this new `Car` class by adding the following snippet to `main.m`.

```
Car *honda = [[Car alloc] init];
[honda startEngine];
NSLog(@"Running: %d", honda.running);
honda.running = NO;                         // Error: read-only property
```

Up until this point, properties have really just been convenient shortcuts that let us avoid writing boilerplate getter and setter methods. This will not be the case for the remaining attributes, which significantly alter the behavior of their properties. They also only apply to properties that store Objective-C objects (opposed to primitive C data types).

The nonatomic Attribute

Atomicity has to do with how properties behave in a threaded environment. When you have more than one thread, it's possible for the setter and the getter to be called at the same time. This means that the getter/setter can be interrupted by another operation, possibly resulting in corrupted data.

Atomic properties lock the underlying object to prevent this from happening, guaranteeing that the get or set operation is working with a complete value. However, it's important to understand that this is only one aspect of thread-safety—using atomic properties does not necessarily mean that your code is thread-safe.

Properties declared with `@property` are atomic by default, and this does incur some overhead. So, if you're not in a multi-threaded environment (or you're implementing your own thread-safety), you'll want to override this behavior with the `nonatomic` attribute, like so:

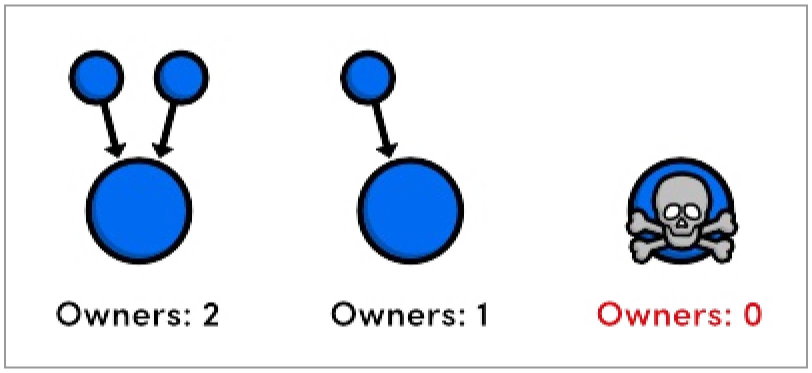
```
@property (nonatomic) NSString *model;
```

There is also a small, practical caveat with atomic properties. Accessors for atomic properties must *both* be either generated or user-defined. Only non-atomic properties let you mix-and-match synthesized accessors with custom ones. You can see this by removing `nonatomic` from the above code and adding a custom getter in `Car.m`.

Memory Management

In any OOP language, objects reside in the computer's memory, and—especially on mobile devices—this is a scarce resource. The goal of a memory management system is to make sure that programs don't take up any more space than they need to by creating and destroying objects in an efficient manner.

Many languages accomplish this through garbage collection, but Objective-C uses a more efficient alternative called **object ownership**. When you start interacting with an object, you're said to *own* that object, which means that it's guaranteed to exist as long as you're using it. When you're done with it, you relinquish ownership, and—if the object has no other owners—the operating system destroys the object and frees up the underlying memory.



Destroying an object with no owners

With the advent of Automatic Reference Counting, the compiler manages all of your object ownership automatically. For the most part, this means that you'll never to worry about how the memory management system actually works. But, you do have to understand the `strong`, `weak` and `copy` attributes of `@property`, since they tell the compiler what kind of relationship objects should have.

The strong Attribute

The `strong` attribute creates an owning relationship to whatever object is assigned to the property. This is the implicit behavior for all object properties, which is a safe default because it makes sure the value exists as long as it's assigned to the property.

Let's take a look at how this works by creating another class called `Person`. It's interface just declares a `name` property:

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

@interface Person : NSObject

@property (nonatomic) NSString *name;

@end
```

The implementation is shown below. It uses the default accessors generated by `@property`. It also overrides `NSObject`'s `description` method, which returns the string representation of the object.

```
// Person.m
#import "Person.h"

@implementation Person

- (NSString *)description {
    return self.name;
}

@end
```

Next, let's add a `Person` property to the `Car` class. Change `Car.h` to the following.

```
// Car.h
```

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

@interface Car : NSObject

@property (nonatomic) NSString *model;
@property (nonatomic, strong) Person *driver;

@end
```

Then, consider the following iteration of `main.m`:

```
// main.m
#import <Foundation/Foundation.h>
#import "Car.h"
#import "Person.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Person *john = [[Person alloc] init];
        john.name = @"John";

        Car *honda = [[Car alloc] init];
        honda.model = @"Honda Civic";
        honda.driver = john;

        NSLog(@"%@ is driving the %@", honda.driver, honda.model);
    }
    return 0;
}
```

Since `driver` is a strong relationship, the `honda` object takes ownership of `john`. This ensures that it will be valid as long as `honda` needs it.

The weak Attribute

Most of the time, the `strong` attribute is intuitively what you want for object properties. However, strong references pose a problem if, for example, we need a reference from `driver` back to the `Car` object he's driving. First, let's add a `car` property to `Person.h`:

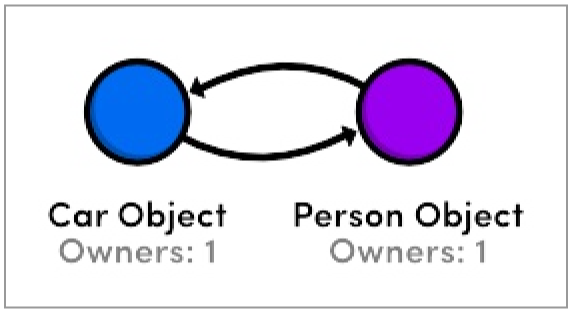
```
// Person.h
#import <Foundation/Foundation.h>

@class Car;

@interface Person : NSObject

@property (nonatomic) NSString *name;
@property (nonatomic, strong) Car *car;

@end
```

The `@class Car` line is a forward declaration of the `Car` class. It's like telling the compiler, "Trust me, the `Car` class exists, so don't try to find it right now." We have to do this instead of our usual `#import` statement because `Car` also imports `Person.h`, and we would have an endless loop of imports. (Compilers don't like endless loops.)

Next, add the following line to `main.m` right after the `honda.driver` assignment:

```
honda.driver = john;
john.car = honda;       // Add this line
```

We now have an owning relationship from `honda` to `john` and another owning relationship from `john` to `honda`. This means that both objects will *always* be owned by another object, so the memory management system won't be able to destroy them even if they're no longer needed.

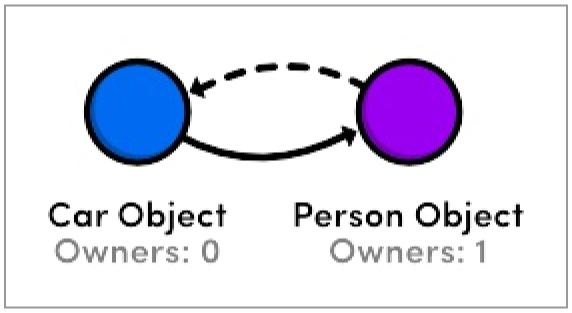


A retain cycle between the `Car` and `Person` classes

This is called a **retain cycle**, which is a form of memory leak, and memory leaks are bad. Fortunately, it's very easy to fix this problem—just tell one of the properties to maintain a **weak reference** to the other object. In `Person.h`, change the `car` declaration to the following:

```
@property (nonatomic, weak) Car *car;
```

The `weak` attribute creates a non-owning relationship to `car`. This allows `john` to have a reference to `honda` while avoiding a retain cycle. But, this also means that there is a possibility that `honda` will be destroyed while `john` still has a reference to it. Should this happen, the `weak` attribute will conveniently set `car` to `nil` in order to avoid a dangling pointer.



A weak reference from the `Person` class to `Car`

A common use case for the `weak` attribute is parent-child data structures. By convention, the parent object should maintain a strong reference with it's children, and the children should store a weak reference back to the parent. Weak references are also an inherent part of the delegate design pattern.

The point to take away is that two objects should never have strong references to each other. The `weak` attribute makes it possible to maintain a cyclical relationship without creating a retain cycle.

The copy Attribute

The `copy` attribute is an alternative to `strong`. Instead of taking ownership of the existing object, it creates a copy of whatever you assign to the property, then takes ownership of that. Only objects that conform to the `NSCopying` protocol can use this attribute.

Properties that represent values (opposed to connections or relationships) are good candidates for copying. For example, developers usually copy `NSString` properties instead of strongly reference them:

```
// Car.h
@property (nonatomic, copy) NSString *model;
```

Now, `Car` will store a brand new instance of whatever value we assign to `model`. If you're working with mutable values, this has the added perk of freezing the object at whatever value it had when it was assigned. This is demonstrated below:

```
// main.m
#import <Foundation/Foundation.h>
#import "Car.h"
```

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Car *honda = [[Car alloc] init];
        NSMutableString *model = [NSMutableString stringWithString:@"Honda Civic"];
        honda.model = model;

        NSLog(@"%@", honda.model);
        [model setString:@"Nissa Versa"];
        NSLog(@"%@", honda.model);            // Still "Honda Civic"
    }
    return 0;
}
```

`NSMutableString` is a subclass of `NSString` that can be edited in-place. If the `model` property didn't create a copy of the original instance, we would be able to see the altered string (`Nissan Versa`) in the second `NSLog()` output.

Other Attributes

The above `@property` attributes are all you should need for modern Objective-C applications (iOS 5+), but there are a few others that you may encounter in older libraries or documentation.

The retain Attribute

The `retain` attribute is the Manual Retain Release version of `strong`, and it has the exact same effect: claiming ownership of assigned values. You shouldn't use this in an Automatic Reference Counted environment.

The unsafe_unretained Attribute

Properties with the `unsafe_unretained` attribute behave similar to `weak` properties, but they don't automatically set their value to `nil` if the referenced object is destroyed. The only reason you should need to use `unsafe_unretained` is to make your class compatible with code that doesn't support the `weak` property.

The assign Attribute

The `assign` attribute doesn't perform any kind of memory-management call when assigning a new value to the property. This is the default behavior for primitive data types, and it used to be a way to implement weak references before iOS 5. Like `retain`, you shouldn't ever need to explicitly use this in modern applications.

Summary

This module presented the entire selection of `@property` attributes, and we hope that you're feeling relatively comfortable modifying the behavior of generated accessor methods. Remember that the goal of all these attributes is to help you focus on *what* data needs to be recorded by letting the compiler automatically determine *how* it's represented. They are summarized below.

Attribute	**Description**
`getter=`	Use a custom name for the getter method.
`setter=`	Use a custom name for the setter method.
`readonly`	Don't synthesize a setter method.
`nonatomic`	Don't guarantee the integrity of accessors in a multi-threaded environment. This is more efficient than the default atomic behavior.
`strong`	Create an owning relationship between the property and the assigned value. This is the default for object properties.
`weak`	Create a non-owning relationship between the property and the assigned value. Use this to prevent retain cycles.
`copy`	Create a copy of the assigned value instead of referencing the existing instance.

Now that we've got properties out of the way, we can take an in-depth look at the other half of Objective-C classes: methods. We'll explore everything from the quirks behind their naming conventions to dynamic method calls.

Methods

Methods represent the actions that an object knows how to perform. They're the logical counterpart to properties, which represent an object's data. You can think of methods as functions that are attached to an object; however, they have a very different syntax.

In this module, we'll explore Objective-C's method naming conventions, which can be frustrating for experienced developers coming from C++, Java, Python, and similar languages. We'll also briefly discuss Objective-C's access modifiers (or lack thereof), and we'll learn how to refer to methods using selectors.

Naming Conventions

Objective-C methods are designed to remove all ambiguities from an API. As a result, method names are horribly verbose, but undeniably descriptive. Accomplishing this boils down to three simple rules for naming Objective-C methods:

1. Don't abbreviate anything.
2. Explicitly state parameter names in the method itself.
3. Explicitly describe the return value of the method.

Keep these rules in mind as you read through the following exemplary interface for a `Car` class.

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

@interface Car : NSObject

// Accessors
- (BOOL)isRunning;
- (void)setRunning:(BOOL)running;
- (NSString *)model;
- (void)setModel:(NSString *)model;

// Calculated values
- (double)maximumSpeed;
- (double)maximumSpeedUsingLocale:(NSLocale *)locale;

// Action methods
- (void)startEngine;
- (void)driveForDistance:(double)theDistance;
- (void)driveFromOrigin:(id)theOrigin toDestination:(id)theDestination;
- (void)turnByAngle:(double)theAngle;
- (void)turnToAngle:(double)theAngle;

// Error handling methods
- (BOOL)loadPassenger:(id)aPassenger error:(NSError **)error;

// Constructor methods
- (id)initWithModel:(NSString *)aModel;
- (id)initWithModel:(NSString *)aModel mileage:(double)theMileage;

// Comparison methods
- (BOOL)isEqualToCar:(Car *)anotherCar;
- (Car *)fasterCar:(Car *)anotherCar;
- (Car *)slowerCar:(Car *)anotherCar;

// Factory methods
+ (Car *)car;
+ (Car *)carWithModel:(NSString *)aModel;
+ (Car *)carWithModel:(NSString *)aModel mileage:(double)theMileage;

// Singleton methods
+ (Car *)sharedCar;

@end
```

Abbreviations

The easiest way to make methods understandable and predictable is to simply avoid abbreviations. Most Objective-C programmer *expect* methods to be written out in full, as this is the convention for all of the standard frameworks, from Foundation to UIKit. This is why the above interface chose `maximumSpeed` over the more concise `maxSpeed`.

Parameters

One of the clearest examples of Objective-C's verbose design philosophy is in the naming conventions for method parameters. Whereas C++, Java, and other Simula-style languages treat a method as a separate entity from its parameters, an Objective-C method name actually contains the names of all its parameters.

For example, to make a `Car` turn by 90 degrees in C++, you would call something like `turn(90)`. But, Objective-C finds this too ambiguous. It's not clear what kind of argument `turn()` should take—it could be the new orientation, or it could be an angle by which to increment your current orientation. Objective-C methods make this explicit by describing the argument with a preposition. The resulting API ensures the method will never be misinterpreted: it's either `turnByAngle:90` or `turnToAngle:90`.

When a method accepts more than one parameter, the name of each argument is also included in the method name. For instance, the above `initWithModel:mileage:` method explicitly labels both the `Model` argument and the `mileage` argument. As we'll see in a moment, this makes for very informative method invocations.

Return Values

You'll also notice that any methods returning a value explicitly state what that value is. Sometimes this is as simple as stating the class of the return type, but other times you'll need to prefix it with a descriptive adjective.

For example, the factory methods start with `car`, which clearly states that the method will return an instance of the `Car` class. The comparison methods `fasterCar:` and `slowerCar:` return the faster/slower of the receiver and the argument, and this is also clearly expressed in the API. It's worth noting that singleton methods should also follow this pattern (e.g., `sharedCar`), since the conventional `instance` method name is ambiguous.

For more information about naming conventions, please visit the official Cocoa Coding Guidlines.

Calling Methods

As discussed in the Instantiation and Usage section, you invoke a method by placing the object and the desired method in square brackets, separated by a space. Arguments are separated from the method name using a colon:

```
[porsche initWithModel:@"Porsche"];
```

When you have more than one parameter, it comes after the initial argument, following the same pattern. Each parameter is paired with a label, separated from other arguments by a space, and set off by a colon:

```
[porsche initWithModel:@"Porsche" mileage:42000.0];
```

It's a lot easier to see the purpose behind the above naming conventions when you approach it from an invocation perspective. They make method calls read more like a human language than a computer one. For example, compare the following method call from Simula-style languages to Objective-C's version:

```
// Python/Java/C++
porsche.drive("Home", "Airport");


// Objective-C
[porsche driveFromOrigin:@"Home" toDestination:@"Airport"];
```

It might be more to type, but that's why Xcode comes with such a nice auto-completion feature. You'll appreciate the verbosity when you leave your code for a few months and come back to fix a bug. This clarity also makes it much easier to work with third-party libraries and to maintain large code bases.

Nested Method Calls

Nesting method calls is a common pattern in Objective-C programs. It's a natural way to pass the result of one call to another. Conceptually, it's the exact same as chaining methods, but the square-bracket syntax makes them look a little bit different:

```
// JavaScript
Car.alloc().init()


// Objective-C
[[Car alloc] init];
```

First, the `[Car alloc]` method is invoked, then the `init` method is called on its return value.

Protected and Private Methods

There are no protected or private access modifiers for Objective-C methods—they are all public. However, Objective-C does provide alternative organizational paradigms that let you *emulate* these features.

"Private" methods can be created by defining them in a class's implementation file while omitting them from its interface file. Since other objects (including subclasses) are never supposed to import the implementation, these methods are effectively hidden from everything but the class itself.

In lieu of protected methods, Objective-C provides categories, which are a more general solution for isolating portions of an API. A full example can be found in "Protected" Methods.

Selectors

Selectors are Objective-C's internal representation of a method name. They let you treat a method as an independent entity, enabling you to separate an action from the object that needs to perform it. This is the basis of the target-action design pattern, which is introduced in the Interface Builder chapter of Ry's Cocoa Tutorial. It's also an integral part of Objective-C's dynamic typing system.

There are two ways to get the selector for a method name. The `@selector()` directive lets you convert a source-code method name to a selector, and the `NSSelectorFromString()` function lets you convert a string to a selector (the latter is not as efficient). Both of these return a special data type for selectors called `SEL`. You can use `SEL` the exact same way as `BOOL`, `int`, or any other data type.

```
// main.m
#import <Foundation/Foundation.h>
#import "Car.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Car *porsche = [[Car alloc] init];
        porsche.model = @"Porsche 911 Carrera";

        SEL stepOne = NSSelectorFromString(@"startEngine");
        SEL stepTwo = @selector(driveForDistance:);
        SEL stepThree = @selector(turnByAngle:quickly:);

        // This is the same as:
        // [porsche startEngine];
        [porsche performSelector:stepOne];

        // This is the same as:
        // [porsche driveForDistance:[NSNumber numberWithDouble:5.7]];
        [porsche performSelector:stepTwo
                      withObject:[NSNumber numberWithDouble:5.7]];

        if ([porsche respondsToSelector:stepThree]) {
            // This is the same as:
            // [porsche turnByAngle:[NSNumber numberWithDouble:90.0]
            //              quickly:[NSNumber numberWithBool:YES]];
            [porsche performSelector:stepThree
                          withObject:[NSNumber numberWithDouble:90.0]
```

```
                                withObject:[NSNumber numberWithBool:YES]];
        }
        NSLog(@"Step one: %@", NSStringFromSelector(stepOne));
    }
    return 0;
}
```

Selectors can be executed on an arbitrary object via `performSelector:` and related methods. The `withObject:` versions let you pass an argument (or two) to the method, but require those arguments to be objects. If this is too limiting for your needs, please see `NSInvocation` for advanced usage. When you're not sure if the target object defines the method, you should use the `respondsToSelector:` check before trying to perform the selector.

The technical name for a method is the primary method name concatenated with all of its parameter labels, separated by colons. This makes colons an integral aspect of method names, which can be confusing to Objective-C beginners. Their usage can be summed up as follows: *parameterless methods never contain a colon, while methods that take a parameter always end in a colon.*

A sample interface and implementation for the above `Car` class is included below. Notice how we have to use `NSNumber` instead of `double` for the parameter types, since the `performSelector:withObject:` method doesn't let you pass primitive C data types.

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

@interface Car : NSObject

@property (copy) NSString *model;

- (void)startEngine;
- (void)driveForDistance:(NSNumber *)theDistance;
- (void)turnByAngle:(NSNumber *)theAngle
            quickly:(NSNumber *)useParkingBrake;

@end
```

```
// Car.m
#import "Car.h"
```

```
@implementation Car

@synthesize model = _model;

- (void)startEngine {
    NSLog(@"Starting the %@'s engine", _model);
}

- (void)driveForDistance:(NSNumber *)theDistance {
    NSLog(@"The %@ just drove %0.1f miles",
          _model, [theDistance doubleValue]);
}

- (void)turnByAngle:(NSNumber *)theAngle
            quickly:(NSNumber *)useParkingBrake {
    if ([useParkingBrake boolValue]) {
        NSLog(@"The %@ is drifting around the corner!", _model);
    } else {
        NSLog(@"The %@ is making a gentle %0.1f degree turn",
              _model, [theAngle doubleValue]);
    }
}

@end
```

Summary

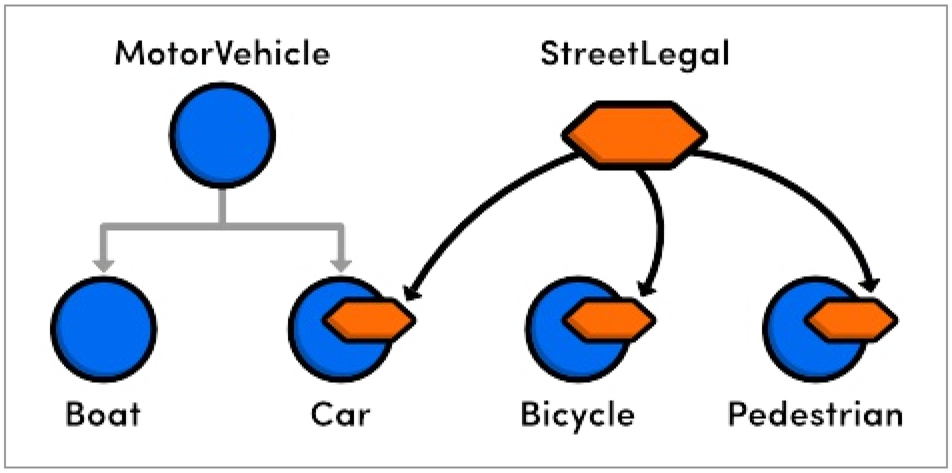
This module explained the reasoning behind Objective-C's method naming conventions. We also learned that there are no access modifiers for Objective-C methods, and how to use `@selector` to dynamically invoke methods.

Adapting to new conventions can be a frustrating process, and the dramatic syntactic differences between Objective-C and other OOP languages won't make your life any easier. Instead of forcing Objective-C into your existing mental model of the programming universe, it helps to approach it in its own right. Try designing a few simple programs before passing judgement on Objective-C's verbose philosophy.

That covers the basics of object-oriented programming in Objective-C. The rest of this tutorial explores more advanced ways to organize your code. First on the list are protocols, which let you share an API between several classes.

Protocols

A protocol is a group of related properties and methods that can be implemented by *any* class. They are more flexible than a normal class interface, since they let you reuse a single API declaration in completely unrelated classes. This makes it possible to represent horizontal relationships on top of an existing class hierarchy.

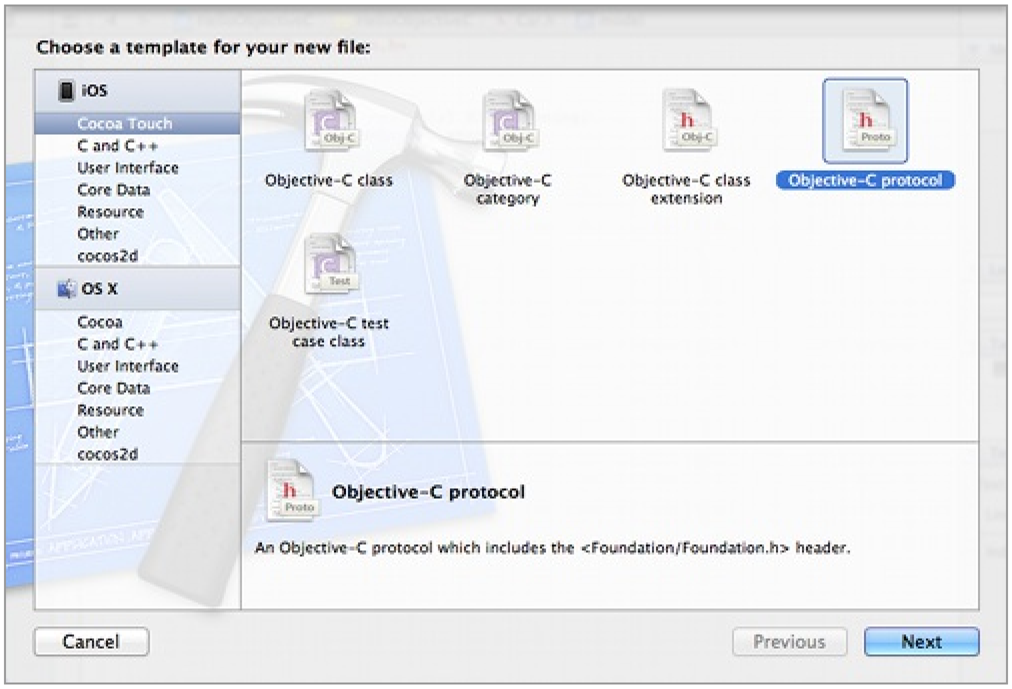


Unrelated classes adopting the `StreetLegal` *protocol*

This is a relatively short module covering the basics behind working with protocols. We'll also see how they fit into Objective-C's dynamic typing system.

Creating Protocols

Like class interfaces, protocols typically reside in a `.h` file. To add a protocol to your Xcode project, navigate to *File > New> File…* or use the *Cmd+N* shortcut. Select *Objective-C protocol* under the *iOS > Cocoa Touch* category.


Choose a template for your new file:

Objective-C protocol

An Objective-C protocol which includes the <Foundation/Foundation.h> header.

Creating a protocol in Xcode

In this module, we'll be working with a protocol called `StreetLegal`. Enter this in the next window, and save it in the project root.

Our protocol will capture the necessary behaviors of a street-legal vehicle. Defining these characteristics in a protocol lets you apply them to arbitrary objects instead of forcing them to inherit from the same abstract superclass. A simple version of the `StreetLegal` protocol might look something like the following:

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

@protocol StreetLegal <NSObject>

- (void)signalStop;
- (void)signalLeftTurn;
- (void)signalRightTurn;

@end
```

Any objects that adopt this protocol are *guaranteed* to implement all of the above methods. The `<NSObject>` after the protocol name incorporates the `NSObject` protocol (not to be confused with the `NSObject` class) into the `StreetLegal` protocol. That is, any objects conforming to the `StreetLegal` protocol are required to conform to the `NSObject` protocol, too.

Adopting Protocols

The above API can be adopted by a class by adding it in angled brackets after the class/superclass name. Create a new classed called `Bicycle` and change its header to the following. Note that you need to import the protocol before you can use it.

```
// Bicycle.h
#import <Foundation/Foundation.h>
#import "StreetLegal.h"

@interface Bicycle : NSObject <StreetLegal>

- (void)startPedaling;
- (void)removeFrontWheel;
- (void)lockToStructure:(id)theStructure;


@end
```

Adopting the protocol is like adding all of the methods in `StreetLegal.h` to `Bicycle.h`. This would work the exact same way even if `Bicycle` inherited from a different superclass. Multiple protocols can be adopted by separating them with commas (e.g., `<StreetLegal, SomeOtherProtocol>`).

There's nothing special about the `Bicycle` implementation—it just has to make sure all of the methods declared by `Bicycle.h` *and* `StreetLegal.h` are implemented:

```
// Bicycle.m
#import "Bicycle.h"


@implementation Bicycle


- (void)signalStop {
    NSLog(@"Bending left arm downwards");
}
- (void)signalLeftTurn {
    NSLog(@"Extending left arm outwards");
}
- (void)signalRightTurn {
    NSLog(@"Bending left arm upwards");
}
- (void)startPedaling {
    NSLog(@"Here we go!");
}
- (void)removeFrontWheel {
    NSLog(@"Front wheel is off."
          "Should probably replace that before pedaling...");
}
- (void)lockToStructure:(id)theStructure {
    NSLog(@"Locked to structure. Don't forget the combination!");
}


@end
```

Now, when you use the `Bicycle` class, you can assume it responds to the API defined by the protocol.

It's as though `signalStop`, `signalLeftTurn`, and `signalRightTurn` were declared in `Bicycle.h`:

```
// main.m
#import <Foundation/Foundation.h>
#import "Bicycle.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Bicycle *bike = [[Bicycle alloc] init];
        [bike startPedaling];
        [bike signalLeftTurn];
        [bike signalStop];
        [bike lockToStructure:nil];
    }
    return 0;
}
```

Type Checking With Protocols

Just like classes, protocols can be used to type check variables. To make sure an object adopts a protocol, put the protocol name after the data type in the variable declaration, as shown below. The next code snippet also assumes that you have created a `Car` class that adopts the `StreetLegal` protocol:

```
// main.m
#import <Foundation/Foundation.h>
#import "Bicycle.h"
#import "Car.h"
#import "StreetLegal.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        id <StreetLegal> mysteryVehicle = [[Car alloc] init];
        [mysteryVehicle signalLeftTurn];

        mysteryVehicle = [[Bicycle alloc] init];
        [mysteryVehicle signalLeftTurn];
    }
    return 0;
}
```

It doesn't matter if `Car` and `Bicycle` inherit from the same superclass—the fact that they both adopt the `StreetLegal` protocol lets us store either of them in a variable declared with `id <StreetLegal>`. This is an example of how protocols can capture common functionality between unrelated classes.

Objects can also be checked against a protocol using the `conformsToProtocol:` method defined by the `NSObject` protocol. It takes a protocol object as an argument, which can be obtained via the `@protocol()` directive. This works much like the `@selector()` directive, but you pass the protocol name instead of a method name, like so:

```
if ([mysteryVehicle conformsToProtocol:@protocol(StreetLegal)]) {
    [mysteryVehicle signalStop];
    [mysteryVehicle signalLeftTurn];
    [mysteryVehicle signalRightTurn];
}
```

Using protocols in this manner is like saying, "Make sure this object has this particular set of functionality." This is a very powerful tool for dynamic typing, as it lets you use a well-defined API without worrying about what kind of object you're dealing with.

Protocols In The Real World

A more realistic use case can be seen in your everyday iOS and OS X application development. The entry point into any app is an "application delegate" object that handles the major events in a program's life cycle. Instead of forcing the delegate to inherit from any particular superclass, the UIKit Framework just makes you adopt a protocol:

```
@interface YourAppDelegate : UIResponder <UIApplicationDelegate>
```

As long as it responds to the methods defined by `UIApplicationDelegate`, you can use *any* object as your application delegate. Implementing the delegate design pattern through protocols instead of subclassing gives developers much more leeway when it comes to organizing their applications.

You can see a concrete example of this in the Interface Builder chapter of Ry's Cocoa Tutorial. It uses the project's default app delegate to respond to user input.

Summary

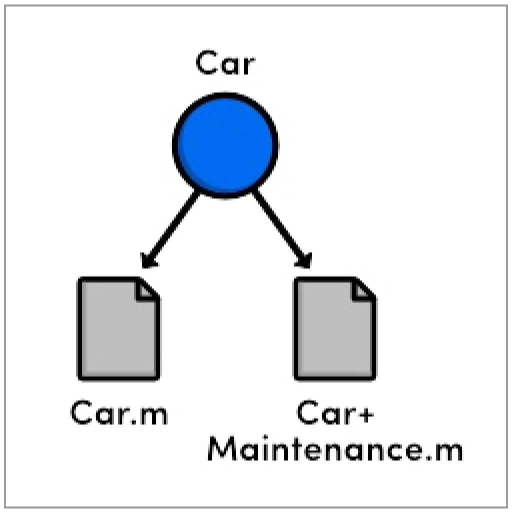
In this module, we added another organizational tool to our collection. Protocols are a way to abstract shared properties and methods into a dedicated file. This helps reduce redundant code and lets you dynamically check if an object supports an arbitrary set of functionality.

You'll find many protocols throughout the Cocoa frameworks. A common use case is to let you alter the behavior of certain classes without the need to subclass them. For instance, the *Table View*, *Outline View*, and *Collection View* UI components all use a data source and delegate object to configure their internal behavior. The data source and delegate are defined as protocols, so you can implement the necessary methods in *any* object you want.

The next module introduces categories, which are a flexible option for modularizing classes and providing opt-in support for an API.

Categories

Categories are a way to split a single class definition into multiple files. Their goal is to ease the burden of maintaining large code bases by modularizing a class. This prevents your source code from becoming monolithic 10000+ line files that are impossible to navigate and makes it easy to assign specific, well-defined portions of a class to individual developers.



Using multiple files to implement a class

In this module, we'll use a category to extend an existing class without touching its original source file. Then, we'll learn how this functionality can be used to emulate protected methods. Extensions are a close relative to categories, so we'll be taking a brief look at those, too.

Setting Up

Before we can start experimenting with categories, we need a base class to work off of. Create or change your existing `Car` interface to the following:

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

@interface Car : NSObject

@property (copy) NSString *model;
@property (readonly) double odometer;

- (void)startEngine;
- (void)drive;
- (void)turnLeft;
- (void)turnRight;
```

```
@end
```

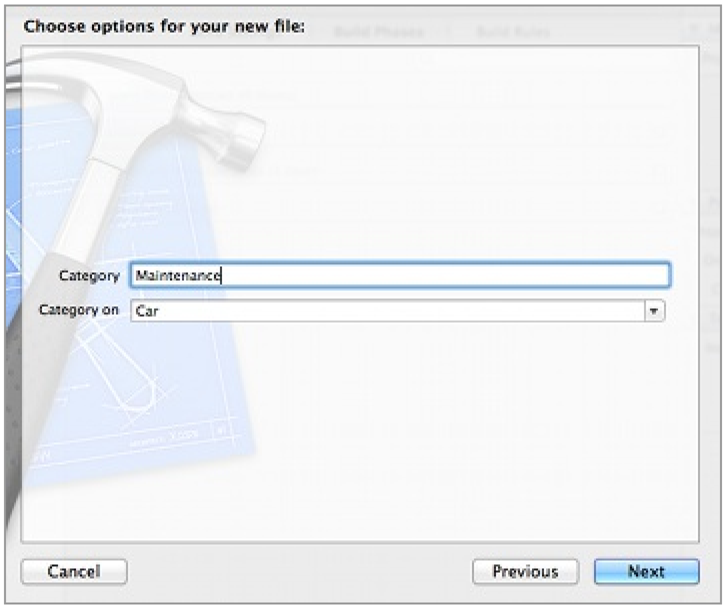
The corresponding implementation just outputs some descriptive messages so we can see when different methods get called:

```
// Car.m
#import "Car.h"

@implementation Car

@synthesize model = _model;

- (void)startEngine {
    NSLog(@"Starting the %@'s engine", _model);
}
- (void)drive {
    NSLog(@"The %@ is now driving", _model);
}
- (void)turnLeft {
    NSLog(@"The %@ is turning left", _model);
}
- (void)turnRight {
    NSLog(@"The %@ is turning right", _model);
}

@end
```

Now, let's say you want to add another set of methods related to car maintenance. Instead of cluttering up these `Car.h` and `Car.m` files, you can place the new methods in a dedicated category.

Creating Categories

Categories work just like normal class definitions in that they are composed of an interface and an implementation. To add a new category to your Xcode project, create a new file and choose the *Objective-C category* template under *iOS* > *Cocoa Touch*. Use `Maintenance` for the *Category* field and `Car` for *Category on*.


Choose options for your new file:

Category: Maintenance

Category on: Car

Creating the `Maintenance` *category*

The only restriction on category names is that they don't conflict with other categories on the same class. The canonical file-naming convention is to use the class name and the category name separated by a plus sign, so you should find a `Car+Maintenance.h` and a `Car+Maintenance.m` in Xcode's *Project Navigator* after saving the above category.

As you can see in `Car+Maintenance.h`, a category interface looks exactly like a normal interface, except the class name is followed by the category name in parentheses. Let's go ahead and add a few methods to the category:

```
// Car+Maintenance.h
#import "Car.h"

@interface Car (Maintenance)

- (BOOL)needsOilChange;
- (void)changeOil;
- (void)rotateTires;
- (void)jumpBatteryUsingCar:(Car *)anotherCar;

@end
```

At runtime, these methods become part of the `Car` class. Even though they're declared in a different file, you will be able to access them as if they were defined in the original `Car.h`.

Of course, you have to implement the category interface for the above methods to actually *do* anything. Again, a category implementation looks almost exactly like a standard implementation, except the category name appears in parentheses after the class name:

```
// Car+Maintenance.m
#import "Car+Maintenance.h"

@implementation Car (Maintenance)

- (BOOL)needsOilChange {
    return YES;
}
- (void)changeOil {
    NSLog(@"Changing oil for the %@", [self model]);
}
- (void)rotateTires {
    NSLog(@"Rotating tires for the %@", [self model]);
}
- (void)jumpBatteryUsingCar:(Car *)anotherCar {
    NSLog(@"Jumped the %@ with a %@", [self model], [anotherCar model]);
}

@end
```

It's important to note that a category can also be used to override existing methods in the base class (e.g., the `Car` class's `drive` method), but **you should never do this**. The problem is that categories are a flat organizational structure. If you override an existing method in `Car+Maintenance.m`, and then decide you want to change its behavior again with another category, there is no way for Objective-C to know which implementation to use. Subclassing is almost always a better option in such a situation.

Using Categories

Any files that use an API defined in a category need to import that category header just like a normal class interface. After importing `Car+Maintenance.h`, all of its methods will be available directly through the `Car` class:

```
// main.m
#import <Foundation/Foundation.h>
#import "Car.h"
#import "Car+Maintenance.h"
```

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Car *porsche = [[Car alloc] init];
        porsche.model = @"Porsche 911 Turbo";
        Car *ford = [[Car alloc] init];
        ford.model = @"Ford F-150";

        // "Standard" functionality from Car.h
        [porsche startEngine];
        [porsche drive];
        [porsche turnLeft];
        [porsche turnRight];

        // Additional methods from Car+Maintenance.h
        if ([porsche needsOilChange]) {
            [porsche changeOil];
        }
        [porsche rotateTires];
        [porsche jumpBatteryUsingCar:ford];
    }
    return 0;
}
```

If you remove the import statement for `Car+Maintenance.h`, the `Car` class will revert to its original state, and the compiler will complain that `needsOilChange`, `changeOil`, and the rest of the methods from the `Maintenance` category don't exist.

"Protected" Methods

But, categories aren't just for spreading a class definition over several files. They are a powerful organizational tool that allow arbitrary files to "opt-in" to a portion of an API by simply importing the category. To everybody else, that API remains hidden.

Recall from the Methods module that protected methods don't actually exist in Objective-C; however, the opt-in behavior of categories can be used to *emulate* protected access modifiers. The idea is to define a "protected" API in a dedicated category, and only import it into subclass implementations. This makes the protected methods available to subclasses, but keeps them hidden from other aspects of the application. For example:

```
// Car+Protected.h
#import "Car.h"

@interface Car (Protected)

- (void)prepareToDrive;

@end
```

```
// Car+Protected.m
#import "Car+Protected.h"

@implementation Car (Protected)

- (void)prepareToDrive {
    NSLog(@"Doing some internal work to get the %@ ready to drive",
          [self model]);
}

@end
```

The `Protected` category shown above defines a single method for internal use by `Car` and its subclasses. To see this in action, let's modify `Car.m`'s `drive` method to use the protected `prepareToDrive` method:

```
// Car.m
#import "Car.h"
#import "Car+Protected.h"

@implementation Car
...
- (void)drive {
    [self prepareToDrive];
    NSLog(@"The %@ is now driving", _model);
}
...
```

Next, let's take a look at how this protected method works by creating a subclass called `Coupe`.

There's nothing special about the interface, but notice how the implementation opts-in to the protected API by importing `Car+Protected.h`. This makes it possible to use the protected `prepareToDrive` method in the subclass. If desired, you can also override the protected method by simply re-defining it in `Coupe.m`.

```
// Coupe.h
#import "Car.h"

@interface Coupe : Car
// Extra methods defined by the Coupe subclass
@end
```

```
// Coupe.m
#import "Coupe.h"
#import "Car+Protected.h"

@implementation Coupe

- (void)startEngine {
    [super startEngine];
    // Call the protected method here instead of in `drive`
    [self prepareToDrive];
}

- (void)drive {
    NSLog(@"VROOOOOOM!");
}

@end
```

To enforce the protected status of the methods in `Car+Protected.h`, it should only be made available to subclass implementations—do *not* import it into other files. In the following `main.m`, you can see the protected `prepareToDrive` method called by `[ford drive]` and `[porsche startEngine]`, but the compiler will complain if you try to call it directly.

```
// main.m
#import <Foundation/Foundation.h>
#import "Car.h"
#import "Coupe.h"
```

```
int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Car *ford = [[Car alloc] init];
        ford.model = @"Ford F-150";
        [ford startEngine];
        [ford drive]; // Calls the protected method

        Car *porsche = [[Coupe alloc] init];
        porsche.model = @"Porsche 911 Turbo";
        [porsche startEngine]; // Calls the protected method
        [porsche drive];

        // "Protected" methods have not been imported,
        // so this will *not* work
        // [porsche prepareToDrive];

        SEL protectedMethod = @selector(prepareToDrive);
        if ([porsche respondsToSelector:protectedMethod]) {
            // This *will* work
            [porsche performSelector:protectedMethod];
        }


    }
    return 0;
}
```

Notice that you *can* access `prepareToDrive` dynamically through `performSelector:`. Once again, *all* methods in Objective-C are public, and there is no way to truly hide them from client code. Categories are merely a convention-based way to control which parts of an API are visible to which files.

Extensions

Extensions are similar to categories in that they let you add methods to a class outside of the main interface file. But, in contrast to categories, an extension's API must be implemented in the *main* implementation file—it *cannot* be implemented in a category.

Remember that private methods can be emulated by adding them to the implementation but not the interface. This works when you have only a few private methods, but can become unwieldy for larger classes. Extensions solve this problem by letting you declare a *formal* private API.

For example, if you wanted to formally add a private `engineIsWorking` method to the `Car` class defined above, you could include an extension in `Car.m`. The compiler complains if the method isn't defined in the main `@implementation` block, but since it's declared in `Car.m` instead of `Car.h`, it remains a *private* method. The extension syntax looks like an empty category:

```
// Car.m
#import "Car.h"


// The class extension
@interface Car ()
- (BOOL)engineIsWorking;
@end


// The main implementation
@implementation Car


@synthesize model = _model;


- (BOOL)engineIsWorking {
    // In the real world, this would probably return a useful value
    return YES;
}
- (void)startEngine {
    if ([self engineIsWorking]) {
        NSLog(@"Starting the %@'s engine", _model);
    }
}
...
@end
```

In addition to declaring formal private API's, extensions can be used to re-declare properties from the public interface. This is often used to make properties internally behave as read-write properties while remaining read-only to other objects. For instance, if we change the above class extension to:

```
// Car.m
#import "Car.h"
```

```
@interface Car ()

@property (readwrite) double odometer;

- (BOOL)engineIsWorking;

@end

...
```

We can then assign values to `self.odometer` inside of the implementation, but trying to do so outside of `Car.m` will result in a compiler error.

Again, re-declaring properties as read-write and creating formal private API's isn't all that useful for small classes. Their real utility comes into play when you need to organize larger frameworks.

Extensions used to see much more action before Xcode 4.3, back when private methods had to be declared *before* they were used. This was inconvenient for many developers, and extensions provided a workaround by acting as forward-declarations of private methods. So, even if you don't use the above pattern in your own projects, you're likely to encounter it at some point in your Objective-C career.

Summary

This module covered Objective-C categories and extensions. Categories are a way to modularize a class by spreading its implementation over many files. Extensions provide similar functionality, except its API must be declared in the *main* implementation file.

Outside of organizing large code libraries, one of the most common uses of categories is to add methods to built-in data types like `NSString` or `NSArray`. The advantage of this is that you don't have to update existing code to use a new subclass, but you need to be very careful not to override existing functionality. For small personal projects, categories really aren't worth the trouble, and sticking with standard tools like subclassing and protocols will save you some debugging headaches down the road.

In the next module, we'll explore another organizational tool called blocks. Blocks are a way to represent and pass around arbitrary statements. This opens the door to a whole new world of programming paradigms.

Blocks

Blocks are Objective-C's anonymous functions. They let you pass arbitrary statements between objects as you would data, which is often more intuitive than referencing functions defined elsewhere. In addition, blocks are implemented as *closures*, making it trivial to capture the surrounding state.

Creating Blocks

Blocks use all the same mechanics as normal functions. You can declare a block variable just like you would declare a function, define the block as though you would implement a function, and then call the block as if it were a function:

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        // Declare the block variable
        double (^distanceFromRateAndTime)(double rate, double time);

        // Create and assign the block
        distanceFromRateAndTime = ^double(double rate, double time) {
            return rate * time;
        };
        // Call the block
        double dx = distanceFromRateAndTime(35, 1.5);

        NSLog(@"A car driving 35 mph will travel "
              @"%.2f miles in 1.5 hours.", dx);
    }
    return 0;
}
```

The caret (`^`) is used to mark the `distanceFromRateAndTime` variable as a block. Like a function declaration, you need to include the return type and parameter types so the compiler can enforce type safety. The `^` behaves in a similar manner to the asterisk before a pointer (e.g., `int *aPointer`) in that it is only required when *declaring* the block, after which you can use it like a normal variable.

The block itself is essentially a function definition—without the function name. The `^double(double` `rate, double time)` signature begins a block literal that returns a `double` and has two parameters that are also doubles (the return type can be omitted if desired). Arbitrary statements can go inside the curly braces (`{}`), just like a normal function.

After assigning the block literal to the `distanceFromRateAndTime` variable, we can *call* that variable as though it were a function.

Parameterless Blocks

If a block doesn't take any parameters, you can omit the argument list in its entirety. And again, specifying the return type of a block literal is always optional, so you can shorten the notation to `^ { ... }`:

```
double (^randomPercent)(void) = ^ {
    return (double)arc4random() / 4294967295;
};
NSLog(@"Gas tank is %.1f%% full",
      randomPercent() * 100);
```

The built-in `arc4random()` function returns a random 32-bit integer. By dividing by the maximum possible value of `arc4random()` (`4294967295`), we get a decimal value between `0` and `1`.

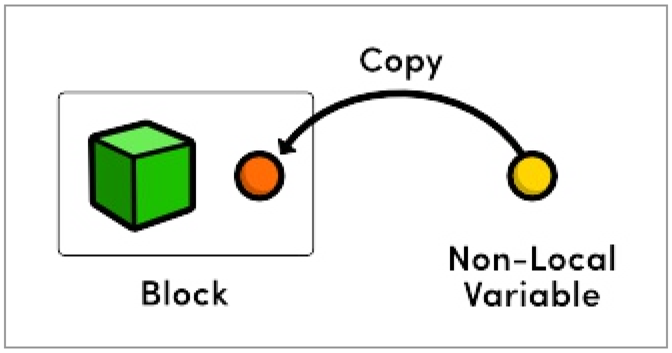
So far, it might seem like blocks are just a complicated way of defining functions. But, the fact that they're implemented as *closures* opens the door to exciting new programming opportunities.

Closures

Inside of a block, you have access to same data as in a normal function: local variables, parameters passed to the block, and global variables/functions. But, blocks are implemented as **closures**, which means that you also have access to **non-local variables**. Non-local variables are variables defined in the block's enclosing lexical scope, but outside the block itself. For example, `getFullCarName` can reference the `make` variable defined before the block:

```
NSString *make = @"Honda";
NSString *(^getFullCarName)(NSString *) = ^(NSString *model) {
    return [make stringByAppendingFormat:@" %@", model];
};
NSLog(@"%@", getFullCarName(@"Accord"));    // Honda Accord
```

Non-local variables are copied and stored with the block as `const` variables, which means they are read-only. Trying to assign a new value to the `make` variable from inside the block will throw a compiler error.



Accessing non-local variables as `const` *copies*

The fact that non-local variables are copied as constants means that a block doesn't just have *access* to non-local variables—it creates a *snapshot* of them. Non-local variables are frozen at whatever value they contain when the block is defined, and the block *always* uses that value, even if the non-local variable changes later on in the program. Watch what happens when we try to change the `make` variable after creating the block:

```
NSString *make = @"Honda";
NSString *(^getFullCarName)(NSString *) = ^(NSString *model) {
    return [make stringByAppendingFormat:@" %@", model];
};
NSLog(@"%@", getFullCarName(@"Accord"));    // Honda Accord

// Try changing the non-local variable (it won't change the block)
make = @"Porsche";
NSLog(@"%@", getFullCarName(@"911 Turbo")); // Honda 911 Turbo
```

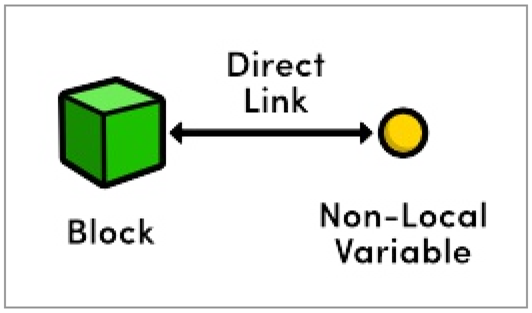
Closures are an incredibly convenient way to work with the surrounding state, as it eliminates the need to pass in extra values as parameters—you simply use non-local variables as if they were defined in the block itself.

Mutable Non-Local Variables

Freezing non-local variables as constant values is a safe default behavior in that it prevents you from accidentally changing them from within the block; however, there are occasions when this is desirable. You can override the `const` copy behavior by declaring a non-local variable with the `__block` storage modifier:

```
__block NSString *make = @"Honda";
```

This tells the block to capture the variable *by reference*, creating a direct link between the `make` variable outside the block and the one inside the block. You can now assign a new value to `make` from outside the block, and it will be reflected in the block, and vice versa.



Accessing non-local variables by reference

Like `static` local variables in normal functions, the `__block` modifier serves as a "memory" between multiple calls to a block. This makes it possible to use blocks as generators. For example, the following snippet creates a block that "remembers" the value of `i` over subsequent invocations.

```
__block int i = 0;
int (^count)(void) = ^ {
    i += 1;
    return i;
};
NSLog(@"%d", count());    // 1
NSLog(@"%d", count());    // 2
NSLog(@"%d", count());    // 3
```

Blocks as Method Parameters

Storing blocks in variables is occasionally useful, but in the real world, they're more likely to be used as method parameters. They solve the same problem as function pointers, but the fact that they can be defined inline makes the resulting code much easier to read.

For example, the following `Car` interface declares a method that tallies the distance traveled by the car. Instead of forcing the caller to pass a constant speed, it accepts a block that defines the car's speed as a function of time.

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

@interface Car : NSObject
```

```
@property double odometer;

- (void)driveForDuration:(double)duration
       withVariableSpeed:(double (^)(double time))speedFunction
                   steps:(int)numSteps;

@end
```

The data type for the block is `double (^)(double time)`, which states that whatever block the caller passes to the method should return a `double` and accept a single `double` parameter. Note that this is almost the exact same syntax as the block variable declaration discussed at the beginning of this module, but without the variable name.

The implementation can then call the block via `speedFunction`. The following example uses a naïve right-handed Riemann sum to approximate the distance traveled over `duration`. The `steps` argument is used to let the caller determine the precision of the estimate.

```
// Car.m
#import "Car.h"

@implementation Car

@synthesize odometer = _odometer;

- (void)driveForDuration:(double)duration
       withVariableSpeed:(double (^)(double time))speedFunction
                   steps:(int)numSteps {
    double dt = duration / numSteps;
    for (int i=1; i<=numSteps; i++) {
        _odometer += speedFunction(i*dt) * dt;
    }
}

@end
```

As you can see in the `main()` function included below, block literals can be defined *within* a method invocation. While it might take a second to parse the syntax, this is still much more intuitive than creating a dedicated top-level function to define the `withVariableSpeed` parameter.

```
// main.m
#import <Foundation/Foundation.h>
#import "Car.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        Car *theCar = [[Car alloc] init];

        // Drive for awhile with constant speed of 5.0 m/s
        [theCar driveForDuration:10.0
               withVariableSpeed:^(double time) {
                           return 5.0;
                       } steps:100];
        NSLog(@"The car has now driven %.2f meters", theCar.odometer);

        // Start accelerating at a rate of 1.0 m/s^2
        [theCar driveForDuration:10.0
               withVariableSpeed:^(double time) {
                           return time + 5.0;
                       } steps:100];
        NSLog(@"The car has now driven %.2f meters", theCar.odometer);
    }
    return 0;
}
```

This is a simple example of the versatility of blocks, but the standard frameworks are chock-full of other use cases. `NSArray` lets you sort elements with a block via the `sortedArrayUsingComparator:` method, and `UIView` uses a block to define the final state of an animation via the `animateWithDuration:animations:` method.

In addition, the `NSOpenPanel` class executes a block after the user selects a file. This convenient behavior is explored in the Persistent Data chapter of Ry's Cocoa Tutorial.

Defining Block Types

Since the syntax for block data types can quickly clutter up your method declarations, it's often useful to `typedef` common block signatures. For instance, the following code creates a new type called `SpeedFunction` that we can use as a more semantic data type for the `withVariableSpeed` argument.

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

// Define a new type for the block
typedef double (^SpeedFunction)(double);

@interface Car : NSObject

@property double odometer;

- (void)driveForDuration:(double)duration
       withVariableSpeed:(SpeedFunction)speedFunction
                   steps:(int)numSteps;

@end
```

Many of the standard Objective-C frameworks also use this technique (e.g., `NSComparator`).

Summary

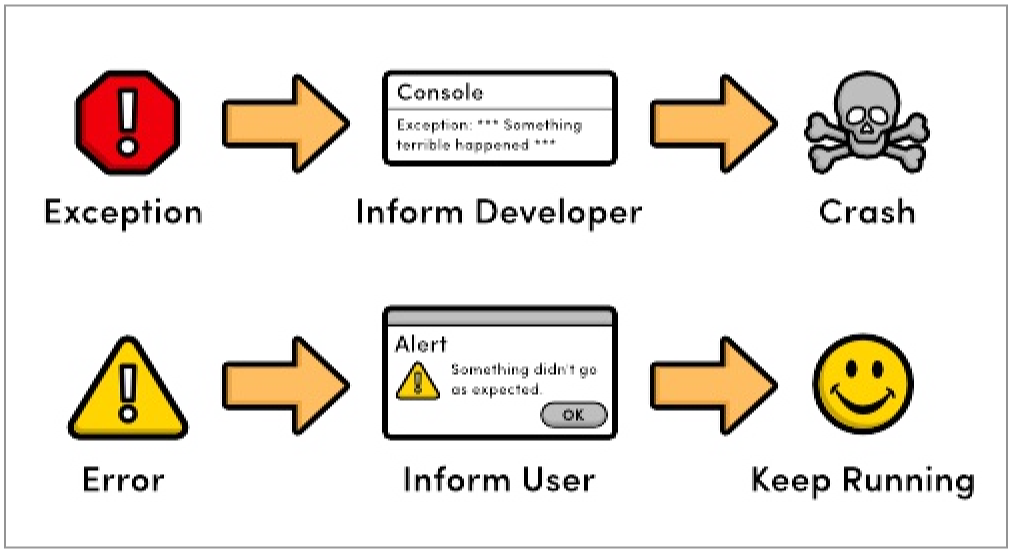
Blocks provide much the same functionality as C functions, but they are much more intuitive to work with (after you get used to the syntax). The fact that they can be defined inline makes it easy to use them inside of method calls, and since they are closures, it's possible to capture the value of surrounding variables with literally no additional effort.

The next module switches gears a little bit and delves into iOS's and OS X's error-handling capabilities. We'll explore two important classes for representing errors: `NSException` and `NSError`.

Exceptions & Errors

Two distinct types of problems can arise while an iOS or OS X application is running. **Exceptions** represent programmer-level bugs like trying to access an array element that doesn't exist. They are designed to inform the developer that an *unexpected* condition occurred. Since they usually result in the program crashing, exceptions should rarely occur in your production code.

On the other hand, **errors** are user-level issues like trying load a file that doesn't exist. Because errors are *expected* during the normal execution of a program, you should manually check for these kinds of conditions and inform the user when they occur. Most of the time, they should not cause your application to crash.


Console
Exception: *** Something terrible happened ***
Exception
Inform Developer
Crash
Alert
Something didn't go as expected.
OK
Error
Inform User
Keep Running

Exceptions vs. errors

This module provides a thorough introduction to exceptions and errors. Conceptually, working with exceptions is very similar to working with errors. First you need to detect the problem, and then you need to handle it. But, as we're about to find out, the underlying mechanics are slightly different

Exceptions

Exceptions are represented by the `NSException` class. It's designed to be a universal way to encapsulate exception data, so you should rarely need to subclass it or otherwise define a custom exception object. `NSException`'s three main properties are listed below.

Property	Description
`name`	An `NSString` that uniquely identifies the exception.
`reason`	An `NSString` that contains a human-readable description of the exception.
`userInfo`	An `NSDictionary` whose key-value pairs contain extra information about the exception. This varies based on the type of exception.

It’s important to understand that exceptions are only used for serious programming errors. The idea is to let you know that something has gone wrong early in the development cycle, after which you’re expected to fix it so it never occurs again. If you’re trying to handle a problem that’s *supposed* to occur, you should be using an error object, not an exception.

Handling Exceptions

Exceptions can be handled using the standard try-catch-finally pattern found in most other high-level programming languages. First, you need to place any code that *might* result in an exception in an `@try` block. Then, if an exception is thrown, the corresponding `@catch()` block is executed to handle the problem. The `@finally` block is called afterwards, regardless of whether or not an exception occurred.

The following `main.m` file raises an exception by trying to access an array element that doesn’t exist. In the `@catch()` block, we simply display the exception details. The `NSException *theException` in the parentheses defines the name of the variable containing the exception object.

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSArray *inventory = @[@"Honda Civic",
                               @"Nissan Versa",
                               @"Ford F-150"];
        int selectedIndex = 3;
        @try {
            NSString *car = inventory[selectedIndex];
            NSLog(@"The selected car is: %@", car);
        } @catch(NSException *theException) {
            NSLog(@"An exception occurred: %@", theException.name);
            NSLog(@"Here are some details: %@", theException.reason);
        } @finally {
            NSLog(@"Executing finally block");
        }
    }
    return 0;
}
```

In the real world, you'll want your `@catch()` block to actually handle the exception by logging the problem, correcting it, or possibly even promoting the exception to an error object so it can be displayed to the user. The default behavior for uncaught exceptions is to output a message to the console and exit the program.

Objective-C's exception-handling capabilities are not the most efficient, so you should only use `@try`/`@catch()` blocks to test for truly exceptional circumstances. Do *not* use it in place of ordinary control flow tools. Instead, check for predictable conditions using standard `if` statements.

This means that the above snippet is actually a very poor use of exceptions. A much better route would have been to make sure that the `selectedIndex` was smaller than the `[inventory count]` using a traditional comparison:

```
if (selectedIndex < [inventory count]) {
    NSString *car = inventory[selectedIndex];
    NSLog(@"The selected car is: %@", car);
} else {
    // Handle the error
}
```

Built-In Exceptions

The standard iOS and OS X frameworks define several built-in exceptions. The complete list can be found here, but the most common ones are described below.

Exception Name	Description
`NSRangeException`	Occurs when you try to access an element that's outside the bounds of a collection.
`NSInvalidArgumentException`	Occurs when you pass an invalid argument to a method.
`NSInternalInconsistencyException`	Occurs when an unexpected condition arises internally.
`NSGenericException`	Occurs when you don't know what else to call the exception.

Note that these values are *strings*, not `NSException` subclasses. So, when you're looking for a specific type of exception, you need to check the `name` property, like so:

```
...
} @catch(NSException *theException) {
    if (theException.name == NSRangeException) {
        NSLog(@"Caught an NSRangeException");
    } else {
        NSLog(@"Ignored a %@ exception", theException);
        @throw;
    }
} ...
```

In an `@catch()` block, the `@throw` directive re-raises the caught exception. We used this in the above snippet to ignore all of the exceptions we didn't want by throwing it up to the next highest `@try` block. But again, a simple `if`-statement would be preferred.

Custom Exceptions

You can also use `@throw` to raise `NSException` objects that contain custom data. The easiest way to create an `NSException` instance is through the `exceptionWithName:reason:userInfo:` factory method. The following example throws an exception inside of a top-level function and catches it in the `main()` function.

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

NSString *getRandomCarFromInventory(NSArray *inventory) {
    int maximum = (int)[inventory count];
    if (maximum == 0) {
        NSException *e = [NSException
                          exceptionWithName:@"EmptyInventoryException"
                          reason:@"*** The inventory has no cars!"
                          userInfo:nil];
        @throw e;
    }
    int randomIndex = arc4random_uniform(maximum);
    return inventory[randomIndex];
}

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        @try {
            NSString *car = getRandomCarFromInventory(@[]);
            NSLog(@"The selected car is: %@", car);

        } @catch(NSException *theException) {

            if (theException.name == @"EmptyInventoryException") {

                NSLog(@"Caught an EmptyInventoryException");

            } else {

                NSLog(@"Ignored a %@ exception", theException);

                @throw;

            }

        }

    }

    return 0;

}
```

While occasionally necessary, you shouldn't really need to throw custom exceptions like this in normal applications. For one, exceptions represent programmer errors, and there are very few times when you should be planning for serious coding mistakes. Second, `@throw` is an expensive operation, so it's always better to use errors if possible.

Errors

Since errors represent *expected* problems, and there are several types of operations that can fail without causing the program to crash, they are much more common than exceptions. Unlike exceptions, this kind of error checking is a normal aspect of production-quality code.

The `NSError` class encapsulates the details surrounding a failed operation. Its main properties are similar to `NSException`.

Property	Description
`domain`	An `NSString` containing the error's domain. This is used to organize errors into a hierarchy and ensure that error codes don't conflict.
`code`	An `NSInteger` representing the ID of the error. Each error in the same domain must have a unique value.
`userInfo`	An `NSDictionary` whose key-value pairs contain extra information about the error. This varies based on the type of error.

The `userInfo` dictionary for `NSError` objects typically contains more information than `NSException`'s version. Some of the pre-defined keys, which are defined as named constants, are listed below.

Key	Value
`NSLocalizedDescriptionKey`	An `NSString` representing the full description of the error. This usually includes the failure reason, too.
`NSLocalizedFailureReasonErrorKey`	A brief `NSString` isolating the reason for the failure.
`NSUnderlyingErrorKey`	A reference to another `NSError` object that represents the error in the next-highest domain.

Depending on the error, this dictionary will also contain other domain-specific information. For example, file-loading errors will have a key called `NSFilePathErrorKey` that contains the path to the requested file.

Note that the `localizedDescription` and `localizedFailureReason` methods are an alternative way to access the first two keys, respectively. These are used in the next section's example.

Handling Errors

Errors don't require any dedicated language constructs like `@try` and `@catch()`. Instead, functions or methods that *may* fail accept an additional argument (typically called `error`) that is a reference to an `NSError` object. If the operation fails, it returns `NO` or `nil` to indicate failure and populates this argument with the error details. If it succeeds, it simply returns the requested value as normal.

Many methods are configured to accept an **indirect reference** to an `NSError` object. An indirect reference is a pointer to a pointer, and it allows the method to point the argument to a brand new `NSError` instance. You can determine if a method's `error` argument accepts an indirect reference by its double-pointer notation: `(NSError **)error`.

The following snippet demonstrates this error-handling pattern by trying to load a file that doesn't exist via `NSString`'s `stringWithContentsOfFile:encoding:error:` method. When the file loads successfully, the method returns the contents of the file as an `NSString`, but when it fails, it directly returns `nil` and *indirectly* returns the error by populating the `error` argument with a new `NSError` object.

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


int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSString *fileToLoad = @"/path/to/non-existent-file.txt";
        NSError *error;
        NSString *content = [NSString stringWithContentsOfFile:fileToLoad
                                                      encoding:NSUTF8StringEncoding
                                                         error:&error];

        if (content == nil) {
            // Method failed
            NSLog(@"Error loading file %@!", fileToLoad);
            NSLog(@"Domain: %@", error.domain);
            NSLog(@"Error Code: %ld", error.code);
            NSLog(@"Description: %@", [error localizedDescription]);
            NSLog(@"Reason: %@", [error localizedFailureReason]);
        } else {
            // Method succeeded
            NSLog(@"Content loaded!");
            NSLog(@"%@", content);
        }
    }
    return 0;
}
```

Notice how the we had to pass the `error` variable to the method using the reference operator. This is because the `error` argument accepts a double-pointer. Also notice how we checked the return value of the method for success with an ordinary `if` statement. You should only try to access the `NSError` reference if the method directly returns `nil`, and you should never use the presence of an `NSError` object to indicate success or failure.

Of course, if you only care about the success of the operation and aren't concern with *why* it failed, you can just pass `NULL` for the `error` argument and it will be ignored.

Built-In Errors

Like `NSException`, `NSError` is designed to be a universal object for representing errors. Instead of subclassing it, the various iOS and OS X frameworks define their own constants for the `domain` and `code` fields. There are several built-in error domains, but the main four are as follows:

```
NSMachErrorDomain
NSPOSIXErrorDomain
NSOSStatusErrorDomain
NSCocoaErrorDomain
```

Most of the errors you'll be working with are in the `NSCocoaErrorDomain`, but if you drill down to the lower-level domains, you'll see some of the other ones. For example, if you add the following line to `main.m`, you'll find an error with `NSPOSIXErrorDomain` for its domain.

```
NSLog(@"Underlying Error: %@", error.userInfo[NSUnderlyingErrorKey]);
```

For most applications, you shouldn't need to do this, but it can come in handy when you need to get at the root cause of an error.

After you've determined your error domain, you can check for a specific error code. The Foundation Constants Reference describes several `enum`'s that define most of the error codes in the `NSCocoaErrorDomain`. For example, the following code searches for a `NSFileReadNoSuchFileError` error.

```
...
if (content == nil) {
    if ([error.domain isEqualToString:@"NSCocoaErrorDomain"] &&
        error.code == NSFileReadNoSuchFileError) {
        NSLog(@"That file doesn't exist!");
        NSLog(@"Path: %@", [[error userInfo] objectForKey:NSFilePathErrorKey]);
    } else {
        NSLog(@"Some other kind of read occurred");
    }
} ...
```

Other frameworks should include any custom domains and error codes in their documentation.

Custom Errors

If you're working on a large project, you'll probably have at least a few functions or methods that can result in an error. This section explains how to configure them to use the canonical error-handling pattern discussed above.

As a best practice, you should define all of your errors in a dedicated header. For instance, a file called `InventoryErrors.h` might define a domain containing various error codes related to fetching items from an inventory.

```
// InventoryErrors.h
```

```
NSString *InventoryErrorDomain = @"com.RyPress.Inventory.ErrorDomain";


enum {
    InventoryNotLoadedError,
    InventoryEmptyError,
    InventoryInternalError
};
```

Technically, custom error domains can be anything you want, but the recommended form is `com.<Company>.<Framework-or-project>.ErrorDomain`, as shown in `InventoryErrorDomain`. The `enum` defines the error code constants.

The only thing that's different about a function or method that is error-enabled is the additional `error` argument. It should specify `NSError **` as its type, as shown in the following iteration of `getRandomCarFromInventory()`. When an error occurs, you point this argument to a new `NSError` object. Also note how we defined `localizedDescription` by manually adding it to the `userInfo` dictionary with `NSLocalizedDescriptionKey`.

```
// main.m
#import <Foundation/Foundation.h>
#import "InventoryErrors.h"

NSString *getRandomCarFromInventory(NSArray *inventory, NSError **error) {
    int maximum = (int)[inventory count];
    if (maximum == 0) {
        if (error != NULL) {
            NSDictionary *userInfo = @{NSLocalizedDescriptionKey: @"Could not"
            " select a car because there are no cars in the inventory."};

            *error = [NSError errorWithDomain:InventoryErrorDomain
                                         code:InventoryEmptyError
                                     userInfo:userInfo];
        }
        return nil;
    }
    int randomIndex = arc4random_uniform(maximum);
    return inventory[randomIndex];
}
```

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

    @autoreleasepool {

        NSArray *inventory = @[];

        NSError *error;

        NSString *car = getRandomCarFromInventory(inventory, &error);


        if (car == nil) {

            // Failed

            NSLog(@"Car could not be selected");

            NSLog(@"Domain: %@", error.domain);

            NSLog(@"Error Code: %ld", error.code);

            NSLog(@"Description: %@", [error localizedDescription]);


        } else {

            // Succeeded

            NSLog(@"Car selected!");

            NSLog(@"%@", car);

        }

    }

    return 0;

}
```

Since it technically is an error and not an exception, this version of `getRandomCarFromInventory()` is the “proper” way to handle it (opposed to Custom Exceptions).

Summary

Errors represent a failed operation in an iOS or OS X application. It’s a standardized way to record the relevant information at the point of detection and pass it off to the handling code. Exceptions are similar, but are designed as more of a development aid. They generally should not be used in your production-ready programs.

How you handle an error or exception is largely dependent on the type of problem, as well as your application. But, most of the time you’ll want to inform the user with something like `UIAlertView` (iOS). or `NSAlert` (OS X). After that, you’ll probably want to figure out what went wrong by inspecting the `NSError` or `NSException` object so that you can try to recover from it.

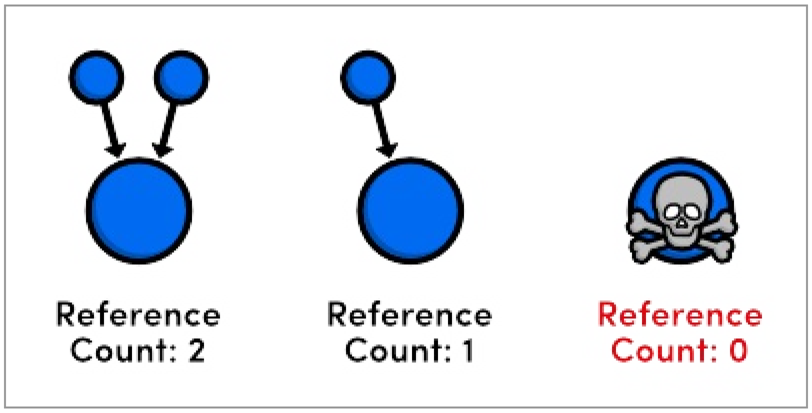
The next module explores some of the more conceptual aspects of the Objective-C runtime. We’ll learn about how the memory behind our objects is managed by experimenting with the (now obsolete) Manual Retain Release system, as well as the practical implications of the newer Automatic

Reference Counting system.

Memory Management

As discussed in the Properties module, the goal of any memory management system is to reduce the memory footprint of a program by controlling the lifetime of all its objects. iOS and OS X applications accomplish this through **object ownership**, which makes sure objects exist as long as they have to, but no longer.

This object-ownership scheme is implemented through a **reference-counting system** that internally tracks how many owners each object has. When you claim ownership of an object, you increase it's reference count, and when you're done with the object, you decrease its reference count. While its reference count is greater than zero, an object is guaranteed to exist, but as soon as the count reaches zero, the operating system is allowed to destroy it.



Destroying an object with zero references

In the past, developers manually controlled an object's reference count by calling special memory-management methods defined by the `NSObject` protocol. This is called **Manual Retain Release (MRR)**. However, Xcode 4.2 introduced **Automatic Reference Counting (ARC)**, which automatically inserts all of these method calls for you. Modern applications should always use ARC, since it's more reliable and lets you focus on your app's features instead of its memory management.

This module explains core reference-counting concepts in the context of MRR, then discusses some of the practical considerations of ARC.

Manual Retain Release

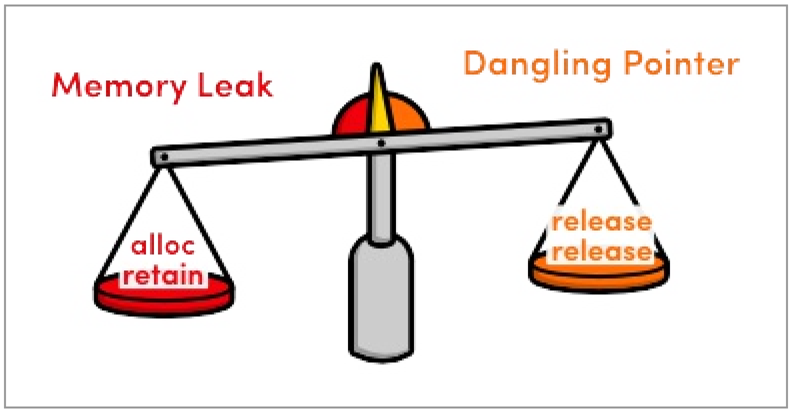
In a Manual Retain Release environment, it's your job to claim and relinquish ownership of every object in your program. You do this by calling special memory-related methods, which are described below.

Method	Behavior
`alloc`	Create an object and claim ownership of it.

`retain`	Claim ownership of an existing object.
`copy`	Copy an object and claim ownership of it.
`release`	Relinquish ownership of an object and destroy it immediately.
`autorelease`	Relinquish ownership of an object but defer its destruction.

Manually controlling object ownership might seem like a daunting task, but it's actually very easy. All you have to do is claim ownership of any object you need and remember to relinquish ownership when you're done with it. From a practical standpoint, this means that you have to balance every `alloc`, `retain`, and `copy` call with a `release` or `autorelease` on the same object.

When you forget to balance these calls, one of two things can happen. If you forget to release an object, its underlying memory is never freed, resulting in a **memory leak**. Small leaks won't have a visible effect on your program, but if you eat up enough memory, your program will eventually crash. On the other hand, if you try to release an object too many times, you'll have what's called a **dangling pointer**. When you try to access the dangling pointer, you'll be requesting an invalid memory address, and your program will most likely crash.

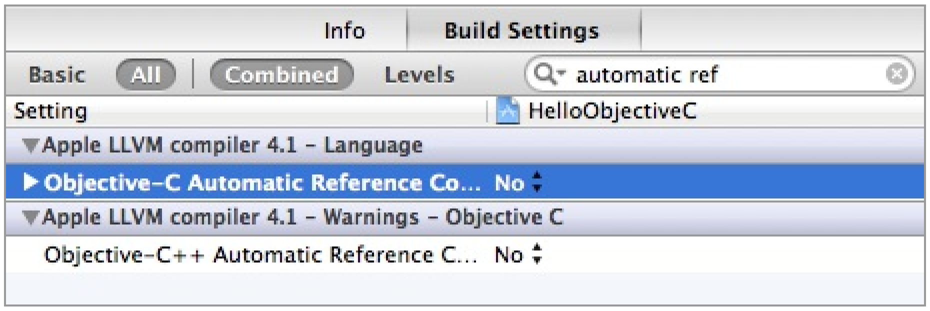


Balancing ownership claims with relinquishes

The rest of this section explains how to avoid memory leaks and dangling pointers through the proper usage of the above methods.

Enabling MRR

Before we can experiment with manual memory management, we need to turn off Automatic Reference Counting. Click the project icon in the *Project Navigator*, make sure the *Build Settings* tab is selected, and start typing `automatic reference counting` in the search bar. The *Objective-C Automatic Reference Counting* compiler option should appear. Change it from *Yes* to *No*.



Setting	HelloObjectiveC
▼Apple LLVM compiler 4.1 – Language	
▶Objective-C Automatic Reference Co...	No
▼Apple LLVM compiler 4.1 – Warnings – Objective C	
Objective-C++ Automatic Reference C...	No

Turning off Automatic Reference Counting

Remember, we're only doing this for instructional purposes—you should never use Manual Retain Release for new projects.

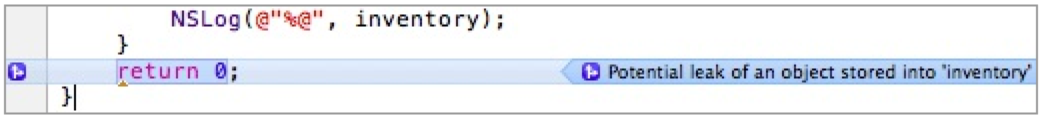
The alloc Method

We've been using the `alloc` method to create objects throughout this tutorial. But, it's not just allocating memory for the object, it's also setting its reference count to `1`. This makes a lot of sense, since we wouldn't be creating the object if we didn't want to keep it around for at least a little while.

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

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSMutableArray *inventory = [[NSMutableArray alloc] init];
        [inventory addObject:@"Honda Civic"];
        NSLog(@"%@", inventory);
    }
    return 0;
}
```

The above code should look familiar. All we're doing is instantiating a mutable array, adding a value, and displaying its contents. From a memory-management perspective, we now own the `inventory` object, which means it's our responsibility to release it somewhere down the road.

But, since we *haven't* released it, our program currently has a memory leak. We can examine this in Xcode by running our project through the static analyzer tool. Navigate to *Product > Analyze* in the menu bar or use the *Shift+Cmd+B* keyboard shortcut. This looks for predictable problems in your code, and it should uncover the following in `main.m`.



```
        NSLog(@"%@", inventory);
    }
    return 0;        Potential leak of an object stored into 'inventory'
}
```

A memory leak! Oh no!

This is a small object, so the leak isn't fatal. However, if it happened over and over (e.g., in a long loop or every time the user clicked a button), the program would eventually run out of memory and crash.

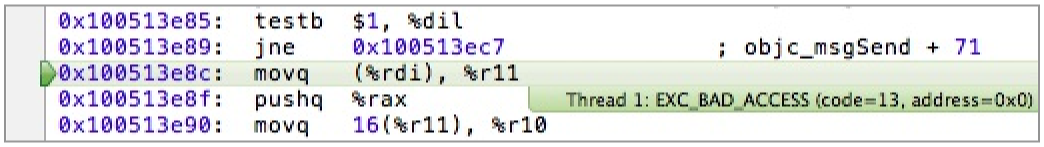
The release Method

The `release` method relinquishes ownership of an object by decrementing its reference count. So, we can get rid of our memory leak by adding the following line after the `NSLog()` call in `main.m`.

```
[inventory release];
```

Now that our `alloc` is balanced with a `release`, the static analyzer shouldn't output any problems. Typically, you'll want to relinquish ownership of an object at the end of the same method in which it was created, as we did here.

Releasing an object too soon creates a dangling pointer. For example, try moving the above line to *before* the `NSLog()` call. Since `release` immediately frees the underlying memory, the `inventory` variable in `NSLog()` now points to an invalid address, and your program will crash with a `EXC_BAD_ACCESS` error code when you try to run it:



```
0x100513e85:  testb  $1, %dil
0x100513e89:  jne    0x100513ec7                 ; objc_msgSend + 71
0x100513e8c:  movq   (%rdi), %r11
0x100513e8f:  pushq  %rax          Thread 1: EXC_BAD_ACCESS (code=13, address=0x0)
0x100513e90:  movq   16(%r11), %r10
```

Trying to access an invalid memory address

The point is, don't relinquish ownership of an object before you're done using it.

The retain Method

The `retain` method claims ownership of an existing object. It's like telling the operating system, "Hey! I need that object too, so don't get rid of it!" This is a necessary ability when other objects need to make sure their properties refer to a valid instance.

As an example, we'll use `retain` to create a strong reference to our `inventory` array. Create a new class called `CarStore` and change its header to the following.

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

```
@interface CarStore : NSObject

- (NSMutableArray *)inventory;
- (void)setInventory:(NSMutableArray *)newInventory;

@end
```

This manually declares the accessors for a property called `inventory`. Our first iteration of `CarStore.m` provides a straightforward implementation of the getter and setter, along with an instance variable to record the object:

```
// CarStore.m
#import "CarStore.h"

@implementation CarStore {
    NSMutableArray *_inventory;
}

- (NSMutableArray *)inventory {
    return _inventory;
}

- (void)setInventory:(NSMutableArray *)newInventory {
    _inventory = newInventory;
}

@end
```

Back in `main.m`, let's assign our `inventory` variable to `CarStore`'s `inventory` property:

```
// main.m
#import <Foundation/Foundation.h>
#import "CarStore.h"

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSMutableArray *inventory = [[NSMutableArray alloc] init];
        [inventory addObject:@"Honda Civic"];
        CarStore *superstore = [[CarStore alloc] init];
        [superstore setInventory:inventory];
        [inventory release];

        // Do some other stuff...

        // Try to access the property later on (error!)
        NSLog(@"%@", [superstore inventory]);
    }
    return 0;
}
```

The `inventory` property in the last line is a dangling pointer because the object was already released earlier in `main.m`. Right now, the `superstore` object has a weak reference to the array. To turn it into a strong reference, `CarStore` needs to claim ownership of the array in its `setInventory:` accessor:

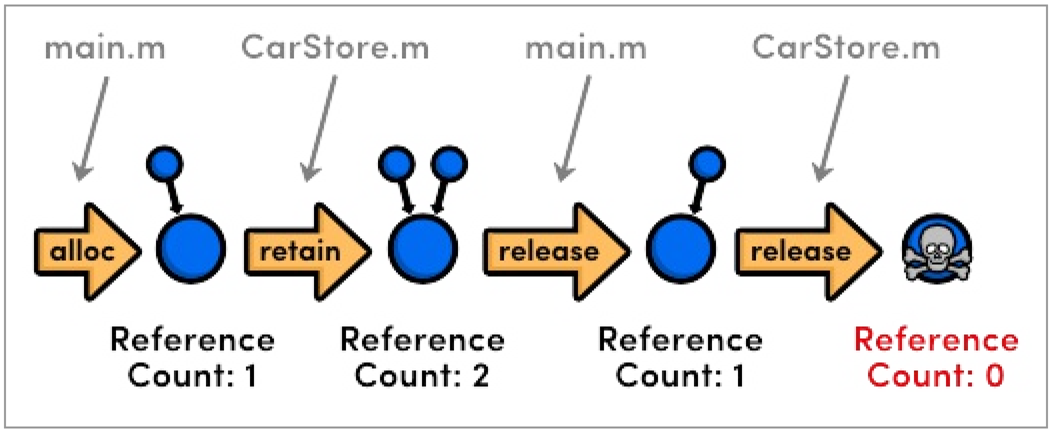
```
// CarStore.m
- (void)setInventory:(NSMutableArray *)newInventory {
    _inventory = [newInventory retain];
}
```

This ensures the `inventory` object won't be released while `superstore` is using it. Notice that the `retain` method returns the object itself, which lets us perform the retain and assignment in a single line.

Unfortunately, this code creates another problem: the `retain` call isn't balanced with a `release`, so we have another memory leak. As soon as we pass another value to `setInventory:`, we can't access the old value, which means we can never free it. To fix this, `setInventory:` needs to call `release` on the old value:

```
// CarStore.m
- (void)setInventory:(NSMutableArray *)newInventory {
    if (_inventory == newInventory) {
        return;
    }
    NSMutableArray *oldValue = _inventory;
    _inventory = [newInventory retain];
    [oldValue release];
}
```

This is basically what the `retain` and the `strong` property attributes do. Obviously, using `@property` is much more convenient than creating these accessors on our own.



Memory management calls on the `inventory` object

The above diagram visualizes all of the memory management calls on the `inventory` array that we created in `main.m`, along with their respective locations. As you can see, all of the `alloc`'s and `retain`'s are balanced with a `release` somewhere down the line, ensuring that the underlying memory will eventually be freed up.

The copy Method

An alternative to `retain` is the `copy` method, which creates a brand new instance of the object and increments the reference count on that, leaving the original unaffected. So, if you want to copy the `inventory` array instead of referring to the mutable one, you can change `setInventory:` to the following.

```
// CarStore.m
- (void)setInventory:(NSMutableArray *)newInventory {
    if (_inventory == newInventory) {
        return;
    }
    NSMutableArray *oldValue = _inventory;
    _inventory = [newInventory copy];
    [oldValue release];
}
```

You may also recall from The `copy` Attribute that this has the added perk of freezing mutable collections at the time of assignment. Some classes provide multiple `copy` methods (much like multiple `init` methods), and it's safe to assume that any method starting with `copy` has the same behavior.

The autorelease method

Like `release`, the `autorelease` method relinquishes ownership of an object, but instead of destroying the object immediately, it defers the actual freeing of memory until later on in the program. This allows you to release objects when you are "supposed" to, while still keeping them around for others to use.

For example, consider a simple factory method that creates and returns a `CarStore` object:

```
// CarStore.h
+ (CarStore *)carStore;
```

Technically speaking, it's the `carStore` method's responsibility to release the object because the caller has no way of knowing that he owns the returned object. So, its implementation should return an autoreleased object, like so:

```
// CarStore.m
+ (CarStore *)carStore {
    CarStore *newStore = [[CarStore alloc] init];
    return [newStore autorelease];
}
```

This relinquishes ownership of the object immediately after creating it, but keeps it in memory long enough for the caller to interact with it. Specifically, it waits until the end of the nearest `@autoreleasepool{}` block, after which it calls a normal `release` method. This is why there's always an `@autoreleasepool{}` surrounding the entire `main()` function—it makes sure all of the autoreleased objects are destroyed after the program is done executing.

All of those built-in factory methods like `NSString`'s `stringWithFormat:` and `stringWithContentsOfFile:` work the exact same way as our `carStore` method. Before ARC, this was a convenient convention, since it let you create objects without worrying about calling `release` somewhere down the road.

If you change the `superstore` constructor from `alloc/init` to the following, you won't have to release it at the end of `main()`.

```
// main.m
CarStore *superstore = [CarStore carStore];
```

In fact, you aren't *allowed* to release the `superstore` instance now because you no longer own it—the `carStore` factory method does. It's very important to avoid explicitly releasing autoreleased objects (otherwise, you'll have a dangling pointer and a crashed program).

The dealloc Method

An object's `dealloc` method is the opposite of its `init` method. It's called right before the object is destroyed, giving you a chance to clean up any internal objects. This method is called automatically by the runtime—you should never try to call `dealloc` yourself.

In an MRR environment, the most common thing you need to do in a `dealloc` method is release objects stored in instance variables. Think about what happens to our current `CarStore` when an instance is deallocated: its `_inventory` instance variable, which has been retained by the setter, never has the chance to be released. This is another form of memory leak. To fix this, all we have to do is add a custom `dealloc` to `CarStore.m`:

```
// CarStore.m
- (void)dealloc {
    [_inventory release];
    [super dealloc];
}
```

Note that you should always call the superclass's `dealloc` to make sure that all of the instance variables in parent classes are properly released. As a general rule, you want to keep custom `dealloc` methods as simple as possible, so you shouldn't try to use them for logic that can be handled elsewhere.

MRR Summary

And that's manual memory management in a nutshell. The key is to balance every `alloc`, `retain`, and `copy` with a `release` or `autorelease`, otherwise you'll encounter either a dangling pointer or a memory leak at some point in your application.

Remember that this section only used Manual Retain Release to understand the internal workings of iOS and OS X memory management. In the real world, much of the above code is actually obsolete, though you might encounter it in older documentation. It's very important to understand that explicitly claiming and relinquishing ownership like this has been completely superseded by Automatic Reference Counting.

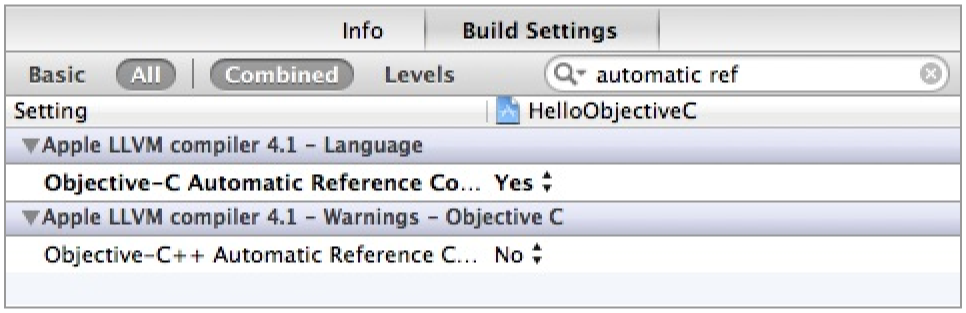
Automatic Reference Counting

Now that you've got your head wrapped around manual memory management, you can forget all about it. Automatic Reference Counting works the exact same way as MRR, but it automatically inserts the appropriate memory-management methods for you. This is a big deal for Objective-C developers, as it lets them focus entirely on *what* their application needs to do rather than *how* it does it.

ARC takes the human error out of memory management with virtually no downside, so the only reason *not* to use it is when you're interfacing with a legacy code base (however, ARC is, for the most part, backward compatible with MRR programs). The rest of this module explains the major changes between MRR and ARC.

Enabling ARC

First, let's go ahead and turn ARC back on in the project's *Build Settings* tab. Change the *Automatic Reference Counting* compiler option to *Yes*. Again, this is the default for all Xcode templates, and it's what you should be using for all of your projects.



Setting | HelloObjectiveC
Apple LLVM compiler 4.1 - Language
Objective-C Automatic Reference Co... Yes
Apple LLVM compiler 4.1 - Warnings - Objective C
Objective-C++ Automatic Reference C... No

Turning on Automatic Reference Counting

No More Memory Methods

ARC works by analyzing your code to figure how what the ideal lifetime of each object should be, then inserts the necessary `retain` and `release` calls automatically. The algorithm requires complete control over the object-ownership in your entire program, which means you're *not allowed* to manually call `retain`, `release`, or `autorelease`.

The only memory-related methods you should ever find in an ARC program are `alloc` and `copy`. You can think of these as plain old constructors and ignore the whole object-ownership thing.

New Property Attributes

ARC introduces new `@property` attributes. You should use the `strong` attribute in place of `retain`, and `weak` in place of `assign`. All of these attributes are discussed in the Properties module.

The dealloc Method in ARC

Deallocation in ARC is also a little bit different. You don't need to release instance variables as we did in The `dealloc` Method—ARC does this for you. In addition, the superclass's `dealloc` is automatically called, so you don't have to do that either.

For the most part, this means you'll never need to include a custom `dealloc` method. One of the few exceptions is when you're using low-level memory allocation functions like `malloc()`. In this case, you'd still have to call `free()` in `dealloc` to avoid a memory leak.

Summary

For the most part, Automatic Reference Counting lets you completely forget about memory management. The idea is to focus on high-level functionality instead of the underlying memory management. The only thing you need to worry about are retain cycles, which was covered in the Properties module.

If you need a more detailed discussion about the nuances of ARC, please visit the Transitioning to ARC Release Notes.

By now, you should know almost everything you need to know about Objective-C. The only thing we haven't covered are the basic data types provided by both C and the Foundation Framework. The next module introduces all of the standard types, from numbers to strings, arrays, dictionaries, and even dates.

Objective-C Data Types

Objective-C inherits all of the primitive types of C, and defines a few of its own, too. But, applications also require higher-level tools like strings, dictionaries, and dates. The Foundation Framework defines several classes that provide the standard, object-oriented data structures found in typical high-level programming languages.

The Primitives module introduces Objective-C's native types, and the rest of the above modules cover the fundamental Foundation classes. Below, you'll find general information about interacting with these classes.

Creating Objects

There are two ways to instantiate objects in Objective-C. First, you have the standard `alloc`/`init` pattern introduced in the Classes module. For example, you can create a new `NSNumber` object with the following:

```
NSNumber *twentySeven = [[NSNumber alloc] initWithInt:27];
```

But, most of the Foundation Framework classes also provide corresponding factory methods, like so:

```
NSNumber *twentySeven = [NSNumber numberWithInt:27];
```

This allocates a new `NSNumber` object for you, autoreleases it, and returns it. Before ARC, this was a convenient way to create objects, but in modern programs, there is no practical difference between these two instantiation patterns.

In addition, numbers, strings, arrays, and dictionaries can be created as literals (e.g., `@"Bill"`). You'll see all three notations throughout the Objective-C literature, as well as in this tutorial.

Comparing Objects

Comparing objects is one of the biggest pitfalls for Objective-C beginners. There are two distinct types of equality comparisons in Objective-C:

- **Pointer comparison** uses the `==` operator to see if two pointers refer to the same memory address (i.e., the same object). It's not possible for different objects to compare equal with this kind of comparison.
- **Value comparison** uses methods like `isEqualToNumber:` to see if two objects represent the same value. It *is* possible for different objects to compare equal with this kind of comparison.

Generally, you'll want to use the second option for more robust comparisons. The relevant methods are introduced alongside each data type (e.g., `NSNumber`'s Comparing Numbers section).

C Primitives

The vast majority of Objective-C's primitive data types are adopted from C, although it does define a few of its own to facilitate its object-oriented capabilities. The first part of this module provides a practical introduction to C's data types, and the second part covers three more primitives that are specific to Objective-C.

The examples in this module use `NSLog()` to inspect variables. In order for them to display correctly, you need to use the correct format specifier in `NSLog()`'s first argument. The various specifiers for C primitives are presented alongside the data types themselves, and all Objective-C objects can be displayed with the `%@` specifier.

The void Type

The `void` type is C's empty data type. Its most common use case is to specify the return type for functions that don't return anything. For example:

```
void sayHello() {
    NSLog(@"This function doesn't return anything");
}
```

The `void` type is not to be confused with the void *pointer*. The former indicates the *absence* of a value, while the latter represents *any* value (well, any pointer, at least).

Integer Types

Integer data types are characterized by their size and whether they are signed or unsigned. The `char` type is always 1 byte, but it's very important to understand that the exact size of the other integer types is implementation-dependent. Instead of being defined as an *absolute* number of bytes, they are defined relative to each other. The only guarantee is that `short <= int <= long <= long long`; however it is possible to determine their exact sizes at runtime.

C was designed to work closely with the underlying architecture, and different systems support different variable sizes. A relative definition provides the flexibility to, for example, define `short`, `int`, and `long` as the same number of bytes when the target chipset can't differentiate between them.

```
BOOL isBool = YES;
NSLog(@"%d", isBool);
NSLog(@"%@", isBool ? @"YES" : @"NO");
```

```
char aChar = 'a';
unsigned char anUnsignedChar = 255;
NSLog(@"The letter %c is ASCII number %hhd", aChar, aChar);
NSLog(@"%hhu", anUnsignedChar);

short aShort = -32768;
unsigned short anUnsignedShort = 65535;
NSLog(@"%hd", aShort);
NSLog(@"%hu", anUnsignedShort);

int anInt = -2147483648;
unsigned int anUnsignedInt = 4294967295;
NSLog(@"%d", anInt);
NSLog(@"%u", anUnsignedInt);

long aLong = -9223372036854775808;
unsigned long anUnsignedLong = 18446744073709551615;
NSLog(@"%ld", aLong);
NSLog(@"%lu", anUnsignedLong);

long long aLongLong = -9223372036854775808;
unsigned long long anUnsignedLongLong = 18446744073709551615;
NSLog(@"%lld", aLongLong);
NSLog(@"%llu", anUnsignedLongLong);
```

The `%d` and `%u` characters are the core specifiers for displaying signed and unsigned integers, respectively. The `hh`, `h`, `l` and `ll` characters are modifiers that tell `NSLog()` to treat the associated integer as a `char`, `short`, `long`, or `long long`, respectively.

It's also worth noting that the `BOOL` type is actually part of Objective-C, not C. Objective-C uses `YES` and `NO` for its Boolean values instead of the `true` and `false` macros used by C.

Fixed-Width Integers

While the basic types presented above are satisfactory for most purposes, it is sometimes necessary to declare a variable that stores a specific number of bytes. This is particularly relevant for algorithms like `arc4random()` that operate on a fixed-width integer.

The `int<n>_t` data types allow you to represent signed and unsigned integers that are exactly 1, 2, 4, or 8 bytes, and the `int_least<n>_t` variants let you constrain the *minimum* size of a variable without specifying an exact number of bytes. In addition, `intmax_t` is an alias for the largest integer type that the system can handle.

```
// Exact integer types
int8_t aOneByteInt = 127;
uint8_t aOneByteUnsignedInt = 255;
int16_t aTwoByteInt = 32767;
uint16_t aTwoByteUnsignedInt = 65535;
int32_t aFourByteInt = 2147483647;
uint32_t aFourByteUnsignedInt = 4294967295;
int64_t anEightByteInt = 9223372036854775807;
uint64_t anEightByteUnsignedInt = 18446744073709551615;

// Minimum integer types
int_least8_t aTinyInt = 127;
uint_least8_t aTinyUnsignedInt = 255;
int_least16_t aMediumInt = 32767;
uint_least16_t aMediumUnsignedInt = 65535;
int_least32_t aNormalInt = 2147483647;
uint_least32_t aNormalUnsignedInt = 4294967295;
int_least64_t aBigInt = 9223372036854775807;
uint_least64_t aBigUnsignedInt = 18446744073709551615;

// The largest supported integer type
intmax_t theBiggestInt = 9223372036854775807;
uintmax_t theBiggestUnsignedInt = 18446744073709551615;
```

Floating-Point Types

C provides three floating-point types. Like the integer data types, they are defined as relative sizes, where `float <= double <= long double`. Literal decimal values are represented as doubles—floats must be explicitly marked with a trailing `f`, and long doubles must be marked with an `L`, as shown below.

```
// Single precision floating-point
float aFloat = -21.09f;
NSLog(@"%f", aFloat);
NSLog(@"%8.2f", aFloat);
```

```
// Double precision floating-point
double aDouble = -21.09;
NSLog(@"%8.2f", aDouble);
NSLog(@"%e", aDouble);

// Extended precision floating-point
long double aLongDouble = -21.09e8L;
NSLog(@"%Lf", aLongDouble);
NSLog(@"%Le", aLongDouble);
```

The `%f` format specifier is used to display floats and doubles as decimal values, and the `%8.2f` syntax determines the padding and the number of points after the decimal. In this case, we pad the output to fill 8 digits and display 2 decimal places. Alternatively, you can format the value as scientific notation with the `%e` specifier. Long doubles require the `L` modifier (similar to `hh`, `l`, etc).

Determining Type Sizes

It's possible to determine the exact size of any data type by passing it to the `sizeof()` function, which returns the number of bytes used to represent the specified type. Running the following snippet is an easy way to see the size of the basic data types on any given architecture.

```
NSLog(@"Size of char: %zu", sizeof(char));   // This will always be 1
NSLog(@"Size of short: %zu", sizeof(short));
NSLog(@"Size of int: %zu", sizeof(int));
NSLog(@"Size of long: %zu", sizeof(long));
NSLog(@"Size of long long: %zu", sizeof(long long));
NSLog(@"Size of float: %zu", sizeof(float));
NSLog(@"Size of double: %zu", sizeof(double));
NSLog(@"Size of size_t: %zu", sizeof(size_t));
```

Note that `sizeof()` can also be used with an array, in which case it returns the number of bytes used by the array. This presents a new problem: the programmer has no idea which data type is required to store the maximum size of an array. Instead of forcing you to guess, the `sizeof()` function returns a special data type called `size_t`. This is why we used the `%zu` format specifier in the above example.

The `size_t` type is dedicated solely to representing memory-related values, and it is guaranteed to be able to store the maximum size of an array. Aside from being the return type for `sizeof()` and other memory utilities, this makes `size_t` an appropriate data type for storing the indices of very large arrays. As with any other type, you can pass it to `sizeof()` to get its exact size at runtime, as shown in the above example.

If your Objective-C programs interact with a lot of C libraries, you're likely to encounter the following application of `sizeof()`:

```
size_t numberOfElements = sizeof(anArray)/sizeof(anArray[0]);
```

This is the canonical way to determine the number of elements in a primitive C array. It simply divides the size of the array, `sizeof(anArray)`, by the size of each element, `sizeof(anArray[0])`.

Limit Macros

While it's trivial to determine the potential range of an integer type once you know how how many bytes it is, C implementations provide convenient macros for accessing the minimum and maximum values that each type can represent:

```
NSLog(@"Smallest signed char: %d", SCHAR_MIN);
NSLog(@"Largest signed char: %d", SCHAR_MAX);
NSLog(@"Largest unsigned char: %u", UCHAR_MAX);

NSLog(@"Smallest signed short: %d", SHRT_MIN);
NSLog(@"Largest signed short: %d", SHRT_MAX);
NSLog(@"Largest unsigned short: %u", USHRT_MAX);

NSLog(@"Smallest signed int: %d", INT_MIN);
NSLog(@"Largest signed int: %d", INT_MAX);
NSLog(@"Largest unsigned int: %u", UINT_MAX);

NSLog(@"Smallest signed long: %ld", LONG_MIN);
NSLog(@"Largest signed long: %ld", LONG_MAX);
NSLog(@"Largest unsigned long: %lu", ULONG_MAX);

NSLog(@"Smallest signed long long: %lld", LLONG_MIN);
NSLog(@"Largest signed long long: %lld", LLONG_MAX);
NSLog(@"Largest unsigned long long: %llu", ULLONG_MAX);

NSLog(@"Smallest float: %e", FLT_MIN);
NSLog(@"Largest float: %e", FLT_MAX);
NSLog(@"Smallest double: %e", DBL_MIN);
NSLog(@"Largest double: %e", DBL_MAX);
NSLog(@"Largest possible array index: %llu", SIZE_MAX);
```

The `SIZE_MAX` macro defines the maximum value that can be stored in a `size_t` variable.

Working With C Primitives

This section takes a look at some common “gotchas“ when working with C’s primitive data types. Keep in mind that these are merely brief overviews of computational topics that often involve a great deal of subtlety.

Choosing an Integer Type

The variety of integer types offered by C can make it hard to know which one to use in any given situation, but the answer is quite simple: use `int`’s unless you have a compelling reason not to.

Traditionally, an `int` is defined to be the native word size of the underlying architecture, so it’s (generally) the most efficient integer type. The only reason to use a `short` is when you want to reduce the memory footprint of very large arrays (e.g., an OpenGL index buffer). The `long` types should only be used when you need to store values that don’t fit into an `int`.

Integer Division

Like most programming languages, C differentiates between integer and floating-point operations. If both operands are integers, the calculation uses integer arithmetic, but if at least one of them is a floating-point type, it uses floating-point arithmetic. This is important to keep in mind for division:

```
int integerResult = 5 / 4;
NSLog(@"Integer division: %d", integerResult);        // 1
double doubleResult = 5.0 / 4;
NSLog(@"Floating-point division: %f", doubleResult);  // 1.25
```

Note that the decimal will always be truncated when dividing two integers, so be sure to use (or cast to) a `float` or a `double` if you need the remainder.

Floating-Point Equality

Floating-point numbers are inherently *not* precise, and certain values simply cannot be represented as a floating-point value. For example, we can inspect the imprecision of the number `0.1` by displaying several decimal places:

```
NSLog(@"%.17f", .1);          // 0.10000000000000001
```

As you can see, 0.1 is not actually represented as 0.1 by your computer. That extra 1 in the 17th digit occurs because converting 1/10 to binary results in a repeating decimal. Of course, a computer cannot store the infinitely many digits required for the exact value, so this introduces a rounding error. The error gets magnified when you start doing calculations with the value, resulting in counterintuitive situations like the following:

```
NSLog(@"%.17f", 4.2 - 4.1); // 0.10000000000000053
if (4.2 - 4.1 == .1) {
    NSLog(@"This math is perfect!");
} else {
    // You'll see this message
    NSLog(@"This math is just a tiny bit off...");
}
```

The lesson here is: don't try to check if two floating-point values are exactly equal, and definitely don't use a floating-point type to store precision-sensitive data (e.g., monetary values). To represent exact quantities, you should use the fixed-point NSDecimalNumber class.

For a comprehensive discussion of the issues surrounding floating-point math, please see What Every Computer Scientist Should Know About Floating-Point Arithmetic by David Goldberg.

Objective-C Primitives

In addition to the data types discussed above, Objective-C defines three of its own primitive types: id, Class, and SEL. These are the basis for Objective-C's dynamic typing capabilities. This section also introduces the anomalous NSInteger and NSUInteger types.

The id Type

The id type is the generic type for all Objective-C objects. You can think of it as the object-oriented version of C's void pointer. And, like a void pointer, it can store a reference to *any* type of object. The following example uses the same id variable to hold a string and a dictionary.

```
id mysteryObject = @"An NSString object";
NSLog(@"%@", [mysteryObject description]);
mysteryObject = @{@"model": @"Ford", @"year": @1967};
NSLog(@"%@", [mysteryObject description]);
```

Recall that all Objective-C objects are referenced as pointers, so when they are statically typed, they must be declared with pointer notation: `NSString *mysteryObject`. However, the `id` type automatically implies that the variable is a pointer, so this is not necessary: `id mysteryObject` (without the asterisk).

The Class Type

Objective-C classes are represented as objects themselves, using a special data type called `Class`. This lets you, for example, dynamically check an object's type at runtime:

```
Class targetClass = [NSString class];
id mysteryObject = @"An NSString object";
if ([mysteryObject isKindOfClass:targetClass]) {
    NSLog(@"Yup! That's an instance of the target class");
}
```

All classes implement a class-level method called `class` that returns its associated class object (apologies for the redundant terminology). This object can be used for introspection, which we see with the `isKindOfClass:` method above.

The SEL Type

The `SEL` data type is used to store selectors, which are Objective-C's internal representation of a method name. For example, the following snippet stores a method called `sayHello` in the `someMethod` variable. This variable could be used to dynamically call a method at runtime.

```
SEL someMethod = @selector(sayHello);
```

Please refer to the Methods module for a thorough discussion of Objective-C's selectors.

NSInteger and NSUInteger

While they aren't technically native Objective-C types, this is a good time to discuss the Foundation Framework's custom integers, `NSInteger` and `NSUInteger`. On 32-bit systems, these are defined to be 32-bit signed/unsigned integers, respectively, and on 64-bit systems, they are 64-bit integers. In other words, they are guaranteed to be the natural word size on any given architecture.

The original purpose of these Apple-specific types was to facilitate the transition from 32-bit architectures to 64-bit, but it's up to you whether or not you want to use `NSInteger` over the basic types (`int`, `long`, `long long`) and the `int<n>_t` variants. A sensible convention is to use `NSInteger` and `NSUInteger` when interacting with Apple's APIs and use the standard C types for everything else.

Either way, it's still important to understand what `NSInteger` and `NSUInteger` represent, as they are used extensively in Foundation, UIKit, and several other frameworks.

NSNumber

The `NSNumber` class is a lightweight, object-oriented wrapper around C's numeric primitives. It's main job is to store and retrieve primitive values, and it comes with dedicated methods for each data type:

```
NSNumber *aBool = [NSNumber numberWithBool:NO];
NSNumber *aChar = [NSNumber numberWithChar:'z'];
NSNumber *aUChar = [NSNumber numberWithUnsignedChar:255];
NSNumber *aShort = [NSNumber numberWithShort:32767];
NSNumber *aUShort = [NSNumber numberWithUnsignedShort:65535];
NSNumber *anInt = [NSNumber numberWithInt:2147483647];
NSNumber *aUInt = [NSNumber numberWithUnsignedInt:4294967295];
NSNumber *aLong = [NSNumber numberWithLong:9223372036854775807];
NSNumber *aULong = [NSNumber numberWithUnsignedLong:18446744073709551615];
NSNumber *aFloat = [NSNumber numberWithFloat:26.99f];
NSNumber *aDouble = [NSNumber numberWithDouble:26.99];

NSLog(@"%@", [aBool boolValue] ? @"YES" : @"NO");
NSLog(@"%c", [aChar charValue]);
NSLog(@"%hhu", [aUChar unsignedCharValue]);
NSLog(@"%hi", [aShort shortValue]);
NSLog(@"%hu", [aUShort unsignedShortValue]);
NSLog(@"%i", [anInt intValue]);
NSLog(@"%u", [aUInt unsignedIntValue]);
NSLog(@"%li", [aLong longValue]);
NSLog(@"%lu", [aULong unsignedLongValue]);
NSLog(@"%f", [aFloat floatValue]);
NSLog(@"%f", [aDouble doubleValue]);
```

It may seem redundant to have an object-oriented version of all the C primitives, but it's necessary if you want to store numeric values in an `NSArray`, `NSDictionary`, or any of the other Foundation Framework collections. These classes require all of their elements to be objects—they do not know how to interact with primitive values. In addition, `NSNumber` makes it possible to pass numbers to methods like `performSelector:withObject:`, as discussed in Selectors.

But, aside from the object-oriented interface, there are a few perks to using `NSNumber`. For one, it provides a straightforward way to convert between primitive data types or get an `NSString` representation of the value:

```
NSNumber *anInt = [NSNumber numberWithInt:42];
float asFloat = [anInt floatValue];
NSLog(@"%.2f", asFloat);
NSString *asString = [anInt stringValue];
NSLog(@"%@", asString);
```

And, like all Objective-C objects, `NSNumber` can be displayed with the `%@` format specifier, which means that you can forget about all of the C-style specifiers introduced in the Primitives module. This makes life a tiny bit easier when trying to debug values:

```
NSNumber *aUChar = [NSNumber numberWithUnsignedChar:255];
NSNumber *anInt = [NSNumber numberWithInt:2147483647];
NSNumber *aFloat = [NSNumber numberWithFloat:26.99f];
NSLog(@"%@", aUChar);
NSLog(@"%@", anInt);
NSLog(@"%@", aFloat);
```

Numeric Literals

Xcode 4.4 introduced numeric literals, which offer a much more convenient alternative to the above factory methods. The `NSNumber` version of `BOOL`'s, `char`'s, `int`'s and `double`'s can all be created by simply prefixing the corresponding primitive type with the `@` symbol; however, `unsigned int`'s, `long`'s, and `float`'s must be appended with the `U`, `L`, or `F` modifiers, as shown below.

```
NSNumber *aBool = @NO;
NSNumber *aChar = @'z';
NSNumber *anInt = @2147483647;
NSNumber *aUInt = @4294967295U;
NSNumber *aLong = @9223372036854775807L;
NSNumber *aFloat = @26.99F;
NSNumber *aDouble = @26.99;
```

In addition to literal values, it's possible to box arbitrary C expressions using the `@()` syntax. This makes it trivial to turn basic arithmetic calculations into `NSNumber` objects:

```
double x = 24.0;
NSNumber *result = @(x * .15);
NSLog(@"%.2f", [result doubleValue]);
```

Immutability

It's important to understand that `NSNumber` objects are immutable—it's not possible to change its associated value after you create it. In this sense, an `NSNumber` instance acts exactly like a primitive value: When you need a new `double` value, you create a new literal—you don't change an existing one.

From a practical standpoint, this means you need to create a new `NSNumber` object every time you change its value. For example, the following loop increments a `counter` object by adding to its primitive value, then re-boxing it into a new `NSNumber` and assigning it to the same variable.

```
NSNumber *counter = @0;
for (int i=0; i<10; i++) {
    counter = @([counter intValue] + 1);
    NSLog(@"%@", counter);
}
```

As you could probably imagine, this isn't the most efficient way to work with numbers. In real applications, you should limit yourself to primitive numeric types for computationally intensive algorithms and wait as long as possible to store the result in an `NSNumber` container.

Comparing Numbers

While it's possible to directly compare `NSNumber` pointers, the `isEqualToNumber:` method is a much more robust way to check for equality. It guarantees that two *values* will compare equal, even if they are stored in different *objects*. For example, the following snippet shows a common case of pointer comparison failure.

```
NSNumber *anInt = @27;
NSNumber *sameInt = @27U;
// Pointer comparison (fails)
if (anInt == sameInt) {
    NSLog(@"They are the same object");
}
// Value comparison (succeeds)
if ([anInt isEqualToNumber:sameInt]) {
    NSLog(@"They are the same value");
}
```

If you need to check for inequalities, you can use the related `compare:` method. Instead of a Boolean value, it returns an NSComparisonResult, which is an enum that defines the relationship between the operands:

Return Value	**Description**
NSOrderedAscending	receiver < argument
NSOrderedSame	receiver == argument
NSOrderedDescending	receiver > argument

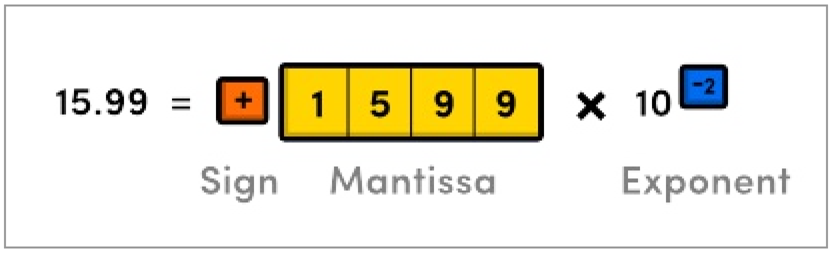
The following example shows these enumerators in action.

```
 NSNumber *anInt = @27;

NSNumber *anotherInt = @42;

NSComparisonResult result = [anInt compare:anotherInt];

if (result == NSOrderedAscending) {

    NSLog(@"27 < 42");

} else if (result == NSOrderedSame) {

    NSLog(@"27 == 42");

} else if (result == NSOrderedDescending) {

    NSLog(@"27 > 42");

}
```

This kind of object-oriented comparison may not be as convenient as the familiar <, ==, and > operators, but abstracting them into methods affords a lot more flexibility to the Foundation Framework classes.

NSDecimalNumber

The `NSDecimalNumber` class provides fixed-point arithmetic capabilities to Objective-C programs. They're designed to perform base-10 calculations without loss of precision and with predictable rounding behavior. This makes it a better choice for representing currency than floating-point data types like `double`. However, the trade-off is that they are more complicated to work with.



Internally, a fixed-point number is expressed as `sign mantissa x 10^exponent`. The sign defines whether it's positive or negative, the mantissa is an unsigned integer representing the significant digits, and the exponent determines where the decimal point falls in the mantissa.

It's possible to manually assemble an `NSDecimalNumber` from a mantissa, exponent, and sign, but it's often easier to convert it from a string representation. The following snippet creates the value `15.99` using both methods.

```
NSDecimalNumber *price;
price = [NSDecimalNumber decimalNumberWithMantissa:1599
                                          exponent:-2
                                        isNegative:NO];
price = [NSDecimalNumber decimalNumberWithString:@"15.99"];
```

Like `NSNumber`, all `NSDecimalNumber` objects are immutable, which means you cannot change their value after they've been created.

Arithmetic

The main job of `NSDecimalNumber` is to provide fixed-point alternatives to C's native arithmetic operations. All five of `NSDecimalNumber`'s arithmetic methods are demonstrated below.

```
NSDecimalNumber *price1 = [NSDecimalNumber decimalNumberWithString:@"15.99"];
NSDecimalNumber *price2 = [NSDecimalNumber decimalNumberWithString:@"29.99"];
NSDecimalNumber *coupon = [NSDecimalNumber decimalNumberWithString:@"5.00"];
NSDecimalNumber *discount = [NSDecimalNumber decimalNumberWithString:@".90"];
NSDecimalNumber *numProducts = [NSDecimalNumber decimalNumberWithString:@"2.0"];
NSDecimalNumber *subtotal = [price1 decimalNumberByAdding:price2];
NSDecimalNumber *afterCoupon = [subtotal decimalNumberBySubtracting:coupon];
NSDecimalNumber *afterDiscount = [afterCoupon decimalNumberByMultiplyingBy:discount];
NSDecimalNumber *average = [afterDiscount decimalNumberByDividingBy:numProducts];
NSDecimalNumber *averageSquared = [average decimalNumberByRaisingToPower:2];

NSLog(@"Subtotal: %@", subtotal);                    // 45.98
NSLog(@"After coupon: %@", afterCoupon);           // 40.98
NSLog((@"After discount: %@"), afterDiscount);       // 36.882
NSLog(@"Average price per product: %@", average);    // 18.441
NSLog(@"Average price squared: %@", averageSquared); // 340.070481
```

Unlike their floating-point counterparts, these operations are guaranteed to be accurate. However, you'll notice that many of the above calculations result in extra decimal places. Depending on the application, this may or may not be desirable (e.g., you might want to constrain currency values to 2 decimal places). This is where custom rounding behavior comes in.

Rounding Behavior

Each of the above arithmetic methods have an alternate `withBehavior:` form that let you define how the operation rounds the resulting value. The `NSDecimalNumberHandler` class encapsulates a particular rounding behavior and can be instantiated as follows:

```
NSDecimalNumberHandler *roundUp = [NSDecimalNumberHandler
                                   decimalNumberHandlerWithRoundingMode:NSRoundUp
                                   scale:2
                                   raiseOnExactness:NO
                                   raiseOnOverflow:NO
                                   raiseOnUnderflow:NO
                                   raiseOnDivideByZero:YES];
```

The `NSRoundUp` argument makes all operations round up to the nearest place. Other rounding options are `NSRoundPlain`, `NSRoundDown`, and `NSRoundBankers`, all of which are defined by `NSRoundingMode`. The `scale:` parameter defines the number of decimal places the resulting value should have, and the rest of the parameters define the exception-handling behavior of any operations. In this case, `NSDecimalNumber` will only raise an exception if you try to divide by zero.

This rounding behavior can then be passed to the `decimalNumberByMultiplyingBy:withBehavior:` method (or any of the other arithmetic methods), as shown below.

```
NSDecimalNumber *subtotal = [NSDecimalNumber decimalNumberWithString:@"40.98"];
NSDecimalNumber *discount = [NSDecimalNumber decimalNumberWithString:@".90"];

NSDecimalNumber *total = [subtotal decimalNumberByMultiplyingBy:discount
                                                   withBehavior:roundUp];
NSLog(@"Rounded total: %@", total);
```

Now, instead of `36.882`, the `total` gets rounded up to two decimal points, resulting in `36.89`.

Comparing NSDecimalNumbers

Like `NSNumber`, `NSDecimalNumber` objects should use the `compare:` method instead of the native inequality operators. Again, this ensures that *values* are compared, even if they are stored in different *instances*. For example:

```
NSDecimalNumber *discount1 = [NSDecimalNumber decimalNumberWithString:@".85"];
NSDecimalNumber *discount2 = [NSDecimalNumber decimalNumberWithString:@".9"];
NSComparisonResult result = [discount1 compare:discount2];
if (result == NSOrderedAscending) {
    NSLog(@"85%% < 90%%");
} else if (result == NSOrderedSame) {
    NSLog(@"85%% == 90%%");
} else if (result == NSOrderedDescending) {
    NSLog(@"85%% > 90%%");
}
```

`NSDecimalNumber` also inherits the `isEqualToNumber:` method from `NSNumber`.

Decimal Numbers in C

For most practical purposes, the `NSDecimalNumber` class should satisfy your fixed-point needs; however, it's worth noting that there is also a function-based alternative available in pure C. This provides increased efficiency over the OOP interface discussed above and is thus preferred for high-performance applications dealing with a large number of calculations.

NSDecimal

Instead of an `NSDecimalNumber` object, the C interface is built around the `NSDecimal` `struct`. Unfortunately, the Foundation Framework doesn't make it easy to create an `NSDecimal` from scratch. You need to generate one from a full-fledged `NSDecimalNumber` using its `decimalValue` method. There is a corresponding factory method, also shown below.

```
NSDecimalNumber *price = [NSDecimalNumber decimalNumberWithString:@"15.99"];
NSDecimal asStruct = [price decimalValue];
NSDecimalNumber *asNewObject = [NSDecimalNumber decimalNumberWithDecimal:asStruct];
```

This isn't exactly an ideal way to create `NSDecimal`'s, but once you have a `struct` representation of your initial values, you can stick to the functional API presented below. All of these functions use `struct`'s as inputs and outputs.

Arithmetic Functions

In lieu of the arithmetic methods of `NSDecimalNumber`, the C interface uses functions like `NSDecimalAdd()`, `NSDecimalSubtract()`, etc. Instead of returning the result, these functions populate the first argument with the calculated value. This makes it possible to reuse an existing `NSDecimal` in several operations and avoid allocating unnecessary structs just to hold intermediary values.

For example, the following snippet uses a single `result` variable across 5 function calls. Compare this to the Arithmetic section, which created a new `NSDecimalNumber` object for each calculation.

```
NSDecimal price1 = [[NSDecimalNumber decimalNumberWithString:@"15.99"] decimalValue];
NSDecimal price2 = [[NSDecimalNumber decimalNumberWithString:@"29.99"] decimalValue];
NSDecimal coupon = [[NSDecimalNumber decimalNumberWithString:@"5.00"] decimalValue];
NSDecimal discount = [[NSDecimalNumber decimalNumberWithString:@".90"] decimalValue];
NSDecimal numProducts = [[NSDecimalNumber decimalNumberWithString:@"2.0"] decimalValue];
NSLocale *locale = [NSLocale currentLocale];
NSDecimal result;

NSDecimalAdd(&result, &price1, &price2, NSRoundUp);
NSLog(@"Subtotal: %@", NSDecimalString(&result, locale));
NSDecimalSubtract(&result, &result, &coupon, NSRoundUp);
NSLog(@"After coupon: %@", NSDecimalString(&result, locale));
NSDecimalMultiply(&result, &result, &discount, NSRoundUp);
NSLog(@"After discount: %@", NSDecimalString(&result, locale));
```

```
NSDecimalDivide(&result, &result, &numProducts, NSRoundUp);
NSLog(@"Average price per product: %@", NSDecimalString(&result, locale));
NSDecimalPower(&result, &result, 2, NSRoundUp);
NSLog(@"Average price squared: %@", NSDecimalString(&result, locale));
```

Notice that these functions accept *references* to `NSDecimal` structs, which is why we need to use the reference operator (`&`) instead of passing them directly. Also note that rounding is an inherent part of each operation—it's not encapsulated in a separate entity like `NSDecimalNumberHandler`.

The `NSLocale` instance defines the formatting of `NSDecimalString()`, and is discussed more thoroughly in the Dates module.

Error Checking

Unlike their OOP counterparts, the arithmetic functions don't raise exceptions when a calculation error occurs. Instead, they follow the common C pattern of using the return value to indicate success or failure. All of the above functions return an `NSCalculationError`, which defines what kind of error occurred. The potential scenarios are demonstrated below.

```
NSDecimal a = [[NSDecimalNumber decimalNumberWithString:@"1.0"] decimalValue];
NSDecimal b = [[NSDecimalNumber decimalNumberWithString:@"0.0"] decimalValue];
NSDecimal result;
NSCalculationError success = NSDecimalDivide(&result, &a, &b, NSRoundPlain);
switch (success) {
    case NSCalculationNoError:
        NSLog(@"Operation successful");
        break;
    case NSCalculationLossOfPrecision:
        NSLog(@"Error: Operation resulted in loss of precision");
        break;
    case NSCalculationUnderflow:
        NSLog(@"Error: Operation resulted in underflow");
        break;
    case NSCalculationOverflow:
        NSLog(@"Error: Operation resulted in overflow");
        break;
    case NSCalculationDivideByZero:
        NSLog(@"Error: Tried to divide by zero");
        break;
    default:
        break;
}
```

Comparing NSDecimals

Comparing `NSDecimal`’s works exactly like the OOP interface, except you use the `NSDecimalCompare()` function:

```
NSDecimal discount1 = [[NSDecimalNumber decimalNumberWithString:@".85"] decimalValue];
NSDecimal discount2 = [[NSDecimalNumber decimalNumberWithString:@".9"] decimalValue];
NSComparisonResult result = NSDecimalCompare(&discount1, &discount2);
if (result == NSOrderedAscending) {
    NSLog(@"85%% < 90%%");
} else if (result == NSOrderedSame) {
    NSLog(@"85%% == 90%%");
} else if (result == NSOrderedDescending) {
    NSLog(@"85%% > 90%%");
}
```

NSString

As we've already seen several times throughout this tutorial, the `NSString` class is the basic tool for representing text in an Objective-C application. Aside from providing an object-oriented wrapper for strings, `NSString` provides many powerful methods for searching and manipulating its contents. It also comes with native Unicode support.

Like `NSNumber` and `NSDecimalNumber`, `NSString` is an immutable type, so you cannot change it after it's been instantiated. It does, however, have a mutable counterpart called `NSMutableString`, which will be discussed at the end of this module.

Creating Strings

The most common way to create strings is using the literal `@"Some String"` syntax, but the `stringWithFormat:` class method is also useful for generating strings that are composed of variable values. It takes the same kind of format string as `NSLog()`:

```
NSString *make = @"Porsche";
NSString *model = @"911";
int year = 1968;
NSString *message = [NSString stringWithFormat:@"That's a %@ %@ from %d!",
                     make, model, year];
NSLog(@"%@", message);
```

Notice that we used the `@"%@"` format specifier in the `NSLog()` call instead of passing the string directly with `NSLog(message)`. Using a literal for the first argument of `NSLog()` is a best practice, as it sidesteps potential bugs when the string you want to display contains `%` signs. Think about what would happen when `message = @"The tank is 50% full"`.

`NSString` provides built-in support for Unicode, which means that you can include UTF-8 characters directly in string literals. For example, you can paste the following into Xcode and use it the same way as any other `NSString` object.

```
NSString *make = @"Côte d'Ivoire";
```

Enumerating Strings

The two most basic `NSString` methods are `length` and `characterAtIndex:`, which return the number of characters in the string and the character at a given index, respectively. You probably won't have to use these methods unless you're doing low-level string manipulation, but they're still good to know:

```
NSString *make = @"Porsche";
for (int i=0; i<[make length]; i++) {
    unichar letter = [make characterAtIndex:i];
    NSLog(@"%d: %hu", i, letter);
}
```

As you can see, `characterAtIndex:` has a return type of `unichar`, which is a `typedef` for `unsigned short`. This value represents the Unicode decimal number for the character.

Comparing Strings

String comparisons present the same issues as `NSNumber` comparisons. Instead of comparing *pointers* with the == operator, you should always use the `isEqualToString:` method for a more robust *value* comparison. The following example shows you how this works, along with the useful `hasPrefix:` and `hasSuffix:` methods for partial comparisons.

```
NSString *car = @"Porsche Boxster";
if ([car isEqualToString:@"Porsche Boxster"]) {
    NSLog(@"That car is a Porsche Boxster");
}
if ([car hasPrefix:@"Porsche"]) {
    NSLog(@"That car is a Porsche of some sort");
}
if ([car hasSuffix:@"Carrera"]) {
    // This won't execute
    NSLog(@"That car is a Carrera");
}
```

And, just like `NSNumber`, `NSString` has a `compare:` method that can be useful for alphabetically sorting strings:

```
NSString *otherCar = @"Ferrari";
NSComparisonResult result = [car compare:otherCar];
if (result == NSOrderedAscending) {
    NSLog(@"The letter 'P' comes before 'F'");
} else if (result == NSOrderedSame) {
    NSLog(@"We're comparing the same string");
} else if (result == NSOrderedDescending) {
    NSLog(@"The letter 'P' comes after 'F'");
}
```

Note that this is a case-sensitive comparison, so uppercase letters will always be *before* their lowercase counterparts. If you want to ignore case, you can use the related `caseInsensitiveCompare:` method.

Combining Strings

The two methods presented below are a way to concatenate `NSString` objects. But, remember that `NSString` is an immutable type, so these methods actually return a *new* string and leave the original arguments unchanged.

```
NSString *make = @"Ferrari";
NSString *model = @"458 Spider";
NSString *car = [make stringByAppendingString:model];
NSLog(@"%@", car);         // Ferrari458 Spider
car = [make stringByAppendingFormat:@" %@", model];
NSLog(@"%@", car);         // Ferrari 458 Spider (note the space)
```

Searching Strings

`NSString`'s search methods all return an `NSRange` struct, which defines a `location` and a `length` field. The `location` is the index of the beginning of the match, and the `length` is the number of characters in the match. If no match was found, `location` will contain `NSNotFound`. For example, the following snippet searches for the `Cabrio` substring.

```
NSString *car = @"Maserati GranCabrio";
NSRange searchResult = [car rangeOfString:@"Cabrio"];
if (searchResult.location == NSNotFound) {
    NSLog(@"Search string was not found");
} else {
    NSLog(@"'Cabrio' starts at index %lu and is %lu characters long",
          searchResult.location,        // 13
          searchResult.length);         // 6
}
```

The next section shows you how to create `NSRange` structs from scratch.

Subdividing Strings

You can divide an existing string by specifying the first/last index of the desired substring. Again, since `NSString` is immutable, the following methods return a new object, leaving the original intact.

```
NSString *car = @"Maserati GranTurismo";
NSLog(@"%@", [car substringToIndex:8]);               // Maserati
NSLog(@"%@", [car substringFromIndex:9]);             // GranTurismo
NSRange range = NSMakeRange(9, 4);
NSLog(@"%@", [car substringWithRange:range]);         // Gran
```

The global `NSMakeRange()` method creates an `NSRange` struct. The first argument specifies the `location` field, and the second defines the `length` field. The `substringWithRange:` method interprets these as the first index of the substring and the number of characters to include, respectively.

It's also possible to split a string into an `NSArray` using the `componentsSeparatedByString:` method, as shown below.

```
NSString *models = @"Porsche,Ferrari,Maserati";
NSArray *modelsAsArray = [models componentsSeparatedByString:@","];
NSLog(@"%@", [modelsAsArray objectAtIndex:1]);        // Ferrari
```

Replacing Substrings

Replacing part of a string is just like subdividing a string, except you provide a replacement along with the substring you're looking for. The following snippet demonstrates the two most common substring replacement methods.

```
NSString *elise = @"Lotus Elise";
NSRange range = NSMakeRange(6, 5);
NSString *exige = [elise stringByReplacingCharactersInRange:range
                                                withString:@"Exige"];
NSLog(@"%@", exige);          // Lotus Exige
NSString *evora = [exige stringByReplacingOccurrencesOfString:@"Exige"
                                                  withString:@"Evora"];
NSLog(@"%@", evora);          // Lotus Evora
```

Changing Case

The `NSString` class also provides a few convenient methods for changing the case of a string. This can be used to normalize user-submitted values. As with all `NSString` manipulation methods, these return new strings instead of changing the existing instance.

```
NSString *car = @"lotUs beSpoKE";
NSLog(@"%@", [car lowercaseString]);      // lotus bespoke
NSLog(@"%@", [car uppercaseString]);      // LOTUS BESPOKE
NSLog(@"%@", [car capitalizedString]);    // Lotus Bespoke
```

Numerical Conversions

`NSString` defines several conversion methods for interpreting strings as primitive values. These are occasionally useful for very simple string processing, but you should really consider `NSScanner` or `NSNumberFormatter` if you need a robust string-to-number conversion tool.

```
NSString *year = @"2012";
BOOL asBool = [year boolValue];
int asInt = [year intValue];
NSInteger asInteger = [year integerValue];
long long asLongLong = [year longLongValue];
float asFloat = [year floatValue];
double asDouble = [year doubleValue];
```

NSMutableString

The `NSMutableString` class is a mutable version of `NSString`. Unlike immutable strings, it's possible to alter individual characters of a mutable string without creating a brand new object. This makes `NSMutableString` the preferred data structure when you're performing several small edits on the same string.

`NSMutableString` inherits from `NSString`, so aside from the ability to manipulate it in place, you can use a mutable string just like you would an immutable string. That is to say, the API discussed above will still work with an `NSMutableString` instance, although methods like `stringByAppendingString:` will still return a `NSString` object—not an `NSMutableString`.

The remaining sections present several methods defined by the `NSMutableString` class. You'll notice that the fundamental workflow for mutable strings is different than that of immutable ones. Instead of creating a new object and replacing the old value, `NSMutableString` methods operate directly on the existing instance.

Creating Mutable Strings

Mutable strings can be created through the `stringWithString:` class method, which turns a literal string or an existing `NSString` object into a mutable one:

```
NSMutableString *car = [NSMutableString stringWithString:@"Porsche 911"];
```

After you've created a mutable string, the `setString:` method lets you assign a new value to the instance:

```
[car setString:@"Porsche Boxster"];
```

Compare this to `NSString`, where you re-assign a new value to the variable. With mutable strings, we don't change the instance reference, but rather manipulate its contents through the mutable API.

Expanding Mutable Strings

`NSMutableString` provides mutable alternatives to many of the `NSString` manipulation methods discussed above. Again, the mutable versions don't need to copy the resulting string into a new memory location and return a new reference to it. Instead, they directly change the existing object's underlying value.

```
NSMutableString *car = [NSMutableString stringWithCapacity:20];
NSString *model = @"458 Spider";

[car setString:@"Ferrari"];
[car appendString:model];
NSLog(@"%@", car);                    // Ferrari458 Spider

[car setString:@"Ferrari"];
[car appendFormat:@" %@", model];
NSLog(@"%@", car);                    // Ferrari 458 Spider

[car setString:@"Ferrari Spider"];
[car insertString:@"458 " atIndex:8];
```

```
NSLog(@"%@", car);                    // Ferrari 458 Spider
```

Also note that, like any well-designed Objective-C class, the method names of `NSString` and `NSMutableString` reflect exactly what they do. The former creates and returns a brand new string, so it uses names like `stringByAppendingString:`. On the other hand, the latter operates on the object itself, so it uses verbs like `appendString:`.

Replacing/Deleting Substrings

It's possible to replace or delete substrings via the `replaceCharactersInRange:withString:` and `deleteCharactersInRange:` methods, as shown below.

```
NSMutableString *car = [NSMutableString stringWithCapacity:20];
[car setString:@"Lotus Elise"];
[car replaceCharactersInRange:NSMakeRange(6, 5)
                   withString:@"Exige"];
NSLog(@"%@", car);                               // Lotus Exige
[car deleteCharactersInRange:NSMakeRange(5, 6)];
NSLog(@"%@", car);                               // Lotus
```

When To Use Mutable Strings

Since `NSString` and `NSMutableString` provide such similar functionality, it can be hard to know when to use one over the other. In general, the static nature of `NSString` makes it more efficient for most tasks; however, the fact that an immutable string can't be changed without generating a new object makes it less than ideal when you're trying to perform several small edits.

The two examples presented in this section demonstrate the advantages of mutable strings. First, let's take a look at an anti-pattern for immutable strings. The following loop generates a string containing all of the numbers between 0 and 999 using `NSString`.

```
// DO NOT DO THIS. EVER.
NSString *indices = @"";
for (int i=0; i<1000; i++) {
    indices = [indices stringByAppendingFormat:@"%d", i];
}
```

Remember that `stringByAppendingFormat:` creates a new `NSString` instance, which means that in each iteration, the entire string gets copied to a new block of memory. The above code allocates 999 string objects that serve only as intermediary values, resulting in an application that requires a whopping 1.76 MB of memory. Needless to say, this is incredibly inefficient.

Now, let’s take a look at the mutable version of this snippet:

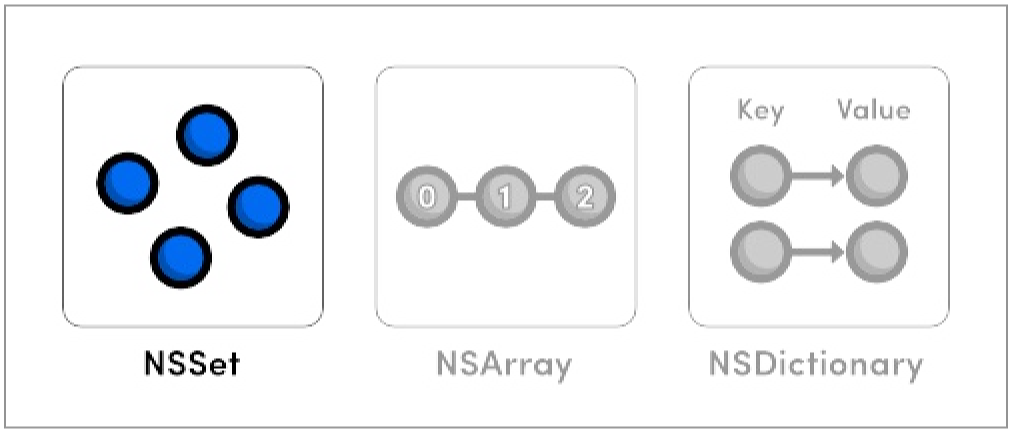
```
NSMutableString *indices = [NSMutableString stringWithCapacity:1];
for (int i=0; i<1000; i++) {
    [indices appendFormat:@"%d", i];
}
```

Since mutable strings manipulate their contents in place, no copying is involved, and we completely avoid the 999 unnecessary allocations. Internally, the mutable string’s storage dynamically expands to accommodate longer values. This reduces the memory footprint to around 19 KB, which is much more reasonable.

So, a good rule of thumb is to use a mutable string whenever you’re running any kind of algorithm that edits or assembles a string in several passes and to use an immutable string for everything else. This also applies to sets, arrays, and dictionaries.

NSSet

NSSet, NSArray, and NSDictionary are the three core collection classes defined by the Foundation Framework. An NSSet object represents a static, unordered collection of distinct objects. Sets are optimized for membership checking, so if you're asking a lot of "is this object part of this group?" kind of questions, you should be using a set—not an array.



The basic collection classes of the Foundation Framework

Collections can only interact with Objective-C objects. As a result, primitive C types like int need to be wrapped in an NSNumber before you can store them in a set, array, or dictionary.

NSSet is immutable, so you cannot add or remove elements from a set after it's been created. You can, however, alter mutable objects that are contained in the set. For example, if you stored an NSMutableString, you're free to call setString:, appendFormat:, and the other manipulation methods on that object. This module also covers NSMutableSet and NSCountedSet.

Creating Sets

An NSSet can be created through the setWithObjects: class method, which accepts a nil-terminated list of objects. Most of the examples in this module utilize strings, but an NSSet instance can record *any* kind of Objective-C object, and it does not have to be homogeneous.

```
NSSet *americanMakes = [NSSet setWithObjects:@"Chrysler", @"Ford",
                                             @"General Motors", nil];
NSLog(@"%@", americanMakes);
```

NSSet also includes a setWithArray: method, which turns an NSArray into an NSSet. Remember that sets are composed of *unique* elements, so this serves as a convenient way to remove all duplicates in an array. For example:

```
NSArray *japaneseMakes = @[@"Honda", @"Mazda",
                           @"Mitsubishi", @"Honda"];
NSSet *uniqueMakes = [NSSet setWithArray:japaneseMakes];
NSLog(@"%@", uniqueMakes);    // Honda, Mazda, Mitsubishi
```

Sets maintain a strong relationship with their elements. That is to say, a set owns each item that it contains. You should be careful to avoid retain cycles when creating sets of custom objects by ensuring that an element in the set never has a strong reference to the set itself.

Enumerating Sets

Fast-enumeration is the preferred method of iterating through the contents of a set, and the `count` method can be used to calculate the total number of items. For example:

```
NSSet *models = [NSSet setWithObjects:@"Civic", @"Accord",
                                      @"Odyssey", @"Pilot", @"Fit", nil];
NSLog(@"The set has %li elements", [models count]);
for (id item in models) {
    NSLog(@"%@", item);
}
```

If you're interested in a block-based solution, you can also use the `enumerateObjectsUsingBlock:` method to process the contents of a set. The method's only parameter is a `^(id obj, BOOL *stop)` block. `obj` is the current object, and the `*stop` pointer lets you prematurely exit the iteration by setting its value to `YES`, as demonstrated below.

```
[models enumerateObjectsUsingBlock:^(id obj, BOOL *stop) {
    NSLog(@"Current item: %@", obj);
    if ([obj isEqualToString:@"Fit"]) {
        NSLog(@"I was looking for a Honda Fit, and I found it!");
        *stop = YES;    // Stop enumerating items
    }
}];
```

The `*stop = YES` line tells the set to stop enumerating once it reaches the `@"Fit"` element. This the block equivalent of the `break` statement.

Note that since sets are unordered, it usually doesn't make sense to access an element outside of an enumeration. Accordingly, `NSSet` does *not* support subscripting syntax for accessing individual elements (e.g., `models[i]`). This is one of the primary differences between sets and arrays/dictionaries.

Comparing Sets

In addition to equality, two `NSSet` objects can be checked for subset and intersection status. All three of these comparisons are demonstrated in the following example.

```
NSSet *japaneseMakes = [NSSet setWithObjects:@"Honda", @"Nissan",
                              @"Mitsubishi", @"Toyota", nil];
NSSet *johnsFavoriteMakes = [NSSet setWithObjects:@"Honda", nil];
NSSet *marysFavoriteMakes = [NSSet setWithObjects:@"Toyota",
                                                  @"Alfa Romeo", nil];

if ([johnsFavoriteMakes isEqualToSet:japaneseMakes]) {
    NSLog(@"John likes all the Japanese auto makers and no others");
}
if ([johnsFavoriteMakes intersectsSet:japaneseMakes]) {
    // You'll see this message
    NSLog(@"John likes at least one Japanese auto maker");
}
if ([johnsFavoriteMakes isSubsetOfSet:japaneseMakes]) {
    // And this one, too
    NSLog(@"All of the auto makers that John likes are Japanese");
}
if ([marysFavoriteMakes isSubsetOfSet:japaneseMakes]) {
    NSLog(@"All of the auto makers that Mary likes are Japanese");
}
```

Membership Checking

Like all Foundation Framework collections, it's possible to check if an object is in a particular `NSSet`. The `containsObject:` method returns a `BOOL` indicating the membership status of the argument. As an alternative, the `member:` returns a reference to the object if it's in the set, otherwise `nil`. This can be convenient depending on how you're using the set.

```
NSSet *selectedMakes = [NSSet setWithObjects:@"Maserati",
                                             @"Porsche", nil];
// BOOL checking
```

```
if ([selectedMakes containsObject:@"Maserati"]) {
    NSLog(@"The user seems to like expensive cars");
}
// nil checking
NSString *result = [selectedMakes member:@"Maserati"];
if (result != nil) {
    NSLog(@"%@ is one of the selected makes", result);
}
```

Again, this is one of the strong suits of sets, so if you're doing a lot of membership checking, you should be using `NSSet` instead of `NSArray` (unless you have a compelling reason not to).

Filtering Sets

You can filter the contents of a set using the `objectsPassingTest:` method, which accepts a block that is called using each item in the set. The block should return `YES` if the current object should be included in the new set, and `NO` if it shouldn't. The following example finds all items that begin with an uppercase letter `C`.

```
NSSet *toyotaModels = [NSSet setWithObjects:@"Corolla", @"Sienna",
                       @"Camry", @"Prius",
                       @"Highlander", @"Sequoia", nil];
NSSet *cModels = [toyotaModels objectsPassingTest:^BOOL(id obj, BOOL *stop) {
    if ([obj hasPrefix:@"C"]) {
        return YES;
    } else {
        return NO;
    }
}];
NSLog(@"%@", cModels);    // Corolla, Camry
```

Because `NSSet` is immutable, the `objectsPassingTest:` method returns a *new* set instead of altering the existing one. This is the same behavior as many of the `NSString` manipulation operations. But, while the *set* is a new instance, it still refers to the same *elements* as the original set. That is to say, filtered elements are not copied—they are referenced.

Combining Sets

Sets can be combined using the `setByAddingObjectsFromSet:` method. Since sets are unique, duplicates will be ignored if both sets contain the same object.

```
NSSet *affordableMakes = [NSSet setWithObjects:@"Ford", @"Honda",
                                 @"Nissan", @"Toyota", nil];
NSSet *fancyMakes = [NSSet setWithObjects:@"Ferrari", @"Maserati",
                                 @"Porsche", nil];
NSSet *allMakes = [affordableMakes setByAddingObjectsFromSet:fancyMakes];
NSLog(@"%@", allMakes);
```

NSMutableSet

Mutable sets allow you to add or delete objects dynamically, which affords a whole lot more flexibility than the static `NSSet`. In addition to membership checking, mutable sets are also more efficient at inserting and removing elements than `NSMutableArray`.

`NSMutableSet` can be very useful for recording the state of a system. For example, if you were writing an application to manage an auto repair shop, you might maintain a mutable set called `repairedCars` and add/remove cars to reflect whether or not they have been fixed yet.

Creating Mutable Sets

Mutable sets can be created with the exact same methods as `NSSet`. Or, you can create an empty set with the `setWithCapacity:` class method. The argument defines the initial amount of space allocated for the set, but it in no way limits the number of items it can hold.

```
NSMutableSet *brokenCars = [NSMutableSet setWithObjects:
                            @"Honda Civic", @"Nissan Versa", nil];
NSMutableSet *repairedCars = [NSMutableSet setWithCapacity:5];
```

Adding and Removing Objects

The big additions provided by `NSMutableSet` are the `addObject:` and `removeObject:` methods. Note that `addObject:` won't actually do anything if the object is already a member of the collection because sets are composed of unique items.

```
NSMutableSet *brokenCars = [NSMutableSet setWithObjects:
                            @"Honda Civic", @"Nissan Versa", nil];
NSMutableSet *repairedCars = [NSMutableSet setWithCapacity:5];
// "Fix" the Honda Civic
```

```
[brokenCars removeObject:@"Honda Civic"];
[repairedCars addObject:@"Honda Civic"];

NSLog(@"Broken cars: %@", brokenCars);     // Nissan Versa
NSLog(@"Repaired cars: %@", repairedCars); // Honda Civic
```

Just like mutable strings, `NSMutableSet` has a different workflow than the static `NSSet`. Instead of generating a new set and re-assigning it to the variable, you can operate directly on the existing set.

You may also find the `removeAllObjects` method useful for completely clearing a set.

Filtering With Predicates

There is no mutable version of the `objectsPassingTest:` method, but you can still filter items with `filterUsingPredicate:`. Predicates are somewhat outside the scope of this tutorial, but suffice it to say that they are designed to make it easier to define search/filter rules. Fortunately, the `NSPredicate` class can be initialized with a block, so we don't need to learn an entirely new format syntax.

The following code snippet is the mutable, predicate-based version of the example from the Filtering Sets section above. Again, this operates directly on the existing set.

```
NSMutableSet *toyotaModels = [NSMutableSet setWithObjects:@"Corolla", @"Sienna",
                              @"Camry", @"Prius",
                              @"Highlander", @"Sequoia", nil];
NSPredicate *startsWithC = [NSPredicate predicateWithBlock:
                            ^BOOL(id evaluatedObject, NSDictionary *bindings) {
                                if ([evaluatedObject hasPrefix:@"C"]) {
                                    return YES;
                                } else {
                                    return NO;
                                }
                            }];
[toyotaModels filterUsingPredicate:startsWithC];
NSLog(@"%@", toyotaModels);    // Corolla, Camry
```

For more information about predicates, please visit the official Predicate Programming Guide.

Set Theory Operations

`NSMutableSet` also provides an API for the basic operations in set theory. These methods let you take the union, intersection, and relative complement of two sets. In addition, the `setSet:` method is also useful for creating a shallow copy of a different set. All of these are included in the following example.

```
NSSet *japaneseMakes = [NSSet setWithObjects:@"Honda", @"Nissan",
                        @"Mitsubishi", @"Toyota", nil];
NSSet *johnsFavoriteMakes = [NSSet setWithObjects:@"Honda", nil];
NSSet *marysFavoriteMakes = [NSSet setWithObjects:@"Toyota",
                             @"Alfa Romeo", nil];

NSMutableSet *result = [NSMutableSet setWithCapacity:5];
// Union
[result setSet:johnsFavoriteMakes];
[result unionSet:marysFavoriteMakes];
NSLog(@"Either John's or Mary's favorites: %@", result);
// Intersection
[result setSet:johnsFavoriteMakes];
[result intersectSet:japaneseMakes];
NSLog(@"John's favorite Japanese makes: %@", result);
// Relative Complement
[result setSet:japaneseMakes];
[result minusSet:johnsFavoriteMakes];
NSLog(@"Japanese makes that are not John's favorites: %@",
      result);
```

Enumeration Considerations

Iterating over a mutable set works the same as a static set, with one very important caveat: you aren't allowed to change the set while you're enumerating it. This is a general rule for any collection class.

The following example demonstrates the *wrong* way to mutate a set in the middle of a for-in loop. We'll be using the rather contrived scenario of removing `@"Toyota"` if any element in the set begins with the letter `T`.

```
// DO NOT DO THIS. EVER.
NSMutableSet *makes = [NSMutableSet setWithObjects:@"Ford", @"Honda",
                       @"Nissan", @"Toyota", nil];
for (NSString *make in makes) {
    NSLog(@"%@", make);
    if ([make hasPrefix:@"T"]) {
        // Throws an NSGenericException:
        // "Collection was mutated while being enumerated"
        [makes removeObject:@"Toyota"];
    }
}
```

The proper way to do this is shown below. Instead of iterating over the set directly, you should create a temporary copy of it with the `allObjects` method and iterate over that. This frees you to alter the original set without any unintended consequences:

```
NSMutableSet *makes = [NSMutableSet setWithObjects:@"Ford", @"Honda",
                       @"Nissan", @"Toyota", nil];
NSArray *snapshot = [makes allObjects];
for (NSString *make in snapshot) {
    NSLog(@"%@", make);
    if ([make hasPrefix:@"T"]) {
        [makes removeObject:@"Toyota"];
    }
}
NSLog(@"%@", makes);
```

NSCountedSet

The `NSCountedSet` class (also called a “bag”) is worth a brief mention. It’s a subclass of `NSMutableSet`, but instead of being limited to *unique* values, it counts the number of times an object has been added to the collection. This is a very efficient way to keep object tallies, as it requires only one instance of an object regardless of how many times it’s been added to the bag.

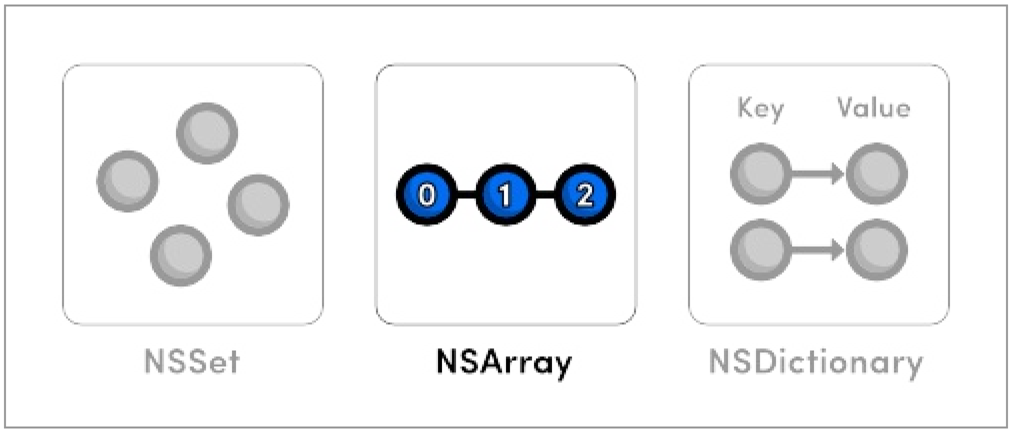
The main difference between a mutable set and `NSCountedSet` is the `countForObject:` method. This will often be used in place of `containsObject:` (which still works as expected). For example:

```
NSCountedSet *inventory = [NSCountedSet setWithCapacity:5];
[inventory addObject:@"Honda Accord"];
[inventory addObject:@"Honda Accord"];
[inventory addObject:@"Nissan Altima"];
NSLog(@"There are %li Accords in stock and %li Altima",
      [inventory countForObject:@"Honda Accord"],    // 2
      [inventory countForObject:@"Nissan Altima"]);  // 1
```

Please see the official documentation for more details.

NSArray

`NSArray` is Objective-C's general-purpose array type. It represents an ordered collection of objects, and it provides a high-level interface for sorting and otherwise manipulating lists of data. Arrays aren't as efficient at membership checking as sets, but the trade-off is that they reliably record the order of their elements.



The basic collection classes of the Foundation Framework

Like `NSSet`, `NSArray` is immutable, so you cannot dynamically add or remove items. Its mutable counterpart, `NSMutableArray`, is discussed in the second part of this module.

Creating Arrays

Immutable arrays can be defined as literals using the `@[]` syntax. This was added relatively late in the evolution of the language (Xcode 4.4), so you're likely to encounter the more verbose `arrayWithObjects:` factory method at some point in your Objective-C career. Both options are included below.

```
NSArray *germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",
                         @"Opel", @"Volkswagen", @"Audi"];
NSArray *ukMakes = [NSArray arrayWithObjects:@"Aston Martin",
                    @"Lotus", @"Jaguar", @"Bentley", nil];

NSLog(@"First german make: %@", germanMakes[0]);
NSLog(@"First U.K. make: %@", [ukMakes objectAtIndex:0]);
```

As you can see, individual items can be accessed through the square-bracket subscripting syntax (`germanMakes[0]`) or the `objectAtIndex:` method. Prior to Xcode 4.4, `objectAtIndex:` was the standard way to access array elements.

Enumerating Arrays

Fast-enumeration is the most efficient way to iterate over an `NSArray`, and its contents are guaranteed to appear in the correct order. It's also possible to use the `count` method with a traditional for-loop to step through each element in the array:

```
NSArray *germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",
                         @"Opel", @"Volkswagen", @"Audi"];
// With fast-enumeration
for (NSString *item in germanMakes) {
    NSLog(@"%@", item);
}
// With a traditional for loop
for (int i=0; i<[germanMakes count]; i++) {
    NSLog(@"%d: %@", i, germanMakes[i]);
}
```

If you're fond of blocks, you can use the `enumerateObjectsUsingBlock:` method. It works the same as `NSSet`'s version, except the index of each item is also passed to the block, so its signature looks like `^(id obj, NSUInteger idx, BOOL *stop)`. And of course, the objects are passed to the block in the same order as they appear in the array.

```
[germanMakes enumerateObjectsUsingBlock:^(id obj,
                                          NSUInteger idx,
                                          BOOL *stop) {
    NSLog(@"%ld: %@", idx, obj);
}];
```

Comparing Arrays

Arrays can be compared for equality with the aptly named `isEqualToArray:` method, which returns `YES` when both arrays have the same number of elements and every pair pass an `isEqual:` comparison. `NSArray` does not offer the same subset and intersection comparisons as `NSSet`.

```
NSArray *germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",
                         @"Opel", @"Volkswagen", @"Audi"];
NSArray *sameGermanMakes = [NSArray arrayWithObjects:@"Mercedes-Benz",
                            @"BMW", @"Porsche", @"Opel",
                            @"Volkswagen", @"Audi", nil];
```

```
if ([germanMakes isEqualToArray:sameGermanMakes]) {
    NSLog(@"Oh good, literal arrays are the same as NSArrays");
}
```

Membership Checking

`NSArray` provides similar membership checking utilities to `NSSet`. The `containsObject:` method works the exact same (it returns `YES` if the object is in the array, `NO` otherwise), but instead of `member:`, `NSArray` uses `indexOfObject:`. This either returns the index of the first occurrence of the requested object or `NSNotFound` if it's not in the array.

```
NSArray *germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",
                         @"Opel", @"Volkswagen", @"Audi"];
// BOOL checking
if ([germanMakes containsObject:@"BMW"]) {
    NSLog(@"BMW is a German auto maker");
}
// Index checking
NSUInteger index = [germanMakes indexOfObject:@"BMW"];
if (index == NSNotFound) {
    NSLog(@"Well that's not quite right...");
} else {
    NSLog(@"BMW is a German auto maker and is at index %ld", index);
}
```

Since arrays can contain more than one reference to the same object, it's possible that the first occurrence isn't the only one. To find other occurrences, you can use the related `indexOfObject:inRange:` method.

Remember that sets are more efficient for membership checking, so if you're querying against a large collection of objects, you should probably be using a set instead of an array.

Sorting Arrays

Sorting is one of the main advantages of arrays. One of the most flexible ways to sort an array is with the `sortedArrayUsingComparator:` method. This accepts an `^NSComparisonResult(id obj1, id obj2)` block, which should return one of the following enumerators depending on the relationship between `obj1` and `obj2`:

Return Value	Description
`NSOrderedAscending`	`obj1` comes before `obj2`
`NSOrderedSame`	`obj1` and `obj2` have no order
`NSOrderedDescending`	`obj1` comes after `obj2`

The following example sorts a list of car manufacturers based on how long their name is, from shortest to longest.

```
NSArray *germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",
                         @"Opel", @"Volkswagen", @"Audi"];
NSArray *sortedMakes = [germanMakes sortedArrayUsingComparator:
    ^NSComparisonResult(id obj1, id obj2) {
        if ([obj1 length] < [obj2 length]) {
            return NSOrderedAscending;
        } else if ([obj1 length] > [obj2 length]) {
            return NSOrderedDescending;
        } else {
            return NSOrderedSame;
        }
}];
NSLog(@"%@", sortedMakes);
```

Like `NSSet`, `NSArray` is immutable, so the sorted array is actually a *new* array, though it still references the same elements as the original array (this is the same behavior as `NSSet`).

Filtering Arrays

You can filter an array with the `filteredArrayUsingPredicate:` method. A short introduction to predicates can be found in the NSSet module, and a minimal example is included below. Just as with the sort method discussed above, this generates a brand new array.

```
NSArray *germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",
                         @"Opel", @"Volkswagen", @"Audi"];

NSPredicate *beforeL = [NSPredicate predicateWithBlock:
    ^BOOL(id evaluatedObject, NSDictionary *bindings) {
        NSComparisonResult result = [@"L" compare:evaluatedObject];
        if (result == NSOrderedDescending) {
            return YES;
        } else {
            return NO;
        }
    }];
NSArray *makesBeforeL = [germanMakes
                         filteredArrayUsingPredicate:beforeL];
NSLog(@"%@", makesBeforeL);    // BMW, Audi
```

Subdividing Arrays

Subdividing an array is essentially the same as extracting substrings from an `NSString`, but instead of `substringWithRange:`, you use `subarrayWithRange:`, as shown below.

```
NSArray *germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",
                         @"Opel", @"Volkswagen", @"Audi"];

NSArray *lastTwo = [germanMakes subarrayWithRange:NSMakeRange(4, 2)];
NSLog(@"%@", lastTwo);    // Volkswagen, Audi
```

Combining Arrays

Arrays can be combined via `arrayByAddingObjectsFromArray:`. As with all of the other immutable methods discussed above, this returns a *new* array containing all of the elements in the original array, along with the contents of the parameter.

```
NSArray *germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",
                         @"Opel", @"Volkswagen", @"Audi"];
NSArray *ukMakes = @[@"Aston Martin", @"Lotus", @"Jaguar", @"Bentley"];

NSArray *allMakes = [germanMakes arrayByAddingObjectsFromArray:ukMakes];
NSLog(@"%@", allMakes);
```

String Conversion

The `componentsJoinedByString:` method concatenates each element of the array into a string, separating them by the specified symbol(s).

```
NSArray *ukMakes = @[@"Aston Martin", @"Lotus", @"Jaguar", @"Bentley"];
```

```
NSLog(@"%@", [ukMakes componentsJoinedByString:@", "]);
```

This can be useful for regular expression generation, file path manipulation, and rudimentary CSV processing; however, if you're doing serious work with file paths/data, you'll probably want to look for a dedicated library.

NSMutableArray

The `NSMutableArray` class lets you dynamically add or remove items from arbitrary locations in the collection. Keep in mind that it's slower to insert or delete elements from a mutable array than a set or a dictionary.

Like mutable sets, mutable arrays are often used to represent the state of a system, but the fact that `NSMutableArray` records the order of its elements opens up new modeling opportunities. For instance, consider the auto repair shop we talked about in the NSSet module. Whereas a set can only represent the status of a collection of automobiles, an array can record the order in which they should be fixed.

Creating Mutable Arrays

Literal arrays are always immutable, so the easiest way to create mutable arrays is still through the `arrayWithObjects:` method. Be careful to use `NSMutableArray`'s version of the method, not `NSArray`'s. For example:

```
NSMutableArray *brokenCars = [NSMutableArray arrayWithObjects:
                               @"Audi A6", @"BMW Z3",
                               @"Audi Quattro", @"Audi TT", nil];
```

You can create empty mutable arrays using the `array` or `arrayWithCapacity:` class methods. Or, if you already have an immutable array that you want to convert to a mutable one, you can pass it to the `arrayWithArray:` class method.

Adding and Removing Objects

The two basic methods for manipulating the contents of an array are the `addObject:` and `removeLastObject` methods. The former adds an object to the end of the array, and the latter is pretty self-documenting. Note that these are also useful methods for treating an `NSArray` as a stack.

```
NSMutableArray *brokenCars = [NSMutableArray arrayWithObjects:
```

```
                                @"Audi A6", @"BMW Z3",
                                @"Audi Quattro", @"Audi TT", nil];
[brokenCars addObject:@"BMW F25"];
NSLog(@"%@", brokenCars);       // BMW F25 added to end
[brokenCars removeLastObject];
NSLog(@"%@", brokenCars);       // BMW F25 removed from end
```

It's most efficient to add or remove items at the *end* of an array, but you can also insert or delete objects at arbitrary locations using `insertObject:atIndex:` and `removeObjectAtIndex:`. Or, if you don't know the index of a particular object, you can use the `removeObject:` method, which is really just a convenience method for `indexOfObject:` followed by `removeObjectAtIndex:`.

```
// Add BMW F25 to front
[brokenCars insertObject:@"BMW F25" atIndex:0];
// Remove BMW F25 from front
[brokenCars removeObjectAtIndex:0];
// Remove Audi Quattro
[brokenCars removeObject:@"Audi Quattro"];
```

It's also possible to replace the contents of an index with the `replaceObjectAtIndex:withObject:` method, as shown below.

```
// Change second item to Audi Q5
[brokenCars replaceObjectAtIndex:1 withObject:@"Audi Q5"];
```

These are the basic methods for manipulating mutable arrays, but be sure to check out the official documentation if you need more advanced functionality.

Sorting With Descriptors

Inline sorts can be accomplished through `sortUsingComparator:`, which works just like the immutable version discussed in Sorting Arrays. However, this section discusses an alternative method for sorting arrays called `NSSortDescriptor`. Sort descriptors typically result in code that is more semantic and less redundant than block-based sorts.

The `NSSortDescriptor` class encapsulates all of the information required to sort an array of dictionaries or custom objects. This includes the property to be compared, the comparison method, and whether the sort is ascending or descending. Once you configure a descriptor(s), you can sort an array by passing it to the `sortUsingDescriptors:` method. For example, the following snippet sorts an array of cars by price and then by model.

```
NSDictionary *car1 = @{
    @"make": @"Volkswagen",
    @"model": @"Golf",
    @"price": [NSDecimalNumber decimalNumberWithString:@"18750.00"]
};
NSDictionary *car2 = @{
    @"make": @"Volkswagen",
    @"model": @"Eos",
    @"price": [NSDecimalNumber decimalNumberWithString:@"35820.00"]
};
NSDictionary *car3 = @{
    @"make": @"Volkswagen",
    @"model": @"Jetta A5",
    @"price": [NSDecimalNumber decimalNumberWithString:@"16675.00"]
};
NSDictionary *car4 = @{
    @"make": @"Volkswagen",
    @"model": @"Jetta A4",
    @"price": [NSDecimalNumber decimalNumberWithString:@"16675.00"]
};
NSMutableArray *cars = [NSMutableArray arrayWithObjects:
                        car1, car2, car3, car4, nil];

NSSortDescriptor *priceDescriptor = [NSSortDescriptor
                                     sortDescriptorWithKey:@"price"
                                                 ascending:YES
                                                  selector:@selector(compare:)];
NSSortDescriptor *modelDescriptor = [NSSortDescriptor
                                     sortDescriptorWithKey:@"model"
                                     ascending:YES
                                     selector:@selector(caseInsensitiveCompare:)];


NSArray *descriptors = @[priceDescriptor, modelDescriptor];
[cars sortUsingDescriptors:descriptors];
NSLog(@"%@", cars);    // car4, car3, car1, car2
```

The descriptor’s `selector` is called on each key’s value, so in the above code, we’re calling `compare:` on `item[@"price"]` and `caseInsensitiveCompare:` on `item[@"model"]` for each pair of items in the array.

Filtering Mutable Arrays

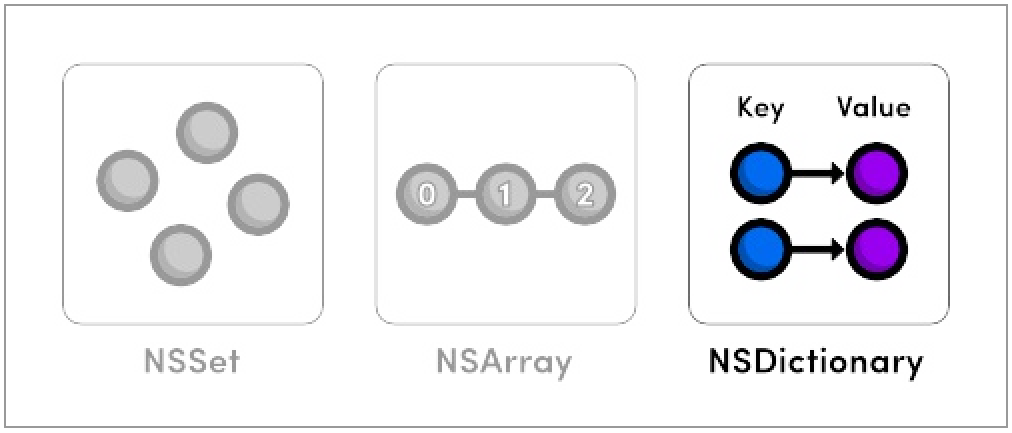
Filtering works the same as it does with immutable arrays, except items are removed from the existing array instead of generating a new one, and you use the `filterUsingPredicate:` method.

Enumeration Considerations

As with all mutable collections, you're not allowed to alter it in the middle of an enumeration. This is covered in the Enumeration Considerations section of the `NSSet` module, but instead of using `allObjects` for the snapshot, you can create a temporary copy of the original array by passing it to the `arrayWithArray:` class method.

NSDictionary

Like an NSSet, the NSDictionary class represents an unordered collection of objects; however, they associate each value with a key, which acts like a label for the value. This is useful for modeling relationships between pairs of objects.



The basic collection classes of the Foundation Framework

NSDictionary is immutable, but the NSMutableDictionary data structure lets you dynamically add and remove entries as necessary.

Creating Dictionaries

Immutable dictionaries can be defined using the literal @{} syntax. But, like array literals, this was added relatively recently, so you should also be aware of the dictionaryWithObjectsAndKeys: and dictionaryWithObjects:forKeys: factory methods. All of these are presented below.

```
// Literal syntax
NSDictionary *inventory = @{
    @"Mercedes-Benz SLK250" : [NSNumber numberWithInt:13],
    @"Mercedes-Benz E350" : [NSNumber numberWithInt:22],
    @"BMW M3 Coupe" : [NSNumber numberWithInt:19],
    @"BMW X6" : [NSNumber numberWithInt:16],
};
// Values and keys as arguments
inventory = [NSDictionary dictionaryWithObjectsAndKeys:
             [NSNumber numberWithInt:13], @"Mercedes-Benz SLK250",
             [NSNumber numberWithInt:22], @"Mercedes-Benz E350",
             [NSNumber numberWithInt:19], @"BMW M3 Coupe",
             [NSNumber numberWithInt:16], @"BMW X6", nil];
// Values and keys as arrays
```

```
NSArray *models = @[@"Mercedes-Benz SLK250", @"Mercedes-Benz E350",
                    @"BMW M3 Coupe", @"BMW X6"];
NSArray *stock = @[[NSNumber numberWithInt:13],
                   [NSNumber numberWithInt:22],
                   [NSNumber numberWithInt:19],
                   [NSNumber numberWithInt:16]];
inventory = [NSDictionary dictionaryWithObjects:stock forKeys:models];
NSLog(@"%@", inventory);
```

The `dictionaryWithObjectsAndKeys:` method treats its argument list as value-key pairs, so every two parameters define a single entry. This ordering is somewhat counterintuitive, so make sure that the value always comes *before* its associated key. The `dictionaryWithObjects:ForKeys:` method is a little bit more straightforward, but you should be careful to ensure that the key array is the same length as the value array.

For common programming tasks, you'll probably never need to use anything but an `NSString` as a key, but it's worth noting that any object that adopts the `NSCopying` protocol and implements the `hash` and the `isEqual:` methods can be used as a key. This is necessary because keys are copied when an entry is added to the array. However, this is not true for values, which are strongly referenced (just like set and array elements).

Accessing Values and Keys

You can use the same subscripting syntax as arrays (`someDict[key]`) to access the value for a particular key. The `objectForKey:` method is the other common way to access values.

```
NSDictionary *inventory = @{
    @"Mercedes-Benz SLK250" : [NSNumber numberWithInt:13],
    @"Mercedes-Benz E350" : [NSNumber numberWithInt:22],
    @"BMW M3 Coupe" : [NSNumber numberWithInt:19],
    @"BMW X6" : [NSNumber numberWithInt:16],
};
NSLog(@"There are %@ X6's in stock", inventory[@"BMW X6"]);
NSLog(@"There are %@ E350's in stock",
      [inventory objectForKey:@"Mercedes-Benz E350"]);
```

Typically, you'll want to access values from keys, as shown above; however, it's possible to do a reverse lookup to get a value's key(s) with the `allKeysForObject:` method. Note that this returns an *array* because multiple keys can map to the same value (but not vice versa). For example, the following extracts all of the keys with a value of `0`.

```
NSDictionary *inventory = @{
    @"Mercedes-Benz SLK250" : [NSNumber numberWithInt:0],
    @"Mercedes-Benz E350" : [NSNumber numberWithInt:0],
    @"BMW M3 Coupe" : [NSNumber numberWithInt:19],
    @"BMW X6" : [NSNumber numberWithInt:16],
};
NSArray *outOfStock = [inventory allKeysForObject:
                        [NSNumber numberWithInt:0]];
NSLog(@"The following cars are currently out of stock: %@",
      [outOfStock componentsJoinedByString:@", "]);
```

Enumerating Dictionaries

As with sets and arrays, fast-enumeration is the most efficient way to enumerate a dictionary, and it loops through the *keys* (not the values). `NSDictionary` also defines a `count` method, which returns the number of entries in the collection.

```
NSDictionary *inventory = @{
    @"Mercedes-Benz SLK250" : [NSNumber numberWithInt:13],
    @"Mercedes-Benz E350" : [NSNumber numberWithInt:22],
    @"BMW M3 Coupe" : [NSNumber numberWithInt:19],
    @"BMW X6" : [NSNumber numberWithInt:16],
};
NSLog(@"We currently have %ld models available", [inventory count]);
for (id key in inventory) {
    NSLog(@"There are %@ %@'s in stock", inventory[key], key);
}
```

You can isolate a dictionary's keys/values with the `allKeys`/`allValues` methods, which return an `NSArray` of each key/value in the collection, respectively. Note that there is no guarantee that these methods will return keys and values in the same order.

```
NSLog(@"Models: %@", [inventory allKeys]);
NSLog(@"Stock: %@", [inventory allValues]);
```

For block-oriented developers, `enumerateKeysAndObjectsUsingBlock:` offers yet another way to enumerate entries. The block is called for each entry, and both the key and the value are passed as arguments:

```
[inventory enumerateKeysAndObjectsUsingBlock:^(id key, id obj, BOOL *stop) {
    NSLog(@"There are %@ %@'s in stock", obj, key);
}];
```

Comparing Dictionaries

Comparing dictionaries works the same as comparing arrays. The `isEqualToDictionary:` method returns `YES` when both dictionaries contain the same key-value pairs:

```
NSDictionary *warehouseInventory = @{
    @"Mercedes-Benz SLK250" : [NSNumber numberWithInt:13],
    @"Mercedes-Benz E350" : [NSNumber numberWithInt:22],
    @"BMW M3 Coupe" : [NSNumber numberWithInt:19],
    @"BMW X6" : [NSNumber numberWithInt:16],
};
NSDictionary *showroomInventory = @{
    @"Mercedes-Benz SLK250" : [NSNumber numberWithInt:13],
    @"Mercedes-Benz E350" : [NSNumber numberWithInt:22],
    @"BMW M3 Coupe" : [NSNumber numberWithInt:19],
    @"BMW X6" : [NSNumber numberWithInt:16],
};
if ([warehouseInventory isEqualToDictionary:showroomInventory]) {
    NSLog(@"Why are we storing so many cars in our showroom?");
}
```

Sorting Dictionary Keys

Dictionaries can't be directly sorted into a new `NSDictionary` instance, but it is possible to sort the *keys* of the dictionary with `keysSortedByValueUsingComparator:`, which accepts a block that should return one of the `NSComparisonResult` enumerators described in the NSArray module. The following example sorts the models from most expensive to most affordable.

```
NSDictionary *prices = @{
    @"Mercedes-Benz SLK250" : [NSDecimalNumber decimalNumberWithString:@"42900.00"],
    @"Mercedes-Benz E350" : [NSDecimalNumber decimalNumberWithString:@"51000.00"],
    @"BMW M3 Coupe" : [NSDecimalNumber decimalNumberWithString:@"62000.00"],
    @"BMW X6" : [NSDecimalNumber decimalNumberWithString:@"55015.00"]
};
NSArray *sortedKeys = [prices keysSortedByValueUsingComparator:
                       ^NSComparisonResult(id obj1, id obj2) {
                           return [obj2 compare:obj1];
                       }];
NSLog(@"%@", sortedKeys);
```

It's important to note that the dictionary's *values* are passed to the block—not the keys themselves. Alternatively, you can always manually sort the results of `allKeys` and `allValues`.

Filtering Dictionary Keys

Similarly, you can't directly filter a dictionary, but the aptly named `keysOfEntriesPassingTest:` method lets you extract the keys passing a particular test. The test is defined as a block that returns `YES` if the current entry's key should be included and `NO` if it shouldn't. The resulting keys are returned as an `NSSet`. For example, the following snippet finds all of the cars that are under $50,000.

```
NSDictionary *prices = @{
    @"Mercedes-Benz SLK250" : [NSDecimalNumber decimalNumberWithString:@"42900.00"],
    @"Mercedes-Benz E350" : [NSDecimalNumber decimalNumberWithString:@"51000.00"],
    @"BMW M3 Coupe" : [NSDecimalNumber decimalNumberWithString:@"62000.00"],
    @"BMW X6" : [NSDecimalNumber decimalNumberWithString:@"55015.00"]
};
NSDecimalNumber *maximumPrice = [NSDecimalNumber decimalNumberWithString:@"50000.00"];
NSSet *under50k = [prices keysOfEntriesPassingTest:
                   ^BOOL(id key, id obj, BOOL *stop) {
                       if ([obj compare:maximumPrice] == NSOrderedAscending) {
                           return YES;
                       } else {
                           return NO;
                       }
                   }];
NSLog(@"%@", under50k);
```

NSMutableDictionary

The `NSMutableDictionary` class lets you add new key-value pairs dynamically. Mutable dictionaries provide similar performance to mutable sets when it comes to inserting and deleting entries, and remember that both of these are a better choice than mutable arrays if you need to constantly alter the collection.

Mutable collections in general lend themselves to representing system states, and mutable dictionaries are no different. A common use case is to map one set of objects to another set of objects. For example, an auto shop application might need to assign broken cars to specific mechanics. One way of modeling this is to treat cars as keys and mechanics as values. This arrangement allows a single mechanic to be responsible for multiple cars, but not vice versa.

Creating Mutable Dictionaries

Mutable dictionaries can be created by calling any of the factory methods defined by `NSDictionary` on the `NSMutableDictionary` class. But, since many of these methods aren't always the most intuitive to work with, you might find it useful to convert a literal dictionary to a mutable one using `dictionaryWithDictionary:`

```
NSMutableDictionary *jobs = [NSMutableDictionary
                             dictionaryWithDictionary:@{
                                 @"Audi TT" : @"John",
                                 @"Audi Quattro (Black)" : @"Mary",
                                 @"Audi Quattro (Silver)" : @"Bill",
                                 @"Audi A7" : @"Bill"
                             }];
NSLog(@"%@", jobs);
```

Adding and Removing Entries

The `setObject:forKey:` and `removeObjectForKey:` methods are the significant additions contributed by `NSMutableDictionary`. The former can be used to either replace existing keys or add new ones to the collection. As an alternative, you can assign values to keys using the dictionary subscripting syntax, also shown below.

```
NSMutableDictionary *jobs = [NSMutableDictionary
                             dictionaryWithDictionary:@{
                                 @"Audi TT" : @"John",
                                 @"Audi Quattro (Black)" : @"Mary",
                                 @"Audi Quattro (Silver)" : @"Bill",
                                 @"Audi A7" : @"Bill"
                             }];
```

```
// Transfer an existing job to Mary
[jobs setObject:@"Mary" forKey:@"Audi TT"];
// Finish a job
[jobs removeObjectForKey:@"Audi A7"];
// Add a new job
jobs[@"Audi R8 GT"] = @"Jack";
```

Combining Dictionaries

Mutable dictionaries can be expanded by adding the contents of another dictionary to its collection via the `addEntriesFromDictionary:` method. This can be used, for example, to combine jobs from two auto shop locations:

```
NSMutableDictionary *jobs = [NSMutableDictionary
                             dictionaryWithDictionary:@{
                                 @"Audi TT" : @"John",
                                 @"Audi Quattro (Black)" : @"Mary",
                                 @"Audi Quattro (Silver)" : @"Bill",
                                 @"Audi A7" : @"Bill"
                             }];
NSDictionary *bakerStreetJobs = @{
    @"BMW 640i" : @"Dick",
    @"BMW X5" : @"Brad"
};
[jobs addEntriesFromDictionary:bakerStreetJobs];
```

This method also presents another option for creating mutable dictionaries:

```
// Create an empty mutable dictionary
NSMutableDictionary *jobs = [NSMutableDictionary dictionary];
// Populate it with initial key-value pairs
[jobs addEntriesFromDictionary:@{
    @"Audi TT" : @"John",
    @"Audi Quattro (Black)" : @"Mary",
    @"Audi Quattro (Silver)" : @"Bill",
    @"Audi A7" : @"Bill"
}];
```

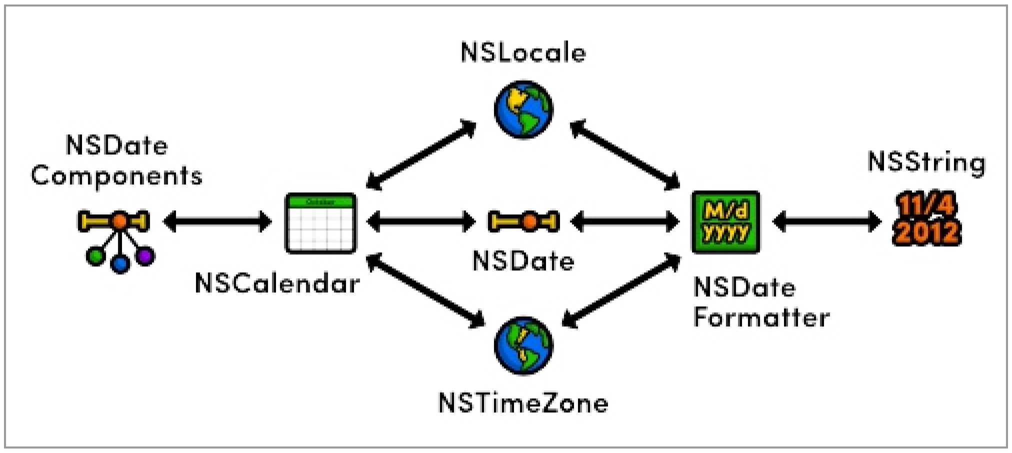
Enumeration Considerations

Dictionaries should not be mutated while you're iterating over them. Please see the Enumeration Considerations section of the *NSSet* module for details.

Depending on how you need to alter the collection, you might be able to use the `allKeys` or `allValues` methods to create snapshots of just the keys or values. If those aren't sufficient you should use the `dictionaryWithDictionary:` class method to create a shallow copy of the entire dictionary.

Date Programming

Date programming is a complex topic in any language, and Objective-C is no different. The Foundation Framework provides extensive support for working with dates in various calendrical systems. Unfortunately, the associated abstractions and dependencies give rise to a plethora of date-related classes that can be hard to navigate if you don't know where to start.



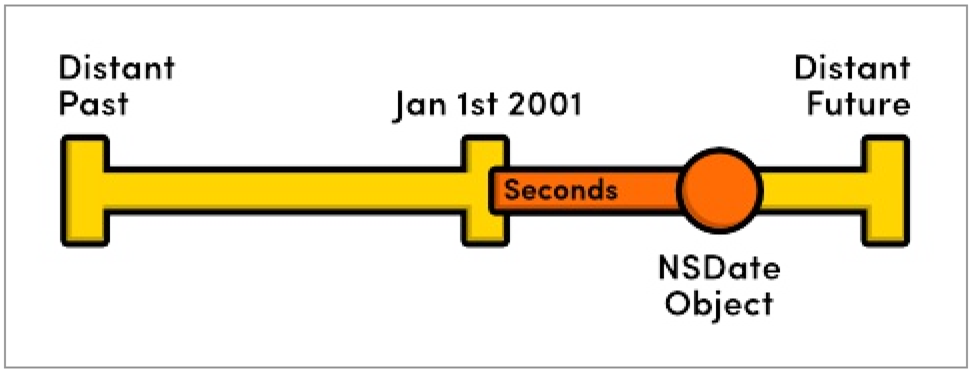
The essential classes for representing and manipulating dates

The above diagram provides a high-level overview of Objective-C's date-handling capabilities. The `NSDate` class represents a specific point in time, which can be reduced to `NSDateComponent`'s (e.g., days, weeks, years) by interpreting it in the context of an `NSCalendar` object. The `NSDateFormatter` class provides a human-readable version of an `NSDate`, and `NSLocale` and `NSTimeZone` encapsulate essential localization information for many calendrical operations.

This module explains all of this in much more detail using several hands-on examples. By the end of this module, you should be able to perform any kind of date operation that you'll ever need.

NSDate

An `NSDate` object represents a specific point in time, independent of any particular calendrical system, time zone, or locale. Internally, it just records the number of seconds from an arbitrary reference point (January 1st, 2001 GMT). For a date to be useful, it generally needs to be interpreted in the context of an `NSCalendar` object.



`NSDate` recording the number of seconds from its reference point

You'll typically want to create date objects using a calendar or a date formatter, as the `NSDate` class only provides a few low-level methods for creating dates from scratch. The `date` method returns an object representing the current date and time, and `dateWithTimeInterval:sinceDate:` generates a relative date using an `NSTimeInterval`, which is a `double` storing the number of seconds from the specified date. As you can see in the following example, a positive interval goes forward in time, and a negative one goes backwards.

```
NSDate *now = [NSDate date];
NSTimeInterval secondsInWeek = 7 * 24 * 60 * 60;
NSDate *lastWeek = [NSDate dateWithTimeInterval:-secondsInWeek
                                      sinceDate:now];
NSDate *nextWeek = [NSDate dateWithTimeInterval:secondsInWeek
                                      sinceDate:now];
NSLog(@"Last Week: %@", lastWeek);
NSLog(@"Right Now: %@", now);
NSLog(@"Next Week: %@", nextWeek);
```

This will output three datetime strings that look something like `2012-11-06 02:24:00 +0000`. They contain the date (expressed as year-month-date), the time of day (expressed as hours:minutes:seconds) and the time zone (expressed as an offset from Greenwich Mean Time (GMT)).

Aside from capturing an absolute point in time, the only real job of `NSDate` is to facilitate comparisons. It defines `isEqualToDate:` and `compare:` methods, which work just like the ones provided by `NSNumber`. In addition, the `earlierDate:` and `laterDate:` methods can be used as a convenient shortcut.

```
NSComparisonResult result = [now compare:nextWeek];
if (result == NSOrderedAscending) {
    NSLog(@"now < nextWeek");
} else if (result == NSOrderedSame) {
    NSLog(@"now == nextWeek");
} else if (result == NSOrderedDescending) {
    NSLog(@"now > nextWeek");
}
NSDate *earlierDate = [now earlierDate:lastWeek];
NSDate *laterDate = [now laterDate:lastWeek];
NSLog(@"%@ is earlier than %@", earlierDate, laterDate);
```

NSDateComponents

The `NSDateComponents` class is a simple data structure for representing the various time periods used by a calendrical system (days, weeks, months, years, etc). Unlike `NSDate`, the meaning of these components are entirely dependent on how it's used.

Consider an `NSDateComponents` object with its `year` property set to `2012`. This could be interpreted as the year `2012` by a Gregorian calendar, the year `1469` by a Buddhist calendar, or two thousand and twelve years relative to some other date.

We'll work more with date components in the next section, but as a simple example, here's how you would create an `NSDateComponents` instance that could be interpreted as November 4th, 2012:

```
NSDateComponents *november4th2012 = [[NSDateComponents alloc] init];
[november4th2012 setYear:2012];
[november4th2012 setMonth:11];
[november4th2012 setDay:4];
NSLog(@"%@", november4th2012);
```

Again, it's important to understand that these components don't *actually* represent November 4th, 2012 until it is interpreted by a Gregorian calendar object as such.

The complete list of component fields can be found below. Note that it's perfectly legal to set only the properties you need and leave the rest as `NSUndefinedDateComponent`, which is the default value for all fields.

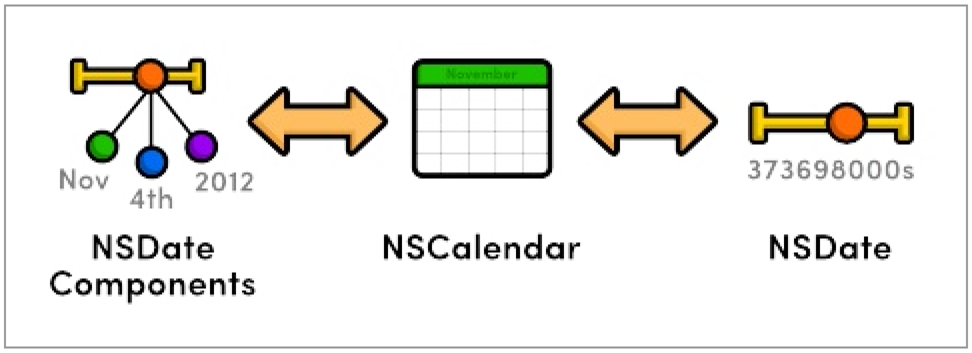
era	week
year	weekday
month	weekdayOrdinal
day	quarter
hour	weekOfMonth
minute	weekOfYear

`second`	`yearForWeekOfYear`

NSCalendar

A calendrical system is a way of breaking down time into manageable units like years, months, weeks, etc. These units are represented as `NSDateComponents` objects; however, not all systems use the same units or interpret them in the same way. So, an `NSCalendar` object is required to give meaning to these components by defining the exact length of a year/month/week/etc.

This provides the necessary context for translating absolute `NSDate` instances into `NSDateComponents`. This is a very important ability, as it lets you work with dates on an intuitive, cultural level instead of a mathematical one. That is, it's much easier to say, "November 4th, 2012" than "373698000 seconds after January 1st, 2001."



Using `NSCalendar` to convert between dates and date components

This section explains how to create different types of calendars, then covers the three main responsibilities of `NSCalendar`: converting `NSDate` objects to components, creating `NSDate` objects from components, and performing calendrical calculations. We'll also take a brief look at `NSCalendarUnit`.

Creating Calendars

`NSCalendar`'s `initWithCalendarIdentifier:` initializer accepts an identifier that defines the calendrical system to use. In addition, the `currentCalendar` class method returns the user's preferred calendar. For most applications, you should opt for `currentCalendar` instead of manually defining one, since it reflects the user's device settings.

```
NSCalendar *gregorian = [[NSCalendar alloc]
                    initWithCalendarIdentifier:NSGregorianCalendar];
NSCalendar *buddhist = [[NSCalendar alloc]
                    initWithCalendarIdentifier:NSBuddhistCalendar];
NSCalendar *preferred = [NSCalendar currentCalendar];
NSLog(@"%@", gregorian.calendarIdentifier);
NSLog(@"%@", buddhist.calendarIdentifier);
NSLog(@"%@", preferred.calendarIdentifier);
```

The `calendarIdentifier` method lets you display the string-representation of the calendar ID, but it is read-only (you can’t change the calendrical system after instantiation). The rest of Cocoa’s built-in calendar identifiers can be accessed via the following constants:

```
NSGregorianCalendar  NSBuddhistCalendar
NSChineseCalendar    NSHebrewCalendar
NSIslamicCalendar    NSIslamicCivilCalendar
NSJapaneseCalendar   NSRepublicOfChinaCalendar
NSPersianCalendar    NSIndianCalendar
NSISO8601Calendar
```

The next few sections walk through the basic usage of `NSCalendar`. If you change the identifier, you can see how different calendars have different interpretations of date components.

From Dates to Components

The following example shows you how to convert an `NSDate` into a culturally significant `NSDateComponents` object.

```
NSDate *now = [NSDate date];
NSCalendar *calendar = [[NSCalendar alloc]
                        initWithCalendarIdentifier:NSGregorianCalendar];
NSCalendarUnit units = NSYearCalendarUnit | NSMonthCalendarUnit | NSDayCalendarUnit;
NSDateComponents *components = [calendar components:units fromDate:now];

NSLog(@"Day: %ld", [components day]);
NSLog(@"Month: %ld", [components month]);
NSLog(@"Year: %ld", [components year]);
```

First, this creates an `NSCalendar` object that represents a Gregorian calendar. Then, it creates a bitmask defining the units to include in the conversion (see `NSCalendarUnits` for details). Finally, it passes these units and an `NSDate` to the `components:fromDate:` method.

From Components to Dates

A calendar can also convert in the other direction, which offers a much more intuitive way to create `NSDate` objects. It lets you define a date using the components of your native calendrical system:

```
NSCalendar *calendar = [[NSCalendar alloc]
```

```
                    initWithCalendarIdentifier:NSGregorianCalendar];
NSDateComponents *components = [[NSDateComponents alloc] init];
[components setYear:2012];
[components setMonth:11];
[components setDay:4];

NSDate *november4th2012 = [calendar dateFromComponents:components];
NSLog(@"%0.0f seconds between Jan 1st, 2001 and Nov 4th, 2012",
          [november4th2012 timeIntervalSinceReferenceDate]);
```

It's not necessary to define the units to include in the conversion, so this is a little more straightforward than translating dates to components. Simply instantiate an `NSDateComponents` object, populate it with the desired components, and pass it to the `dateFromComponents:` method.

Remember that `NSDate` represents an absolute point in time, independent of any particular calendrical system. So, by calling `dateFromComponents:` on different `NSCalendar` objects, you can reliably compare dates from incompatible systems.

Calendrical Calculations

The third job of `NSCalendar` is to provide a high-level API for date-related calculations. First, we'll take a look at how calendars let you add components to a given `NSDate` instance. The following example generates a date object that is exactly one month away from the current date.

```
NSDate *now = [NSDate date];
NSCalendar *calendar = [[NSCalendar alloc]
                    initWithCalendarIdentifier:NSGregorianCalendar];
NSDateComponents *components = [[NSDateComponents alloc] init];
[components setMonth:1];
NSDate *oneMonthFromNow = [calendar dateByAddingComponents:components
                                                    toDate:now
                                                   options:0];
NSLog(@"%@", oneMonthFromNow);
```

All you have to do is create an `NSDateComponents` object, record the components you want to add, and pass it to `dateByAddingComponents:toDate:options:`. This is a good example of how the meaning of date components is entirely dependent on how it's used. Instead of representing a date, the above components represent a *duration*.

The `options` argument should either be `0` to let components overflow into higher units or

NSWrapCalendarComponents to prevent this behavior. For example, if you set month to 14, passing 0 as an option tells the calendar to interpret it as 1 year and 2 months, whereas NSWrapCalendarComponents interprets it as 2 months and completely ignores the extra year.

Also note that dateByAddingComponents:toDate:options: is a calendar-aware operation, so it compensates for 30 vs. 31 day months (in the Gregorian calendar). This makes NSCalendar a more robust way of adding dates than NSDate's low-level dateWithTimeInterval:sinceDate: method.

The other major calendrical operation provided by NSCalendar is components:fromDate:toDate:options:, which calculates the interval between two NSDate objects. For example, you can determine the number of weeks since NSDate's internal reference point as follows:

```
NSDate *start = [NSDate dateWithTimeIntervalSinceReferenceDate:0];
NSDate *end = [NSDate date];
NSCalendar *calendar = [NSCalendar currentCalendar];
NSCalendarUnit units = NSWeekCalendarUnit;
NSDateComponents *components = [calendar components:units
                                           fromDate:start
                                             toDate:end
                                            options:0];
NSLog(@"It has been %ld weeks since January 1st, 2001",
      [components week]);
```

NSCalendarUnit

The first argument of components:fromDate: and components:fromDate:toDate:options: determine the properties that will be populated on the resulting NSDateComponents object. The possible values are defined by NSCalendarUnit, which enumerates the following constants:

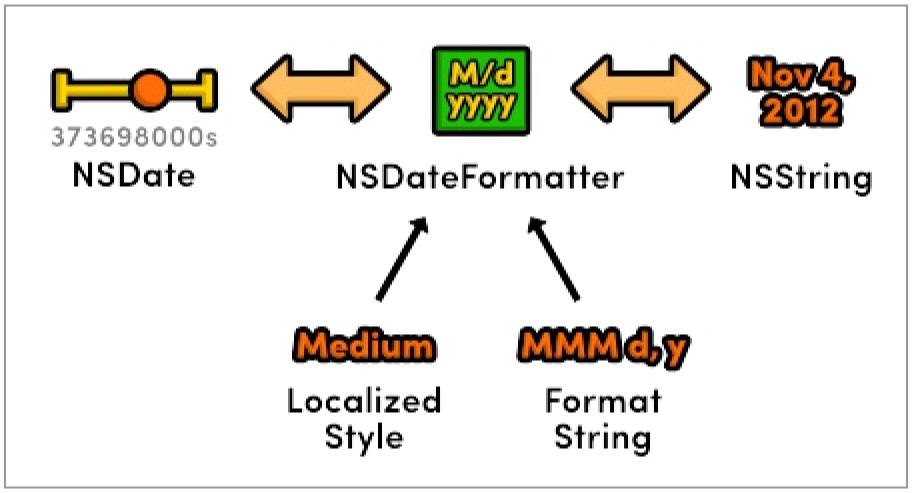
```
NSEraCalendarUnit      NSWeekdayCalendarUnit
NSYearCalendarUnit     NSWeekdayOrdinalCalendarUnit
NSMonthCalendarUnit    NSQuarterCalendarUnit
NSDayCalendarUnit      NSWeekOfMonthCalendarUnit
NSHourCalendarUnit     NSWeekOfYearCalendarUnit
NSMinuteCalendarUnit   NSYearForWeekOfYearCalendarUnit
NSSecondCalendarUnit   NSCalendarCalendarUnit
NSWeekCalendarUnit     NSTimeZoneCalendarUnit
```

When you need more than one of these units, you should combine them into a bitmask using the bitwise OR operator (|). For example, to include the day, hour, and minute, you would use the following value for the first parameter of components:fromDate:.

```
NSDayCalendarUnit | NSHourCalendarUnit | NSMinuteCalendarUnit
```

NSDateFormatter

The `NSDateFormatter` class makes it easy to work with the human-readable form of a date. Whereas calendars decompose a date into an `NSDateComponents` object, date formatters convert between `NSDate`'s and `NSString`'s.



Using a `NSDateFormatter` to convert between dates and strings

There are two ways to use a date formatter: 1) with localized styles or 2) with a custom format string. The former method is a better choice if you're displaying content to users, since it incorporates their preferences and device settings. The latter is useful when you need to know *exactly* what the resulting string representations will look like.

Localized Styles

Localized styles define how a date should look using abstract descriptions instead of specific date components. For example, instead of defining a date as `MM/dd/yy`, a style would say it should be displayed in its "short" form. This lets the system adapt the representation to the user's language, region, and preferences while still offering a level of control to the developer.

The following snippet formats a date using the "short" style. As you can see, separate styles are used for the date and the time. The `stringFromDate:` method formats an `NSDate` object according to the provided styles.

```
NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
[formatter setDateStyle:NSDateFormatterShortStyle];
[formatter setTimeStyle:NSDateFormatterShortStyle];
```

```
NSDate *now = [NSDate date];
NSString *prettyDate = [formatter stringFromDate:now];
NSLog(@"%@", prettyDate);
```

I'm an English speaker from the United States, so this will output `11/4/2012 8:09 PM`. If you have different default settings, you'll see the traditional date formatting used in your particular language/region. We'll take a closer look at this behavior in the NSLocale section. The complete list of formatting styles are included below.

```
NSDateFormatterNoStyle
NSDateFormatterShortStyle
NSDateFormatterMediumStyle
NSDateFormatterLongStyle
NSDateFormatterFullStyle
```

Note that the `NSDateFormatterNoStyle` constant can be used to omit the date or time from the output string.

Custom Format Strings

Again, the styles discussed above are the preferred way to define user-visible dates. However, if you're working with dates on a programmatic level, you may find the `setDateFormat:` method useful. This accepts a format string that defines precisely how the date will appear. For example, you can use `M.d.y` to output the month, date, and year separated by periods as follows:

```
// Formatter configuration
NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
NSLocale *posix = [[NSLocale alloc] initWithLocaleIdentifier:@"en_US_POSIX"];
[formatter setLocale:posix];
[formatter setDateFormat:@"M.d.y"];
// Date to string
NSDate *now = [NSDate date];
NSString *prettyDate = [formatter stringFromDate:now];
NSLog(@"%@", prettyDate);
```

As a best practice, you should always set the `locale` property of the `NSDateFormatter` before using custom format strings. This ensures that the user's default locale won't affect the output, which would result in subtle, hard-to-reproduce bugs. In the above case, the POSIX locale ensures that the date is displayed as `11.4.2012`, regardless of the user's settings.

Custom date formats also present an alternative way to create date objects. Using the above configuration, it's possible to convert the string `11.4.2012` into an `NSDate` with the `dateFromString:` method. If the string can't be parsed, it will return `nil`.

```
// String to date
NSString *dateString = @"11.4.2012";
NSDate *november4th2012 = [formatter dateFromString:dateString];
NSLog(@"%@", november4th2012);
```

In addition to `M`, `d`, and `y`, the Unicode Technical Standard #35 defines a plethora of other date format specifiers. This document contains a lot of information not really necessary to use `NSDateFormatter` correctly, so you may find the following list of sample format strings to be a more practical quick-reference. The date used to generate the output string is November 4th, 2012 8:09 PM Central Standard Time.

Format String	**Output String**
`M/d/y`	`11/4/2012`
`MM/dd/yy`	`11/04/12`
`MMM d, ''yy`	`Nov 4, '12`
`MMMM`	`November`
`E`	`Sun`
`EEEE`	`Sunday`
`'Week' w 'of 52'`	`Week 45 of 52`
`'Day' D 'of 365'`	`Day 309 of 365`
`QQQ`	`Q4`
`QQQQ`	`4th quarter`
`m 'minutes past' h`	`9 minutes past 8`
`h:mm a`	`8:09 PM`
`HH:mm:ss's'`	`20:09:00s`
`HH:mm:ss:SS`	`20:09:00:00`
`h:mm a zz`	`8:09 PM CST`
`h:mm a zzzz`	`8:09 PM Central Standard Time`
`yyyy-MM-dd HH:mm:ss Z`	`2012-11-04 20:09:00 -0600`

Notice that literal text needs to be wrapped in single quotes to prevent it from being parsed by the formatter.

NSLocale

An `NSLocale` object represents a set of conventions for a particular language, region, or culture. Both `NSCalendar` and `NSDateFormatter` rely on this information to localize many of their core operations.

The previous section already showed you how to create a custom locale via the `initWithLocaleIdentifier:` initializer. Let's take a look at the impact a locale change can have on an `NSDateFormatter` by running the same example using Egyptian Arabic instead of POSIX:

```
NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
NSLocale *egyptianArabic = [[NSLocale alloc] initWithLocaleIdentifier:@"ar_EG"];
[formatter setLocale:egyptianArabic];
[formatter setDateFormat:@"M.d.y"];

NSDate *now = [NSDate date];
NSString *prettyDate = [formatter stringFromDate:now];
NSLog(@"%@", prettyDate);
```

Instead of `11.4.2012`, the date is now displayed with Arabic digits: ١١.٤.٢٠١٢. This is why setting the locale before using custom format strings is so important—if you don't, `NSDateFormatter` will use the system default, and your format strings will return unexpected results for international users.

Locale identifiers are composed of a language abbreviation followed by a country abbreviation. In the above example, `ar` stands for Arabic, and `EG` represents the Egyptian dialect. You can obtain a complete list of available locale identifiers with the following (it's a very long list):

```
NSLog(@"%@", [NSLocale availableLocaleIdentifiers]);
```

But, whenever possible, you'll want to stick with the user's preferred locale. This makes sure that your app conforms to their expectations (you don't want to force an Arabic speaker to read English). You can access the preferred locale through the `currentLocale` class method, as shown below. Note that this is the default used by `NSCalendar` and `NSDateFormatter`.

```
NSLocale *preferredLocale = [NSLocale currentLocale]);
```

Custom `NSLocale` objects are usually only required for specialized applications or testing purposes, but it's still good to understand how iOS and OS X applications automatically translate your data for an international audience.

NSTimeZone

It's very important to understand that a string like `11.4.2012 8:09 PM` does *not* specify a precise point in time—8:09 PM in Chicago occurs 8 hours after 8:09 PM in Cairo. The time zone is a required piece of information for creating an absolute `NSDate` instance. Accordingly, `NSCalendar` and `NSDateFormatter` both require an `NSTimeZone` object to offset their calculations appropriately.

A time zone can be created explicitly with either its full name or its abbreviated one. The following example demonstrates both methods and shows how a time zone can alter the `NSDate` produced by a formatter. The generated dates are displayed in Greenwich Mean Time, and you can see that 8:09 PM in Chicago is indeed 8 hours after 8:09 PM in Cairo.

```
NSTimeZone *centralStandardTime = [NSTimeZone timeZoneWithAbbreviation:@"CST"];
NSTimeZone *cairoTime = [NSTimeZone timeZoneWithName:@"Africa/Cairo"];

NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
NSLocale *posix = [[NSLocale alloc] initWithLocaleIdentifier:@"en_US_POSIX"];
[formatter setLocale:posix];
[formatter setDateFormat:@"M.d.y h:mm a"];
NSString *dateString = @"11.4.2012 8:09 PM";

[formatter setTimeZone:centralStandardTime];
NSDate *eightPMInChicago = [formatter dateFromString:dateString];
NSLog(@"%@", eightPMInChicago);      // 2012-11-05 02:09:00 +0000

[formatter setTimeZone:cairoTime];
NSDate *eightPMInCairo = [formatter dateFromString:dateString];
NSLog(@"%@", eightPMInCairo);        // 2012-11-04 18:09:00 +0000
```

Note that the `NSDateFormatter`'s `timeZone` property is a fallback, so format strings that contain one of the `z` specifiers will use the parsed value as expected. You can use `NSCalendar`'s `timeZone` property to the same effect.

Complete lists of time zone names and abbreviations can be accessed via `knownTimeZoneNames` and `abbreviationDictionary`, respectively. However, the latter does not necessarily provide values for *every* time zone.

```
NSLog(@"%@", [NSTimeZone knownTimeZoneNames]);
NSLog(@"%@", [NSTimeZone abbreviationDictionary]);
```

All users have a preferred time zone, whether it's configured explicitly or determined automatically based on their region. The `localTimeZone` class method is the best option for accessing the current time zone.

```
NSTimeZone *preferredTimeZone = [NSTimeZone localTimeZone];
```

As with locales, it's generally a better idea to stick with the default time zone. Custom `NSTimeZone` objects are typically only necessary for scheduling applications where time zones need to be explicitly associated with individual events.

Conclusion

We hope that you've enjoyed *Ry's Objective-C Tutorial*. Our goal was to walk you through the language behind iOS and OS X application development, and if you've been following along from the beginning of this tutorial, you should be feeling pretty comfortable with classes, properties, methods, protocols, categories, blocks, and the major data types in both C and Objective-C.

The next step is to take your fantastic Objective-C foundation and combine it with the Cocoa/Cocoa Touch frameworks to build some awesome OS X and iOS apps. Apple's Start Developing iOS Apps Today guide is a great jumping-off point for new iOS developers, but if you're interested in game development, be sure to check back here in March 2013 for *Ry's Cocos2d Tutorial*.

Please feel free to contact us with any questions or comments about this tutorial or Objective-C in general. If you've created an app using the skills you learned here, we'd love to hear about that, too.