Values and Collections

Although Objective-C is an object-oriented programming language, it is a superset of C, which means you can use any of the standard C *scalar* (non-object) types like `int`, `float` and `char` in Objective-C code. There are also additional scalar types available in Cocoa and Cocoa Touch applications, such as `NSInteger`, `NSUInteger` and `CGFloat`, which have different definitions depending on the target architecture.

Scalar types are used in situations where you just don't need the benefits (or associated overheads) of using an object to represent a value. While strings of characters are usually represented as instances of the `NSString` class, numeric values are often stored in scalar local variables or properties.

It's possible to declare a C-style array in Objective-C, but you'll find that collections in Cocoa and Cocoa Touch applications are usually represented using instances of classes like `NSArray` or `NSDictionary`. These classes can only be used to collect Objective-C objects, which means that you'll need to create instances of classes like `NSValue`, `NSNumber` or `NSString` in order to represent values before you can add them to a collection.

The previous chapters in this guide make frequent use of the `NSString` class and its initialization and class factory methods, as well as the Objective-C `@"string"` literal, which offers a concise syntax to create an `NSString` instance. This chapter explains how to create `NSValue` and `NSNumber` objects, using either method calls or through Objective-C value literal syntax.

Basic C Primitive Types Are Available in Objective-C

Each of the standard C scalar variable types is available in Objective-C:

```
int someInteger = 42;
float someFloatingPointNumber = 3.1415;
double someDoublePrecisionFloatingPointNumber = 6.02214199e23;
```

as well as the standard C operators:

```
int someInteger = 42;
someInteger++;            // someInteger == 43

int anotherInteger = 64;
anotherInteger--;         // anotherInteger == 63

anotherInteger *= 2;      // anotherInteger == 126
```

If you use a scalar type for an Objective-C property, like this:

```
@interface XYZCalculator : NSObject
@property double currentValue;
@end
```

it's also possible to use C operators on the property when accessing the value via dot syntax, like this:

```
@implementation XYZCalculator
- (void)increment {
    self.currentValue++;
}
- (void)decrement {
```

```
    self.currentValue--;
}
- (void)multiplyBy:(double)factor {
    self.currentValue *= factor;
}
@end
```

Dot syntax is purely a syntactic wrapper around accessor method calls, so each of the operations in this example is equivalent to first using the get accessor method to get the value, then performing the operation, then using the set accessor method to set the value to the result.

Objective-C Defines Additional Primitive Types

The `BOOL` scalar type is defined in Objective-C to hold a Boolean value, which is either `YES` or `NO`. As you might expect, `YES` is logically equivalent to `true` and `1`, while `NO` is equivalent to `false` and `0`.

Many parameters to methods on Cocoa and Cocoa Touch objects also use special scalar numeric types, such as `NSInteger` or `CGFloat`.

For example, the `NSTableViewDataSource` and `UITableViewDataSource` protocols (described in the previous chapter) both have methods requesting the number of rows to display:

```
@protocol NSTableViewDataSource <NSObject>
- (NSInteger)numberOfRowsInTableView:(NSTableView *)tableView;
...
@end
```

These types, like `NSInteger` and `NSUInteger`, are defined differently depending on the target architecture. When building for a 32-bit environment (such as for iOS), they are 32-bit signed and unsigned integers respectively; when building for a 64-bit environment (such as for the modern OS X runtime) they are 64-bit signed and unsigned integers respectively.

It's best practice to use these platform-specific types if you might be passing values across API boundaries (both internal and exported APIs), such as arguments or return values in method or function calls between your application code and a framework.

For local variables, such as a counter in a loop, it's fine to use the basic C types if you know that the value is within the standard limits.

C Structures Can Hold Primitive Values

Some Cocoa and Cocoa Touch API use C structures to hold their values. As an example, it's possible to ask a string object for the range of a substring, like this:

```
    NSString *mainString = @"This is a long string";
    NSRange substringRange = [mainString rangeOfString:@"long"];
```

An `NSRange` structure holds a location and a length. In this case, `substringRange` will hold a range of `{10,4}`—the "`l`" at the start of `@"long"` is the character at zero-based index `10` in `mainString`, and `@"long"` is `4` characters in length.

Similarly, if you need to write custom drawing code, you'll need to interact with Quartz, which requires structures based around the `CGFloat` data type, like `NSPoint` and `NSSize` on OS X and `CGPoint` and `CGSize` on iOS. Again, `CGFloat` is defined differently depending on the target architecture.

For more information on the Quartz 2D drawing engine, see *Quartz 2D Programming Guide*.

Objects Can Represent Primitive Values

If you need to represent a scalar value as an object, such as when working with the collection classes described in the next section, you can use one of the basic value classes provided by Cocoa and Cocoa Touch.

Strings Are Represented by Instances of the NSString Class

As you've seen in the previous chapters, `NSString` is used to represent a string of characters, like `Hello World`. There are various ways to create `NSString` objects, including standard allocation and initialization, class factory methods or literal syntax:

```
NSString *firstString = [[NSString alloc] initWithCString:"Hello World!"
                                                  encoding:NSUTF8StringEncoding];
NSString *secondString = [NSString stringWithCString:"Hello World!"
                                              encoding:NSUTF8StringEncoding];
NSString *thirdString = @"Hello World!";
```

Each of these examples effectively accomplishes the same thing—creating a string object that represents the provided characters.

The basic `NSString` class is immutable, which means its contents are set at creation and cannot later be changed. If you need to represent a different string, you must create a new string object, like this:

```
NSString *name = @"John";
name = [name stringByAppendingString:@"ny"];    // returns a new string object
```

The `NSMutableString` class is the mutable subclass of `NSString`, and allows you to change its character contents at runtime using methods like `appendString:` or `appendFormat:`, like this:

```
NSMutableString *name = [NSMutableString stringWithString:@"John"];
[name appendString:@"ny"];   // same object, but now represents "Johnny"
```

Format Strings Are Used to Build Strings from Other Objects or Values

If you need to build a string containing variable values, you need to work with a **format string**. This allows you to use format specifiers to indicate how the values are inserted:

```
int magicNumber = ...
NSString *magicString = [NSString stringWithFormat:@"The magic number is %i",
magicNumber];
```

The available format specifiers are described in String Format Specifiers. For more information about strings in general, see the *String Programming Guide*.

Numbers Are Represented by Instances of the NSNumber Class

The `NSNumber` class is used to represent any of the basic C scalar types, including `char`, `double`, `float`, `int`, `long`, `short`, and the `unsigned` variants of each, as well as the Objective-C Boolean type, `BOOL`.

As with `NSString`, you have a variety of options to create `NSNumber` instances, including allocation and initialization or the class factory methods:

```
NSNumber *magicNumber = [[NSNumber alloc] initWithInt:42];
NSNumber *unsignedNumber = [[NSNumber alloc] initWithUnsignedInt:42u];
NSNumber *longNumber = [[NSNumber alloc] initWithLong:42l];

NSNumber *boolNumber = [[NSNumber alloc] initWithBOOL:YES];
```

```
NSNumber *simpleFloat = [NSNumber numberWithFloat:3.14f];
NSNumber *betterDouble = [NSNumber numberWithDouble:3.1415926535];

NSNumber *someChar = [NSNumber numberWithChar:'T'];
```

It's also possible to create `NSNumber` instances using Objective-C literal syntax:

```
NSNumber *magicNumber = @42;
NSNumber *unsignedNumber = @42u;
NSNumber *longNumber = @42l;

NSNumber *boolNumber = @YES;

NSNumber *simpleFloat = @3.14f;
NSNumber *betterDouble = @3.1415926535;

NSNumber *someChar = @'T';
```

These examples are equivalent to using the `NSNumber` class factory methods.

Once you've created an `NSNumber` instance it's possible to request the scalar value using one of the accessor methods:

```
int scalarMagic = [magicNumber intValue];
unsigned int scalarUnsigned = [unsignedNumber unsignedIntValue];
long scalarLong = [longNumber longValue];

BOOL scalarBool = [boolNumber boolValue];

float scalarSimpleFloat = [simpleFloat floatValue];
double scalarBetterDouble = [betterDouble doubleValue];

char scalarChar = [someChar charValue];
```

The `NSNumber` class also offers methods to work with the additional Objective-C primitive types. If you need to create an object representation of the scalar `NSInteger` and `NSUInteger` types, for example, make sure you use the correct methods:

```
NSInteger anInteger = 64;
NSUInteger anUnsignedInteger = 100;

NSNumber *firstInteger = [[NSNumber alloc] initWithInteger:anInteger];
NSNumber *secondInteger = [NSNumber numberWithUnsignedInteger:anUnsignedInteger];

NSInteger integerCheck = [firstInteger integerValue];
NSUInteger unsignedCheck = [secondInteger unsignedIntegerValue];
```

All `NSNumber` instances are immutable, and there is no mutable subclass; if you need a different number, simply use another `NSNumber` instance.

> **Note:** `NSNumber` is actually a class cluster. This means that when you create an instance at runtime, you'll get a suitable concrete subclass to hold the provided value. Just treat the created object as an instance of `NSNumber`.

Represent Other Values Using Instances of the NSValue Class

The `NSNumber` class is itself a subclass of the basic `NSValue` class, which provides an object wrapper around a single value or data item. In addition to the basic C scalar types, `NSValue` can also be used to represent pointers and structures.

The `NSValue` class offers various factory methods to create a value with a given standard structure, which makes it easy to create an instance to represent, for example, an `NSRange`, like the example from earlier in the chapter:

```
    NSString *mainString = @"This is a long string";
    NSRange substringRange = [mainString rangeOfString:@"long"];
    NSValue *rangeValue = [NSValue valueWithRange:substringRange];
```

It's also possible to create `NSValue` objects to represent custom structures. If you have a particular need to use a C structure (rather than an Objective-C object) to store information, like this:

```
typedef struct {
    int i;
    float f;
} MyIntegerFloatStruct;
```

you can create an `NSValue` instance by providing a pointer to the structure as well as an encoded Objective-C type. The `@encode()` compiler directive is used to create the correct Objective-C type, like this:

```
    struct MyIntegerFloatStruct aStruct;
    aStruct.i = 42;
    aStruct.f = 3.14;

    NSValue *structValue = [NSValue value:&aStruct
                             withObjCType:@encode(MyIntegerFloatStruct)];
```

The standard C reference operator (`&`) is used to provide the address of `aStruct` for the `value` parameter.

Most Collections Are Objects

Although it's possible to use a C array to hold a collection of scalar values, or even object pointers, most collections in Objective-C code are instances of one of the Cocoa and Cocoa Touch collection classes, like `NSArray`, `NSSet` and `NSDictionary`.

These classes are used to manage groups of objects, which means any item you wish to add to a collection must be an instance of an Objective-C class. If you need to add a scalar value, you must first create a suitable `NSNumber` or `NSValue` instance to represent it.

Rather than somehow maintaining a separate copy of each collected object, the collection classes use strong references to keep track of their contents. This means that any object that you add to a collection will be kept alive at least as long as the collection is kept alive, as described in Manage the Object Graph through Ownership and Responsibility.

In addition to keeping track of their contents, each of the Cocoa and Cocoa Touch collection classes make it easy to perform certain tasks, such as enumeration, accessing specific items, or finding out whether a particular object is part of the collection.

The basic `NSArray`, `NSSet` and `NSDictionary` classes are immutable, which means their contents are set at creation. Each also has a mutable subclass to allow you to add or remove objects at will.

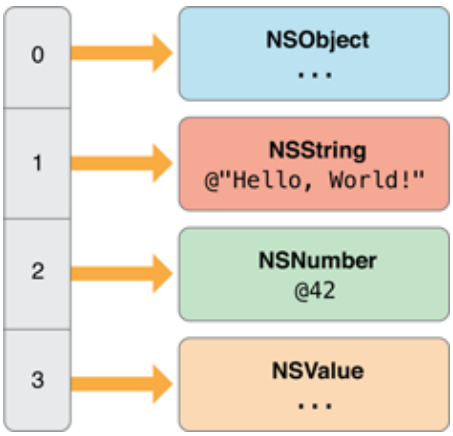
For more information on the different collection classes available in Cocoa and Cocoa Touch, see *Collections Programming Topics*.

Arrays Are Ordered Collections

An `NSArray` is used to represent an *ordered collection* of objects. The only requirement is that each item is an Objective-C object— there's no requirement for each object to be an instance of the same class.

To maintain order in the array, each element is stored at a zero-based index, as shown in Figure 6-1.

Figure 6-1 An Array of Objective-C Objects



Creating Arrays

As with the value classes described earlier in this chapter, you can create an array through allocation and initialization, class factory methods, or literal syntax.

There are a variety of different initialization and factory methods available, depending on the number of objects:

```
+ (id)arrayWithObject:(id)anObject;
+ (id)arrayWithObjects:(id)firstObject, ...;
- (id)initWithObjects:(id)firstObject, ...;
```

The `arrayWithObjects:` and `initWithObjects:` methods both take a nil-terminated, variable number of arguments, which means that you must include `nil` as the last value, like this:

```
    NSArray *someArray =
  [NSArray arrayWithObjects:someObject, someString, someNumber, someValue, nil];
```

This example creates an array like the one shown earlier, in Figure 6-1. The first object, `someObject`, will have an array index of `0`; the last object, `someValue`, will have an index of `3`.

It's possible to truncate the list of items unintentionally if one of the provided values is `nil`, like this:

```
    id firstObject = @"someString";
    id secondObject = nil;
    id thirdObject = @"anotherString";
    NSArray *someArray =
  [NSArray arrayWithObjects:firstObject, secondObject, thirdObject, nil];
```

In this case, `someArray` will contain only `firstObject`, because the `nil secondObject` would be interpreted as the end of the list of items.

Literal Syntax

It's also possible to create an array using an Objective-C literal, like this:

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

You should not terminate the list of objects with `nil` when using this literal syntax, and in fact `nil` is an invalid value. You'll get an exception at runtime if you try to execute the following code, for example:

```
id firstObject = @"someString";
id secondObject = nil;
NSArray *someArray = @[firstObject, secondObject];
// exception: "attempt to insert nil object"
```

If you do need to represent a `nil` value in one of the collection classes, you should use the `NSNull` singleton class, as described in Represent nil with NSNull.

Querying Array Objects

Once you've created an array, you can query it for information like the number of objects, or whether it contains a given item:

```
NSUInteger numberOfItems = [someArray count];

if ([someArray containsObject:someString]) {
    ...
}
```

You can also query the array for an item at a given index. You'll get an out-of-bounds exception at runtime if you attempt to request an invalid index, so you should always check the number of items first:

```
if ([someArray count] > 0) {
    NSLog(@"First item is: %@", [someArray objectAtIndex:0]);
}
```

This example checks whether the number of items is greater than zero. If so, it logs a description of the first item, which has an index of zero.

Subscripting

There's also a subscript syntax alternative to using `objectAtIndex:`, which is just like accessing a value in a standard C array. The previous example could be re-written like this:

```
if ([someArray count] > 0) {
    NSLog(@"First item is: %@", someArray[0]);
}
```

Sorting Array Objects

The `NSArray` class also offers a variety of methods to sort its collected objects. Because `NSArray` is immutable, each of these methods returns a new array containing the items in the sorted order.

As an example, you can sort an array of strings by the result of calling `compare:` on each string, like this:

```
NSArray *unsortedStrings = @[@"gammaString", @"alphaString", @"betaString"];
NSArray *sortedStrings =
            [unsortedStrings sortedArrayUsingSelector:@selector(compare:)];
```

Mutability

Although the `NSArray` class itself is immutable, this has no bearing on any collected objects. If you add a mutable string to an immutable array, for example, like this:

```
NSMutableString *mutableString = [NSMutableString stringWithString:@"Hello"];
NSArray *immutableArray = @[mutableString];
```

there's nothing to stop you from mutating the string:

```
if ([immutableArray count] > 0) {
    id string = immutableArray[0];
    if ([string isKindOfClass:[NSMutableString class]]) {
        [string appendString:@" World!"];
    }
}
```

If you need to be able to add or remove objects from an array after initial creation, you'll need to use `NSMutableArray`, which adds a variety of methods to add , remove or replace one or more objects:

```
NSMutableArray *mutableArray = [NSMutableArray array];
[mutableArray addObject:@"gamma"];
[mutableArray addObject:@"alpha"];
[mutableArray addObject:@"beta"];

[mutableArray replaceObjectAtIndex:0 withObject:@"epsilon"];
```

This example creates an array that ends up with the objects `@"epsilon"`, `@"alpha"`, `@"beta"`.

It's also possible to sort a mutable array in place, without creating a secondary array:

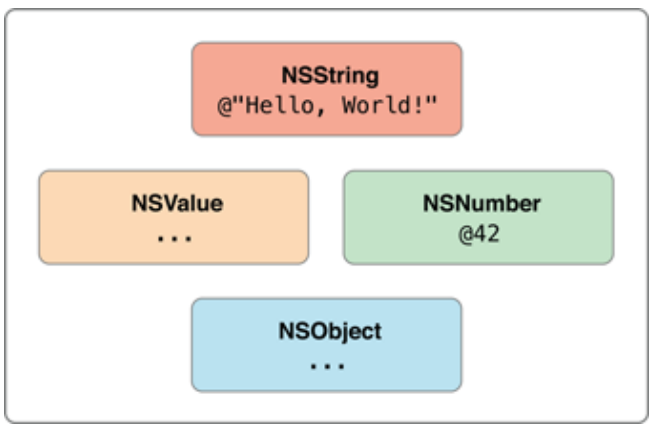
```
[mutableArray sortUsingSelector:@selector(caseInsensitiveCompare:)];
```

In this case the contained items will be sorted into the ascending, case insensitive order of `@"alpha"`, `@"beta"`, `@"epsilon"`.

Sets Are Unordered Collections

An `NSSet` is similar to an array, but maintains an unordered group of **distinct** objects, as shown in Figure 6-2.

Figure 6-2 A Set of Objects



Because sets don't maintain order, they offer a performance improvement over arrays when it comes to testing for membership.

The basic `NSSet` class is again immutable, so its contents must be specified at creation, using either allocation and initialization or a class factory method, like this:

```
NSSet *simpleSet =
  [NSSet setWithObjects:@"Hello, World!", @42, aValue, anObject, nil];
```

As with `NSArray`, the `initWithObjects:` and `setWithObjects:` methods both take a nil-terminated, variable number of arguments. The mutable `NSSet` subclass is `NSMutableSet`.

Sets only store one reference to an individual object, even if you try and add an object more than once:

```
NSNumber *number = @42;
NSSet *numberSet =
  [NSSet setWithObjects:number, number, number, number, nil];
// numberSet only contains one object
```

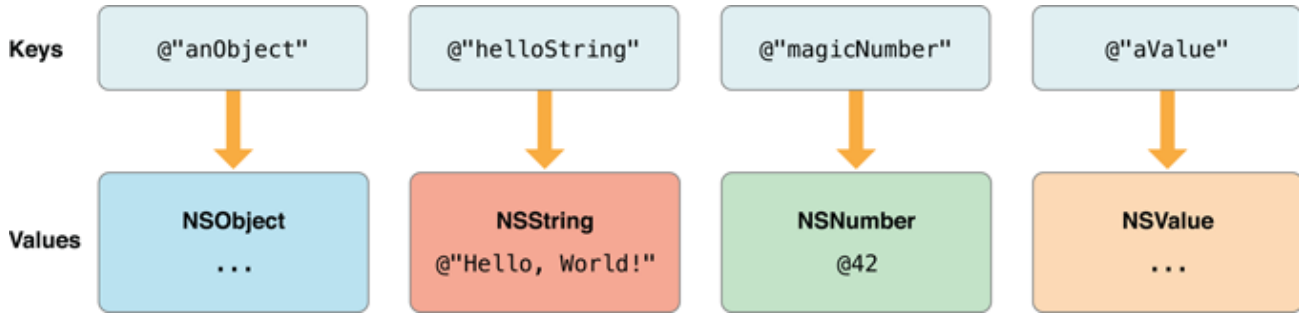
For more information on sets, see Sets: Unordered Collections of Objects.

Dictionaries Collect Key-Value Pairs

Rather than simply maintaining an ordered or unordered collection of objects, an `NSDictionary` stores objects against given keys, which can then be used for retrieval.

It's best practice to use string objects as dictionary keys, as shown in Figure 6-3.

Figure 6-3 A Dictionary of Objects



Note: It's possible to use other objects as keys, but it's important to note that each key is copied for use by a dictionary and so must support `NSCopying`.

If you wish to be able to use Key-Value Coding, however, as described in *Key-Value Coding Programming Guide*, you must use string keys for dictionary objects.

Creating Dictionaries

You can create dictionaries using either allocation and initialization, or class factory methods, like this:

```
NSDictionary *dictionary = [NSDictionary dictionaryWithObjectsAndKeys:
                anObject, @"anObject",
          @"Hello, World!", @"helloString",
                    @42, @"magicNumber",
                 aValue, @"aValue",
                         nil];
```

Note that for the `dictionaryWithObjectsAndKeys:` and `initWithObjectsAndKeys:` methods, each object is specified before its key, and again, the list of objects and keys must be nil-terminated.

Literal Syntax

Objective-C also offers a literal syntax for dictionary creation, like this:

```
NSDictionary *dictionary = @{
                @"anObject" : someObject,
            @"helloString" : @"Hello, World!",
            @"magicNumber" : @42,
                 @"aValue" : someValue
};
```

Note that for dictionary literals, the key is specified before its object and is not nil-terminated.

Querying Dictionaries

Once you've created a dictionary, you can ask it for the object stored against a given key, like this:

```
NSNumber *storedNumber = [dictionary objectForKey:@"magicNumber"];
```

If the object isn't found, the `objectForKey:` method will return `nil`.

There's also a subscript syntax alternative to using `objectForKey:`, which looks like this:

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

Mutability

If you need to add or remove objects from a dictionary after creation, you need to use the `NSMutableDictionary` subclass, like this:

```
[dictionary setObject:@"another string" forKey:@"secondString"];
[dictionary removeObjectForKey:@"anObject"];
```

Represent nil with NSNull

It's not possible to add `nil` to the collection classes described in this section because `nil` in Objective-C means "no object." If you need to represent "no object" in a collection, you can use the `NSNull` class:

```
NSArray *array = @[ @"string", @42, [NSNull null] ];
```

`NSNull` is a singleton class, which means that the `null` method will always return the same instance. This means that you can check whether an object in an array is equal to the shared `NSNull` instance:

```
for (id object in array) {
    if (object == [NSNull null]) {
        NSLog(@"Found a null object");
    }
}
```

Use Collections to Persist Your Object Graph

The `NSArray` and `NSDictionary` classes make it easy to write their contents directly to disk, like this:

```
NSURL *fileURL = ...
NSArray *array = @[@"first", @"second", @"third"];


BOOL success = [array writeToURL:fileURL atomically:YES];
if (!success) {
    // an error occured...
}
```

If every contained object is one of the *property list* types (`NSArray`, `NSDictionary`, `NSString`, `NSData`, `NSDate` and `NSNumber`), it's possible to recreate the entire hierarchy from disk, like this:

```
NSURL *fileURL = ...
NSArray *array = [NSArray arrayWithContentsOfURL:fileURL];
if (!array) {
    // an error occurred...
}
```

For more information on property lists, see *Property List Programming Guide*.

If you need to persist other types of objects than just the standard property list classes shown above, you can use an archiver object, such as `NSKeyedArchiver`, to create an archive of the collected objects.

The only requirement to create an archive is that each object must support the `NSCoding` protocol. This means that each object must know how to *encode* itself to an archive (by implementing the `encodeWithCoder:` method) and *decode* itself when read from an existing archive (the `initWithCoder:` method).

The `NSArray`, `NSSet` and `NSDictionary` classes, and their mutable subclasses, all support `NSCoding`, which means you can persist complex hierarchies of objects using an archiver. If you use Interface Builder to lay out windows and views, for example, the resulting nib file is just an archive of the object hierarchy that you've created visually. At runtime, the nib file is unarchived to a hierarchy of objects using the relevant classes.

For more information on Archives, see *Archives and Serializations Programming Guide*.

Use the Most Efficient Collection Enumeration Techniques

Objective-C and Cocoa or Cocoa Touch offer a variety of ways to enumerate the contents of a collection. Although it's possible to use a traditional C `for` loop to iterate over the contents, like this:

```
int count = [array count];
for (int index = 0; index < count; index++) {
    id eachObject = [array objectAtIndex:index];
    ...
}
```

it's best practice to use one of the other techniques described in this section.

Fast Enumeration Makes It Easy to Enumerate a Collection

Many collection classes conform to the `NSFastEnumeration` protocol, including `NSArray`, `NSSet` and `NSDictionary`. This means that you can use fast enumeration, an Objective-C language-level feature.

The fast enumeration syntax to enumerate the contents of an array or set looks like this:

```
for (<Type> <variable> in <collection>) {
```

```
    ...
}
```

As an example, you might use fast enumeration to log a description of each object in an array, like this:

```
for (id eachObject in array) {
    NSLog(@"Object: %@", eachObject);
}
```

The `eachObject` variable is set automatically to the current object for each pass through the loop, so one log statement appears per object.

If you use fast enumeration with a dictionary, you iterate over the dictionary *keys*, like this:

```
for (NSString *eachKey in dictionary) {
    id object = dictionary[eachKey];
    NSLog(@"Object: %@ for key: %@", object, eachKey);
}
```

Fast enumeration behaves much like a standard C `for` loop, so you can use the `break` keyword to interrupt the iteration, or `continue` to advance to the next element.

If you are enumerating an ordered collection, the enumeration proceeds in that order. For an `NSArray`, this means the first pass will be for the object at index `0`, the second for object at index `1`, etc. If you need to keep track of the current index, simply count the iterations as they occur:

```
int index = 0;
for (id eachObject in array) {
    NSLog(@"Object at index %i is: %@", index, eachObject);
    index++;
}
```

You cannot mutate a collection during fast enumeration, even if the collection is mutable. If you attempt to add or remove a collected object from within the loop, you'll generate a runtime exception.

Most Collections Also Support Enumerator Objects

It's also possible to enumerate many Cocoa and Cocoa Touch collections by using an `NSEnumerator` object.

You can ask an `NSArray`, for example, for an `objectEnumerator` or a `reverseObjectEnumerator`. It's possible to use these objects with fast enumeration, like this:

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

In this example, the loop will iterate over the collected objects in reverse order, so the last object will be first, and so on.

It's also possible to iterate through the contents by calling the enumerator's `nextObject` method repeatedly, like this:

```
id eachObject;
while ( (eachObject = [enumerator nextObject]) ) {
    NSLog(@"Current object is: %@", eachObject);
}
```

In this example, a `while` loop is used to set the `eachObject` variable to the next object for each pass through the loop. When there are no more objects left, the `nextObject` method will return `nil`, which evaluates as a logical value of false so the loop stops.

Note: Because it's a common programmer error to use the C assignment operator (=) when you mean the equality operator (==), the compiler will warn you if you set a variable in a conditional branch or loop, like this:

```
if (someVariable = YES) {
    ...
}
```

If you really do mean to reassign a variable (the logical value of the overall assignment is the final value of the left hand side), you can indicate this by placing the assignment in parentheses, like this:

```
if ( (someVariable = YES) ) {
    ...
}
```

As with fast enumeration, you cannot mutate a collection while enumerating. And, as you might gather from the name, it's faster to use fast enumeration than to use an enumerator object manually.

Many Collections Support Block-Based Enumeration

It's also possible to enumerate `NSArray`, `NSSet` and `NSDictionary` using blocks. Blocks are covered in detail in the next chapter.