Collections Programming Topics

Data Management: Data Types & Collections

2010-07-08

Contents

Introduction

This document describes objects that let you group together other types of objects in arrays, sets, or dictionaries.

All developers using Cocoa need to understand how collections work.

Organization of This Document

The various types of collection and the operations you can perform on them are discussed in the following articles:

Arrays: Ordered Collections

Arrays are ordered collections of objects. Cocoa provides several array classes, NSArray, NSMutableArray (a subclass of NSArray), and NSPointerArray.

Array Fundamentals

Arrays are ordered collections that can contain any sort of object—the collection does not have to be homogeneous. For example, an array could contain any combination of Cat and Dog objects, and if it's mutable you could add a Vet object, but it could not contain an int or a float. NSPointerArray differs from NSArray and NSMutableArray in that the former can contain nil values (if you need to add a representation of a null value to an instance of NSArray, use NSNull) and its contents do not need to be objects; moreover, it also offers memory management options using an instance of NSPointerFunctions.

An NSArray object manages a static array—that is, once you have created the array you cannot add objects to it or remove objects from it—you can, however, modify individual elements themselves (if they support modification). For example, given an NSArray object that contains just a single Dog object, you could not add another Dog, or a Cat. You could, however, change the Dog's name.

An NSMutableArray object manages a dynamic—or mutable—array, which allows the addition and deletion of entries at any time, automatically allocating memory as needed. For example, given an NSMutableArray object that contains just a single Dog object, you can add another Dog, or a Cat, or any other object. You can also, as with an NSArray instance, change the Dog's name—and in general, anything that you can do with an NSArray object you can do with an NSMutableArray object.

You can easily create an instance of one type of array from the other using the initializer initWithArray: or the convenience constructor arrayWithArray:. For example, if you have an instance of NSArray, myArray, you can create a mutable copy as follows:

```
NSMutableArray *myMutableArray = [NSMutableArray arrayWithArray:myArray];
```

In general, you instantiate an array by sending one of the array... messages to either the NSArray or NSMutableArray class object. The array... messages return an array containing the elements you pass in as arguments. And when you add an object to an Objective-C array, the object isn't copied, but rather receives a retain message before its id is added to the array. When an array is deallocated, each element is sent a release message.

NSArray's two primitive methods—count and objectAtIndex—provide the basis for all other methods in its interface. The count method returns the number of elements in the array. objectAtIndex gives you access to the array elements by index, with index values starting at 0.

Mutable Arrays

NSMutableArray's primitive methods, listed below, provide the basis for its ability to add, replace, and remove elements:

```
addObject:
insertObject:atIndex:
removeLastObject
removeObjectAtIndex:
replaceObjectAtIndex:withObject:
```

The other methods in NSMutableArray provide convenient ways of inserting an object into a specific slot in the array and removing an object based on its identity or position in the array, as illustrated in the following example.

```
NSMutableArray *array = [NSMutableArray array];
[array addObject:[NSColor blackColor]];
[array insertObject:[NSColor redColor] atIndex:0];
[array insertObject:[NSColor blueColor] atIndex:1];
[array addObject:[NSColor whiteColor]];
[array removeObjectsInRange:(NSMakeRange(1, 2))];
// array now contains redColor and whiteColor
```

In a managed memory environment, when an object is removed from a mutable array it receives a release message. This means that *if an array is the only owner of an object*, then the object is deallocated when it is removed. If you want to use the object after its removal, you should therefore typically send it a retain message before it's removed from the array. For example, if in the following example anArray is the only owner of anObject, the third statement below would result in a run-time error:

```
id anObject = [anArray objectAtIndex:0];
[anArray removeObjectAtIndex:0];
// if no object other than anArray owned anObject, the next line causes a crash
[anObject someMessage];
```

If you want to use an object after removing it from an array, you should retain it first as shown in the following example:

```
id anObject = [[anArray objectAtIndex:0] retain];
[anArray removeObjectAtIndex:0];
[anObject someMessage];
// remember to send anObject a release message when you have finished with it
```

Using Arrays

The NSArray methods objectEnumerator and reverseObjectEnumerator grant sequential access to the elements of the array, differing only in the direction of travel through the elements (see "Enumeration: Traversing a Collection's Elements" (page 23)). NSArray's makeObjectsPerformSelector: and makeObjectsPerformSelector:withObject: methods let you send messages to all objects in the array.

You can extract a subset of the array (`subarrayWithRange:`) or concatenate the elements of an array of `NSString` objects into a single string (`componentsJoinedByString:`). In addition, you can compare two arrays using the `isEqualToArray:` and `firstObjectCommonWithArray:` methods. Finally, you can create a new array that contains the objects in an existing array and one or more additional objects with `arrayByAddingObject:` or `arrayByAddingObjectsFromArray:`.

There are two principal methods you can use to determine whether an object is present in an array, `indexOfObject:` and`indexOfObjectIdenticalTo:`—there are also two variants, `indexOfObject:inRange:` and `indexOfObjectIdenticalTo:inRange:` that you can use to search a range within an array. The `indexOfObject:` methods test for equality by sending elements in the array an `isEqual:` message; the `indexOfObjectIdenticalTo:` methods test for equality using pointer comparison. The difference is illustrated in the following example:

```
NSString *yes0 = @"yes";
NSString *yes1 = @"YES";
NSString *yes2 = [NSString stringWithFormat:@"%@", yes1];

NSArray *yesArray = [NSArray arrayWithObjects: yes0, yes1, yes2, nil];

NSUInteger index;

index = [yesArray indexOfObject:yes2];
// index is 1

index = [yesArray indexOfObjectIdenticalTo:yes2];
// index is 2
```

Dictionaries: Collections of Keys and Values

Dictionaries manage pairs of keys and values. Use a dictionary when you need a convenient and efficient way to retrieve data associated with an arbitrary key. An `NSDictionary` object manages a static array; that is, an array whose keys and values cannot be removed, replaced, or added to. However, the individual elements can be modified. An `NSMutableDictionary`, a subclass of `NSDictionary`, allows you to add and delete entries at any time, automatically allocating memory as needed. You can easily convert one type of dictionary to the other with the initializer that takes a dictionary as an argument.

Dictionary Fundamentals

A key-value pair within a dictionary is called an entry. Each entry consists of one object that represents the key, and a second object which is that key's value. Within a dictionary, the keys are unique—that is, no two keys in a single dictionary are equal (as determined by `equals` or `isEqual:`). A key does not have to be a string, it can be any object that adopts the `NSCopying` protocol.

Methods that add entries to dictionaries—whether as part of initialization (for all dictionaries) or during modification (for mutable dictionaries)— don't add each key object to the dictionary directly, but copy each key argument and add the copy to the dictionary. In a reference-counted environment, each corresponding value object receives a `retain` message to ensure that it won't be deallocated before the dictionary is finished with it.

Important: Because the dictionary copies each key, keys must conform to the `NSCopying` protocol). You should also bear this in mind when choosing what objects to use as keys. Although you can use any object that adopts the `NSCopying` protocol, it is typically bad design to use large objects such as instances of `NSImage` since this may incur performance penalties.

Internally, a dictionary uses a hash table to organize its storage and to provide rapid access to a value given the corresponding key. However, the methods defined for dictionaries insulate you from the complexities of working with hash tables, hashing functions, or the hashed value of keys. The methods take keys directly, not their hashed form.

Using Mutable Dictionaries

You must be careful when removing an entry from a mutable dictionary, since the key and value objects that make up the entry receive `release` messages. If there are no further references to the objects, they're deallocated. Note that if your program keeps a reference to such an object, the reference will become invalid unless you remember to send the object a `retain` message before it's removed from the dictionary. For example, the third statement below will result in a run-time error if the dictionary is the only owner of the `anObject`:

```
id anObject = [aDictionary objectForKey:theKey];

[aDictionary removeObjectForKey:theKey];
[anObject someMessage]; // likely crash
```

To guard against this possibility, you can retain a value object before you remove it, as illustrated in this example:

```
id anObject = [[aDictionary objectForKey:theKey] retain];

[aDictionary removeObjectForKey:theKey];
[anObject someMessage];
```

Sorting a Dictionary

`NSDictionary` provides the method `keysSortedByValueUsingSelector:` which returns an array of the dictionary's keys in the order they would be in if the dictionary were sorted by its values, as illustrated in the following example:

```
NSDictionary *dict = [NSDictionary dictionaryWithObjectsAndKeys:
    [NSNumber numberWithInt:63], @"Mathematics",
    [NSNumber numberWithInt:72], @"English",
    [NSNumber numberWithInt:55], @"History",
    [NSNumber numberWithInt:49], @"Geography",
    nil];

NSArray *sortedKeysArray =
    [dict keysSortedByValueUsingSelector:@selector(compare:)];
// sortedKeysArray contains: (Geography, History, Mathematics, English)
```

Subclassing NSDictionary and NSMutableDictionary

There should typically be little need to subclass `NSDictionary` or `NSMutableDictionary`. If you do need to customize behavior of these classes, it is often better to consider composition rather than subclassing.

If you do need to subclass `NSDictionary` or `NSMutableDictionary`, you need to take into account that they are represented by class clusters—there are therefore several primitive methods for which you must provide implementations.

`NSDictionary`'s three primitive methods—`count`, `objectForKey:`, and `keyEnumerator`—provide the basis for all of the other methods in its interface. The `count` method returns the number of entries in the dictionary. `objectForKey:` returns the value associated with a given key. `keyEnumerator` returns an object that lets you iterate through each of the keys in the dictionary. `NSDictionary`'s other methods operate by invoking one or more of these primitives. The non-primitive methods provide convenient ways of accessing multiple entries at once. And the Objective-C methods `description...` and `writeToFile:atomically:` cause a dictionary to write a representation of itself to a string or to a file, respectively.

`NSMutableDictionary`'s primitive methods—`setObjectForKey:` and `removeObjectForKey:`—add modification operations to the basic operations it inherits from `NSDictionary`. Its other methods invoke one or both of these primitives. They provide convenient ways of adding or removing multiple entries at a time.

Sets: Unordered Collections of Objects

A set is an unordered collection of objects. Cocoa provides three classes to represent sets. An `NSSet` object is a static unordered collection of objects. You establish a static set's entries when it's created, and thereafter you cannot remove, replace, or add to them. `NSMutableSet`, a subclass of `NSSet`, is a dynamic set of objects. A dynamic—or mutable—set allows the addition and deletion of entries at any time, automatically allocating memory as needed. `NSMapTable` is similar to `NSMutableSet` (although it does not inherit from `NSSet`) in that represents a dynamic set of objects, but it provides additional features for memory management of its contents.

Set Fundamentals

You can use sets as an alternative to arrays when the order of elements isn't important and performance in testing whether an object is contained in the set is a consideration. While arrays are ordered, testing them for membership is slower than testing sets.

When using sets in Objective-C code, the objects in a set must respond to the `NSObject` protocol methods `hash` and `isEqual:` (see the `NSObject` protocol for more information). Note that if mutable objects are stored in a set, either the `hash` method of the objects shouldn't depend on the internal state of the mutable objects or the mutable objects shouldn't be modified while they're in the set (note that it can be difficult to know whether or not a given object is in a collection).

`NSSet` provides a number of initializer methods such as `setWithObjects:` and `initWithArray:` that return an `NSSet` object containing the elements (if any) you pass in as arguments. Objects added to a set are not copied (unless you pass `YES` as the argument to `initWithSet:copyItems:`); rather, an object is added directly to a set. In a managed memory environment, an object receives a `retain` message when it's added; in a garbage collected environment, it is strongly referenced. The `set` method is a convenience method to create an empty immutable set.

Mutable Sets

You can create an `NSMutableSet` object using any of the initializers provided by `NSSet`. You can create an `NSMutableSet` object from an instance of `NSSet` (or vice versa) using `setWithSet:` or `initWithSet:`.

`NSMutableSet` provides methods for adding objects to a set: `addObject:` adds a single object to the set. `addObjectsFromArray:` adds all objects from a specified array to the set. And `unionSet:` adds all the objects from another set.

`NSMutableSet` provides these methods to remove objects from a set: `intersectSet:`, `removeAllObjects`, `removeObject:`, and `minusSet:` .

In a managed memory environment, when an object is removed from a mutable set it receives a `release` message. This means that *if a set is the only owner of an object*, then (by standard rules of memory management) the object is deallocated when it is removed. If you want to use the object after its removal, you should therefore typically send it a `retain` message before it's removed from the set. For example, if in the following example `aSet` is the only owner of `anObject`, the third statement below would result in a run-time error:

```
id anObject = [aSet anyObject];
[aSet removeObject:anObject];
// if no object other than aSet owned anObject, the next line causes a crash
[anObject someMessage];
```

If you want to use an object after removing it from a set, you should retain it first as shown in the following example:

```
id anObject = [[aSet anyObject] retain];
[aSet removeObject:anObject];
[anObject someMessage];
// remember to send anObject a release message when you have finished with it
```

Using Sets

`NSSet` provides methods for querying the elements of the set. `allObjects` returns an array containing the objects in a set. `anyObject` returns some object in the set. `count` returns the number of objects currently in the set. `member:` returns the object in the set that is equal to a specified object. Additionally, `intersectsSet:` tests for set intersection, `isEqualToSet:` tests for set equality, and `isSubsetOfSet:` tests for one set being a subset of another.

The `NSSet` method `objectEnumerator` lets you traverse elements of the set one by one. And the Objective-C methods `makeObjectsPerformSelector:` and `makeObjectsPerformSelector:withObject:` provides for sending messages to individual objects in the set.

Counted Sets: Unordered Collection of Indistinct Objects

`NSCountedSet`, a subclass of `NSMutableSet`, is a set to which you can add a particular object more than once; in other words, the elements of the set aren't necessarily distinct. A counted set is also known as a bag. The `NSCountedSet` class is available only in Objective-C.

Each distinct object inserted into an `NSCountedSet` object has a counter associated with it. `NSCountedSet` keeps track of the number of times objects are inserted and requires that objects be removed the same number of times. Thus, there is only one instance of an object in an `NSCountedSet` even if the object has been added to the set multiple times. The `countForObject:` method returns the number of times the specified object has been added to this set.

Sorting and Filtering NSArray Objects

Sorting Arrays

There are several ways to sort an `NSArray` object; these methods return a new array: `sortedArrayUsingDescriptors:`, `sortedArrayUsingFunction:context:`, `sortedArrayUsingFunction:context:hint:`, and `sortedArrayUsingSelector:`—you can use them with an immutable or a mutable array; these methods sort the contents of a mutable array: `sortUsingDescriptors:`, `sortUsingFunction:context:`, and `sortUsingSelector:`.

Important: If you sort a list that is to be shown to the end user, you should always use a localized comparison. For a general overview of the issues related to sorting, see Collation Introduction.

Sorting With Functions and Selectors

The following example illustrates the use of `sortedArrayUsingSelector:`,`sortedArrayUsingFunction:context:`, and `sortedArrayUsingFunction:context:hint:`. The most complex of these methods is `sortedArrayUsingFunction:context:hint:`. The hinted sort is most efficient when you have a large array (N entries) that you sort once and then change only slightly (P additions and deletions, where P is much smaller than N). You can reuse the work you did in the original sort by conceptually doing a merge sort between the N "old" items and the P "new" items. To obtain an appropriate hint, you use `sortedArrayHint` when the original array has been sorted, and keep hold of it until you need it (when you want to re-sort the the array after it has been modified).

```
NSInteger alphabeticSort(id string1, id string2, void *reverse)
{
    if ((NSInteger *)reverse == NO) {
        return [string2 localizedCaseInsensitiveCompare:string1];
    }
    return [string1 localizedCaseInsensitiveCompare:string2];
}

NSMutableArray *anArray =
    [NSMutableArray arrayWithObjects:@"aa", @"ab", @"ac", @"ad", @"ae", @"af",
 @"ag",
        @"ah", @"ai", @"aj", @"ak", @"al", @"am", @"an", @"ao", @"ap", @"aq",
@"ar", @"as", @"at",
        @"au", @"av", @"aw", @"ax", @"ay", @"az", @"ba", @"bb", @"bc", @"bd",
@"bf", @"bg", @"bh",
        @"bi", @"bj", @"bk", @"bl", @"bm", @"bn", @"bo", @"bp", @"bq", @"br",
@"bs", @"bt", @"bu",
        @"bv", @"bw", @"bx", @"by", @"bz", @"ca", @"cb", @"cc", @"cd", @"ce",
@"cf", @"cg", @"ch",
        @"ci", @"cj", @"ck", @"cl", @"cm", @"cn", @"co", @"cp", @"cq", @"cr",
@"cs", @"ct", @"cu",
        @"cv", @"cw", @"cx", @"cy", @"cz", nil];
// note: anArray is sorted
NSData *sortedArrayHint = [anArray sortedArrayHint];

[anArray insertObject:@"be" atIndex:5];

NSArray *sortedArray;

// sort using a selector
sortedArray =
        [anArray
sortedArrayUsingSelector:@selector(localizedCaseInsensitiveCompare:)];

// sort using a function
int reverseSort = NO;
sortedArray =
       [anArray sortedArrayUsingFunction:alphabeticSort context:&reverseSort];

// sort with a hint
sortedArray =
        [anArray sortedArrayUsingFunction:alphabeticSort
                                  context:&reverseSort
                                     hint:sortedArrayHint];
```

Sorting With Sort Descriptors

Sort descriptors (instances of `NSSortDescriptor`) provide a convenient and abstract way to describe a sort ordering. Sort descriptors provide several useful features. You can use them in conjunction with Cocoa bindings to sort the contents of, for example, a table view, and you can use them with Core Data to order the results of a fetch request.

If you use the methods `sortedArrayUsingDescriptors:` or `sortUsingDescriptors:`, sort descriptors provide an easy way to sort a collection of objects using a number of their properties. Consider the following example. Given an array of dictionaries (custom objects would work in the same way):

```
NSString *LAST = @"lastName";
NSString *FIRST = @"firstName";

NSMutableArray *array = [NSMutableArray array];
NSArray *sortedArray;

NSDictionary *dict;
dict = [NSDictionary dictionaryWithObjectsAndKeys:
                     @"Jo", FIRST, @"Smith", LAST, nil];
[array addObject:dict];

dict = [NSDictionary dictionaryWithObjectsAndKeys:
                     @"Joe", FIRST, @"Smith", LAST, nil];
[array addObject:dict];

dict = [NSDictionary dictionaryWithObjectsAndKeys:
                     @"Joe", FIRST, @"Smythe", LAST, nil];
[array addObject:dict];
```

```
dict = [NSDictionary dictionaryWithObjectsAndKeys:
                     @"Joanne", FIRST, @"Smith", LAST, nil];
[array addObject:dict];

dict = [NSDictionary dictionaryWithObjectsAndKeys:
                     @"Rupert", FIRST, @"Psmith", LAST, nil];
[array addObject:dict];
```

you can sort the contents of the array by last name then first name as follows:

```
// The results are likely to be shown to a user
// Note the use of the localizedCaseInsensitiveCompare: selector
NSSortDescriptor *lastDescriptor =
    [[[NSSortDescriptor alloc] initWithKey:LAST
                               ascending:YES

selector:@selector(localizedCaseInsensitiveCompare:)] autorelease];
NSSortDescriptor *firstDescriptor =
    [[[NSSortDescriptor alloc] initWithKey:FIRST
                               ascending:YES

selector:@selector(localizedCaseInsensitiveCompare:)] autorelease];

NSArray *descriptors = [NSArray arrayWithObjects:lastDescriptor, firstDescriptor,
 nil];
sortedArray = [array sortedArrayUsingDescriptors:descriptors];
```

It is conceptually and programmatically easy to change the sort ordering and to arrange by first name then last name:

```
NSSortDescriptor *lastDescriptor =
        [[[NSSortDescriptor alloc] initWithKey:LAST
                                   ascending:NO

selector:@selector(localizedCaseInsensitiveCompare:)] autorelease];
NSSortDescriptor *firstDescriptor =
    [[[NSSortDescriptor alloc] initWithKey:FIRST
                               ascending:NO

selector:@selector(localizedCaseInsensitiveCompare:)] autorelease];
NSArray *descriptors = [NSArray arrayWithObjects:firstDescriptor, lastDescriptor,
 nil];
sortedArray = [array sortedArrayUsingDescriptors:descriptors];
```

In particular, it is straightforward to create the sort descriptors from user input.

By contrast, the following code illustrates the first sorting using a function.

```
NSInteger lastNameFirstNameSort(id person1, id person2, void *reverse)
{
    NSString *name1 = [person1 valueForKey:LAST];
    NSString *name2 = [person2 valueForKey:LAST];

   NSComparisonResult comparison = [name1 localizedCaseInsensitiveCompare:name2];
    if (comparison == NSOrderedSame) {

        name1 = [person1 valueForKey:FIRST];
        name2 = [person2 valueForKey:FIRST];
        comparison = [name1 localizedCaseInsensitiveCompare:name2];
    }

    if ((BOOL *)reverse == NO) {
        return 0 - comparison;
    }
    return comparison;
}

BOOL reverseSort = YES;
sortedArray = [array sortedArrayUsingFunction:lastNameFirstNameSort
    context:&reverseSort];
```

This approach is considerably less flexible.

Filtering Arrays

iOS Note: The predicate classes—`NSPredicate`, `NSCompoundPredicate`, and `NSComparisonPredicate`—are present only in the Mac OS X version of Foundation.

`NSArray` and `NSMutableArray` provide methods to filter array contents. `NSArray` provides `filteredArrayUsingPredicate:` which returns a new array containing objects in the receiver that match the specified predicate. `NSMutableArray` adds `filterUsingPredicate:` which evaluates the receiver's content against the specified predicate and leaves only objects that match. These methods are illustrated in the following example. For more about predicates, see *Predicate Programming Guide*.

```
NSMutableArray *array =
    [NSMutableArray arrayWithObjects:@"Bill", @"Ben", @"Chris", @"Melissa",
nil];

NSPredicate *bPredicate =
    [NSPredicate predicateWithFormat:@"SELF beginswith[c] 'b'"];
NSArray *beginWithB =
    [array filteredArrayUsingPredicate:bPredicate];
// beginWithB contains { @"Bill", @"Ben" }.

NSPredicate *sPredicate =
    [NSPredicate predicateWithFormat:@"SELF contains[c] 's'"];
[array filterUsingPredicate:sPredicate];
// array now contains { @"Chris", @"Melissa" }
```

Enumeration: Traversing a Collection's Elements

`NSEnumerator` is a simple abstract class whose subclasses enumerate collections of other objects. Collection objects—such as arrays, sets, and dictionaries—provide special `NSEnumerator` objects with which to enumerate their contents. You send `nextObject` repeatedly to a newly created `NSEnumerator` object to have it return the next object in the original collection. When the collection is exhausted, it returns `nil`. You can't "reset" an enumerator after it's exhausted its collection. To enumerate a collection again, you must create a new enumerator.

Note: On Mac OS X v10.5 and later it is more efficient to enumerate a collection using fast enumeration (see "Fast Enumeration" in *The Objective-C Programming Language*).

Collection classes such as `NSArray`, `NSSet`, and `NSDictionary` include methods that return an enumerator appropriate to the type of collection. For instance, `NSArray` has two methods that return an `NSEnumerator` object: `objectEnumerator` and `reverseObjectEnumerator`. `NSDictionary` also has two methods that return an `NSEnumerator` object: `keyEnumerator` and `objectEnumerator`. These methods let you enumerate the contents of an `NSDictionary` by key or by value, respectively.

In Objective-C, an `NSEnumerator` retains the collection over which it's enumerating (unless implemented differently by a custom subclass).

It is not safe to remove, replace, or add to a mutable collection's elements while enumerating through it. If you need to modify a collection during enumeration, you can either: make a copy of the collection and enumerate using the copy; or, collect the information you require during the enumeration and apply the changes afterwards. The second pattern is illustrated in the following example.

```
NSMutableDictionary *myMutableDictionary = ... ;
NSMutableArray *keysToDeleteArray =
    [NSMutableArray arrayWithCapacity:[myMutableDictionary count]];
NSString *aKey;
NSEnumerator *keyEnumerator = [myMutableDictionary keyEnumerator];
while (aKey = [keyEnumerator nextObject])
{
    if ( /* test criteria for key or value */ ) {
        [keysToDeleteArray addObject:aKey];
    }
}
[myMutableDictionary removeObjectsForKeys:keysToDeleteArray];
```

Document Revision History

This table describes the changes to *Collections Programming Topics*.

Date	Notes
2010-07-08	Added links to related concepts.
2009-07-23	Corrected typographical errors.
2009-02-04	Corrected a code example showing indexOfObjectIdenticalTo:.
2008-06-05	Added note stating that the predicate classes are not available in iOS.
2007-10-31	Updated for Mac OS X v10.5. Fixed various minor errors.
2007-07-10	Changed examples in Sorting and Filtering NSArray Objects to use localized comparisons.
2006-11-07	Augmented description of searching for objects in an array.
2006-09-05	Revised "Arrays" article to clarify usage patterns.
2006-06-28	Changed document name from "Collections."
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.