# Collections Programming Topics

Developer

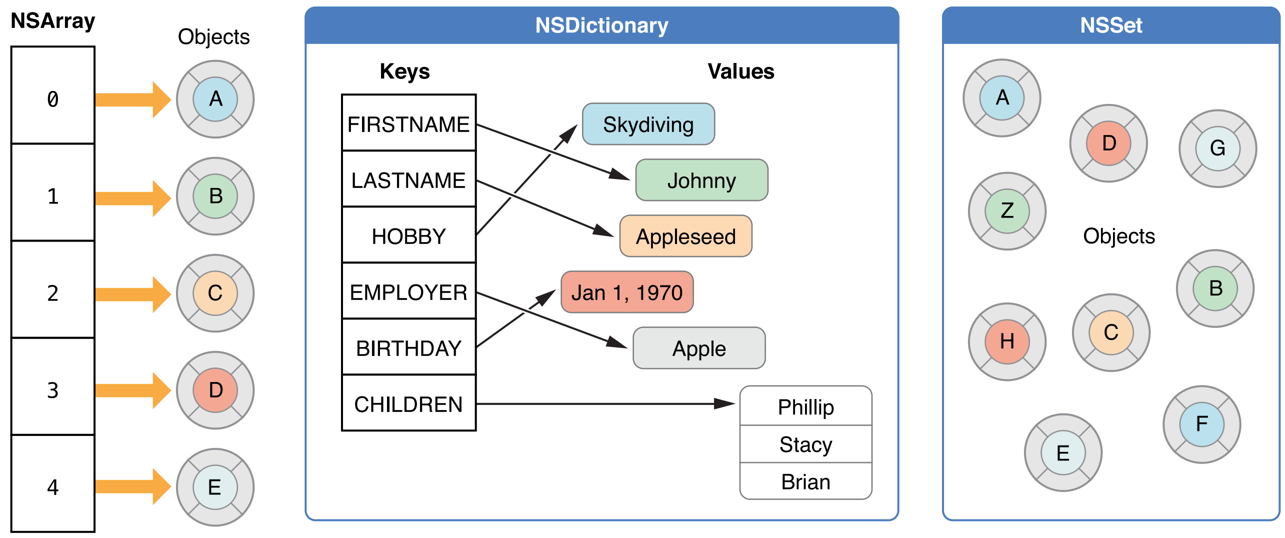
# Contents

# Figures and Listings

Objective-CSwift

# About Collections

In Cocoa and Cocoa Touch, a collection is a Foundation framework class used for storing and managing groups of objects. Its primary role is to store objects in the form of either an array, a dictionary, or a set.



These classes ease the task of managing groups of objects. Foundation collections are efficient and used extensively by OS X and iOS.

## At a Glance

Collections share a number of characteristics. Most collections hold only objects and have both a mutable and an immutable variant.

All collections share a number of common tasks, which include:

- Enumerating the objects in a collection
- Determining whether an object is in a collection
- Accessing individual elements in a collection

Mutable collections also allow some additional tasks:

- Adding objects to a collection

- Removing objects from a collection

While collections share many characteristics, there are also important differences. As a result, you will find some collections better suited to a particular task than others. Because how well a collection performs depends on how it is used, you should choose the collection best suited to a particular task.

## Accessing Indexes and Easily Enumerating Elements: Arrays

Arrays (such as `NSArray` and `NSMutableArray`) are ordered collections which allow indexed access to their contents. You might use an array to store the information to be presented in a table view because the order matters.

**Relevant Chapters:** Arrays: Ordered Collections (page 11)

## Associating Data with Arbitrary Keys: Dictionaries

Dictionaries (such as `NSDictionary` and `NSMutableDictionary`) are unordered collections that allow keyed-value access to their contents. They also allow for fast insertion and deletion operations. Dictionaries are useful for storing values that have meaning based on their key. For example, you might have a dictionary of information about California, with capital as a key and Sacramento as the corresponding value.

**Relevant Chapters:** Dictionaries: Collections of Keys and Values (page 23)

## Offering Fast Insertion, Deletion, and Membership Checks: Sets

Sets (such as `NSSet` , `NSMutableSet`, and `NSCountedSet`) are unordered collections of objects. Sets allow for fast insertion and deletion operations. They also allow you to quickly see whether an object is in a collection. `NSSet` and `NSMutableSet` store collections of distinct objects, while `NSCountedSet` stores a collection of non-distinct objects. For example, suppose you have a number of city objects and you want to visit each one only once. If you store each city that you visit in a set, you can quickly and easily see whether you have visited it.

**Relevant Chapters:** Sets: Unordered Collections of Objects (page 30)

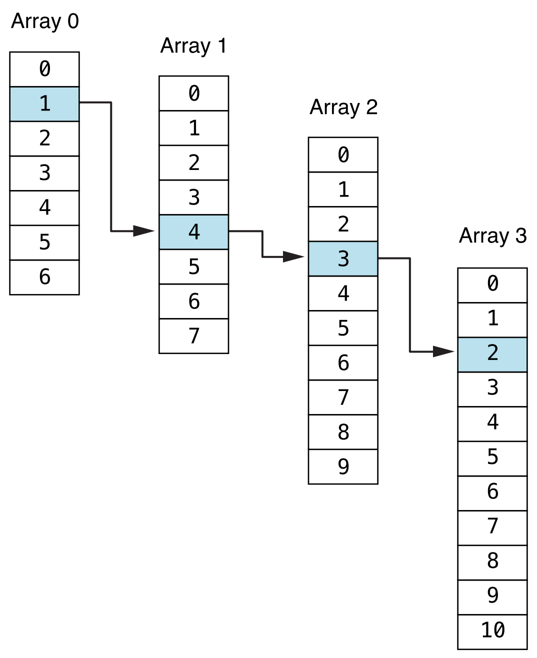
## Storing Subsets of Arrays: Index Sets

Index sets (such as `NSIndexSet` and `NSMutableIndexSet`) are helper objects that extend the capabilities of arrays. They allow you to store a subset of an array by storing the indexes into the array rather than by creating a new array. You might use an index set to allow a user to select multiple entries from a list of entries. For example, suppose you have a table view and you allow the user to select some of the rows. Because the rows are stored as an array, you could store the selections as an index set into that array.

**Relevant Chapters:** Index Sets: Storing Indexes into an Array (page 35)

## Storing Paths Through Nested Arrays: Index Paths

Index paths store the location of information in a more complicated collection hierarchy, specifically nested arrays. Cocoa provides the `NSIndexPath` class for this purpose. For example, the index path 1.4.3.2 specifies the path shown here:



While they are not collections in the strictest sense, index paths ease the task of managing nested arrays. The `UITableView` class makes extensive use of index paths to store locations within a table view.

**Relevant Chapters:** Index Paths: Storing a Path Through Nested Arrays (page 39)

## Customizing Memory and Storage Options: Pointer Collection Classes (OS X)

If you need collections to store arbitrary pointers or integers, or need to make use of zeroing weak references in a garbage-collected environment, there are the three pointer collection classes: `NSPointerArray`, `NSMapTable`, and `NSHashTable`. These are similar to `NSMutableArray`, `NSMutableDictionary`, and `NSMutableSet`, respectively. The three pointer collection classes allow additional options for specifying how the collection manages its contents. You can, for example, use pointer equality instead of invoking `isEqual:` during comparisons. Unlike all other collection classes, `NSPointerArray` is allowed to hold a `NULL` pointer.

**Relevant Chapters:** Arrays: Ordered Collections (page 11), Dictionaries: Collections of Keys and Values (page 23), Sets: Unordered Collections of Objects (page 30), and Pointer Function Options (page 47).

## Working with Collections: Copying and Enumerating

In addition to class specific behavior, there are some tasks which are shared in similar form between the collection classes. Two of these tasks are copying a collection and enumerating its contents.

When you need to create a new collection with the contents of another, you can choose either a shallow or a deep copy into the other. In a shallow copy each object is retained when it is added to the new collection and ownership is shared by two or more collections. In a deep copy each object is sent a `copyWithZone:` message as it is added to the collection, instead of being retained.

If you need to check each item in a collection for some condition or to perform some action on the entries selectively, you can use one of the provided ways of enumerating the contents of a collection. The two main methods of enumeration are fast enumeration and block-based enumeration.

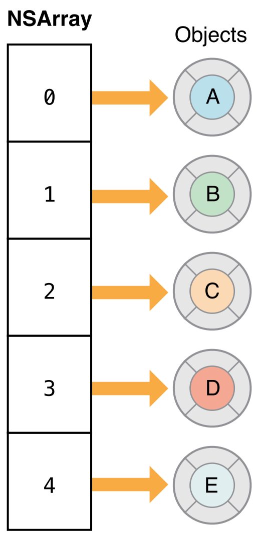
**Relevant Chapters:** Copying Collections (page 41) and Enumeration: Traversing a Collection's Elements (page 44)

# Arrays: Ordered Collections

Objective-CSwift

Arrays are ordered collections of any sort of object. For example, the objects contained by the array in Figure 1 can be any combination of cat and dog objects, and if the array is mutable you can add more dog objects. The collection does not have to be homogeneous.

**Figure 1** Example array



## Array Fundamentals

An `NSArray` object manages an immutable array—that is, after you have created the array, you cannot add, remove, or replace objects. You can, however, modify individual elements themselves (if they support modification). The mutability of the collection does not affect the mutability of the objects inside the collection. You should use an immutable array if the array rarely changes, or changes wholesale.

An `NSMutableArray` object manages a mutable array, which allows the addition and deletion of entries, 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` object, 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 should use a mutable array if the array changes incrementally or is very large—as large collections take more time to initialize.

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. The `array...` messages return an array containing the elements you pass in as arguments. And when you add an object to an `NSMutableArray` object, the object isn't copied, (unless you pass `YES` as the argument to `initWithArray:copyItems:`). Rather, a strong reference to the object is added to the array. For more information on copying and memory management, see Copying Collections (page 41).

In `NSArray`, two main methods—`count` and `objectAtIndex:`—provide the basis for all other methods in its interface:

- `count` returns the number of elements in the array.
- `objectAtIndex:` gives you access to the array elements by index, with index values starting at 0.

**Note:** Most operations on an array take constant time: accessing an element, adding or removing an element at either end, and replacing an element. Inserting an element into the middle of an array takes linear time.

## Mutable Arrays

In `NSMutableArray`, the main methods, listed below, provide the basis for its ability to add, replace, and remove elements:

`addObject:`

`insertObject:atIndex:`

`removeLastObject`

`removeObjectAtIndex:`

`replaceObjectAtIndex:withObject:`

If you do not need an object to be placed at a specific index or to be removed from the middle of the collection, you should use the `addObject:` and `removeLastObject` methods because it is faster to add and remove at the end of an array than in the middle.

The other methods in `NSMutableArray` provide convenient ways of inserting an object into a slot in the array and removing an object based on its identity or position in the array, as illustrated in Listing 1.

**Listing 1** Adding to and removing from arrays

```
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
```

## Using Arrays

You can access objects in an array by index using the `objectAtIndex:` method. For example, if you have an array of `NSString` objects, you can access the third string in the array as follows:

```
NSString *someString = [arrayOfStrings objectAtIndex:2];
```

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. Similarly, the `NSArray` methods `makeObjectsPerformSelector:` and `makeObjectsPerformSelector:withObject:` let you send messages to all objects in the array. In most cases, fast enumeration should be used as it is faster and more flexible than using an `NSEnumerator` or the `makeObjectsPerformSelector:` method. For more on enumeration, see Enumeration: Traversing a Collection's Elements (page 44).

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 Listing 2.

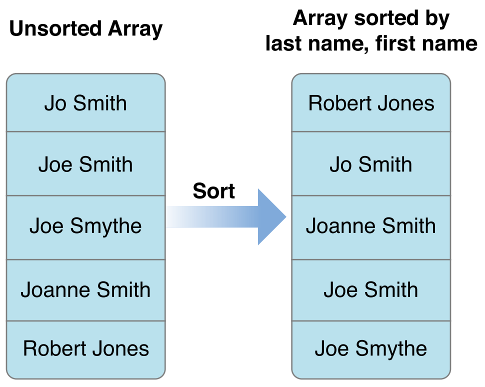
**Listing 2** Searching for an object in an array

```
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
```

## Sorting Arrays

You may need to sort an array based on some criteria. For instance, you may need to place a number of user-created strings into alphabetic order, or you may need to place numbers into increasing or decreasing order. Figure 2 shows an array sorted by last name then first name. Cocoa provides convenient ways to sort the contents of an array, such as sort descriptors, blocks, and selectors.

**Figure 2** Sorting arrays



## 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 easily perform most sort operations with minimal custom code. You can also use sort descriptors in conjunction with Cocoa bindings to sort the contents of, for example, a table view. You can also 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. Given an array of dictionaries (custom objects work in the same way), you can sort its contents by last name then first name. Listing 3 shows how to create that array and then sort with descriptors. (Figure 2 (page 14) shows an illustration of this example.)

**Listing 3** Creating and sorting an array of dictionaries

```
//First create the array of dictionaries
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:
                     @"Robert", first, @"Jones", last, nil];
[array addObject:dict];

//Next we sort the contents of the array by last name then first name

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

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, as shown in Listing 4.

**Listing 4** Sorting by first name, last name

```
NSSortDescriptor *lastDescriptor =
    [[NSSortDescriptor alloc] initWithKey:last
                                ascending:NO
                            selector:@selector(localizedCaseInsensitiveCompare:)];
NSSortDescriptor *firstDescriptor =
    [[NSSortDescriptor alloc] initWithKey:first
                                ascending:NO
                            selector:@selector(localizedCaseInsensitiveCompare:)];
```

```
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, Listing 5 illustrates the first sorting using a function. This approach is considerably less flexible.

**Listing 5** Sorting with a function is less flexible

```
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 == YES) {
        return 0 - comparison;
    }
    return comparison;
}

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

## Sorting with Blocks

You can use blocks to help sort an array based on custom criteria. The `sortedArrayUsingComparator:` method of `NSArray` sorts the array into a new array, using the block to compare the objects. `NSMutableArray`'s `sortUsingComparator:` sorts the array in place, using the block to compare the objects. Listing 6 illustrates sorting with a block.

**Listing 6** Blocks ease custom sorting of arrays

```
NSArray *sortedArray = [array sortedArrayUsingComparator: ^(id obj1, id obj2) {


     if ([obj1 integerValue] > [obj2 integerValue]) {
          return (NSComparisonResult)NSOrderedDescending;
     }


     if ([obj1 integerValue] < [obj2 integerValue]) {
          return (NSComparisonResult)NSOrderedAscending;
     }
     return (NSComparisonResult)NSOrderedSame;
}];
```

## Sorting with Functions and Selectors

Listing 7 illustrates the use of the methods `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 array after it has been modified).

**Listing 7** Sorting using selectors and functions

```
NSInteger alphabeticSort(id string1, id string2, void *reverse)
{
    if (*(BOOL *)reverse == YES) {
        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
BOOL reverseSort = NO;
sortedArray =
        [anArray sortedArrayUsingFunction:alphabeticSort context:&reverseSort];


// sort with a hint
```

```
sortedArray =
        [anArray sortedArrayUsingFunction:alphabeticSort
                                  context:&reverseSort
                                     hint:sortedArrayHint];
```

**Important:** If you sort an array of strings that is to be shown to the end user, you should always use a localized comparison. In other languages or contexts, the correct order of a sorted array can be different.

For a general overview of the issues related to sorting, see Collation Introduction.

## Filtering Arrays

The `NSArray` and `NSMutableArray` classes 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 Listing 8. For more about predicates, see *Predicate Programming Guide*.

**Listing 8** Filtering arrays with predicates

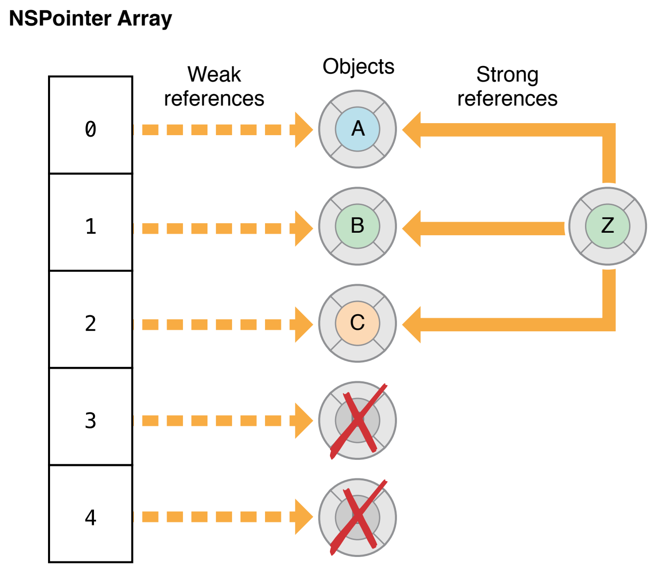
```
NSMutableArray *array =
    [NSMutableArray arrayWithObjects:@"Bill", @"Ben", @"Chris", @"Melissa", nil];

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" }
```

You can also filter an array using an `NSIndexSet` object. `NSArray` provides `objectsAtIndexes:`, which returns a new array containing the objects at the indexes in the provided index set. `NSMutableArray` adds `removeObjectsAtIndexes:`, which allows you to filter the array in place using an index set. For more information on index sets, see Index Sets: Storing Indexes into an Array (page 35).

## Pointer Arrays

The `NSPointerArray` class is configured by default to hold objects much as `NSMutableArray` does, except that it can hold `nil` values and that the `count` method reflects those `nil` values. It also allows additional storage options that you can tailor for specific cases, such as when you need advanced memory management options or when you want to hold a specific type of pointer. For example, the pointer array in Figure 3 is configured to hold weak references to its contents. You can also specify whether you want to copy objects entered into the array.

**Figure 3** Pointer array object ownership



You can use an `NSPointerArray` object when you want an ordered collection that uses weak references. For example, suppose you have a global array that contains some objects. Because global objects are never collected, none of its contents can be deallocated unless they are held weakly. Pointer arrays configured to hold objects weakly do not own their contents. If there are no strong references to objects within such a pointer array, those objects can be deallocated. For example, the pointer array in Figure 3 (page 21) holds weak references to its contents. Object D and Object E will deallocated.

To create a pointer array , create or initialize it using `pointerArrayWithOptions:` or `initWithOptions:` and the appropriate `NSPointerFunctionsOptions` options. Alternatively you can initialize it using `initWithPointerFunctions:` and appropriate instances of `NSPointerFunctions`. For more information on the various pointer functions options, see Pointer Function Options (page 47).

The `NSPointerArray` class also defines a number of convenience constructors for creating a pointer array with strong or weak references to its contents. For example, `pointerArrayWithWeakObjects` creates a pointer array that holds weak references to its contents. These convenience constructors should only be used if you are storing objects.

To configure a pointer array to use arbitrary pointers, you can initialize it with both the `NSPointerFunctionsOpaqueMemory` and `NSPointerFunctionsOpaquePersonality` options. For example, you can add a pointer to an `int` value using the approach shown in Listing 9.

**Listing 9** Pointer array configured for non-object pointers

```
NSPointerFunctionsOptions options=(NSPointerFunctionsOpaqueMemory |
     NSPointerFunctionsOpaquePersonality);

NSPointerArray *ptrArray=[NSPointerArray pointerArrayWithOptions: options];

 [ptrArray addPointer: someIntPtr];
```

You can then access an integer as show below.

```
NSLog(@" Index 0 contains: %i", *(int *) [ptrArray pointerAtIndex: 0] );
```

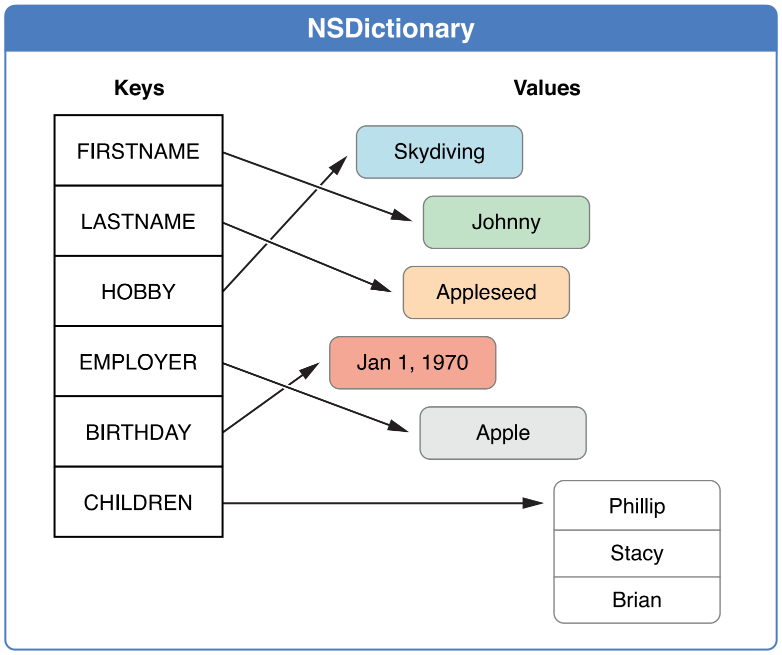
When configured to use arbitrary pointers, a pointer array has the risks associated with using pointers. For example, if the pointers refer to stack–based data created in a function, those pointers are not valid outside of the function, even if the pointer array is. Trying to access them will lead to undefined behavior.

# Dictionaries: Collections of Keys and Values

SwiftObjective-C

Dictionaries manage pairs of keys and values. 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 `isEqual:`). A key can be any object that adopts the `NSCopying` protocol and implements the `hash` and `isEqual:` methods. Figure 1 shows a dictionary which contains information about a hypothetical person. As shown, a value contained in the dictionary can be any object, even another collection.

**Figure 1** Example dictionary



## Dictionary Fundamentals

An `NSDictionary` object manages an immutable dictionary—that is, after you create the dictionary, you cannot add, remove or replace keys and values. You can, however, modify individual values themselves (if they support modification), but the keys must not be modified. The mutability of the collection does not affect the mutability of the objects inside the collection. You should use an immutable dictionary if the dictionary rarely changes, or changes wholesale.

An `NSMutableDictionary` object manages a mutable dictionary, which allows the addition and deletion of entries at any time, automatically allocating memory as needed. You should use a mutable dictionary if the dictionary changes incrementally, or is very large—as large collections take more time to initialize.

You can easily create an instance of one type of dictionary from the other using the initializer `initWithDictionary:` or the convenience constructor `dictionaryWithDictionary:`.

In general, you instantiate a dictionary by sending one of the `dictionary...` messages to either the `NSDictionary` or `NSMutableDictionary` class. The `dictionary...` messages return a dictionary containing the keys and values you pass in as arguments. Objects added as values to a dictionary are not copied (unless you pass `YES` as the argument to `initWithDictionary:copyItems:`). Rather, a strong reference to the object is added to the dictionary. For information on how a dictionary handles key objects, see Using Custom Keys (page 26). For more information on copying and memory management, see Copying Collections (page 41).

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 in their hashed form.

**Note:** If the key objects have a good hash function, accessing an element, setting an element, and removing an element all take constant time. With a poor hash function (one that causes frequent hash collisions), these operations take up to linear time. Classes such as `NSString` that are part of Foundation have a good hash function.

## Using Mutable Dictionaries

When removing an entry from a mutable dictionary, remember that the dictionary's strong reference to the key and value objects that make up the entry are discarded. If there are no further strong references to the objects, they're deallocated.

Adding objects to a mutable dictionary is relatively straightforward. To add a single key-value pair, or to replace the object for a particular key, use the `setObject:forKey:` instance method, as shown in Listing 1.

**Listing 1** Adding objects to a dictionary

```
NSString *last = @"lastName";
NSString *first = @"firstName";
```

```
NSMutableDictionary *dict = [NSMutableDictionary dictionaryWithObjectsAndKeys:
        @"Jo", first, @"Smith", last, nil];
NSString *middle = @"middleInitial";

 [dict setObject:@"M" forKey:middle];
```

You can also add entries from another dictionary using the `addEntriesFromDictionary:` instance method. If both dictionaries contain the same key, the receiver's previous value object for that key is released and the new object takes its place. For example, after the code in Listing 2 executes, `dict` would have a value of "Jones" for the key "lastName".

**Listing 2** Adding entries from another dictionary

```
NSString *last = @"lastName";
NSString *first = @"firstName";
NSString *suffix = @"suffix";
NSString *title = @"title";

NSMutableDictionary *dict = [NSMutableDictionary dictionaryWithObjectsAndKeys:
    @"Jo", first, @"Smith", last, nil];

NSDictionary *newDict = [NSDictionary dictionaryWithObjectsAndKeys:
    @"Jones", last, @"Hon.", title, @"J.D.", suffix, nil];

 [dict addEntriesFromDictionary: newDict];
```

## 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 Listing 3.

**Listing 3** Sorting dictionary keys by value

```
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
```

You can also use blocks to easily sort a dictionary's keys based on their corresponding values. The `keysSortedByValueUsingComparator:` method of `NSDictionary` allows you to use a block to compare the keys to be sorted into a new array. Listing 4 illustrates sorting with a block.

**Listing 4** Blocks ease custom sorting of dictionaries

```
NSArray *blockSortedKeys = [dict keysSortedByValueUsingComparator: ^(id obj1, id
obj2) {

     if ([obj1 integerValue] > [obj2 integerValue]) {
          return (NSComparisonResult)NSOrderedDescending;
     }

     if ([obj1 integerValue] < [obj2 integerValue]) {
          return (NSComparisonResult)NSOrderedAscending;
     }
     return (NSComparisonResult)NSOrderedSame;
}];
```

## Using Custom Keys

In most cases, Cocoa-provided objects such as `NSString` objects should be sufficient for use as keys. In some cases, however, it may be necessary to use custom objects as keys in a dictionary. When using custom objects as keys, there are some important points to keep in mind.

Keys must conform to 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. Instead, they copy each key argument and add the copy to the dictionary. After being copied into the dictionary, the dictionary-owned copies of the keys should not be modified.

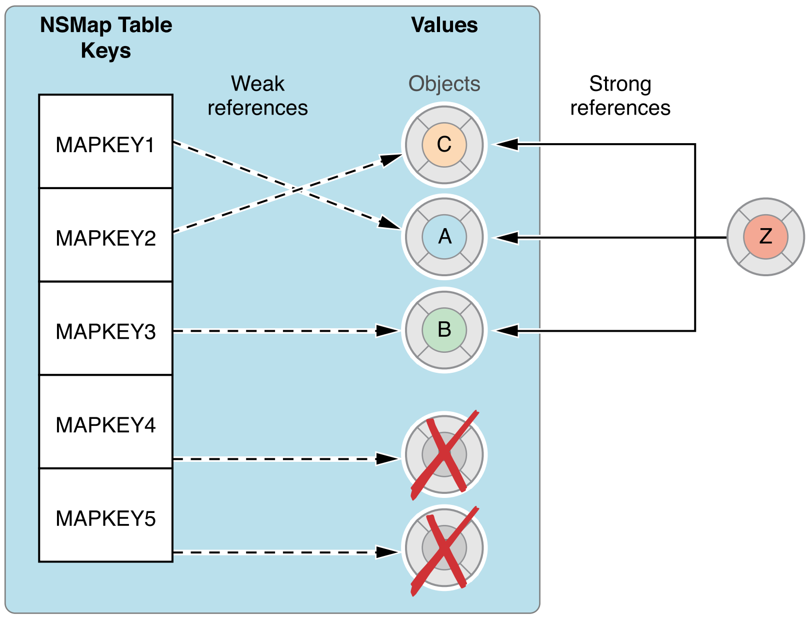
Keys must implement the `hash` and `isEqual:` methods because a dictionary uses a hash table to organize its storage and to quickly access contained objects. In addition, performance in a dictionary is heavily dependent on the hash function used. With a bad hash function, the decrease in performance can be severe. For more information on the `hash` and `isEqual:` methods see `NSObject`.

**Important:** Because the dictionary copies each key, keys must conform to the `NSCopying` protocol. Bear this in mind when choosing what objects to use as keys. Although you can use any object that adopts the `NSCopying` protocol and implements the `hash` and `isEqual:` methods, it is typically bad design to use large objects, such as instances of `NSImage`, because doing so may incur performance penalties.

## Using Map Tables

The `NSMapTable` class is configured by default to hold objects much like `NSMutableDictionary`. It also allows additional storage options that you can tailor for specific cases, such as when you need advanced memory management options, or when you want to hold a specific type of pointer. For example, the map table in Figure 2 is configured to hold weak references to its value objects. You can also specify whether you want to copy objects entered into the array.

**Figure 2** Map table object ownership



You can use an `NSMapTable` object when you want a collection of key-value pairs that uses weak references. For example, suppose you have a global map table that contains some objects. Because global objects are never collected, none of its contents can be deallocated unless they are held weakly. Map tables configured to hold objects weakly do not own their contents. If there are no strong references to objects within such a map table, those objects are deallocated. For example, the map table in Figure 2 (page 27) holds weak references to its contents. Object D and Object E will be deallocated, but the rest of the objects remain.

To create a map table, create or initialize it using `mapTableWithKeyOptions:valueOptions:` or `initWithKeyOptions:valueOptions:capacity:` and the appropriate pointer functions options. Alternatively, you can initialize it using `initWithKeyPointerFunctions:valuePointerFunctions:capacity:` and appropriate instances of `NSPointerFunctions`. For more information on the pointer functions options, see Pointer Function Options (page 47).

The `NSMapTable` class also defines a number of convenience constructors for creating a map table with strong or weak references to its contents. For example, `mapTableWithStrongToWeakObjects` creates a map table that holds strong references to its keys and weak references to its values. These convenience constructors should only be used if you are storing objects.

**Important:** Only the options listed in `NSMapTableOptions` guarantee that the rest of the API will work correctly—including copying, archiving, and fast enumeration. While other `NSPointerFunctions` options are used for certain configurations, such as to hold arbitrary pointers, not all combinations of the options are valid. With some combinations, the map table may not work correctly or may not even be initialized correctly.

To configure a map table to use arbitrary pointers, initialize it with both the `NSPointerFunctionsOpaqueMemory` and `NSPointerFunctionsOpaquePersonality` value options. Key and value options do not have to be the same. When using a map table to contain arbitrary pointers, the C function API for `void *` pointers should be used. For more information, see Managing Map Tables. For example, you can add a pointer to an `int` value using the approach shown in Listing 5. Note that the map table uses `NSString` objects as keys, and that keys are copied into the map table.

**Listing 5** Map table configured for non-object pointers

```
NSPointerFunctionsOptions keyOptions=NSPointerFunctionsStrongMemory |
     NSPointerFunctionsObjectPersonality | NSPointerFunctionsCopyIn;
NSPointerFunctionsOptions valueOptions=NSPointerFunctionsOpaqueMemory |
     NSPointerFunctionsOpaquePersonality;
```

```
NSMapTable *mapTable = [NSMapTable mapTableWithKeyOptions:keyOptions
        valueOptions:valueOptions];

NSString *key1 = @"Key1";
NSMapInsert(mapTable, key1, someIntPtr);
```

You can then access that integer by using the `NSMapGet` function.

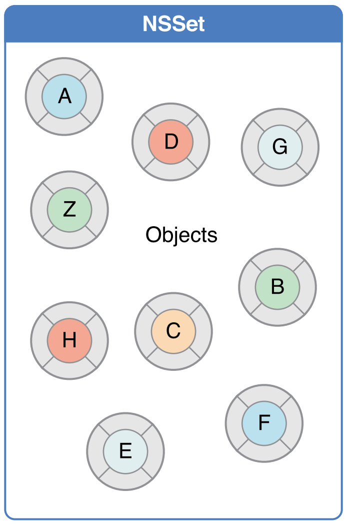
```
NSLog(@" Key1 contains: %i", *(int *) NSMapGet(mapTable, @"Key1"));
```

When configured to use arbitrary pointers, a map table has the risks associated with using pointers. For example, if the pointers refer to stack-based data created in a function, those pointers are not valid outside of the function, even if the map table is. Trying to access them will lead to undefined behavior.

# Sets: Unordered Collections of Objects

Objective-CSwift
A set is an unordered collection of objects, as shown in Figure 1. You can use sets as an alternative to arrays when the order of elements isn't important and performance of testing whether an object is in the set is important. Even though arrays are ordered, testing them for membership is slower than testing sets.

**Figure 1** Example set



## Set Fundamentals

An `NSSet` object manages an immutable set of distinct objects—that is, after you create the set, you cannot add, remove, or replace objects. You can, however, modify individual objects themselves (if they support modification). The mutability of the collection does not affect the mutability of the objects inside the collection. You should use an immutable set if the set rarely changes, or changes wholesale.

`NSMutableSet`, a subclass of `NSSet`, is a mutable set of distinct objects, which allows the addition and deletion of entries at any time, automatically allocating memory as needed. You should use a mutable set if the set changes incrementally, or is very large—as large collections take more time to initialize.

`NSCountedSet`, a subclass of `NSMutableSet`, is a mutable 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 set keeps a counter associated with each distinct object inserted. `NSCountedSet` objects keep track of the number of times objects are inserted and require that objects be removed the same number of times to completely remove the object from the set. Thus, there is only one instance of an object in a counted set, even if the object has been added multiple times. The `countForObject:` method returns the number of times the specified object has been added to this set.

The objects in a set must respond to the `NSObject` protocol methods `hash` and `isEqual:` (see `NSObject` for more information). 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. For example, a mutable dictionary can be put in a set, but you must not change it while it is in there. (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, a strong reference to the object is added to the set. For more information on copying and memory management, see Copying Collections (page 41).

Sets, excluding `NSCountedSet`, are the preferred collections if you want to be assured that no object is represented more than once, and there is no net effect for adding an object more than once.

**Note:** If the objects in the set have a good hash function, accessing an element, setting an element, and removing an element all take constant time. With a poor hash function (one that causes frequent hash collisions), these operations take up to linear time. Classes such as `NSString` that are part of Foundation have a good hash function.

## 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:`.

The `NSMutableSet` class 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.
- `unionSet:` adds all the objects from another set which are not already present.

The `NSMutableSet` class additionally provides these methods to remove objects from a set:

- `intersectSet:` removes all the objects which are not in another set.
- `removeAllObjects` removes all the objects from the set.

- `removeObject:` removes a particular object from the set.
- `minusSet:` removes all the objects which are in another set.

Because `NSCountedSet` is a subclass of `NSMutableSet`, it inherits all of these methods. However, some of them behave slightly differently with `NSCountedSet`. For example:

- `unionSet:` adds all the objects from another set, even if they are already present.
- `intersectSet:` removes all the objects which are not in another set. If there are multiple instances of the same object in either set, the resulting set contains that object as many times as the set with fewer instances.
- `minusSet:` removes all the objects which are in another set. If an object is present more than once in the counted set, it only removes one instance of it.

## Using Sets

The `NSSet` class provides methods for querying the elements of the set:

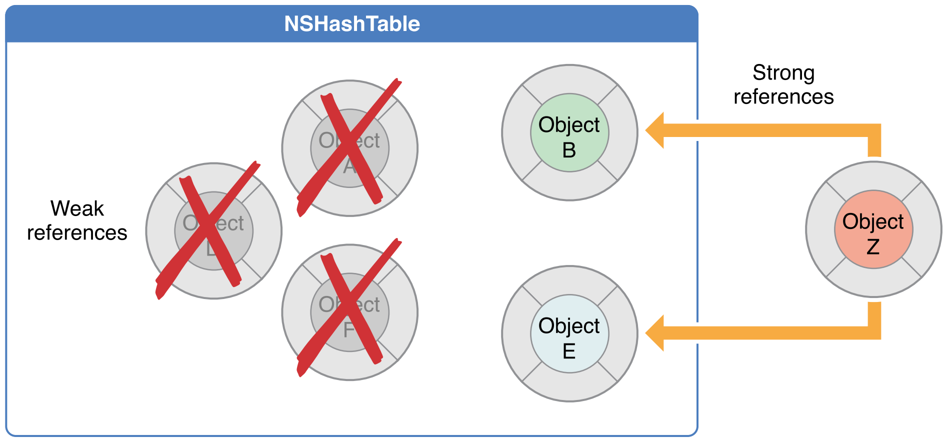
- `allObjects` returns an array containing the objects in a set.
- `anyObject` returns some object in the set. (The object is chosen at convenience *not* at random.)
- `count` returns the number of objects currently in the set.
- `member:` returns the object in the set that is equal to a specified object.
- `intersectsSet:` tests whether two sets share at least one object.
- `isEqualToSet:` tests whether two sets are equal.
- `isSubsetOfSet:` tests whether all of the objects contained in the set are also present in another set.

The `NSSet` method `objectEnumerator` lets you traverse elements of the set one by one. And the`makeObjectsPerformSelector:` and `makeObjectsPerformSelector:withObject:` methods provide for sending messages to individual objects in the set. In most cases, fast enumeration should be used because it is faster and more flexible than using an `NSEnumerator` or the `makeObjectsPerformSelector:` method. For more on enumeration, see Enumeration: Traversing a Collection's Elements (page 44).

## Hash Tables

The `NSHashTable` class is configured by default to hold objects much like `NSMutableSet` does. It also allows additional storage options that you can tailor for specific cases, such as when you need advanced memory management options, or when you want to hold a specific type of pointer. For example, the map table in Figure 2 is configured to hold weak references to its elements. You can also specify whether you want to copy objects entered into the set.

**Figure 2** Hash table object ownership



You can use an `NSHashTable` object when you want an unordered collection of elements that uses weak references. For example, suppose you have a global hash table that contains some objects. Because global objects are never collected, none of its contents can be deallocated unless they are held weakly. Hash tables configured to hold objects weakly do not own their contents. If there are no strong references to objects within such a hash table, those objects are deallocated. For example, the hash table in Figure 2 (page 33) holds weak references to its contents. Objects A, C, and Z will be deallocated, but the rest of the objects remain.

To create a hash table, initialize it using the `initWithOptions:capacity:` method and the appropriate pointer functions options. Alternatively, you can initialize it using `initWithPointerFunctions:capacity:` and appropriate instances of `NSPointerFunctions`. For more information on the various pointer functions options, see Pointer Function Options (page 47).

The `NSHashTable` class also defines the `hashTableWithWeakObjects` convenience constructor for creating a hash table with weak references to its contents. It should only be used if you are storing objects.

**Important:** Only the options listed in `NSHashTableOptions` guarantee that the rest of the API will work correctly—including copying archiving and fast enumeration. While other `NSPointerFunctions` options are used for certain configurations, such as to hold arbitrary pointers, not all combinations of the options are valid. With some combinations the hash table may not work correctly, or may not even be initialized correctly.

To configure a hash table to use arbitrary pointers, initialize it with both the `NSPointerFunctionsOpaqueMemory` and `NSPointerFunctionsOpaquePersonality` options. When using a hash table to contain arbitrary pointers, the C function API for `void *` pointers should be used. For more information, see Hash Tables. For example, you can add a pointer to an `int` value using the approach shown in Listing 1.

**Listing 1** Hash table configured for non-object pointers

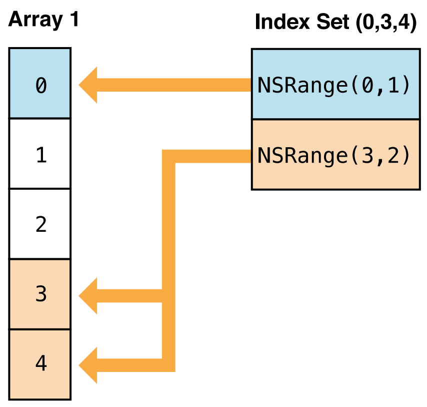
```
NSHashTable *hashTable=[[NSHashTable alloc] initWithOptions:
    NSPointerFunctionsOpaqueMemory |NSPointerFunctionsOpaquePersonality
    capacity: 1];

NSHashInsert(hashTable, someIntPtr);
```

When configured to use arbitrary pointers, a hash table has the risks associated with using pointers. For example, if the pointers refer to stack-based data created in a function, those pointers are not valid outside of the function, even if the hash table is. Trying to access them will lead to undefined behavior.

# Index Sets: Storing Indexes into an Array

You use index sets to store indexes into some other data structure, such as an `NSArray` object. Each index in an index set can only appear once, which is why index sets are not suitable for storing arbitrary collections of integers. Because index sets (as in Figure 1) make use of ranges to store indexes, they are usually more efficient than storing a collection of integer values, such as in an array.

**Figure 1** Index set and array interaction



## Index Set Fundamentals

An `NSIndexSet` object manages an immutable set of indexes—that is, after you create the index set, you cannot add indexes to it or remove indexes from it.

An `NSMutableIndexSet` object manages a mutable index set, which allows the addition and deletion of indexes at any time, automatically allocating memory as needed.

You can easily create an instance of one type of index set from the other using the initializer `initWithIndexSet:`. This is particularly useful if you want to create an immutable index set containing disjoint sets of indexes, which are typically created using mutable index sets. For example, if you have an `NSMutableIndexSet` object named `myIndexes`, which has had the indexes added to it, you can create an immutable copy as follows:

```
NSIndexSet *myImmutableIndexes=[[NSIndexSet alloc] initWithIndexSet: myIndexes];
```

You can also initialize an index set from a single index or a range of indexes by using the `initWithIndex:` or `initWithIndexesInRange:` method.

## Mutable Index Sets

The methods of the `NSMutableIndexSet` class allow you to add or remove additional indexes or index ranges. You can, for example, store disjoint sets of indexes and modify preexisting sets of indexes as needed. Some of these methods are listed below:

- `addIndex:`
- `addIndexesInRange:`
- `removeIndex:`
- `removeIndexesInRange:`

If you have an empty `NSMutableIndexSet` object named `myDisjointIndexes`, you can fill it with the indexes: 1, 2, 5, 6, 7, and 10, as shown in Listing 1.

**Listing 1** Adding indexes to a mutable index set

```
[myDisjointIndexes addIndexesInRange: NSMakeRange(1,2)];
[myDisjointIndexes addIndexesInRange: NSMakeRange(5,3)];
[myDisjointIndexes addIndex: 10];
```

## Iterating Through Index Sets

To access all of the objects indexed by an index set, it may be convenient to iterate sequentially through the index set. Iterating through the index set, rather than through the corresponding array, is more efficient, as it allows you to examine only the indexes that you are interested in. If you have an `NSArray` object named `anArray` and an `NSIndexSet` object named `anIndexSet`, you can iterate forward through an index set as shown in Listing 2.

**Listing 2** Forward iteration through an index set

```
NSUInteger index=[anIndexSet firstIndex];

while(index != NSNotFound)
```

```
{

     NSLog(@" %@",[anArray objectAtIndex:index]);
     index=[anIndexSet indexGreaterThanIndex: index];
}
```

Sometimes it may be necessary to iterate backward through an index set, for example, when you want to selectively remove objects from indexes from an `NSMutableArray` object. You can iterate backward through an index set as shown in Listing 3.

**Listing 3** Reverse iteration through an index set

```
NSUInteger index=[anIndexSet lastIndex];

while(index != NSNotFound)
{

     if([[aMutableArray objectAtIndex: index] isEqualToString:@"G"]){
          [aMutableArray removeObjectAtIndex:index];
     }
     index=[anIndexSet indexLessThanIndex: index];
}
```

The above approach should be used only if you want to selectively remove objects referred to by an index set. If you want to remove the objects at all the indexes in an index set, use `removeObjectsAtIndexes:` instead.

## Index Sets and Blocks

Index sets are especially powerful when used in conjunction with blocks. Blocks allow you to create index sets that designate the members of an array which pass some test. For example, if you have an unsorted array of numbers and you want to create an index set which holds indexes to all numbers less than 20, you use something similar to Listing 4.

**Listing 4** Creating an index set from an array using a block

```
NSIndexSet *lessThan20=[someArray indexesOfObjectsPassingTest:^(id obj, NSUInteger
 index, BOOL *stop){
     if ([obj isLessThan:[NSNumber numberWithInt:20]]){
          return YES;
     }
     return NO;
}];
```

Index sets can also be used in block-based enumeration of an array. To enumerate only the indexes of the array contained in the index set, use the `enumerateObjectsAtIndexes:options:usingBlock:` method.

Alternatively, the index set itself can be enumerated using a block with the `enumerateIndexesUsingBlock:` method. For example, you can perform some task for each object whose index is in the set. You can even access objects from multiple arrays provided the index set is valid for the arrays used, as in Listing 5.

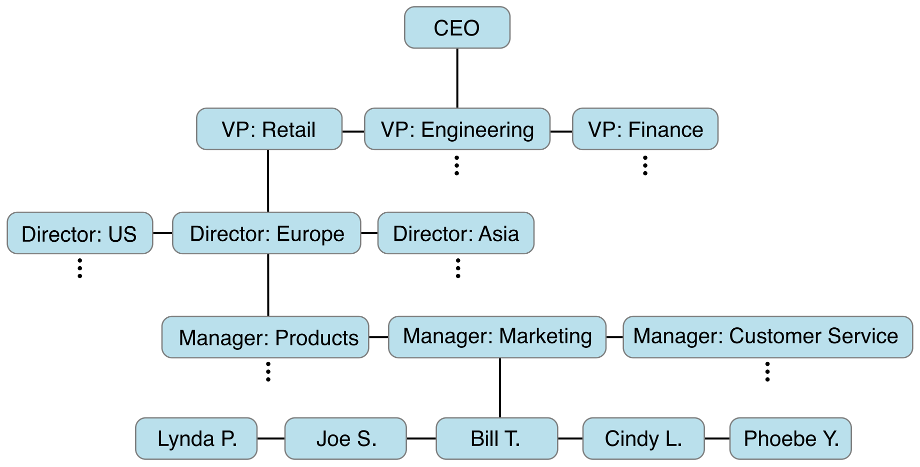
**Listing 5** Enumerating an index set to access multiple arrays

```
[anIndexSet enumerateIndexesUsingBlock:^(NSUInteger idx, BOOL *stop){
    if([[firstArray objectAtIndex: idx] isEqual:[secondArray objectAtIndex: idx]]){
          NSLog(@"Objects at %i Equal",idx);
     }
}];
```

# Index Paths: Storing a Path Through Nested Arrays

Index paths store the path through a nested set of arrays and are used to retrieve an object in a more complicated collection hierarchy, such as a tree. Figure 1, for example, shows a nested set of arrays which represents the hierarchy of a hypothetical company.

**Figure 1** Nested arrays and index paths



## Index Path Fundamentals

If you consider the hierarchy of the hypothetical company shown in Figure 1 (page 39), the root array consists of a single entry for the CEO. The array below that consists of the various vice presidents. Below each vice president is an array of directors, and so on. If you want to store the position of a particular employee on the Europe marketing team, for instance, a simple index is not enough. Instead a path through the nested arrays is necessary. In this case, Bill T. can be represented by the index path 0.0.1.1.2.

You can create an index path using from a single index or from a C-style array of `NSUInteger` values. Listing 1 shows how to create the index path to Bill T.

**Listing 1** Creating an index path from an array

```
NSUInteger arrayLength = 5;
NSUInteger integerArray[] = {0,0,1,1,2};
```

```
NSIndexPath *aPath = [[NSIndexPath alloc] initWithIndexes:integerArray
length:arrayLength];
```

You can also create an index path automatically from many of the more complex hierarchy collection classes. See the `indexPath` method of the `NSTreeNode` class for an example.

## Using Index Paths

`NSIndexPath` provides methods for querying the elements in the path. For example, `indexAtPosition:` returns the index stored at the given position in the index path. You can also create new index paths by adding a new index or removing the last index. A few classes use index paths extensively to manage their contents. `NSTreeController` is one such example. For more information on `NSTreeController` and index paths, see *Cocoa Bindings Programming Topics* .

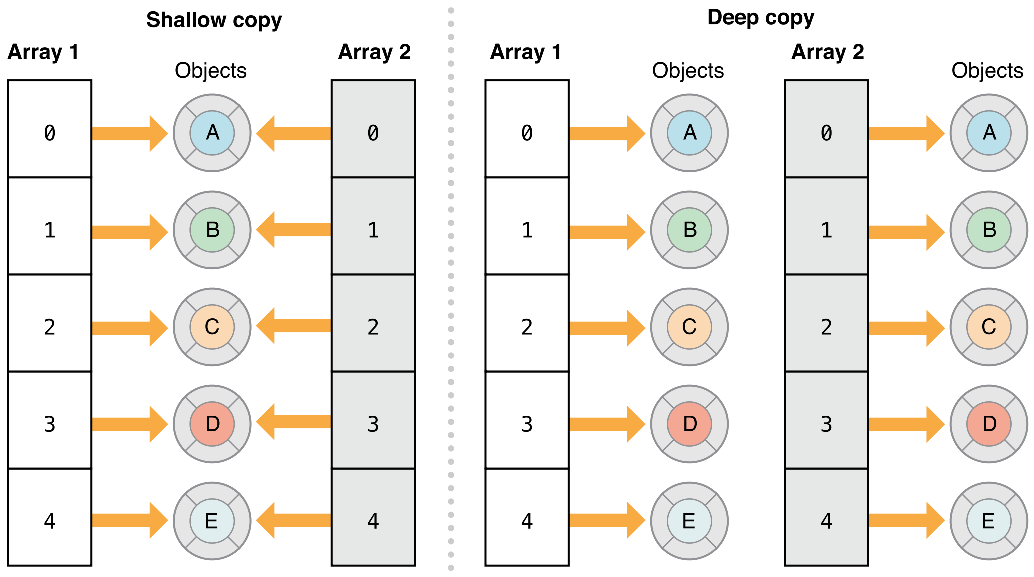
In iOS, `UITableView` and its delegate and data source use index paths to manage much of their content and to handle user interaction. To assist with this, UIKit adds programming interfaces to `NSIndexPath` to incorporate the rows and sections of a table view more fully into index paths. For more information, see *NSIndexPath UIKit Additions* . For instance, index paths are used to designate user selections using the `tableView:didSelectRowAtIndexPath:` delegate method.

# Copying Collections

Objective-CSwift

There are two kinds of object copying: shallow copies and deep copies. The normal copy is a shallow copy that produces a new collection that shares ownership of the objects with the original. Deep copies create new objects from the originals and add those to the new collection. This difference is illustrated by Figure 1.

**Figure 1** Shallow copies and deep copies



## Shallow Copies

There are a number of ways to make a shallow copy of a collection. When you create a shallow copy, the objects in the original collection are sent a `retain` message and the pointers are copied to the new collection. Listing 1 shows some of the ways to create a new collection using a shallow copy.

**Listing 1** Making a shallow copy

```
NSArray *shallowCopyArray = [someArray copyWithZone:nil];


NSDictionary *shallowCopyDict = [[NSDictionary alloc]
initWithDictionary:someDictionary copyItems:NO];
```

These techniques are not restricted to the collections shown. For example, you can copy a set with the `copyWithZone:` method—or the `mutableCopyWithZone:` method—or an array with `initWithArray:copyItems:` method.

## Deep Copies

There are two ways to make deep copies of a collection. You can use the collection's equivalent of `initWithArray:copyItems:` with `YES` as the second parameter. If you create a deep copy of a collection in this way, each object in the collection is sent a `copyWithZone:` message. If the objects in the collection have adopted the `NSCopying` protocol, the objects are deeply copied to the new collection, which is then the sole owner of the copied objects. If the objects do not adopt the `NSCopying` protocol, attempting to copy them in such a way results in a runtime error. However, `copyWithZone:` produces a shallow copy. This kind of copy is only capable of producing a one-level-deep copy. If you only need a one-level-deep copy, you can explicitly call for one as in Listing 2.

**Listing 2** Making a deep copy

```
NSArray *deepCopyArray=[[NSArray alloc] initWithArray:someArray copyItems:YES];
```

This technique applies to the other collections as well. Use the collection's equivalent of `initWithArray:copyItems:` with `YES` as the second parameter.

If you need a true deep copy, such as when you have an array of arrays, you can archive and then unarchive the collection, provided the contents all conform to the `NSCoding` protocol. An example of this technique is shown in Listing 3.

**Listing 3** A true deep copy

```
NSArray* trueDeepCopyArray = [NSKeyedUnarchiver unarchiveObjectWithData:
          [NSKeyedArchiver archivedDataWithRootObject:oldArray]];
```

## Copying and Mutability

When you copy a collection, the mutability of that collection or the objects it contains can be affected. Each method of copying has slightly different effects on the mutability of the objects in a collection of arbitrary depth:

- `copyWithZone:` makes the surface level immutable. All deeper levels have the mutability they previously had.
- `initWithArray:copyItems:` with `NO` as the second parameter gives the surface level the mutability of the class it is allocated as. All deeper levels have the mutability they previously had.
- `initWithArray:copyItems:` with `YES` as the second parameter gives the surface level the mutability of the class it is allocated as. The next level is immutable, and all deeper levels have the mutability they previously had.
- Archiving and unarchiving the collection leaves the mutability of all levels as it was before.

# Enumeration: Traversing a Collection’s Elements

Cocoa defines three main ways to enumerate the contents of a collection. These include fast enumeration and block-based enumeration. There is also the `NSEnumerator` class, though it has generally been superseded by fast enumeration.

## Fast Enumeration

Fast enumeration is the preferred method of enumerating the contents of a collection because it provides the following benefits:

- The enumeration is more efficient than using `NSEnumerator` directly.
- The syntax is concise.
- The enumerator raises an exception if you modify the collection while enumerating.
- You can perform multiple enumerations concurrently.

The behavior for fast enumeration varies slightly based on the type of collection. Arrays and sets enumerate their contents, and dictionaries enumerate their keys. `NSIndexSet` and `NSIndexPath` do not support fast enumeration. You can use fast enumeration with collection objects, as shown in Listing 1.

**Listing 1** Using fast enumeration with a dictionary

```
for (NSString *element in someArray) {
     NSLog(@"element: %@", element);
}

NSString *key;
for (key in someDictionary){
     NSLog(@"Key: %@, Value %@", key, [someDictionary objectForKey: key]);
}
```

For more information on fast enumeration, see Fast Enumeration in *The Objective-C Programming Language*.

## Using Block-Based Enumeration

`NSArray`, `NSDictionary`, and `NSSet` allow enumeration of their contents using blocks. To enumerate with a block, invoke the appropriate method and specify the block to use. Listing 2 demonstrates block-based enumeration for an `NSArray` object.

**Listing 2** Block-based enumeration of an array

```
NSArray *anArray = [NSArray arrayWithObjects:@"A", @"B", @"D", @"M", nil];
NSString *string = @"c";

[anArray enumerateObjectsUsingBlock:^(id obj, NSUInteger index, BOOL *stop){
     if ([obj localizedCaseInsensitiveCompare:string] == NSOrderedSame) {
          NSLog(@"Object Found: %@ at index: %i",obj, index);
          *stop = YES;
     }
} ];
```

For an `NSSet` object, you can use similar code, as shown in Listing 3.

**Listing 3** Block-based enumeration of a set

```
NSSet *aSet = [NSSet setWithObjects: @"X", @"Y", @"Z", @"Pi", nil];
NSString *aString = @"z";

[aSet enumerateObjectsUsingBlock:^(id obj, BOOL *stop){
     if ([obj localizedCaseInsensitiveCompare:aString]==NSOrderedSame) {
          NSLog(@"Object Found: %@", obj);
          *stop = YES;
     }
} ];
```

For `NSArray` enumeration, the `index` parameter is useful for concurrent enumeration. Without this parameter, the only way to access the index would be to use the `indexOfObject:` method, which is inefficient. The `stop` parameter is important for performance, because it allows the enumeration to stop early based on some condition determined within the block. The block-based enumeration methods for the other collections are slightly different in name and in block signature. See the respective class references for the method definitions.

## Using an Enumerator

`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.

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`. The `NSDictionary` class also has two methods that return an `NSEnumerator` object: `keyEnumerator` and `objectEnumerator`. These methods let you enumerate the contents of an `NSDictionary` object by key or by value, respectively.

In Objective-C, an `NSEnumerator` object retains the collection over which it's enumerating (unless it is 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 Listing 4.

**Listing 4** Enumerating a dictionary and removing objects

```
NSMutableDictionary *myMutableDictionary = <#Get a mutable dictionary#> ;
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];
```

# Pointer Function Options

The pointer collection classes (`NSPointerArray`, `NSMapTable`, and `NSHashTable`) allow you to further customize the collection to tailor it to your memory and storage needs. The options specified by `NSPointerFunctionsOptions` provide a convenient interface for customizing how the collection manages the pointers it contains.

## Pointer Collection Fundamentals

Pointer collections are configured using options from three different categories: memory options, personality options, and copying behavior. Not all combinations of memory, personality, and copying options are valid.

Memory options specify the expected behavior for when items are added to the collection, removed from the collection, or copied. A few of the more common options include:

- `NSPointerFunctionsStrongMemory`, which is used for a collection that holds strong references to its contents.
- `NSPointerFunctionsZeroingWeakMemory`, which is used for a collection that holds weak references to its contents.
- `NSPointerFunctionsOpaqueMemory`, which is used for cases when ownership of the contents is managed completely outside a collection. It is often used for collections that hold pointers to primitive types such as integers or C-strings.

Personality options specify the type of pointers stored in the collection, such as pointers to objects or pointers to other data types. They also specify what happens for hashing and equality tests. A few of the more common options include:

- `NSPointerFunctionsObjectPersonality`, which is used for a collection that holds objects and uses `isEqual:` to determine equality.
- `NSPointerFunctionsObjectPointerPersonality`, which is often used for a collection that holds objects and uses direct comparison to determine equality.
- `NSPointerFunctionsOpaquePersonality`, which is often used for a collection that holds pointers to primitive types such as integers or C-strings.

Copy options specify whether the collection should copy the elements entered into the collection. If the `NSPointerFunctionsCopyIn` option is specified, the collection copies the elements entered; otherwise, it does not.

If you need greater customization than the `NSPointerFunctionsOptions` allow, you can use the `NSPointerFunctions` class to define custom functions for operations like memory allocation, hashing, and equality testing. For example, if you have a collection of `struct`s, you would need to specify the size of the `struct`.

## Configuring Pointer Collections to Hold Objects

If you want to configure a pointer collection to hold objects, there are a few options. For objects, only two personality options make sense:

- `NSPointerFunctionsObjectPersonality`, the default object option, uses the `isEqual:` method for determining equality.
- `NSPointerFunctionsObjectPointerPersonality`, also a viable option, uses pointer equality to determine equality.

You can also choose to use either strong references or zeroing weak references. If you choose to use strong references, you can also choose whether you want objects to be copied when they are added to the collection.

For example, if you want a collection to hold weak references to objects and use `isEqual:` to determine equality, you can specify the options as follows:

```
NSPointerFunctionsOptions collectionOptions = NSPointerFunctionsObjectPersonality
            | NSPointerFunctionsZeroingWeakMemory;
```

After specifying the options, `collectionOptions` can then be passed to a collection during initialization.

## Configuring Pointer Collections for Arbitrary Pointer Use

If you want to configure a pointer collection to hold arbitrary (non-object) pointers, you have some flexibility to configure the collection based on the type of pointer the collection will hold. For the most flexibility, you can select the `NSPointerFunctionsOpaquePersonality`, which allows you to hold pointers to most primitive types. You can also select one of the type-specific options:

- `NSPointerFunctionsIntegerPersonality` holds integer pointers.

- `NSPointerFunctionsStructPersonality` holds pointers to structs. If you specify this option you must set the `sizeFunction` property of the `NSPointerFunctions` object that you use.
- `NSPointerFunctionsCStringPersonality` holds pointers to C-strings.

You should typically use `NSPointerFunctionsOpaqueMemory` when dealing with arbitrary pointers because it is compatible with all of the personality options. If you need to, you can use `NSPointerFunctionsMallocMemory` or `NSPointerFunctionsMachVirtualMemory` with the opaque, C-string, and struct personalities, although this is not typically recommended.

The only arbitrary pointer configurations that support copy-in behavior are the C-string and struct personalities when using either malloc or Mach virtual memory.

If you want a collection to hold arbitrary pointers using opaque memory, you can specify the options as follows:

```
NSPointerFunctionsOptions collectionOptions = NSPointerFunctionsOpaquePersonality
          | NSPointerFunctionsOpaqueMemory;
```

After specifying the options, `collectionOptions` can then be passed to a collection during initialization.

# Document Revision History

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

| Date | Notes |
|---|---|
| 2010-09-01 | Added information about index sets, index paths, hash tables, and map tables. |
| 2009-08-14 | 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 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. |