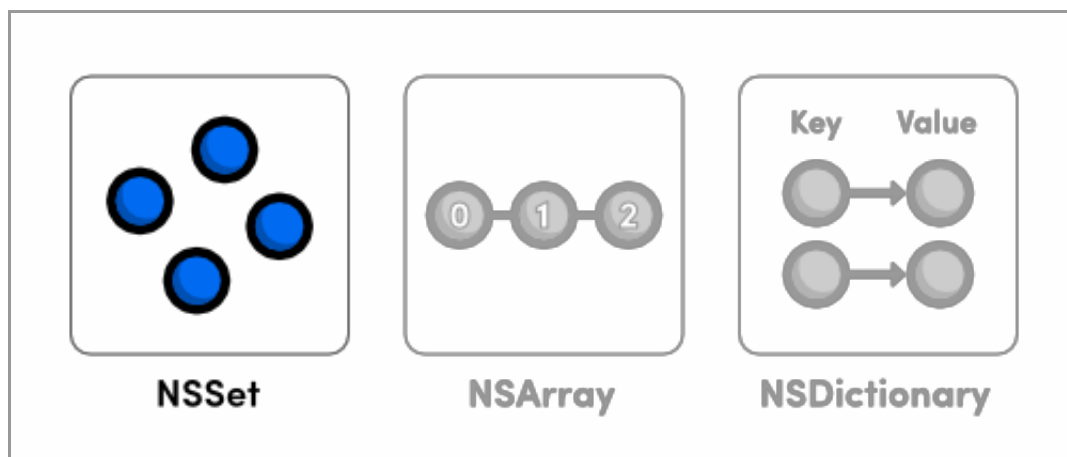


- [RyPress](#)
- [Tutorials](#)
- [Sponsors](#)
- [About](#)
- [Contact](#)

[◀ Back to Objective-C Data Types](#)

NSSet

`NSSet`, `NSArray`, and `NSDictionary` are the three core collection classes defined by the Foundation Framework. An `NSSet` object represents a static, unordered collection of distinct objects. Sets are optimized for membership checking, so if you're asking a lot of "is this object part of this group?" kind of questions, you should be using a set—not an [array](#).



The basic collection classes of the Foundation Framework

Collections can only interact with Objective-C objects. As a result, primitive C types like `int` need to be wrapped in an [NSNumber](#) before you can store them in a set, array, or dictionary.

`NSSet` is immutable, so you cannot add or remove elements from a set after it's been created. You can, however, alter mutable objects that are contained in the set. For example, if you stored an `NSMutableString`, you're free to call `setString:`, `appendFormat:`, and the other manipulation methods on that object. This module also covers [NSMutableSet](#) and [NSCountedSet](#).

Creating Sets

An `NSSet` can be created through the `setWithObjects:` class method, which accepts a `nil`-terminated list of objects. Most of the examples in this module utilize strings, but an `NSSet` instance can record *any* kind of Objective-C object, and it does not have to be homogeneous.

```
NSSet *americanMakes = [NSSet setWithObjects:@"Chrysler", @"Ford",
                                             @"General Motors", nil];

NSLog(@"%@", americanMakes);
```

`NSSet` also includes a `setWithArray:` method, which turns an `NSArray` into an `NSSet`. Remember that sets are composed of *unique* elements, so this serves as a convenient way to remove all duplicates in an array. For example:

```
NSArray *japaneseMakes = @[@"Honda", @"Mazda",
                           @"Mitsubishi", @"Honda"];

NSSet *uniqueMakes = [NSSet setWithArray:japaneseMakes];

NSLog(@"%@", uniqueMakes);    // Honda, Mazda, Mitsubishi
```

Sets maintain a [strong relationship](#) with their elements. That is to say, a set owns each item that it contains. You should be careful to avoid retain cycles when creating sets of custom objects by ensuring that an element in the set never has a strong reference to the set itself.

Enumerating Sets

Fast-enumeration is the preferred method of iterating through the contents of a set, and the `count` method can be used to calculate the total number of items. For example:

```
NSSet *models = [NSSet setWithObjects:@"Civic", @"Accord",
                                       @"Odyssey", @"Pilot", @"Fit", nil];

NSLog(@"The set has %li elements", [models count]);

for (id item in models) {
    NSLog(@"%@", item);
}
```

```
}
```

If you're interested in a [block](#)-based solution, you can also use the `enumerateObjectsUsingBlock:` method to process the contents of a set. The method's only parameter is a `^(id obj, BOOL *stop)` block. `obj` is the current object, and the `*stop` pointer lets you prematurely exit the iteration by setting its value to `YES`, as demonstrated below.

```
[models enumerateObjectsUsingBlock:^(id obj, BOOL *stop) {  
    NSLog(@"Current item: %@", obj);  
    if ([obj isEqualToString:@"Fit"]) {  
        NSLog(@"I was looking for a Honda Fit, and I found it!");  
        *stop = YES;    // Stop enumerating items  
    }  
}];
```

The `*stop = YES` line tells the set to stop enumerating once it reaches the `@"Fit"` element. This is the block equivalent of the `break` statement.

Note that since sets are unordered, it usually doesn't make sense to access an element outside of an enumeration. Accordingly, `NSSet` does *not* support subscripting syntax for accessing individual elements (e.g., `models[i]`). This is one of the primary differences between sets and [arrays/dictionaries](#).

Comparing Sets

In addition to equality, two `NSSet` objects can be checked for subset and intersection status. All three of these comparisons are demonstrated in the following example.

```
NSSet *japaneseMakes = [NSSet setWithObjects:@"Honda", @"Nissan",  
                                           @"Mitsubishi", @"Toyota", nil];  
NSSet *johnsFavoriteMakes = [NSSet setWithObjects:@"Honda", nil];  
NSSet *marysFavoriteMakes = [NSSet setWithObjects:@"Toyota",  
                                                  @"Alfa Romeo", nil];  
  
if ([johnsFavoriteMakes isEqualToSet:japaneseMakes]) {  
    NSLog(@"John likes all the Japanese auto makers and no others");  
}
```

```
}  
if ([johnsFavoriteMakes intersectsSet:japaneseMakes]) {  
    // You'll see this message  
    NSLog(@"John likes at least one Japanese auto maker");  
}  
if ([johnsFavoriteMakes isSubsetOfSet:japaneseMakes]) {  
    // And this one, too  
    NSLog(@"All of the auto makers that John likes are Japanese");  
}  
if ([marysFavoriteMakes isSubsetOfSet:japaneseMakes]) {  
    NSLog(@"All of the auto makers that Mary likes are Japanese");  
}
```

Membership Checking

Like all Foundation Framework collections, it's possible to check if an object is in a particular NSSet. The `containsObject:` method returns a `BOOL` indicating the membership status of the argument. As an alternative, the `member:` returns a reference to the object if it's in the set, otherwise `nil`. This can be convenient depending on how you're using the set.

```
NSSet *selectedMakes = [NSSet setWithObjects:@"Maserati",  
                                           @"Porsche", nil];  
  
// BOOL checking  
if ([selectedMakes containsObject:@"Maserati"]) {  
    NSLog(@"The user seems to like expensive cars");  
}  
  
// nil checking  
NSString *result = [selectedMakes member:@"Maserati"];  
if (result != nil) {  
    NSLog(@"%@ is one of the selected makes", result);  
}
```

Again, this is one of the strong suits of sets, so if you're doing a lot of membership checking, you should be using NSSet instead of NSArray (unless you have a compelling reason not to).

Filtering Sets

You can filter the contents of a set using the `objectsPassingTest:` method, which accepts a block that is called using each item in the set. The block should return YES if the current object should be included in the new set, and NO if it shouldn't. The following example finds all items that begin with an uppercase letter C.

```
NSMutable *toyotaModels = [NSMutable setWithObjects:@"Corolla", @"Sienna",
                                                    @"Camry", @"Prius",
                                                    @"Highlander", @"Sequoia", nil];

NSMutable *cModels = [toyotaModels objectsPassingTest:^(BOOL id obj, BOOL *stop) {
    if ([obj hasPrefix:@"C"]) {
        return YES;
    } else {
        return NO;
    }
}];

NSLog(@"%@", cModels);    // Corolla, Camry
```

Because NSMutable is immutable, the `objectsPassingTest:` method returns a *new* set instead of altering the existing one. This is the same behavior as many of the NSString manipulation operations. But, while the *set* is a new instance, it still refers to the same *elements* as the original set. That is to say, filtered elements are not copied—they are referenced.

Combining Sets

Sets can be combined using the `setByAddingObjectsFromSet:` method. Since sets are unique, duplicates will be ignored if both sets contain the same object.

```
NSMutable *affordableMakes = [NSMutable setWithObjects:@"Ford", @"Honda",
                                                        @"Nissan", @"Toyota", nil];

NSMutable *fancyMakes = [NSMutable setWithObjects:@"Ferrari", @"Maserati",
                                                  @"Porsche", nil];

NSMutable *allMakes = [affordableMakes setByAddingObjectsFromSet:fancyMakes];

NSLog(@"%@", allMakes);
```

NSMutableSet

Mutable sets allow you to add or delete objects dynamically, which affords a whole lot more flexibility than the static `NSSet`. In addition to membership checking, mutable sets are also more efficient at inserting and removing elements than `NSMutableArray`.

`NSMutableSet` can be very useful for recording the state of a system. For example, if you were writing an application to manage an auto repair shop, you might maintain a mutable set called `repairedCars` and add/remove cars to reflect whether or not they have been fixed yet.

Creating Mutable Sets

Mutable sets can be created with the exact same methods as `NSSet`. Or, you can create an empty set with the `setWithCapacity:` class method. The argument defines the initial amount of space allocated for the set, but it in no way limits the number of items it can hold.

```
NSMutableSet *brokenCars = [NSMutableSet setWithObjects:
                            @"Honda Civic", @"Nissan Versa", nil];
NSMutableSet *repairedCars = [NSMutableSet setWithCapacity:5];
```

Adding and Removing Objects

The big additions provided by `NSMutableSet` are the `addObject:` and `removeObject:` methods. Note that `addObject:` won't actually do anything if the object is already a member of the collection because sets are composed of unique items.

```
NSMutableSet *brokenCars = [NSMutableSet setWithObjects:
                            @"Honda Civic", @"Nissan Versa", nil];
NSMutableSet *repairedCars = [NSMutableSet setWithCapacity:5];
// "Fix" the Honda Civic
[brokenCars removeObject:@"Honda Civic"];
```

```
[repairedCars addObject:@"Honda Civic"];

NSLog(@"Broken cars: %@", brokenCars);    // Nissan Versa
NSLog(@"Repaired cars: %@", repairedCars); // Honda Civic
```

Just like mutable strings, NSMutableSet has a different workflow than the static NSSet. Instead of generating a new set and re-assigning it to the variable, you can operate directly on the existing set.

You may also find the removeAllObjects method useful for completely clearing a set.

Filtering With Predicates

There is no mutable version of the objectsPassingTest: method, but you can still filter items with filterUsingPredicate:. Predicates are somewhat outside the scope of this tutorial, but suffice it to say that they are designed to make it easier to define search/filter rules. Fortunately, the NSPredicate class can be initialized with a block, so we don't need to learn an entirely new format syntax.

The following code snippet is the mutable, predicate-based version of the example from the [Filtering Sets](#) section above. Again, this operates directly on the existing set.

```
NSMutableSet *toyotaModels = [NSMutableSet setWithObjects:@"Corolla", @"Sienna",
                                                         @"Camry", @"Prius",
                                                         @"Highlander", @"Sequoia", nil];

NSPredicate *startsWithC = [NSPredicate predicateWithBlock:
    ^BOOL(id evaluatedObject, NSDictionary *bindings) {
        if ([evaluatedObject hasPrefix:@"C"]) {
            return YES;
        } else {
            return NO;
        }
    }
];

[toyotaModels filterUsingPredicate:startsWithC];
NSLog(@"%@", toyotaModels);    // Corolla, Camry
```

For more information about predicates, please visit the official [Predicate Programming Guide](#).

Set Theory Operations

`NSMutableSet` also provides an API for the basic operations in set theory. These methods let you take the union, intersection, and relative complement of two sets. In addition, the `setSet:` method is also useful for creating a shallow copy of a different set. All of these are included in the following example.

```
NSMutableSet *japaneseMakes = [NSMutableSet setWithObjects:@"Honda", @"Nissan",
                                                           @"Mitsubishi", @"Toyota", nil];
NSMutableSet *johnsFavoriteMakes = [NSMutableSet setWithObjects:@"Honda", nil];
NSMutableSet *marysFavoriteMakes = [NSMutableSet setWithObjects:@"Toyota",
                                                                @"Alfa Romeo", nil];

NSMutableSet *result = [NSMutableSet setWithCapacity:5];
// Union
[result setSet:johnsFavoriteMakes];
[result unionSet:marysFavoriteMakes];
NSLog(@"Either John's or Mary's favorites: %@", result);
// Intersection
[result setSet:johnsFavoriteMakes];
[result intersectSet:japaneseMakes];
NSLog(@"John's favorite Japanese makes: %@", result);
// Relative Complement
[result setSet:japaneseMakes];
[result minusSet:johnsFavoriteMakes];
NSLog(@"Japanese makes that are not John's favorites: %@",
      result);
```

Enumeration Considerations

Iterating over a mutable set works the same as a static set, with one very important caveat: you aren't allowed to change the set while you're enumerating it. This is a general rule for any collection class.

The following example demonstrates the *wrong* way to mutate a set in the middle of a for-in loop. We'll be using the rather contrived scenario of removing @"Toyota" if any element in the set begins with the letter T.


```
// DO NOT DO THIS. EVER.
NSMutableSet *makes = [NSMutableSet setWithObjects:@"Ford", @"Honda",
                                                  @"Nissan", @"Toyota", nil];
for (NSString *make in makes) {
    NSLog(@"%@", make);
    if ([make hasPrefix:@"T"]) {
        // Throws an NSGenericException:
        // "Collection was mutated while being enumerated"
        [makes removeObject:@"Toyota"];
    }
}
```

The proper way to do this is shown below. Instead of iterating over the set directly, you should create a temporary copy of it with the `allObjects` method and iterate over that. This frees you to alter the original set without any unintended consequences:

```
NSMutableSet *makes = [NSMutableSet setWithObjects:@"Ford", @"Honda",
                                                  @"Nissan", @"Toyota", nil];
NSArray *snapshot = [makes allObjects];
for (NSString *make in snapshot) {
    NSLog(@"%@", make);
    if ([make hasPrefix:@"T"]) {
        [makes removeObject:@"Toyota"];
    }
}
NSLog(@"%@", makes);
```

NSCountedSet

The `NSCountedSet` class (also called a “bag”) is worth a brief mention. It’s a subclass of `NSMutableSet`, but instead of being limited to *unique* values, it counts the number of times an object has been added to the collection. This is a very efficient way to keep object tallies, as it requires only one instance of an object regardless of how many times it’s been added to the bag.

The main difference between a mutable set and `NSMutableSet` is the `countForObject:` method. This will often be used in place of `containsObject:` (which still works as expected). For example:

```
NSMutableSet *inventory = [NSMutableSet setWithCapacity:5];
[inventory addObject:@"Honda Accord"];
[inventory addObject:@"Honda Accord"];
[inventory addObject:@"Nissan Altima"];
NSLog(@"There are %li Accords in stock and %li Altima",
      [inventory countForObject:@"Honda Accord"], // 2
      [inventory countForObject:@"Nissan Altima"]); // 1
```

Please see the [official documentation](#) for more details.

[Continue to NSMutableArray ›](#)

- © 2012–2013 RyPress.com
- All Rights Reserved
- Terms of Service