NSArray Class Cluster

lass Cluster Description

The NSArray class clusters manage arrays of objects. The cluster's two public classes, NSArray and NSMutableArray, declare the programmatic interface for static and dynamic arrays, respectively.

The objects you create using these classes are referred to as *arrays*. Because of the nature of class clusters, arrays are not actual instances of the NSArray or NSMutableArray classes but of one of their private subclasses. Although an array's class is private, its interface is public, as declared by these abstract superclasses, NSArray and NSMutableArray.

Generally, you instantiate an array by sending one of the **array...** messages to either the NSArray or NSMutableArray class object. These methods return an array containing the elements you pass in as arguments. (Note that arrays can't contain **nil**.) In general, objects that you add to an array aren't copied; rather, each object receives a **retain** message before its **id** is added to the array. When an object is removed from an array, it's sent a **release** message.

The NSArray class adopts the NSCopying and NSMutableCopying protocols, making it convenient to convert an array of one type to the other.

NSArray

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSMutableCopying NSObject (NSObject)
Declared In:	Foundation/NSArray.h

Class at a Glance

Purpose

An NSArray stores an immutable array of objects.

Principal Attributes

- A count of the number of objects in the array.
- The set of objects contained in the array.

Creation

+ array	Returns an empty array.
+ arrayWithArray:	Returns an array containing the elements from another array.
+ arrayWithContentsOfFile:	Returns an array initialized from the contents of a file.
+ arrayWithObject:	Returns an array containing a single object.
+ arrayWithObjects:	Returns an array containing multiple objects.
+ arrayWithObjects:count:	Returns an array containing a specified number of objects.

Commonly Used Methods

– count:	Returns the number of objects currently in the array.
– objectAtIndex:	Returns the object located at the specified index.

Primitive Methods

– count
– objectAtIndex:

lass Description

NSArray declares the programmatic interface to an object that manages an unchanging array of objects. 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.

The methods **objectEnumerator** and **reverseObjectEnumerator** also grant sequential access to the elements of the array, differing only in the direction of travel through the elements. These methods are provided so that arrays can be traversed in a manner similar to that used for objects of other collection classes such as NSDictionary. See the **objectEnumerator** method description for a code exerpt that shows how to use these methods to access the elements of an array.

NSArray provides methods for querying the elements of the array. **indexOfObject:** searches the array for the object that matches its argument. To determine whether the search is successful, each element of the array is sent an **isEqual:** message, as declared in the NSObject protocol. Another method, **indexOfObjectIdenticalTo:**, is provided for the less common case of determining whether a specific object is present in the array. **indexOfObjectIdenticalTo:** tests each element in the array to see whether its **id** matches that of the argument.

NSArray's **makeObjectsPerform:** and **makeObjectsPerform:withObject:** methods let you send messages to all objects in the array. To act on the array as a whole, a variety of other methods are defined. You can create a sorted version of the array (**sortedArrayUsingSelector:** and **sortedArrayUsingFunction:context:**), extract a subset of the array (**subarrayWithRange:**), or concatenate the elements of an array of NSStrings into a single string (**componentsJoinedByString:**). In addition, you can compare two arrays using the **isEqualToArray:** and **firstObjectCommonWithArray:** methods. Finally, you can create new arrays that contain the objects in an existing array and one or more additional objects with **arrayByAddingObject:** and **arrayByAddingObjectsFromArray:**.

dopted Protocols

NSCoding	– encodeWithCoder: – initWithCoder:
NSCopying	– copyWithZone:
NSMutableCopying	– mutableCopyWithZone:

Method Types

Creating an array	+ allocWithZone
	+ array
	+ arrayWithArray:
	+ arrayWithContentsOfFile:
	+ arrayWithObject:
	+ arrayWithObjects:
	+ arrayWithObjects:count:
	– initWithArray:
	– initWithContentsOfFile:
	– initWithObjects:
	– initWithObjects:count:
Querying the array	– containsObject:
	– count
	– getObjects:
	– getObjects:range:
	– indexOfObject:
	– indexOfObject:inRange:
	– indexOfObjectIdenticalTo:
	– indexOfObjectIdenticalTo:inRange:
	– lastObject
	– objectAtIndex:
	– objectEnumerator
	– reverseObjectEnumerator
Sending messages to elements	– makeObjectsPerform:
	– makeObjectsPerform:withObject:
	– makeObjectsPerformSelector:
	– makeObjectsPerformSelector:withObject:
Comparing arrays	– firstObjectCommonWithArray:
	– isEqualToArray:
Deriving new arrays	– arrayByAddingObject:
	– arrayByAddingObjectsFromArray:
	– sortedArrayHint
	– sortedArrayUsingFunction:context:
	– sortedArrayUsingFunction:context:hint:
	– sortedArrayUsingSelector:
	– subarrayWithRange:
Working with string elements	– componentsJoinedByString:
	– pathsMatchingExtensions:

Creating a description of the array	– description
	– descriptionWithLocale:
	– descriptionWithLocale:indent:
	– writeToFile:atomically:

Class Methods

allocWithZone:

+ (id)**allocWithZone:**(NSZone *)*zone*

Creates and returns an uninitialized array in the specified zone. If the receiver is the NSArray class object, an instance of an immutable private subclass is returned; otherwise, an object of the receiver's class is returned.

Typically, you create temporary arrays using the **array...** class methods, not the **allocWithZone:** and **init...** methods. Note that it's your responsibility to free objects created with the **allocWithZone:** method.

array

+ (id)**array**

Creates and returns an empty array. This method is used by mutable subclasses of NSArray.

See also: + **arrayWithObject:**, + **arrayWithObjects:**

arrayWithArray:

+ (id)arrayWithArray:(NSArray *)*anArray*

Creates and returns an array containing the objects in *anArray.*

See also: + **arrayWithObjects:**, **– initWithObjects:**

arrayWithContentsOfFile:

+ (id)**arrayWithContentsOfFile:**(NSString *)*aPath*

Creates and returns an array containing the contents of the file specified by *aPath*. The file identified by *aPath* must contain a string representation produced by the **writeToFile:atomically:** method. In addition, the array representation must contain only property list objects (NSString, NSData, NSArray, or NSDictionary objects).

Returns **nil** if the file can't be opened or if the contents of the file can't be parsed into an array.

arrayWithObject:

+ (id)**arrayWithObject:**(id)*anObject*

Creates and returns an array containing the single element *anObject*.

See also: + **array**, + **arrayWithObjects:**

arrayWithObjects:

+ (id)**arrayWithObjects:**(id)*firstObj*, ...

Creates and returns an array containing the objects in the argument list. The argument list is a comma-separated list of objects ending with **nil**.

This code example creates an array containing three different types of elements (assuming *aPath* exists):

```
NSArray *myArray;
NSData *someData = [NSData dataWithContentsOfFile:aPath];
NSValue *aValue = [NSNumber numberWithInt:5];
NSString *aString = @"a string";

myArray = [NSArray arrayWithObjects:someData, aValue, aString, nil];
```

See also: + **array**, + **arrayWithObject:**

arrayWithObjects:count:

+ (id)**arrayWithObjects:**(id *)*objects* **count:**(unsigned)*count*

Creates and returns an array containing *count* objects from *objects*.

See also: – **getObjects:**, – **getObjects:range:**

nstance Methods

arrayByAddingObject:

– (NSArray *)**arrayByAddingObject:**(id)*anObject*

Returns a new array that is a copy of the receiver with *anObject* added to the end. Since *anObject* is added to the array, it receives a **retain** message. If *anObject* is **nil**, an NSInvalidArgumentException is raised.

See also: – **addObject:** (NSMutableArray)

arrayByAddingObjectsFromArray:

– (NSArray *)**arrayByAddingObjectsFromArray:**(NSArray *)*otherArray*

Returns a new array that is a copy of the receiver with the objects contained in *otherArray* added to the end.

See also: – **addObjectsFromArray:** (NSMutableArray)

componentsJoinedByString:

– (NSString *)**componentsJoinedByString:**(NSString *)*separator*

Constructs and returns an NSString that is the result of interposing *separator* between the elements of the receiver's array. For example, this code excerpt causes *myTextObject* to display the path **/NextDeveloper/Examples** (assuming *stream* exists):

```
NSArray *pathArray = [NSArray arrayWithObjects:@"NextDeveloper",
    @"Examples", nil];
NSLog("The path is /%@.\n",
    [pathArray componentsJoinedByString:@"/"]);
[myTextObject readText:stream];
```

Each element of the receiver's array must be an NSString or an error occurs. If the receiver has no elements, an NSString representing an empty string is returned.

See also: – **componentsSeparatedByString:** (NSString)

containsObject:

– (BOOL)**containsObject:**(id)*anObject*

Returns YES if *anObject* is present in the array. This method determines whether an object is prsent in the array by sending an **isEqual:** message to each of the array's objects (and passing *anObject* as the parameter to each **isEqual:** message).

See also: – **indexOfObject:**, – **indexOfObjectIdenticalTo:**, – **isEqual:** (NSObject)

count

– (unsigned int)**count**

Returns the number of objects currently in the array.

See also: – **objectAtIndex:**

description

@protocol NSObject
– (NSString *)**description**

Returns a string that represents the contents of the receiver, formatted as a property list.

See also: **– descriptionWithLocale:, – descriptionWithLocale:indent:**

descriptionWithLocale:

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*locale*

Returns a string that represents the contents of the receiver, formatted as a property list. *locale* specifies options used for formatting each of the receiver's elements (where recognized); specify **nil** if you don't want the elements formatted.

For a description of how *locale* is applied to each element in the receiving array, see **descriptionWithLocale:indent:**.

See also: **– description, – descriptionWithLocale:indent:**

descriptionWithLocale:indent:

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*locale* **indent:**(unsigned int)*level*

Returns a string that represents the contents of the receiver, formatted as a property list. *locale* specifies options used for formatting each of the receiver's elements; specify **nil** if you don't want the elements formatted. *level* allows you to specify a level of indent, to make the output more readable: set *level* to 0 to use four spaces to indent, or 1 to indent the output with a tab character.

The returned NSString contains the string representations of each of the receiver's elements, in order, from first to last. To obtain the string representation of a given element, **descriptionWithLocale:indent:** proceeds as follows:

- If the element is an NSString, it is used as is.
- If the element responds to **descriptionWithLocale:indent:**, that method is invoked to obtain the element's string representation.
- If the element responds to **descriptionWithLocale:**, that method is invoked to obtain the element's string representation.
- If none of the above conditions are met, the element's string representation is obtained by invoking its **description** method.

See also: **– description, – descriptionWithLocale:**

firstObjectCommonWithArray:

– (id)**firstObjectCommonWithArray:**(NSArray *)*otherArray*

Returns the first object contained in the receiver that's equal to an object in *otherArray*. If no such object is found, this method returns **nil**. This method uses **isEqual:** to check for object equality.

See also: – **containsObject:**, – **isEqual:** (NSObject)

getObjects:

– (void)**getObjects:**(id *)*aBuffer*

Copies the objects contained in the receiver to *aBuffer*.

See also: + **arrayWithObjects:count:**

getObjects:range:

– (void)**getObjects:**(id *)*aBuffer* **range:**(NSRange)*aRange*

Copies the objects contained in the receiver that fall within the specified range to *aBuffer*.

See also: + **arrayWithObjects:count:**

hash

@protocol NSObject
– (unsigned int)**hash**

Returns an unsigned integer that can be used as a table address in a hash table structure. For an array, **hash** returns the number of elements in the array. If two arrays are equal (as determined by the **isEqual:** method), they will have the same hash value.

See also: – **isEqual:** (NSObject)

indexOfObject:

– (unsigned int)**indexOfObject:**(id)*anObject*

Searches the receiver for *anObject* and returns the lowest index whose corresponding array value is equal to *anObject*. Objects are considered equal if they have the same **id** or if **isEqual:** returns YES. If none of the objects in the receiver are equal to *anObject*, **indexOfObject:** returns NSNotFound.

See also: – **containsObject:**, – **indexOfObjectIdenticalTo:**, – **isEqual:** (NSObject)

indexOfObject:inRange:

– (unsigned)**indexOfObject:**(id)*anObject* **inRange:**(NSRange)*aRange*

Searches the specified range within the receiver for *anObject* and returns the lowest index whose corresponding array value is equal to *anObject*. Objects are considered equal if they have the same **id** or if **isEqual:** returns YES. If none of the objects in the specified range are equal to *anObject*, **indexOfObject:** returns NSNotFound.

See also: – **containsObject:**, – **indexOfObjectIdenticalTo:**, – **isEqual:** (NSObject)

indexOfObjectIdenticalTo:

– (unsigned int)**indexOfObjectIdenticalTo:**(id)*anObject*

Searches the receiver for *anObject* (testing for equality by comparing object **id**s) and returns the lowest index whose corresponding array value is equal to *anObject*. If none of the objects in the receiver are equal to *anObject*, **indexOfObject:** returns NSNotFound.

See also: – **containsObject:**, – **indexOfObject:**, – **isEqual:** (NSObject)

indexOfObjectIdenticalTo:inRange:

– (unsigned)**indexOfObjectIdenticalTo:**(id)*anObject* **inRange:**(NSRange)*aRange*

Searches the specified range within the receiver for *anObject* (testing for equality by comparing object **id**s) and returns the lowest index whose corresponding array value is equal to *anObject*. If none of the objects in the specified range are equal to *anObject*, **indexOfObject:** returns NSNotFound.

See also: – **containsObject:**, – **indexOfObject:**, – **isEqual:** (NSObject)

initWithArray:

– (id)**initWithArray:**(NSArray *)*anArray*

Initializes a newly allocated array by placing in it the objects contained in *array*. Each object in *array* receives a **retain** message as it’s added to the array. After an immutable array has been initialized in this way, it can’t be modified. Returns **self**.

See also: + **arrayWithObject:**, – **initWithObjects:**

initWithContentsOfFile:

– (id)**initWithContentsOfFile:**(NSString *)*aPath*

Initializes a newly allocated array with the contents of the file specified by *aPath*. The file identified by *aPath* must contain a string representation produced by the **writeToFile:atomically:** method. In addition, the array representation must contain only property list objects (NSString, NSData, NSArray, or NSDictionary objects).

Returns **self** if the receiver is successfully initialized, or **nil** if the file can’t be opened or if the contents of the file can’t be parsed into an array.

See also: – **writeToFile:atomically:**

initWithObjects:

– (id)**initWithObjects:**(id)*firstObj*, ...

Initializes a newly allocated array by placing in it the objects in the argument list. This list is a comma-separated list of objects ending with **nil**. Objects are retained as they’re added to the array. After an immutable array has been initialized in this way, it can’t be modified. Returns **self**.

See also: – **initWithObjects:count:**, + **arrayWithObjects:**, – **initWithArray:**

initWithObjects:count:

– (id)**initWithObjects:**(id *)*objects* **count:**(unsigned int)*count*

Initializes a newly allocated array by placing in it *count* objects from the *objects* array. Each object in the *objects* array receives a **retain** message as it’s added to the array. After an immutable array has been initialized in this way, it can’t be modified. Returns **self**.

See also: – **initWithObjects:**, + **arrayWithObjects:**, – **initWithArray:**

isEqual:

@protocol NSObject
– (BOOL)**isEqual:**(id)*anObject*

Returns YES if the receiver and *anObject* are equal; otherwise returns NO. A YES return value indicates that the receiver and *anObject* are both instances of classes that inherit from NSArray and that they both contain the same objects (as determined by the **isEqualToArray:** method).

See also: – **isEqualToArray:**

isEqualToArray:

– (BOOL)**isEqualToArray:**(NSArray *)*otherArray*

Compares the receiving array to *otherArray*. If the contents of *otherArray* are equal to the contents of the receiver, this method returns YES. If not, it returns NO.

Two arrays have equal contents if they each hold the same number of objects and objects at a given index in each array satisfy the **isEqual:** test.

See also: – **isEqual:** (NSObject)

lastObject

– (id)**lastObject**

Returns the object in the array with the highest index value. If the array is empty, **lastObject** returns **nil**.

See also: – **removeLastObject**

makeObjectsPerform:

– (void)**makeObjectsPerform:**(SEL)*aSelector*

Sends the *aSelector* message to each object in the array in reverse order (starting with the last object and continuing backwards through the array to the first object). The *aSelector* method must be one that takes no arguments. It shouldn't have the side effect of modifying the receiving array. The messages are sent using the **perform:** method declared in the NSObject protocol.

See also: – **makeObjectsPerform:withObject:,** – **perform:** (NSObject)

makeObjectsPerformSelector:

– (void)**makeObjectsPerformSelector:**(SEL)*aSelector*

Same as **makeObjectsPerform:**.

makeObjectsPerformSelector:withObject:

– (void)**makeObjectsPerformSelector:**(SEL)*aSelector* **withObject:**(id)*anObject*

Same as **makeObjectsPerform:withObject:**.

makeObjectsPerform:withObject:

– (void)**makeObjectsPerform:**(SEL)*aSelector* **withObject:**(id)*anObject*

Sends the *aSelector* message to each object in the array in reverse order (starting with the last object and continuing backwards through the array to the first object). The message is sent each time with *anObject* as an argument, so the *aSelector* method must be one that takes a single argument of type **id**. The *aSelector* method shouldn't, as a side effect, modify the receiving array. The messages are sent using the **perform:with:** method declared in the NSObject protocol.

See also: – **makeObjectsPerform:,** – **perform:withObject:** (NSObject)

objectAtIndex:

– (id)**objectAtIndex:**(unsigned int)*index*

Returns the object located at *index*. If *index* is beyond the end of the array (that is, if *index* is greater than or equal to the value returned by **count**), an NSRangeException is raised.

See also: – **count**

objectEnumerator

– (NSEnumerator *)**objectEnumerator**

Returns an enumerator object that lets you access each object in the array, in order, starting with the element at index 0, as in:

```
NSEnumerator *enumerator = [myArray objectEnumerator];
id anObject;

while ((anObject = [enumerator nextObject])) {
     /*  code to act on each element as it is returned */
}
```

When this method is used with mutable subclasses of NSArray, your code shouldn't modify the array during enumeration.

See also: – **reverseObjectEnumerator,** – **nextObject** (NSEnumerator)

pathsMatchingExtensions:

– (NSArray *)**pathsMatchingExtensions:**(NSArray *)*filterTypes*

Returns a new array that contains those string objects in the receiver that have a filename extension (as determined by NSString's **pathExtension** method) that matches one of the extensions in *filterTypes*. *filterTypes* should be an array of NSStrings, each of which identifies a filename extension to be matched (such as “tiff” or “eps”). Filenames that don’t have an extension aren’t included in the result. This method can be used to identify those files with a particular extension (or set of extensions) within a directory.

reverseObjectEnumerator

– (NSEnumerator *)**reverseObjectEnumerator**

Returns an enumerator object that lets you access each object in the array, in order, from the element at the highest index down to the element at index 0. Your code shouldn’t modify the array during enumeration.

See also: – **objectEnumerator,** – **nextObject** (NSEnumerator)

sortedArrayHint

– (NSData *)**sortedArrayHint**

Analyzes the receiver and returns a “hint” that speeds the sorting of the array when the hint is supplied to **sortedArrayUsingFunction:context:hint:**.

sortedArrayUsingFunction:context:

– (NSArray *)**sortedArrayUsingFunction:**(int(*)(id, id, void *))*comparator* **context:**(void *)*context*

Returns a new array that lists the receiver’s elements in ascending order as defined by the comparison function *comparator*. The new array contains references to the receiver’s elements, not copies of them. The retain count is incremented for each element in the receiving array.

The comparison function is used to compare two elements at a time and should return NSOrderedAscending if the first element is smaller than the second, NSOrderedDescending if the first element is larger than the second, and NSOrderedSame if the elements are equal. Each time the comparison function is called, it’s passed *context* as its third argument. This allows the comparison to be based on some outside parameter, such as whether character sorting is case-sensitive or case-insensitive.

Given *anArray* (an array of NSNumber objects) and a comparison function of this type:

```
int intSort(id num1, id num2, void *context)
{
    int v1 = [num1 intValue];
    int v2 = [num2 intValue];
    if (v1 < v2)
        return NSOrderedAscending;
    else if (v1 > v2)
        return NSOrderedDescending;
    else
        return NSOrderedSame;
}
```

A sorted version of *anArray* is created in this way:

```
NSArray *sortedArray;
sortedArray = [anArray sortedArrayUsingFunction:intSort
    context:NULL];
```

See also: – **sortedArrayUsingSelector:**

sortedArrayUsingFunction:context:hint:

– (NSArray *)**sortedArrayUsingFunction:**(int (*)(id, id, void *))*compare* **context:**(void *)*context* **hint:**(NSData *)*hint*

Similar to **sortedArrayUsingFunction:context:**, except that it uses the supplied *hint* to speed the sorting process. To obtain an appropriate *hint*, use **sortedArrayHint**. When you know that the array is nearly sorted, this method is faster than **sortedArrayUsingFunction:context:**.

sortedArrayUsingSelector:

– (NSArray *)**sortedArrayUsingSelector:**(SEL)*comparator*

Returns an array that lists the receiver's elements in ascending order, as determined by the comparison method specified by the selector *comparator*. The new array contains references to the receiver's elements, not copies of them. The retain count is incremented for each element in the receiving array.

The *comparator* message is sent to each object in the array, and has as its single argument another object in the array. The comparator method is used to compare two elements at a time and should return NSOrderedAscending if the receiver is smaller than the argument, NSOrderedDescending if the receiver is larger than the argument, and NSOrderedSame if they are equal.

For example, an array of NSStrings can be sorted by using the **compare:** method declared in the NSString class. Assuming *anArray* exists, a sorted version of the array can be created in this way:

```
NSArray *sortedArray = [anArray sortedArrayUsingSelector:@selector(compare:)];
```

See also: – **sortedArrayUsingFunction:context:**

subarrayWithRange:

– (NSArray *)**subarrayWithRange:**(NSRange)*range*

Returns a new array containing the receiver's elements that fall within the limits specified by *range*. If *range* isn't within the receiver's range of elements, an NSRangeException is raised. Each object receives a **retain** message as it's added to the array.

For example, the following code example creates an array containing the elements found in the first half of *wholeArray* (assuming that *wholeArray* exists).

```
NSArray *halfArray;
NSRange theRange;

theRange.location = 0;
theRange.length = [wholeArray count] / 2;

halfArray = [wholeArray subarrayWithRange:theRange];
```

writeToFile:atomically:

– (BOOL)**writeToFile:**(NSString *)*path* **atomically:**(BOOL)*flag*

Writes the contents of the receiver to the file specified by *path*. If the receiver's contents are all property list objects (NSString, NSData, NSArray, or NSDictionary objects), the file written by this method can be used to initialize a new array with the class method **arrayWithContentsOfFile:** or the instance method **initWithContentsOfFile:**.

If *flag* is YES, the array is written to an auxiliary file, and then the auxiliary file is renamed to *path*. If *flag* is NO, the array is written directly to *path*. The YES option guarantees that *path*, if it exists at all, won't be corrupted even if the system should crash during writing.

If *path* contains a tilde (~) character, you must expand it with **stringByExpandingTildeInPath:** before invoking this method.

This method returns YES if the file is written successfully, and NO otherwise.

See also: – **initWithContentsOfFile:**

NSMutableArray

Inherits From:	NSArray : NSObject
Conforms To:	NSCoding NSCopying NSMutableCopying (NSArray) NSObject (NSObject)
Declared In:	Foundation/NSArray.h

Class at a Glance

Purpose

An NSMutableArray stores a modifiable array of objects.

Principal Attributes

- A count of the number of objects in the array.
- The set of objects contained in the array.

Creation

+ arrayWithCapacity:	An empty array with enough allocated memory to hold a specified number of objects

Commonly Used Methods

– insertObject:atIndex:	Inserts an object at a specified index.
– removeObject:	Removes all occurrences of an object.
– removeObjectAtIndex:	Removes the object at a given index.
– replaceObjectAtIndex:withObject:	Replaces the object at a given index.

Primitive Methods

– addObject:
– replaceObjectAtIndex:withObject:
– removeLastObject

lass Description

The NSMutableArray class declares the programmatic interface to objects that manage a modifiable array of objects. This class adds insertion and deletion operations to the basic array-handling behavior inherited from NSArray.

NSMutableArray methods are conceptually based on these three primitive methods:

addObject:
replaceObjectAtIndex:withObject:
removeLastObject

The other methods in its interface 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.

When an object is removed from a mutable array, it receives a **release** message. If there are no further references to the object, the object is 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 array. For example, if anObject isn't retained before removing it from the array, the third statement below could result in a run-time error:

```
id anObject = [[anArray objectAtIndex:0] retain];
[anArray removeObjectAtIndex:0];
[anObject someMessage];
```

A Note for Those Creating Subclasses of NSMutableArray

Although conceptually the NSMutableArray class has three primitive methods, two others also access the array's data directly. These methods are:

insertObject:atIndex:
removeObjectAtIndex:

These methods could be implemented using the primitives listed above but doing so would incur unnecessary overhead. For instance, objects would receive **retain** and **release** messages as they were shifted to accommodate the insertion or deletion of an element.

lethod Types

Creating an NSMutableArray	+ arrayWithCapacity:
	– initWithCapacity:

Adding and replacing objects	– addObject: – addObjectsFromArray: – insertObject:atIndex: – replaceObjectAtIndex:withObject: – replaceObjectsInRange:withObjectsFromArray: – replaceObjectsInRange:withObjectsFromArray:range: – setArray:
Removing objects	– removeAllObjects – removeLastObject – removeObject: – removeObject:inRange: – removeObjectAtIndex: – removeObjectIdenticalTo: – removeObjectIdenticalTo:inRange: – removeObjectsFromIndices:numIndices: – removeObjectsInArray: – removeObjectsInRange:
Rearranging objects	– sortUsingFunction:context: – sortUsingSelector:

Class Methods

arrayWithCapacity:

+ (id)**arrayWithCapacity:**(unsigned int)*numItems*

Creates and returns an NSMutableArray, giving it enough allocated memory to hold *numItems* objects. NSMutableArrays expand as needed, so *numItems* simply establishes the object's initial capacity.

See also: – **initWithCapacity:**

nstance Methods

addObject:

– (void)**addObject:**(id)*anObject*

Inserts *anObject* at the end of the receiver. The object receives a **retain** message as it's added to the array. If *anObject* is **nil**, an NSInvalidArgumentException is raised.

See also: – **addObjectsFromArray:**, – **removeObject:**, – **setArray:**

addObjectsFromArray:

– (void)**addObjectsFromArray:**(NSArray *)*otherArray*

Adds the objects contained in *otherArray* to the end of the receiver’s array of objects.

See also: **setArray:**, – **removeObject:**

initWithCapacity:

– (id)**initWithCapacity:**(unsigned int)*numItems*

Initializes a newly allocated array, giving it enough memory to hold *numItems* objects. Mutable arrays expand as needed, so *numItems* simply establishes the object’s initial capacity. Returns **self**.

See also: – **arrayWithCapacity:**

insertObject:atIndex:

– (void)**insertObject:**(id)*anObject* **atIndex:**(unsigned int)*index*

Inserts *anObject* into the receiver at *index*. If *index* is already occupied, the objects at *index* and beyond are shifted down one slot to make room. *index* cannot be greater than the number of elements in the array. *anObject* receives a **retain** message as it’s added to the array. This method raises an NSInvalidArgumentException if *anObject* is **nil** and raises an NSRangeException if *index* is greater than the number of elements in the array.

Note that NSArrays are not like C arrays. That is, even though you do an “initWithCapacity:,” the specified capacity is regarded as a “hint”; the actual size of the array is still 0. Because of this, you can only insert new objects in ascending order—with no gaps. Once you add two objects, the array's size is 2, so you can add objects at indexes 0, 1, or 2. Index 3 is illegal and out of bounds; if you try to add an object at index 3 (when the size of the array is 2), NSMutableArray raises an exception.

See also: – **removeObjectAtIndex:**

removeAllObjects

– (void)**removeAllObjects**

Empties the receiver of all its elements. Each removed object is sent a **release** message.

See also: – **removeObject:**, – **removeLastObject**, – **removeObjectAtIndex:**, – **removeObjectIdenticalTo:**

removeLastObject

– (void)**removeLastObject**

Removes the object with the highest-valued index in the array and sends it a **release** message. **removeLastObject** raises an NSRangeException if there are no objects in the array.

See also: – **removeAllObjects**, – **removeObject:**, – **removeObjectAtIndex:**, – **removeObjectIdenticalTo:**

removeObject:

– (void)**removeObject:**(id)*anObject*

Removes all occurrences of *anObject* in the array. This method uses **indexOfObject:** to locate matches and then removes them by using **removeObjectAtIndex:**. Thus, matches are determined on the basis of an object's response to the **isEqual:** message.

See also: – **removeAllObjects**, – **removeLastObject**, – **removeObjectAtIndex:**, – **removeObjectIdenticalTo:**, – **removeObjectsInArray:**

removeObject:inRange:

– (void)removeObject:(id)*anObject* **inRange:(NSRange)***aRange*

Removes all occurrences of *anObject* within the specified range in the array. Matches are determined on the basis of an object's response to the **isEqual:** message and by comparing **id**s.

See also: – **removeAllObjects**, – **removeLastObject**, – **removeObjectAtIndex:**, – **removeObjectIdenticalTo:**, – **removeObjectsInArray:**

removeObjectAtIndex:

– (void)**removeObjectAtIndex:**(unsigned int)*index*

Removes the object at *index* and moves all elements beyond *index* up one slot to fill the gap. The removed object receives a **release** message. This method raises an NSRangeException if *index* is beyond the end of the array.

See also: **– insertObjectAtIndex:**, – **removeAllObjects**, – **removeLastObject**, – **removeObject:**, – **removeObjectIdenticalTo:, – removeObjectsFromIndices:numIndices:**

removeObjectIdenticalTo:

– (void)**removeObjectIdenticalTo:**(id)*anObject*

Removes all occurrences of *anObject* in the array. This method uses the **indexOfObjectIdenticalTo:** method to locate matches and then removes them by using **removeObjectAtIndex:**. Thus, matches are determined on the basis of an object's **id**.

See also: – **removeAllObjects**, – **removeLastObject**, – **removeObject:**, – **removeObjectAtIndex:**

removeObjectIdenticalTo:inRange:

– (void)removeObjectIdenticalTo:(id)*anObject* **inRange:(NSRange)***aRange*

Removes all occurrences of *anObject* within the specified range in the array. Matches are determined by comparing object **id**s.

See also: – **removeAllObjects**, – **removeLastObject**, – **removeObject:**, – **removeObjectAtIndex:**

removeObjectsFromIndices:numIndices:

– (void)**removeObjectsFromIndices:**(unsigned int *)*indices* **numIndices:**(unsigned int)*count*

This method is similar to **removeObjectAtIndex:**, but allows you to efficiently remove multiple objects with a single operation. *count* indicates the number of objects to be removed, while *indices* points to the first in a list of indexes. Note that if you sort the list of indexes in ascending order, you will improve the speed of this operation.

This method does not distribute and therefore should be used sparingly.

See also: – **insertObjectAtIndex:**, – **removeObjectAtIndex:**, **– removeObjectsInRange:**

removeObjectsInArray:

– (void)**removeObjectsInArray:**(NSArray *)*otherArray*

This method is similar to **removeObject:**, but allows you to efficiently remove large sets of objects with a single operation. It assumes that all elements in *otherArray*—which are the objects to be removed—respond to **hash** and **isEqual:**.

This method does not distribute and therefore should be used sparingly.

See also: – **removeAllObjects**, – **removeObjectIdenticalTo:**

removeObjectsInRange:

– (void)removeObjectsInRange:(NSRange)*aRange*

Removes each of the objects within the specified range in the receiver using **removeObjectAtIndex:**.

replaceObjectAtIndex:withObject:

– (void)**replaceObjectAtIndex:**(unsigned int)*index* **withObject:**(id)*anObject*

Replaces the object at *index* with *anObject*. *anObject* receives a **retain** message as it's added to the array, and the previous object at *index* receives a **release** message. This method raises an NSInvalidArgumentException if *anObject* is **nil** and raises an NSRangeException if *index* is beyond the end of the array.

See also: – **insertObjectAtIndex:**, – **removeObjectAtIndex:**

replaceObjectsInRange:withObjectsFromArray:

– (void)replaceObjectsInRange:(NSRange)*aRange*
withObjectsFromArray:(NSArray *)*otherArray*

Replaces the objects in the receiver specified by *aRange* with all of the objects from *otherArray*. If *otherArray* has fewer objects than are specified by *aRange*, the extra objects in the receiver are removed. If *otherArray* has more objects than are specified by *aRange*, the extra objects from *otherArray* are inserted into the receiver.

See also: – **insertObject:atIndex:**, – **removeObjectAtIndex:**, – **replaceObjectAtIndex:withObject:**

replaceObjectsInRange:withObjectsFromArray:range:

– (void)replaceObjectsInRange:(NSRange)*aRange*
withObjectsFromArray:(NSArray *)*otherArray* **range:(NSRange)***otherRange*

Replaces the objects in the receiver specified by *aRange* with the objects in *otherArray* specified by *otherRange*. *aRange* and *otherRange* don't have to be equal; if *aRange* is greater than *otherRange*, the extra objects in the receiver are removed. If *otherRange* is greater than *aRange*, the extra objects from *otherArray* are inserted into the receiver.

See also: – **insertObject:atIndex:**, – **removeObjectAtIndex:**, – **replaceObjectAtIndex:withObject:**

setArray:

– (void)**setArray:**(NSArray *)*otherArray*

Sets the receiver’s elements to those in *otherArray*. Shortens the receiver, if necessary, so that it contains no more than the number of elements in *otherArray*. Replaces existing elements in the receiver with the elements in *otherArray*, releasing those objects that are being replaced and retaining those objects that are replacing them. Finally, if there are more elements in *otherArray* than there are in the receiver, the additional items are then added (and **retain** is sent to each object as it is added to the receiver).

See also: **– addObjectsFromArray:, – replaceObjectAtIndex:withObject:**

sortUsingFunction:context:

– (void)**sortUsingFunction:**(int (*)(id, id, void *))*compare* **context:**(void *)*context*

Sorts the receiver’s elements in ascending order as defined by the comparison function *compare*. The comparison function is used to compare two elements at a time and should return NSOrderedAscending if the first element is smaller than the second, NSOrderedDescending if the first element is larger than the second, and NSOrderedSame if the elements are equal. Each time the comparison function is called, it’s passed *context* as its third argument. This allows the comparison to be based on some outside parameter, such as whether character sorting is case-sensitive or case-insensitive.

See also: **– sortUsingSelector:**, **– sortedArrayUsingFunction:context:** (NSArray)

sortUsingSelector:

– (void)**sortUsingSelector:**(SEL)*comparator*

Sorts the receiver’s elements in ascending order, as determined by the comparison method specified by the selector *comparator*. The *comparator* message is sent to each object in the array, and has as its single argument another object in the array. The comparator method is used to compare two elements at a time and should return NSOrderedAscending if the receiver is smaller than the argument, NSOrderedDescending if the receiver is larger than the argument, and NSOrderedSame if they are equal.

See also: **– sortUsingFunction:context:**, **– sortedArrayUsingSelector:** (NSArray)