FCOrderedCollection

Inherits From:	FCCollection : Object
Declared In:	FCOrderedCollection.h

Class Description

FCOrderedCollection is a subclass of FCCollection that implements the behavior of an ordered collection.   "Order" is defined as each object in the collection having a unique index with which it can be accessed.

All collection subclasses which require an ordering are subclasses of this class. This includes lists, stacks, queues, and sorted sets.

FCOrderedCollection uses a List object as its object storage medium, and thus is fast at adding objects and retrieving objects sequentially, but will take O(n) to determine if an object is a member of the collection. Some subclasses have optimized the membership check; see the class documentation for the instantiable FCOrderedCollection subclasses.

FCOrderedCollection is an abstract superclass.   You cannot instantiate it directly; in fact, some of its methods are simply stubs in the superclass and return errors when invoked.   Its basic purpose is to provide common methods and an orthogonal interface to its eight instantiable subclasses, all of which fully adhere to the interface described here.   In the documentation below, the term "collection" refers to any non-abstract subclass of FCOrderedCollection.

FCOrderedCollection inherits from FCCollection.   The interface documented here only covers the methods that are new or different in FCOrderedCollection, but all the methods in FCCollection will work on an FCUnorderedCollection as well.   Refer to the documentation on FCCollection to complete the description of an FCUnorderedCollection.

Instance Variables

Inherited from Object
None declared in this class.

Inherited from FCCollection
id **_fc_contents** ;
Class **_fc_class** ;

SEL **_fc_sortSelector** ;
BOOL **_fc_archiveByReference** ;

Declared in FCOrderedCollection
None declared in this class.

Method Types

Creating instances	+alloc +allocFromZone:
Asking About the Contents	-firstObject -lastObject -objectAt: -indexOf: -indexOfEqualObject:
Changing the Contents	-removeFirstObject -removeLastObject -removeObjectAt:
Making Related Collections	-copyFrom:to:
Iterating	-loopIndex:
Comparing and Sorting	-sortByCompare:

Class Methods

alloc

+ **alloc;**

This method cannot be used to create an FCOrderedCollection object. FCOrderedCollection is an abstract superclass, you should call **alloc** only on its instantiable subclasses. The method is implemented only to prevent you from using it; if you do use it, it generates an error message.

allocFromZone:

+ **allocFromZone:**(NXZone *)*zone;*

This method cannot be used to create an FCOrderedCollection object. FCOrderedCollection is an abstract superclass, you should call **allocFromZone:** only on its instantiable subclasses. The method is implemented only to prevent you from using it; if you do use it, it generates an error message.

Instance Methods

copyFrom:to:

- **copyFrom:**(unsigned)*first* **to:**(unsigned)*last;*

Returns a new collection object which contains the subset of objects from positions *first* to *last* in the receiving collection. Returns **nil** if the indices are out of bounds.

See also: **- collectObjects:** (FCCollection)

firstObject

- **firstObject;**

Returns the **id** of the first object in the collection, or **nil** if the collection is empty.

See also: **- lastObject, -objectAt:**

indexOf:

- (unsigned)**indexOf:anObject;**

Returns the index of the first occurrence of *anObject* in the collection, or NX_NOT_IN_LIST if *anObject* isn't in the collection.

indexOfEqualObject:

- (unsigned)**indexOfEqualObject:anObject;**

Returns the index of the first object in the collection which thinks itself equal to *anObject* . Equality is tested by sending the isEqual: message to all the elements in the collection.

Returns NX_NOT_IN_LIST if no objects think themselves equal.

See also: **- contains:**

lastObject

- **lastObject;**

Returns the **id** of the last object in the collection, or **nil** if the collection is empty.

See also: **- firstObject, -objectAt:**

loopIndex:

- (unsigned)**loopIndex:**(FCLoopState *)*loopState;*

Returns an unsigned integer representation of the opaque *loopState* loop counter. You can use this integer as an index into the collection; e.g., [employees objectAt:[employees loopIndex:loopState]].

See also: - startLoop: (FCCollection)**, - nextObject:** (FCCollection)**, FOR_EACH()** (FCCollection)**, FOR_EACH_EXCEPT_FIRST()** (FCCollection)**, FOR_EACH_SELECTED()** (FCCollection)**, FOR_EACH_BACKWARDS()**

objectAt:

- **objectAt:**(unsigned)*index;*

Returns the **id** of the object located at slot *index* , or **nil** if *index* is beyond the end of the collection.

See also: - firstObject, -lastObject, - count (FCCollection)

removeFirstObject

- **removeFirstObject;**

Removes the first object from the collection and returns it. Returns **nil** if the collection is empty.

See also: - removeLastObject, - removeObjectAt:

removeLastObject

- **removeLastObject;**

Removes the last object from the collection and returns it. Returns **nil** if the collection is empty.

See also: **- removeFirstObject, - removeObjectAt:**

removeObjectAt:

- **removeObjectAt:**(unsigned)*index;*

Removes the object located at *index* and returns it. If there's no object at *index* , this method returns **nil** .

The positions of the remaining objects in the collection are adjusted so there's no gap.

See also: **- removeFirstObject, - removeLastObject, - removeObject:** (FCCollection)

sortByCompare:

- **sortByCompare:**(SEL)*theSelector;*

Sorts the objects in the collection. Objects are compared by sending them the *theSelector* message, which must take an **id** (of the comparison object) as its sole argument. The return values for the *theSelector* method should be of type **FCCompareType** ; FC_COMPARE_EQUAL_TO, FC_COMPARE_GREATER_THAN, or FC_COMPARE_LESS_THAN, depending on whether the receiving object is equal to, greater than, or less than the argument object, respectively.

For example, the following method in the Employee class would allow employees to be sorted by age:

```
 - (FCCompareType)compareAge:otherEmployee
{
   return compareInts([self age], [otherEmployee age]);
}
```

The actual sort would be done by calling [employees sortByCompare:@selector(compareAge:)].

See also: **compareFloats(), compareChars(), compareStrings(), compareShorts(), compareInts(), compareLongs()**

Macros

FOR_EACH_BACKWARDS()

FOR_EACH_BACKWARDS(*item* , *collection* , *block* **)**

Loops backwards through *collection* one object at a time, placing each object in *item* , then executing *block* . The advantage of going backwards is that you can safely add elements to the end of the collection or delete the current element while in the loop. (If you attempt either of these while looping forwards you will loop infinitely or skip elements, respectively.)

Here's an example of deleting all employees who make over $10:

```
    FOR_EACH_BACKWARDS(person, employees, {
      if ([person salary] > 10.0)
          [employees removeObject:person];
   })
```

FOR_EACH_BACKWARDS() loops can be nested to arbitrary depth.

See also: **- startLoop:, - nextObject:, FOR_EACH()** (FCCollection)**, FOR_EACH_EXCEPT_FIRST()** (FCCollection)**, FOR_EACH_SELECTED()** (FCCollection)