FCCollection

Inherits From: Object

Declared In: FCCollection.h

Class Description

FCCollection is the ultimate superclass for a family of classes that provide capabilities to manipulate collections of objects.

FCCollection is conceptually analogous to the List class. Both contain collections of **id** s, and allow you to add and remove objects. Both add objects to the collection by reference, rather than actually copying the objects. Neither class allows **nil** objects to be inserted into the collection.

FCCollection implements many of the same methods as List, and wherever applicable duplicates List's method interfaces exactly.

FCCollection, however, provides more methods for controlling the contents of the collection (e.g., ± **uniqueElements** and ± **setContentClass:**), manipulating the objects in the collection (e.g., ± **addObjectsFrom:**), and creating related collections (e.g., ± **selectObjects:** and ±

collectObjects:). Further, its subclasses add features beyond List's, and have more efficient internal implementations for specific applications.

FCCollection 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 FCCollection.

To determine which FCCollection subclass you should use in a particular situation, refer first to the 'Foundation Classes and Function' introduction document, and then to the class documentation for the appropriate subclasses.

Instance Variables

Inherited from Object
None declared in this class.

Declared in FCCollection
id **_fc_contents** ;
Class **_fc_class** ;
SEL **_fc_sortSelector** ;
BOOL **_fc_archiveByReference** ;

_fc_contents	contents of the collection
_fc_class	class of the content objects

_fc_sortSelector	selector used for sorting in subclasses
_fc_archiveByReference	archiving mode

Method Types

Initializing the Class	+initialize
Creating instances	+alloc +allocFromZone:
Initializing a Collection	-init -initCount: -initForClass:
Freeing the Collection	-free -freeObjects
Copying Methods	-copy -copyFromZone: -copyAs: -copyAs:fromZone: -deepCopy -deepCopyFromZone: -copyAs:fromZone:deep:
Accessing the Behavior of the Collection	-archiveByReference -setArchiveByReference: -contentClass -setContentClass:

	-uniqueElements -isOrdered -isSorted
Asking About the Contents	-count -isEmpty -contains: -equalObject: -containsEqualObject: -occurrencesOf: -isDisjointFrom: -containsSubset: -trueForAll: -trueForAny:
Changing the Contents	-addObject: -addObjectsFrom: -removeObject: -removeObjectsFrom: -removeObjectsNotContainedIn: -removeAllOccurrencesOfObject: -empty -replaceObject:with:
Making Related Collections	-selectObjects: -rejectObjects: -collectObjects: -collectObjects:as:
Iterating	-startLoop: -nextObject:

	-peekNextObject:
Comparing and Sorting	-isEqual: -compare:
Sending Messages to the Objects	-makeObjectsPerform: -makeObjectsPerform:with:
Archiving	-write: -read:

Class Methods

alloc

+ **alloc;**

This method cannot be used to create an FCCollection object. FCCollection 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 FCCollection object. FCCollection 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.

initialize

+ **initialize;**

Sets the version of the class for use in Objective-C archiving. You should not use this method directly, it is called for you when the class is first used.

Instance Methods

addObject:

- **addObject:anObject;**

Subclasses should implement this method to add *anObject* into the collection. Ordered subclasses will add *anObject* to the end of the list. Subclasses with unique objects will fail and return **nil** if *anObject* is already present in the collection. If the programmer has set a content class, **addObject:** will fail if *anObject* isn't a kind of that class.

FCCollection's implementation of this method prints an error message and aborts. Existing subclasses override this to provide the correct behavior.

See also: - removeObject:, - addObjectsFrom:, - isOrdered, - uniqueElements, - setContentClass:

addObjectsFrom:

- **addObjectsFrom:otherCollection;**

Adds all the objects in *otherCollection* to this collection. The objects are not removed from *otherCollection* . The objects are added with the - **addObject:** method, so any objects that fail that method's tests will be skipped.

See also: - addObject:

archiveByReference

- (BOOL)**archiveByReference;**

Returns YES if the objects in the collection will be archived using **NXWriteObjectByReference** , and NO if the objects will be archived with **NXWriteObject** . The default is NO; you can change this with the - **setArchiveByReference:** method.

See also: - setArchiveByReference:, NXWriteObjectByReference, NXWriteObject (Common Functions)

collectObjects:

- **collectObjects:**(SEL)*idSelector;*

Returns a new collection of the same class as the receiver that contains the collected return values from every object being sent the *idSelector* message.

The *idSelector* method should take no arguments, and return either an **id** or **nil** . Every object in the collection must respond to *idSelector* .

See also: **- collectObjects:as:, - selectObjects:, - rejectObjects:**

collectObjects:as:

- **collectObjects:**(SEL)*idSelector* **as:newClass;**

Returns a new FCCollection subclass of type *newClass* that contains the collected return values from every object being sent the *idSelector* message.

The *idSelector* method should take no arguments, and return either an **id** or **nil** . Every object in the collection must respond to *idSelector* .

See also: **- collectObjects:, - selectObjects:, - rejectObjects:**

compare:

- (FCCompareType)**compare:anObject;**

Returns FC_COMPARE_EQUAL_TO if the receiver is of the same class, contains the same elements, and has the same instance variables as *anObject* . Returns FC_COMPARE_NOT_EQUAL_TO if the element collections differ. Returns FC_COMPARE_CANT_COMPARE if the class or instance variables differ (e.g., the contentClasses are different).

Note that this routine is only partially implemented in FCCollection; it checks the instance variables of the two collection objects, but doesn't check that the contents are the same. This latter duty is the subclasses' responsibility; both FCOrderedCollection and FCUnorderedCollection subclass this method correctly.

See also: **- isEqual:**

contains:

- (BOOL)**contains:anObject;**

Returns YES if *anObject* is a member of the collection.

See also: **- containsEqualObject:**

containsEqualObject:

- (BOOL)**containsEqualObject:anObject;**

Returns YES if an object which thinks itself equal to *anObject* is a member of the collection. Equality is tested by sending the isEqual: message to all the elements in the collection.

See also: **- equalObject:, - contains:**

containsSubset:

- (BOOL)**containsSubset:otherCollection;**

Returns YES if this collection contains all the elements in *otherCollection* .

See also: **- isDisjointFrom:**

contentClass

- **contentClass;**

Returns the class of the objects contained in the collection, if the collection's class has been

explicitly set. Returns **nil** if no class checking is in force (e.g., the collection may be heterogeneous). The default is **nil** .

See also: **- setContentClass:, - init, - initForClass:**

copy

- **copy;**

Returns a new collection object with the same contents and instance variables (e.g., contentClass, sortSelector) as the receiver. The objects in the receiving collection aren't copied; therefore, both collections contain pointers to the same set of objects.

See also: **- copyFromZone:, - copyAs:, - deepCopy**

copyAs:

- **copyAs:aClass;**

Returns a new FCCollection subclass object of type *aClass* with the same contents as the receiver. The objects in the receiving collection aren't copied; therefore, both collections contain pointers to the same set of objects.

Note that copying from some subclasses to some others may give unexpected results; for example, copying an FCBag with duplicates into an FCSet will result in a new set with the duplicates missing.

See also: **- copyAs:fromZone:, - copy, - deepCopy**

copyAs:fromZone:

- **copyAs:aClass fromZone:**(NXZone *)*zone;*

Returns a new FCCollection subclass object of type *aClass* with the same contents as the receiver. The objects in the receiving collection aren't copied; therefore, both collections contain pointers to the same set of objects. Memory for the new collection is allocated from *zone* .

Note that copying from some subclasses to some others may give unexpected results; for example, copying an FCBag with duplicates into an FCSet will result in a new set with the duplicates missing.

See also: **- copyAs:, - copyFromZone:, - deepCopyFromZone:**

copyAs:fromZone:deep:

- **copyAs:aClass fromZone:**(NXZone *)*zone* **deep:**(BOOL)*deep;*

Returns a new FCCollection subclass object of type *aClass* with the same contents as the receiver. If *deep* is YES, the objects in the receiving collection are copies of the ones in the original collection; otherwise both collections contain pointers to the same set of objects. Memory for the new collection is allocated from *zone* .

Note that copying from some subclasses to some others may give unexpected results; for example, copying an FCBag with duplicates into an FCSet will result in a new set with the duplicates missing.

Subclasses that need to customize their copying behavior should override this method.

See also: **- copyAs:, - copyFromZone:, - deepCopyFromZone:**

copyFromZone:

- **copyFromZone:**(NXZone *)*zone;*

Returns a new collection object with the same contents and instance variables (e.g., contentClass, sortSelector) as the receiver. The objects in the receiving collection aren't copied; therefore, both collections contain pointers to the same set of objects. Memory for the new collection is allocated from *zone* .

See also: **- copy, - copyAs:fromZone:, - deepCopyFromZone:**

count

- (unsigned)**count;**

Returns the number of objects currently in the collection. In a non-unique collection, this will count multiple instances of the same object multiple times.

deepCopy

- **deepCopy;**

Returns a new collection object with the same contents as the receiver. The objects in the receiving collection **are** copied by sending them the - copy method; therefore, the new collection contains a different set of objects than the receiver.

Note that the objects in the collection must properly implement the - copy method in order for - deepCopy to work.

See also: **- copy, - copyAs:, - copy** (Object)

deepCopyFromZone:

- **deepCopyFromZone:**(NXZone *)*zone;*

Returns a new collection object with the same contents as the receiver. The objects in the receiving collection **are** copied by sending them the - copy method; therefore, the new collection contains a different set of objects than the receiver. Memory for the new collection is allocated from *zone* .

Note that the objects in the collection must properly implement the - copy method in order for - deepCopy to work.

See also: **- deepCopy, - copyFromZone:, - copyAs:fromZone:, - copyFromZone:** (Object)

empty

- **empty;**

Empties the collection of all its objects without freeing them, and returns self.

See also: **- freeObjects**

equalObject:

- **equalObject:anObject;**

Returns an object which thinks itself equal to *anObject* if there is such an object in the collection. Equality is tested by sending the isEqual: message to all the elements in the collection.

If there are several objects that think themselves equal, one is picked at random. Returns **nil** if no

objects think themselves equal.

See also: **- containsEqualObject:, - contains:**

free

- **free;**

Deallocates the FCCollection object and the memory it allocated for the array of object **id**s. However, the objects contained in the collection aren't freed.

See also: **- freeObjects, - empty**

freeObjects

- **freeObjects;**

Removes every object from the collection, sends each one of them a **free** message, and returns **self** . Even if an object appears twice in the collection it will only receive one **free** message. The FCCollection object itself isn't **free** d; this method returns a valid **self** .

The methods that free the objects should not modify the collection.

See also: **- empty, - free**

init

- **init;**

Initializes the receiver, a new FCCollection object, but doesn't allocate any memory for its

collection of objects. The initial capacity of the collection will be 0. Minimal amounts of memory will be allocated when objects are added.

The **init** method does not set a content class for the collection; it allows you to add objects of any type. If you wish to restrict the type of objects you can add use **initForClass:** instead, or send **setContentClass:** to the receiver.

Because FCCollection is an abstract superclass, you will only send this message to subclasses of FCCollection, rather than FCCollection directly.

See also: - initCount:, - initForClass:, - setContentClass

initCount:

- **initCount:**(unsigned)*numSlots;*

Initializes the receiver, a new FCCollection object, and allocates enough memory for it to hold *numSlots* objects.

The **initCount:** method does not set a content class for the collection; it allows you to add objects of any type. If you wish to restrict the type of objects you can add use **initForClass:** instead, or send **setContentClass:** to the receiver.

Because FCCollection is an abstract superclass, you will send this message to subclasses of FCCollection, rather than FCCollection directly.

See also: - init, - initForClass:, - setContentClass

initForClass:

- **initForClass:theClass;**

Initializes the receiver, a new FCCollection object, and sets it to only allow objects that are a kind of *theClass* to be added to its collection. The initial capacity of the collection will be 0. Minimal amounts of memory will be allocated when objects are added.

Because FCCollection is an abstract superclass, you will send this message to subclasses of FCCollection, rather than FCCollection directly.

See also: - init, - initCount:, - setContentClass

isDisjointFrom:

- (BOOL)**isDisjointFrom:otherCollection;**

Returns YES if this collection shares no elements with *otherCollection* .

See also: - containsSubset:

isEmpty

- (BOOL)**isEmpty;**

Returns YES if there are no objects in the collection.

See also: - empty

isEqual:

- (BOOL)**isEqual:anObject;**

Returns YES if the receiver is of the same class, contains the same elements, and has the same content class (if any) and archiving mode as *anObject* .

See also: **- compare:**

isOrdered

- (BOOL)**isOrdered;**

FCCollection instances return YES to indicate that objects in their collections are ordered. Ordered collections have a unique, sequential index for each object in the collection.

Some subclasses (e.g., FCUnorderedCollection) override this method to return NO to indicate they maintain no ordering.

See also: **- isSorted, - isOrdered** (FCUnorderedCollection)

isSorted

- (BOOL)**isSorted;**

FCCollection instances return NO to indicate that objects in their collections are not sorted. Sorted collections keep all their objects constantly in a programmer-specified sorted order.

Some subclasses (e.g., FCSortedCollection) override this method to return YES to indicate they are sorted.

See also: **- isOrdered, - isSorted** (FCSortedCollection)

makeObjectsPerform:

- **makeObjectsPerform:**(SEL)*aSelector;*

Sends the *aSelector* message to each object in the collection. Ordered collections will send the message to the objects in reverse order, unordered collections will send the message in random order. The *aSelector* method must be one that takes no arguments. It should not modify the collection.

See also: **- makeObjectsPerform:with:**

makeObjectsPerform:with:

- **makeObjectsPerform:**(SEL)*aSelector* **with:anObject;**

Sends the *aSelector* message to each object in the collection. Ordered collections will send the message to the objects in reverse order, unordered collections will send the message in random order. 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 should not modify the collection.

See also: **- makeObjectsPerform:**

nextObject:

- **nextObject:**(FCLoopState *)*loopState;*

Subclasses implement this method to return the next object in the collection according to the opaque loop counter *loopState* . *loopState* is incremented. No modification of the collection should be done while iterating through it. This method will return **nil** if there are no more elements in the

collection.

Both ordered and unordered subclasses of FCCollection can be stepped through sequentially with this method; with unordered subclasses this is the only way to step through all the objects.

FCCollection's implementation of this method prints an error message and aborts. Existing subclasses override this to provide the correct behavior.

Note that for simple loops, it's much more efficient to use FOR_EACH() or one of its variations to loop through all objects, as they don't do a method call for - **nextObject:** every time through the loop.

See also: **- startLoop:, - peekNextObject:, FOR_EACH(), FOR_EACH_EXCEPT_FIRST(), FOR_EACH_SELECTED(), FOR_EACH_BACKWARDS()** (FCOrderedCollection)

occurrencesOf:

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

Returns the number of times *anObject* appears in the collection. For FCCollection this will always be 0 or 1, but for subclasses which respond NO to **uniqueElements** this method can return any non-negative integer.

See also: **- contains, - containsEqualObject:**

peekNextObject:

- **peekNextObject:**(FCLoopState *)*loopState;*

Subclasses implement this method to return the next object in the collection according to the

opaque loop counter *loopState* . *loopState* is unmodified. This method will return **nil** if there are no more elements in the collection.

FCCollection's implementation of this method prints an error message and aborts. Existing subclasses override this to provide the correct behavior.

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

read:

- **read:**(NXTypedStream *)*stream;*

Reads the FCCollection from the typed stream *stream* . Used by the Objective-C archiving functions. You should not call this method directly.

See also: **- setArchiveByReference:, - archiveByReference, - write:**

rejectObjects:

- **rejectObjects:**(SEL)*booleanSelector;*

Returns a new collection object which contains all the objects from the current collection except those which respond YES to the message *booleanSelector* .

The *booleanSelector* method should take no arguments and return YES or NO. Objects which don't respond to *booleanSelector* are included in the new collection. It is the programmer's responsibility to free the new collection.

See also: **- selectObjects:, - collectObjects:, - collectObjects:as:**

removeAllOccurrencesOfObject:

- **removeAllOccurrencesOfObject:anObject;**

Removes all occurrences of *anObject* from this collection. This is equivalent to calling **removeObject:** in subclasses which have unique elements.

See also: **- removeObject:, - uniqueElements**

removeObject:

- **removeObject:anObject;**

Subclasses should implement this method to remove *anObject* from the collection and return it. Subclasses with non-unique elements will only remove a single reference if multiples exist. If *anObject* isn't in the collection, this method will return **nil** .

FCCollection's implementation of this method prints an error message and aborts. Existing subclasses override this to provide the correct behavior.

See also: **- addObject:, - removeObjectsFrom:, - removeAllOccurrencesOfObject:**

removeObjectsFrom:

- **removeObjectsFrom:otherCollection;**

Removes from this collection all the objects in *otherCollection* that appear in this collection. *otherCollection* is not affected. The objects are removed with the - **removeObject:** method.

See also: **- removeObject:, - removeObjectsNotContainedIn:, - empty**

removeObjectsNotContainedIn:

- **removeObjectsNotContainedIn:otherCollection;**

Removes all the objects in this collection, except those which also appear in *otherCollection* . *otherCollection* is not affected.

See also: **- removeObject:, - removeObjectsFrom:, - empty**

replaceObject:with:

- **replaceObject:anObject with:newObject;**

Removes the first occurrence of *anObject* from the collection and adds *newObject,* returning *anObject* . However, if *newObject* is **nil** or *anObject* isn't in the collection, nothing is done and **nil** is returned. If a subclass of FCCollection requires unique elements and *newObject* is already in the collection, the replace will fail and **nil** will be returned.

Ordered, unsorted subclasses will put *newObject* in the spot where *anObject* was; other subclasses don't define where *newObject* will appear.

See also: **- removeObject:, - addObject:**

selectObjects:

- **selectObjects:**(SEL)*booleanSelector;*

Returns a new collection object which contains only those objects from the current collection that respond YES to the message *booleanSelector* .

The *booleanSelector* method should take no arguments and return YES or NO. Objects which don't respond to *booleanSelector* are left out of the new collection. It is the programmer's responsibility to free the new collection.

See also: **- rejectObjects:, - collectObjects:, - collectObjects:as:**

setArchiveByReference:

- **setArchiveByReference:**(BOOL)*flag;*

Determines whether the objects in the collection will be archived using **NXWriteObjectByReference** (YES) or **NXWriteObject** (NO). The default is **NXWriteObject** .

See also: **- archiveByReference, NXWriteObjectByReference, NXWriteObject** (Common Functions)

setContentClass:

- **setContentClass:theClass;**

Sets the class of the objects that will be contained in the collection. This does not change the class of existing objects in the collection, it merely tells the collection to enforce that only objects that are a kind of *theClass* may be added to the collection.

If *theClass* is **nil** no class checking is done, and the collection may be heterogeneous. This is the default behavior for collections not initialized with - **initForClass:** .

See also: **- init, - initForClass:, - setContentClass:**

startLoop:

- **startLoop:**(FCLoopState *)*loopState;*

Subclasses implement this method to initialize *loopState* , an FCLoopState structure that's required when iterating through the collection. Iterating through all elements of a collection involves initializing an iteration state, conceptually private to FCCollection, and then progressing until all entries have been visited. An example of sending the *doSomething* message to all elements in a collection follows:

```
unsigned int count = 0;
FCLoopState  state;
id element;

[collection startLoop:&state];
while (element = [collection nextObject: &loopState])
    [element doSomething];
}
```

FCCollection's implementation of this method prints an error message and aborts. Existing subclasses override this to provide the correct behavior.

See also: - nextObject:, - peekNextObject:, FOR_EACH(), FOR_EACH_EXCEPT_FIRST(), FOR_EACH_SELECTED()

trueForAll:

- (BOOL)**trueForAll:**(SEL)*booleanSelector;*

Returns YES if all of the elements in this collection return YES when sent the method *booleanSelector* . The *booleanSelector* method should take no arguments and return YES or NO. Also returns NO if any of the elements don't respond to *booleanSelector* .

See also: **- trueForAny:**

trueForAny:

- (BOOL)**trueForAny:**(SEL)*booleanSelector;*

Returns YES if any of the elements in this collection return YES when sent the method *booleanSelector* . The *booleanSelector* method should take no arguments and return YES or NO. This method does not require that all the elements respond to *booleanSelector* .

See also: **- trueForAll:**

uniqueElements

- (BOOL)**uniqueElements;**

FCCollection instances return NO to indicate that a single object may appear multiple times in their collections.

Some subclasses (e.g., sets) override this method to return YES to indicate objects can only appear once in their collections.

See also: **- uniqueElements** (FCSet, FCOrderedSet, FCSortedSet)

write:

- **write:**(NXTypedStream *)*stream;*

Writes the FCCollection and all the objects it contains to the typed stream *stream* . The objects are

written using either **NXWriteObject** or **NXWriteObjectByReference** , depending on whether the collection is set to **archiveByReference** .

This method is only used by the Objective-C archiving functions. You should not call this method directly.

See also: **- setArchiveByReference:, - archiveByReference, - read:**

Macros

FOR_EACH()

FOR_EACH(*item* , *collection* , *block* **)**

Loops forward through *collection* one object at a time, placing each object in *item* , then executing *block* . Here is an example of increasing each employee's salary by 10%:

```
FOR_EACH(person, employees, {
   [person setSalary:[[person salary] * 1.10]];
})
```

FOR_EACH() loops can be nested to arbitrary depth. You should not modify the collection inside this loop; if you add objects you may loop over the added objects as well and loop infinitely, and if you delete objects you'll modify the meaning of the loop counter and end up skipping objects. If you wish to step through a collection and add or delete objects, you must use a subclass of FCOrderedCollection and the **FOR_EACH_BACKWARDS()** macro.

See also: **- startLoop:, - nextObject:, FOR_EACH_EXCEPT_FIRST(), FOR_EACH_FIRST_REST(), FOR_EACH_SELECTED(), FOR_EACH_BACKWARDS()** (FCOrderedCollection)

FOR_EACH_EXCEPT_FIRST()

FOR_EACH_EXCEPT_FIRST(*item* , *collection* , *block* **)**

Loops through all but the first object in *collection* , placing each object in *item* , then executing *block* . In ordered collections this object will be [collection objectAt:0], on unordered collections the first object is chosen randomly (making this macro a bad idea for unordered collections).

FOR_EACH_EXCEPT_FIRST() loops can be nested to arbitrary depth. You should not modify the collection inside this loop (see the discussion in **FOR_EACH()** , above).

See also: - startLoop:, - nextObject:, FOR_EACH(), FOR_EACH_FIRST_REST(), FOR_EACH_SELECTED(), FOR_EACH_BACKWARDS() (FCOrderedCollection)

FOR_EACH_FIRST_REST()

FOR_EACH_FIRST_REST(*item* , *collection* , *firstBlock* , *restBlock* **)**

First executes *firstBlock* with *item* set to the first object in *collection* . In ordered collections the object will be [collection objectAt:0], on unordered collections the first object is chosen randomly.

FOR_EACH_FIRST_REST() then loops through all but the first object in *collection* , placing each object in *item* , then executing *block* . **FOR_EACH_FIRST_REST()** loops can be nested to arbitrary depth. You should not modify the collection inside this loop (see the discussion in **FOR_EACH()** , above).

Here is an example of finding the minimum salary of all employees:

```
FOR_EACH_FIRST_REST(person, employees, {
   smallestSalary = [person salary];
},{
```

```
    smallestSalary = MIN(smallestSalary, [person salary]);
})
```

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

FOR_EACH_SELECTED()

FOR_EACH_SELECTED(*item* , *collection* , *condition* , *block* **)**

Loops through *collection* one object at a time, placing each object in *item* . For every iteration, if *condition* is **TRUE** , executes *block* . Here is an example of increasing by 10% the salary of all employees making less than $10,000:

```
FOR_EACH_SELECTED(person, employees, ([person salary] < 10000), {
    [person setSalary:[[person salary] * 1.10]];
})
```

FOR_EACH_SELECTED() loops can be nested to arbitrary depth. You should not modify the collection inside this loop (see the discussion in **FOR_EACH()** , above).

See also: **- startLoop:, - nextObject:, FOR_EACH(), FOR_EACH_EXCEPT_FIRST(), FOR_EACH_FIRST_REST(), FOR_EACH_BACKWARDS()** (FCOrderedCollection)