SortedList

INHERITS FROM List
DECLARED IN SortedList.h

CLASS DESCRIPTION

A SortedList is like a List, but its contents are always kept in sorted order. Objects can be added to the SortedList only through an addObject: method; direct insertion of objects into a specific slot is not allowed, since this would not assure that the list would always remain in the correct order.

The SortedList can use a variety of criteria by which it is to be kept in order. Several have been predefined, and you can define other, more complex methods for comparing the relative values of objects.

The SortedList is told of an accessor method. The SortedList will use that accessor method's return value to compare the relative values of objects and keep the data structure sorted. All objects placed in the list should respond to the accessor method.

If the objects placed in the SortedList have only a simple key, such as a single integer, floating point, or double precision numeric value, or an NXAtom string value, and the objects have a defined method for returning the value of the key, no redefinition or subclassing of the objects placed in the SortedList is necessary. More complex key values that include several several scalar quantities may require that the objects included in the list be subclassed so that a method capable of comparing the relative values of two objects is present.

For example, assume that we want to store objects in the SortedList by the value of a double precision instance variable in the object, arrivalTime. The method -(double)**getArrivalTime** returns the value of the instance variable. We could allocate and initialize a new list to hold the objects in sorted order through the following:

```
objectList = [[SortedList alloc] initCount:2];
[objectList setSortOrder:ASCENDING];
```

```
[objectList setKeyDataType:DOUBLE];
[objectList setKeyMethod:@selector(getArrivalTime)];

[objectList addObject:foo];
```

All objects added to the SortedList will be kept in ascending order by the value of the double precision variable returned by **-getArrivalTime**.

If a more complex key is required to compare the object's relative values the objects should be able to compare themselves. For example, it might be desirable to sort the objects in the list by a structure with several fields. If this is the case, the objects placed in the SortedList should define a method that takes a single argument of *id*, incorporates all the necessary logic involved in the comparison, and that returns an integer greater than, equal to, or less than zero, if the receiver is greater than, less than, or equal to the argument, respectively.

Suppose we need to sort an object of class Household by address, a complex structure that includes ZIP code, street name, and house address. The Household object might implement a method called **-addressComparedTo:***anotherHousehold*. This would compare the address of the receiving object to that of the object passed as the argument, and return an integer greater than zero if the receiving object has an address "greater" than *anotherHousehold*, an integer less than zero if the address is "less" than *anotherHousehold*, or zero if the two are equal. The SortedList would be used in much the same way as it would be if the comparison operator were a simple scalar:

```
objectList = [[SortedList alloc] initCount:2];
[objectList setSortOrder:ASCENDING];
[objectList setKeyDataType:OTHER];
[objectList setKeyMethod:@selector(addressComparedTo:)];

[objectList addObject:foo];
```

The only difference is that the objects in the list must be able to compare themselves, while objects sorted by

simple scalar types need only specifiy a method to access the instance variable.

INSTANCE VARIABLES

inherited from Object	Class	isa;
Inherited from List	id	*dataPtr;
	unsigned int	numElements;
	unsigned int	maxElements;
Declared in SortedList	int	sortOrder;
	BOOL	caseSensitive;
	SEL	keyMethod;
	int	keyDataType;

sortOrder	Ascending or descending sort order.
caseSensitive	Whether capitialization matters when comparing strings.
keyMethod	The key value accessor method.
keyDataType	Type of data returned by the accessor method.

METHOD TYPES

Initializing the class	+initialize
Initializing a new SortedList object	-initCount:

copying and freeing a SortedList	-copy -copyFromZone
Comparing SortedLists	-(BOOL)isEqual
Manipulating objects	-addObject: -addObjectIfAbsent:
Sorting criteria	-setSortOrder: -(int)sortOrder -setCaseSensitive: -(BOOL)caseSensitive -setKeyMethod: -(SEL)keyMethod -setKeyDataType: -(int)keyDataType
Comparing objects	-(int)compare:to:
Sorting a list	-insertionSort;
Debugging	-printKeyValues;
Cause run-time error	-insertObject:at: -replaceObjectAt:with: -replaceObject:with:
Archiving methods	-read: -write:

CLASS METHODS

initialize

+initialize

Sets the class version number. Updated classes with more or different instance variables should change this version number so that the read: method can correctly handle new older data structures.

INSTANCE METHODS

addObject:

-addObject:*anObject*

Inserts *anObject* in the SortedList in sorted order. Returns **self**. If the object does not respond to the selector **keyMethod** the object will not be inserted and a run-time error will be reported.

See also: **-addObjectIfAbsent:**

addObjectIfAbsent:

- addObjectIfAbsent:*anObject*

Inserts *anObject* in the SortedList and returns **self**, provided that *anObject* isn't already in the SortedList. If *anObject* is in the SortedList, it won't be inserted, but **self** is still returned.

caseSensitive

-(BOOL)**caseSensitive**

Returns the status of the *caseSensitive* variable. If this variable is set to YES, comparisons of NXAtom strings will be case sensitive. If the value is NO, capitalization will not be considered when sorting strings. This variable only affects the SortedList when the data type is ATOM; it will have no effect if NXAtoms are not the

key value type.

see also: **-setCaseSensitive:**

compare:to:

-(int)**compare:***thisObject* **to:***thatObject*

Compares the key values of the two objects, and returns an integer greater than zero if *thisObject* is larger than *thatObject* in the sort order. If the data type is one of the four predefined data typesÐINT, ATOM, DOUBLE, or FLOATÐthe method uses the accessor method saved in keyMethod to query the objects and compare their relative values. If the key data type is defined to be OTHER, it is assumed that the key accessor method is capable of taking a single argument of *id* and that the objects in the list are capable of comparing their own relative values. In effect, *thisObject* will be sent the accessor method with an argument of *thatObject*; the accessor method should return a value greater, less than, or equal to zero, depending on how their (perhaps complex) keys fall in the user-defined sort order.

The default sort order is ASCENDING. If this is set to DESCENDING, the compare:to: method will return a value less than zero if the *thisObject* is larger than *thatObject*.

See also: **setKeyMethod:, setSortOrder:**

copy

- copy

Returns a new List object with the same contents as the receiver. The objects in the List aren't copied; therefore, both Lists contain pointers to the same set of objects. Memory for the new List is allocated from the same zone as the receiver. All the instance variables (sortOrder, keyMethod, etc) are replicated as well.

See also: **- copyFromZone:**

copyFromZone:

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

Returns a new List object, allocated from *zone*, with the same contents as the receiver. The objects in the List aren't copied; therefore, both Lists contain pointers to the same set of objects. The instance variables of the new SortedList (sortOrder, keyMethod, etc) are the same as those of the old list.

See also: - **copy**

initCount:

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

Initializes the receiver, a new SortedList object, by allocating enough memory for it to hold *numSlots* objects. Also initializes SortedList to be ascending, case sensitive, and have an undefined key data type. Both the key data type and the key method should be initialized by explicit initialization calls. Returns **self**.

This method is the designated initializer for the class. It should be used immediately after memory for the SortedList has been allocated and before any objects have been assigned to it; it shouldn't be used to reinitialize a SortedList that's already in use. Returns **self**.

See also: - **capacity**

insertObject:at:

-insertObject:*anObject* **at:**(unsigned int)*index*

Not implemented. Calling this method will cause a run-time error. Objects cannot be added to a SortedList at an arbitrary position; instead use addObject and let the SortedList take care of what position it winds up in.

See also: **replaceObjectAt:with:, replaceObject:with:**

insertionSort

-insertionSort

Sorts the contents of the SortedList. This can be useful if the accessor method changes during program execution, leaving the contents of the SortedList in a disordered state according to the new key values. It should not be called for routine object additions; those will be added in the correct position. This is not an exceptionally speedy implementation of a sort routine (nor is this entire class optimized for speed). Returns **self**.

keyDataType

-(int)**keyDataType**

Returns the type of the key's data. There are four predefined types of data: INT, DOUBLE, FLOAT, and ATOM. Any method that returns data of this type can be used to provide key values for the sorting criteria. Complex data types should be defined as OTHER. If this data type is specified, the objects in the list should be able to compare their relative values through a method that takes a single argument of type *id*.

See also: **setkeyDataType:**

keyMethod

-(SEL)**keyMethod**

Returns the key value accessor method. This is the method that is used to retrieve the values the SortedList is ordered by.

See also: **setkeyMethod:**

printKeyValues

-printKeyValues

Prints out the index numbers and the key values of the SortedList object. This is really only useful for debugging, and is a code relic. But relics are sometimes useful. Returns **self**.

read:

-read:(NXTypedStream)*stream*

Unarchives the object from an NXTypedStream.

See also: **write:**

replaceObject:with:

replaceObject:*anObject* **with:***newObject*

See the comments for **insertObject:at:** .

replaceObjectAt:with:

replaceObjectAt:(unsigned int)*index* **with:***newObject*

See the comments for **insertObject:at:** .

setCaseSensitive:

setCaseSensitive:(BOOL)*isIt*

If the keyDataType is ATOM, this sets the sort order to be either case sensitive or not. It has no effect on the SortedList if the keyDataType is not ATOM. Returns **self.**

See also: **caseSensitive**

setKeyDataType:

-setKeyDataType:(int)*theKeyDataType*

Alerts the SortedList to the type of data returned by the keyMethod. Predefined types are the constants INT, ATOM, DOUBLE, and FLOAT. Complex data types can be specified with the constant OTHER. If this is the

case, the objects in the SortedList must be able to respond to a message that takes a single argument of type id, and returns an integer greater than, less than, or equal to zero, depending on the relative values of the object's key data when sorted in ascending order. Returns **self**.

See also: **keyDataType**

setKeyMethod:

-setKeyMethod:(SEL)*theMethod*

Sets the accessor method for the sorting key. The return data type of the key should also be set with the **setKeyDataType:** method. The predefined data types are INT, DOUBLE, FLOAT, and ATOM. Any method that returns a value of these types can be used to order the list. If more complex keys are used, the objects placed in the list should have a method that takes a single id as an argument, and returns an integer greater than zero, less than zero, or zero if the receiver is greater than the argument; the data type should also be set to OTHER.

There may already be objects in the SortedList arranged according to another accessor method; these objects might not respond to the new accessor method. If there are one or more pre-existing objects in the SortedList that do not respond to the new key method, **setKeyMethod:** will return **nil**. Otherwise, the list is completely resorted according to the new criteria, and **self** is returned. Since sorting requires a method to compare objects the **setKeyDataType:** method should be called to set the new data type before this method is used. (The order of the method invocations doesn't matter if the SortedList is empty.) Returns **self**.

See also: **keyMethod, setKeyDataType: keyDataType**

setSortOrder:

-setSortOrder:(int)*theSortOrder*

Sets the sort order, either ASCENDING or DESCENDING. In an ASCENDING sort, objects with relatively small keys are placed first on the list. If the sort order changes the entire SortedList is rearranged to reflect the

new ordering criterion. Returns **self**.

See also: **SortOrder**

write:

write:(NXTypedStream*)*stream*

Archives the object to a typed stream.

CONSTANTS AND DEFINED TYPES

ASCENDING	0
DESCENDING	1

INT	0
ATOM	1
DOUBLE	2
FLOAT	3
OTHER	4

CURRENT_SORTED_LIST_VERSION 1

```
typedef double (*keyValueFunctionDouble)(id, SEL);
typedef int    (*keyValueFunctionInt)(id, SEL);
typedef NXAtom (*keyValueFunctionAtom)(id, SEL);
typedef float  (*keyValueFunctionFloat)(id, SEL);
```