Collections Library

Documentation and Implementation Status

Overview

The object types of the collections library establish a general-purpose foundation to maintain object references or other values as members of structured collections. Customization options consolidate a wide range of basic collection structures into a few core types (Array, List, Set, Map). These types are defined strictly by their interface, not their internal implementation. Specialized options, however, give control and flexibility for efficient, low-level use (such as implementing other libraries).

Documentation

The collections library follows the `library definition conventions`

`(defobj/library.html)`
. It has following standard sections of documentation:

- `Usage Guide`
`(collections/guide.html)`
- `Advanced Usage Guide`
`(collections/adv.html)`

- `Interface Reference`
`(collections/ref.html)`
- `Subclassing Reference`
`(collections/subclass.html)`

- `Interface Design Notes`
`(collections/design.html)`
- `Implementation Notes`
`(collections/impl.html)`

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Collections Library: Documentation and Implementation Status

The collections library has an almost complete set of Interface Reference documents, and the design of the interface is almost entirely final. The implementation of the collections library, however, doesn't yet implement many of the more advanced features of this interface.

Collections library development has concentrated on the specific features needed by the activity library. This library uses collections for its underlying support of actions to be executed in schedules. The collections library itself is still being completed for use as a general-purpose library.

The Array and List types have all basic capability fully implemented. Set and Map have basic messages defined, but currently rely on a crude implementation based on sorted lists. (This implementation is more than adequate for the usually small and relatively static schedule structures in the activity library, and is the most forgiving when change does take place within members being traversed.) Much more efficient implementations based on both balanced trees and hash tables are in the works.

OrderedSet is currently supported only with the low-level option for an internal member slot, and the implemented messages do not match the ones documented in the Interface Reference. Support for groups of duplicate members is missing from both Set and Map.

None of the special options for restricted usage modes on any type of collection (e.g., read-only restriction) has been implemented. Stack and Queue are not implemented, but are nothing more than restricted uses of a List. There is no support for any member type except id or smaller (other member types will depend on a data type facility to be supplied by defobj).

Even though incomplete, the portion of capability that is implemented has been exercised very heavily. The interfaces to Set and Map structures will remain the same even as their underlying implementations improve, so there is no harm in using them. See the GridTurtle test programs
(../grid.html)
for the most complete examples of collections usage.

The collections library follows the documentation structure suggested by the library interface conventions
(../defobj/library.html)
of the defobj library. There are placeholders at least for each section of documentation (some of which merely indicate that the section is not available yet) so that all links should at least link up with something, whether or not there's anything there.

Throughout the documentation, a parenthesized comment that starts with (.. indicates an editorial comment on the current status of implementation or documentation.

The documentation priority for all libraries is to complete at least their interface reference documents, so that there is the equivalent of Unix "man pages" that summarize

Collections Library: Usage Guide

This section of documentation is not yet available. In the meantime, see the GridTurtle test programs
(../grid.html)
for the most complete examples of collections library usage. If you can't find an example there that exercises a message or option that you want to use, chances are it's not implemented.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Collections Library: Interface Reference

Dependencies

Following are the other header files imported by <collections.h>:

```
#import <defobj.h>
```

The collections library follows the library interface conventions
(../defobj/library.html)
of the defobj library
(../defobj.html)
. It also depends on standard supertypes and classes defined by this library. Initialization of the collections library automatically initializes the defobj library as well. Since defobj also requires the collections library, both must always be linked into an application together.

Object Types (with links to reference pages)

```
Collection
 (Collection.html)
          String
 (String.html)
      OutputStream
 (Stream.html)
   Index
 (Index.html)

     /       |         \
    /        |          \
 Array
 (Array.html)
     List,
 (List.html)
           KeyedCollection,
 (KeyedCltn.html)

             ListIndex
 (List.html)
       KeyedIndex
 (KeyedCltn.html)
```

- Map
(Map.html)
- String
(String.html)
- OutputStream
(Stream.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Description

A collection is a grouping of object references or other data values which are assigned explicitly as members of the collection. Depending on the subtype, collection members may also be maintained in various associations with each other, such as an ordering of members or an association of members with identifying key values. Major Collection subtypes include Array, List, Set and Map. The Collection supertype establishes common conventions (and associated messages) supported by all types of collections.

> All collections support traversal over their members using a separate object called an Index. All collections also follow common rules regarding the types of data values which may be added as members. The next two subsections summarize these basic features of all collections.

Index objects

An index is a special type of object that references a current position in an enumeration sequence over a collection. An enumeration sequence contains every member of a collection exactly once. Every collection defines an associated type of object for its index. The index type of a collection defines additional ways in which the members of a collection may be processed beyond any messages available directly on the collection. Often the operations of an index provide the most basic and flexible means for manipulating individual members of a collection. An index object into a collection may be created at any time. An index is the basic means to traverse through all members of a collection. Multiple indexes on the same collection may all exist at the same time and may reference the same or different positions. Depending on the collection type, it may be possible to modify a collection through its indexes.

> Once an index is created, the sequence of members in its enumeration sequence is guaranteed to remain the same, provided that no new members are added to the underlying collection, or existing members removed. If a member is located once at a particular position, it is guaranteed to remain at that position as long as the index itself remains.
>
> Many collection types define an explicit ordering over their members. For such collections the sequence of members referred to by an index will always be consistent with this ordering. An explicit, total ordering also guarantees that all indexes of the same collection have the same member sequence.
>
> If no such ordering is defined, however, some particular sequence of all the collection still becomes associated with each created index. All collection members are guaranteed to be contained somewhere in the enumeration sequence for any particular index, but two indexes on the same collection are not guaranteed to have the same sequence.
>
> The Index type corresponds to the iterator types defined as part of many other object-orien libraries. The name Index is shorter and emphasizes the more abstract and multi-function role of these basic support objects for any collection. For more background on design of indexes and iterators, see the Interface Design Notes for the collections library.

being created. Members (and keys, too, if applicable) are added to the new collection in the same order as defined by traversing the initial value collection with a local temporary index. Create-time options are also applied as noted under SetInitialValue. Some collection subtypes may have further special rules for initial values; if so, these are documented with the subtype.

[aCollection copy: aZone] is equivalent to [[aCollection getType] create: aZone setInitialValue: aCollection].

Create-time options

```
CREATING
- (void) setMemberType: aDataType;
- (void) setIndexSafety: indexSafety;
- (void) setIndexUpdateHandler: (handler_t)handler : (void *)arg;
SETTING
- (void) setReplaceOnly: (BOOL)replaceOnly;
USING
- getMemberType;
- getIndexSafety;
- (handler_t) getIndexUpdateHandler: (void *)arg;
- (BOOL) getReplaceOnly;
```

MemberType

(.. This option is not currently implemented for any collection types.)

The MemberType option may be used to declare the type of member which a collection contains. Its value must be an objec having one of the ValueType types defined in defobj. (..Currently no ValueType objects are implemented, so MemberType is not supported.

The member type defaults to the Symbol value Unknown. Specifying a member type has no effect on the collection if the storage size of the data type is no larger than a pointer of id type. The member type, however, may be helpful as documentation, and members can be checked for conformance to the type as part of debug options. If the data type has a size greater than id, the collection automatically allocates and frees space for the member value whenever member values are added or removed. For such a collection, all member values must be passed to the collection not as the value itself, but as a pointer to the value (cast to id as necessary). If a new member is being added, the value will be copied to a new member location contained within the collection. Member values returned by the collection will be returned as pointers to this internal value, not as the value itself. In the case of member values removed from the collection, a nil pointer is returned.

ReplaceOnly

(.. This option is not currently implemented for any collection types.)

This boolean-valued option restricts valid usage of a collection by excluding all operations which add or remove members. For some collection subtypes, a replace-only restriction can obtain many of the same performance advantages as a read-only collection, but without disabling replace operations as well. Just like the ReadOnly option, the ReplaceOnly option may be reset after a collection is created, provided it was not originally set to true.

Collection behavior

```
- (int) getCount;
```

getCount returns the integer number of members currently contained in the collection. All collections maintain their count internally so that no traversal of collection members is required simply to return this value.

```
- atOffset: (int)offset;
- atOffset: (int)offset put: anObject;

- getFirst;
- getLast;
```

An offset is an integer value that gives relative position of a member in the enumeration sequence of a collection. Offsets start the count of the first member at zero, just like C array indexing.

Offsets provide an alternate means to access the members of a collection, without creating a separate index object. Some collection subtypes (such as Array) support fast, direct access by integer member offset, while others support member offsets only as a shorthand for sequential access through every preceding member. Access by offsets is supported on all collections regardless of whether its speed on a particular collection type.

The atOffset: message returns the member at a particular member offset. The atOffset: put: message replaces the member at a particular offset with a new value, and returns the previous member value at this offset. Both these messages raise the error OffsetOutOfR if the offset is less than zero or greater than or equal to the count of members in the collection.

The getFirst message is equivalent to [aCollection atOffset: 0], and the getLast message is equivalent to [aCollection atOffset: [aCollection getCount] - 1].

```
- (BOOL) contains: aMember;
- remove: aMember;
- (void) removeAll;
```

The contains: message returns true if the collection contains any member value which matches the value passed as its argument. Depending on the collection subtype, this may require traversing

Name

Index – reference into an enumeration sequence for a collection

Synopsis

```
//
// Index -- reference into an enumeration sequence for a collection
//
@deftype Index <DefinedObject, Copy, Drop>- getCollection;

- next;
- prev;
- findNext: anObject;
- findPrev: anObject;

- get;
- put: anObject;
- remove;

- getLoc;
- (void) setLoc: locSymbol;
- (int) getOffset;
- setOffset: (int)offset;

- (int) compare: anIndex;
@end
```

Description

An index is a reference into an enumeration sequence of a collection. Such an enumeration sequence contains all members of the collection in some order. This order will always be consistent with ordering of members in the collection, assuming there is such an ordering. Otherwise, the sequence will still contain all members in some order that remains fixed provided that new members are not added or removed from the collection.

An index is created by a begin: or createIndex: message against a collection. Each major collection type has its own corresponding index type, which supports specialized types of processing against the valid contents of that kind of collection. Once created, an index is a separate object from the collection itself, but it remains valid only so long as the collection itself still exists. Multiple indexes may exist at the same time against the same collection, and each index maintains its own position within an enumeration sequence for the collection.

The next message positions the index to the next valid member after its current position, or to a special position after the end of all valid members. The prev message works similarly, but positions to a valid member preceding the current position, or to a special position preceding all valid members. In addition to repositioning the index, both messages return the new member value to which they are positioned, or nil if there is no such member.

Following is a standard form of loop which may be used to enumerate the members of a collection using an index:

```
index = [aCollection begin: aZone];
  while ( (member = [index next]) ) {
    // do something with member ...
  }
```

When first created, the current position of an index is always set to the special position preceding all valid members. From this initial position, this loop accesses each succeeding member in turn. It assigns each member value to a local variable ("member" in this example) until a nil value is returned.

```
- findNext: anObject;
- findPrev: anObject;
```

findNext: and findPrev: repeatedly perform next or prev until the member value of the index matches their argument. nil is returned if the index reaches the end of valid members without matching the argument. They are only convenience shorthands for loops which could otherwise be programmed using next or prev.

```
- get;
- put: anObject;
- remove;
```

get returns the member value at which the index is currently positioned, or nil if the index is not positioned at a member The put: message replaces the member value at the current index position with its argument. Neither of these messages changes the current position of the index.

The get message provides an alternate way to obtain the current member value in a loop that traverses a collection; its return value is the same as next or prev would return when first positioning to a new member.

The remove message removes the member at the current location of an index, and returns the member value removed. The index position is set to a special position between the members which previously preceded and followed the removed member. If there is no preceding or following member, the index is set to the special location before the start or after the end of all members. After a current member is removed, there is no member at the current index location, but a subsequent next or prev message will continue with the same member that would have been accessed had the current member not been removed.

Both put: and remove raise the InvalidIndexLoc error if the index is not positioned at a current member.

```
- getLoc;
- (void) setLoc: locSymbol;
```

depends on the specific type of the collection, just as noted for the atOffset: message on Collection. In the worst case, this operation is linear in the magnitude of the offset.

```
- (int) compare: anIndex;
```

The compare: message compares the current location of one index with the current location of another index passed as its argument. If the two indexes have the same location, compare: returns zero. Otherwise, compare: returns +1 or -1 according to whether the argument index precedes or follows the receiver index in the enumeration sequence of the collection. If either of the two indexes has an unknown offset, and the location of the other index is anything other than Start or End or an immediately adjacent member, compare: returns the UnknownOffset integer value.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

however, the Array type can be used to wrap an existing C array for external access as an object collection. It can also provide access to an internal C array for direct manipulation using C expressions. These forms of low-level access support hybrid modes of use in which advantages of both low-level manipulation and external object access can be combined.

Inherited behavior

```
@deftype Array <Collection, CREATABLE>
```

The Array type is directly creatable, and supports all standard messages of Collection except removal of individual members. The messages based on an integer offset, either on the collection (atOffset:, atOffset:put:), or an index (setOffset:) all execute in fast constant time. Members of an array are fully ordered according to these integer offsets. Sequential access to members through its members is also fully supported. The Array type disables the remove message inherited from Collection; the message is defined, but any attempt to remove a member will raise an error that the operation is not supported.

While the IndexSafety option of an Array defaults to Unsafe, indexes are inherently safe under all operations with the exception of resizing an array or resetting its member allocation. (The only individual member operation which modifies contents of an array is replacing a member value, and this does not affect existing indexes.) The UnsafeAtMember value of IndexSafety is not supported since individual members cannot be added or removed. A value of SafeAlways makes existing indexes safe even under resizing or reallocation. If an existing index referred to a member location that is no longer contained in the array, the index location is reset to the End location.

The default value of the ReplaceOnly option is true, and cannot be overridden.

The type of index returned by begin: on an array is simply Index. There is no special index type for Array because there are no additional messages beyond those already defined by Index.

Create-time options

```
SETIING
- (void) setCount: (int)count;
- (void) setDefaultMember: memberValue;

- (void) setMemberBlock: (void *)members setCount: (int)count;
USING
- (void) setDefaultMember: memberValue;
- getDefaultMember;

- (void) setMemberBlock: (void *)members setCount: (int)count;
```

being used, the external program is entirely responsible for assuring that the MemberBlock value is a pointer to valid allocated memory containing at least the number of member slots given by setCount:.

With an external member allocation, the array itself will not attempt to either allocate this memory or free it when the array is dropped. Dropping the array only removes its reference to the external allocation.

The pointer to allocated memory is given as a pointer to id type, but if a data type given as MemberType has size greater than id, the pointer must actually point to an array of slots each of which has the size of the data type.

Since an array neither allocates nor frees an external member allocation, the same region of allocated memory may be referenced by multiple arrays, including overlapping member ranges each defined by starting location and count. This flexibility enables alternate subrange views of a single, contiguous initial allocation by means of separately created external collections.

Create combination messages

```
CREATING
+ create: aZone setCount: (int)count;
+ create: aZone setMemberBlock:(id *)members setCount:(int)count;
```

In addition to the standard inherited create messages, the Array type supports two forms of combination messages which both create the array and set its initial options. These are provided both for convenience and to provide the fastest possible array creation for simple cases in which speed might be important. The two cases supported are creating an array with a member count for its internal members already set, and creating an array which refers to an external member allocation. If any more options than these are needed, the normal method of setting options between createBegin: and createEnd messages should be used instead.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Inherited behavior

```
@deftype List <Collection, CREATABLE>
```

The List type is supports all messages of Collection. If created with default options, it provides no special speedup of accesses by integer offset.

Unless the EndsOnly option is specified, the default value of IndexSafety is UnsafeAtMember This means that indexes are inherently safe under all operations except those which affect the member at an index or immediately adjacent to an index between members. If the EndsOnly option is specified, the default value of IndexSafety is Unsafe. In this case, indexes could be invalidated by any operation that adds or removes members. The SafeAlways can be specified for any case to get a guarantee of full index safety.

ListIndex is a special index type just for list collections. It provides the only method to add members within the interior of a list.

Create-time options

```
CREATING
- (void) setEndsOnly: (BOOL)endsOnly;
USING
- (BOOL) getEndsOnly;
```

EndsOnly

(.. This option is not implemented yet.) The EndsOnly option restricts a list so that members can be added or removed only at either end. This restriction permits an alternate implementation that both reduces memory usage and enables fast, direct access to members by integer offset. It obtains key performance advantages of an array while preserving the dynamic extendability of a list.

List behavior

```
- (void) addFirst: anObject;
- (void) addLast: anObject;
```

The addFirst: and addLast: messages add a new member to either end of a list. With addFirst:, the new member becomes the first member of the array. With addLast:, it becomes the last member.

```
- removeFirst;
- removeLast;
```

removeFirst and removeLast remove either the first or last member of a list. They each return the value of the member just removed.

Name

Stack, Queue – restricted forms of lists

(.. Neither of these types is currently implemented. They are nothing more than restricted cases of a List, so if you need them use a List instead.)

Synopsis

```
// Stack -- form of list with add/remove both at start of list
//
@deftype Stack <List>- (void) push: anObject;
- pop;
@end

//
// Queue -- form of list with add at start of list and remove at last
//
@deftype Queue <List>- (void) enqueue: anObject;
- dequeue;
@end
```

Description

A Stack is a subtype of List which restricts the addition and removal of members to the start of a list. Queue is a subtype of List which restricts addition of members to the start of the list and removal to the end. Both types define specially named messages to reflect the specialized usage of their resulting add and remove operations.

Interface Reference

Inherited behavior

```
@deftype Stack <List>@deftype Queue <List>
...
```

Name

KeyedCollection, KeyedIndex – supertype of Set and Map types

Synopsis

```
//
// KeyedCollection -- supertype of Set and Map types
//
@deftype KeyedCollection <Collection>CREATING
- (void) setDuplicatesOption: duplicatesOption;
- (void) setGroupType: aCollectionType;
- (void) setSingletonGroups: (BOOL)singletonGroups;

- (void) setSorted: (BOOL)sorted;
- (void) setCompareFunction: (idXid_t)aFunction;
-
(void)        setBucketFunction: (bucket_t)aFunction setBucketCount: (int)n;

- (void) setPartiallyOrdered: (BOOL)partiallyOrdered;
- (void) setMemberSlot: (int)byteOffset;
SETTING
- (void) setPartialOrderContext: aKeyedCollectionCollection;
USING
- getDupOption;
- getGroupType;
- (BOOL) getSingletonGroups;

- (BOOL) getSorted;
- (int(*)()) getCompareFunction;
- (int(*)()) getBucketFunction;
- (int) getMaxBuckets;

- (BOOL) getPartiallyOrdered;
- getPartialOrderContext;

- (int) getMemberSlot;

- at: aKey;
- (BOOL) containsKey: aKey;
- (int) getCountAtKey: aKey;

- removeKey: aKey;
```

Inherited behavior

```
@deftype KeyedCollection <Collection>
```

The KeyedCollection type inherits all standard behavior of Collection. The KeyedCollection type is not itself creatable; it only serves as a common supertype for Set and Map collection types.

The default value of IndexSafety for a KeyedCollection is Unsafe, but values of either UnsafeAtMember or SafeAlways can be requested to guarantee a higher level of safety.

Create-time options

(.. None of these options is implemented yet, except for CompareFunction.)

```
CREATING
- (void) setDupOption: dupOption;
- (void) setGroupType: aCollectionType;
- (void) setSingletonGroups: singletonGroups;

- (void) setSorted: (BOOL)sorted;
- (void) setCompareFunction: (idXid_t)aFunction;

- (void) setPartiallyOrdered: (BOOL)partiallyOrdered;

- (void) setMemberSlot: (int)byteOffset;
SETTING
-
(void)        setBucketFunction: (bucket_t)aFunction setBucketCount: (int)n;
- (void) setPartialOrderContext: aKeyedCollection;
USING
- getDupOption;
- getGroupType;
- (BOOL) getSingletonGroups;

- (BOOL) getSorted;
- (int(*)()) getCompareFunction;

- (BOOL) getPartiallyOrdered;
- getPartialOrderContext;

- (int) getMemberSlot;

- (int(*)()) getBucketFunction;
- (int) getBucketCount;
```

Sorted

If this option is true, the immediate members of a keyed collection (included collections of duplicate members, if any) are totally ordered according to an ordering relation defined on key values. The default order for sorting is determined by a standard compare: message on a key value to be added vs. existing keys of the collection. This default method to establish order can be overridden using the CompareFunction option.

The default for the Sorted function is true, unless a value is also given for the option BucketFunction.

CompareFunction

Use the function pointed to by the argument as the method for determining whether two members have the same key value, and also for ordering the keys of the collection if the Sorted option is true. The argument is declared with the following function pointer type:

```
typedef int (*idXid_t)( id, id );
```

The function given will be called whenever one key value needs to be compared with another. Multiple calls to the function typically occur whenever members are added or removed from the collection, until the correct member for insertion or removal is determined.

The compare function accepts two key values as its arguments. Like member types, key types declared with the id pointer type, but may contain any type of value (such as an integer) up to the size of an id. An explicit compare function can support any type of value as a key, regardless of whether it is an object that supports a standard compare: message.

The compare function is called repeatedly by the collection to compare two key values. The function should return zero if the key values are equal, -1 for the first key argument less than the second, and +1 for the first greater than the second. If a keyed collection is not sorted, either -1 or +1 may be returned for unequal keys, regardless of whether one might be taken as greater or less than the other.

BucketFunction

The BucketFunction option specifies a function that is called whenever a member of an unsorted collection is added or removed. It may also be used for a sorted collection to provide faster insertion and removal of members. The responsibility of a "bucket function" is to assign the key value to some integer in a range from zero up to some maximum value. These integers identify "buckets" to which the same key value will also be assigned whenever passed to the function. If the collection is unsorted, these buckets represent hash buckets, and for speed should be relatively sparse (few keys likely in any one bucket, given the expected size of the collection). If the collection is sorted, these buckets represent ordered buckets to which the same range of key values will always be assigned. Such sorted buckets can speed up the search for a key location within a sorted collection. Sorted buckets can also be sparse to minimize the need for further sorting within each bucket. If many keys end up in the same sorted bucket, the sorting process normally used without buckets is still used, so there is less performance impact than with hash keys of an unsorted collection.

getMemberSlot

The MemberSlot option indicates that space has been reserved within each member that allows it to contain a link directly to its position in the enumeration for a collection. If such space has been reserved, special messages can be used that rapidly position an index directly to the member. Operations to remove members are also much faster.

The value of MemberSlot specifies the offset in bytes from the start of each member where the space for its position link has been reserved. The following typedefs are provided to declare the space required for a position link within a member allocation:

```
typedef struct memberData { void *memberData[2]; }             member_t;
typedef struct memberData { void *memberData[2]; id owner; }  dupmember_t;
```

The first type declares the required space for a position link in a collection that does not accept duplicate key values. The second type is required if duplicate keys are permitted.

The offset of the position link from the start of a member may be either positive or negative, in the range of -2048 to +2047. The default value of MemberSlot is UnknownOff (a large negative value), which specifies that no slot for an internal position link is available within each member.

If a member slot is defined, a memory pointer of some kind must always be used for the value of every member. These members may be object id pointers, but other types of memory pointers are acceptable as well.

The contents of the first two words of position link vary according to the position of a member in a collection. The third word of a dupmember_t link contains the id of the collection in which a member is directly contained. This collection is either an internally created collection containing members with duplicate keys, or the collection to which the member was added if there is no other member with the same key.

KeyedCollection behavior

(.. None of the messages or other behavior relating to duplicate key values are currently implemented.)

```
- at: aKey;
- (BOOL) containsKey: aKey;
- (int) getCountAtKey: aKey;  // unimplemented

- removeKey: aKey;
```

The at: message returns the existing member of the collection which matches the key value passed as its argument, or nil if there is no key value in the collection which matches. If duplicate entries for this key exist, the entire collection of duplicate members created for the key value is returned instead.

The containsKey: message returns true if the key value passed as its argument is contained in the collection, and false otherwise. The getCountAtKey message returns

```
- setMember: aMember;
```

(.. This message is not currently implemented.) The setMember: message repositions the index to the member passed as its argument. This message is valid only if an internal member slot was defined for the collection with the MemberSlot option.

Roger Burkhart <rmb@santafe.edu>
(mailto:rmb@santafe.edu)

Last modified: 1997-12-19 (mgd)

@end

The add: message adds a new member to a set. It returns true if member added matched the key of any member already contained in the Set. Whether or not the member was actually added to the collection or not depends on the setting of DupOption.

The add:setIndex: message adds a new member to a set just like add:, but also has a side effect of setting the position of the index passed as the setIndex: argument to the member which was just added. If an index positioned to the new member is needed anyway, setting the index in the same operation avoids repeating a search for the position at which the member key belongs.

The replace: message replaces a member stored at a given key with another member value that matches the same key. The new value to replace the existing one is passed as the argument. replace: returns the member value that was replaced, or nil if the collection contained no member with a matching key.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Name

Map – collection of associations from key objects to member objects

Synopsis

```
//
// Map -- collection of associations from key objects to member objects
//
@deftype Map <KeyedCollection, CREATABLE>CREATING
- (void) setKeyType: aDataType;
- (void) setKeySlot: (int)byteOffset;
USING
-          getKeyType;
- (int) getKeySlot;

- (BOOL) at: aKey insert: anObject;
- (BOOL) at: aKey insert: anObject setIndex: anIndex;
- at: aKey replace: anObject;

- (BOOL) insertGroup: aKey;

- createIndex: aZone setKey: aKey;
@end

@deftype MapIndex <KeyedIndex>- getKey;
- setKey: aKey;

- get: (id *)key;
- next: (id *)key;
- prev: (id *)key;

- replaceKey: aKey;
end
```

Description

Map is a subtype of KeyedCollection in which the key value associated with each member is independen of the member itself. Whenever a new member is added to the collection, a key value to be associated with the member must be supplied also. A Map defines a mapping from key values to member values.

(.. This message is not currently implemented. It will eventually give a fast way to create an index already positioned at an entry with a particular key value.)

MapIndex behavior

```
- getKey;
- setKey: aKey;
```

The getKey message returns the key value associated with the current location of the index. It returns nil if the index is not currently positioned at a member. The setKey: messages repositions the index to an entry having a key value that matches its argument. If there is more than one entry matching this key value, the index is positioned to the first entry that matches.

```
- get: (id *)key;
- next: (id *)key;
- prev: (id *)key;
```

These messages are similar to those defined by the Index supertype, except that they combine the return of a member value with an output value for an associated key value. In each case, the key value is passed as an (id *) output argument.

```
- replaceKey: aKey;
```

(.. This message is not currently implemented.)

> replaceKey: replaces a key value with another key value that still compares equal to a key value already present. It may be used to replace an allocation of one object, already contained in a Map, by a different object allocation. replaceKey: returns nil if no existing key value matches its key value argument; otherwise it returns the key value which was replaced by its argument.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Create-time options

```
CREATING
+ create: aZone setC: (const char *)cstring;
SETTING
- (void) setC: (const char *)cstring;
```

...

String behavior

```
- (const char *)getC;
- (void) appendC: (const char *)cstring;

- (int) getCount;
- (int) compare: aString;
@end
```

...

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

...

Special create: message

```
+ create: aZone setFileStream: (FILE *)fileStream;
```

...

OutputStream behavior

```
- (void) appendC: (const char *)cstring;
@end
```

...

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Collections Library: Interface Design Notes

Design Objectives

A collections library is one of the most important foundation services for object-oriented programming. Most object-oriented systems provide at least the start of a general-purpose collections library. The GNUSTEP, project, for example, provides the libobject library (currently in alpha test at ftp://alpha.prep.ai.mit.edu/) which includes a collections library along with other services intended to parallel those of the OpenStep framework developed by Next.

Swarm has implemented its own collection library to meet the specialized needs of its agent simulation framework. Major objectives of the Swarm collections library include the following:

... (.. more to come, obviously)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08