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)

Last modified: 1998-01-08

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 all basic capability. A second priority is to complete the complementary "Usage Guide" documents. Unlike the reference documents, the Usage Guide will have a task-oriented organization, and will lead the initial user through actual code examples in the rough order a typical user is likely to need them. It will serve the role of a tutorial on each library.

The Usage Guide code examples have not yet been developed. For the time being, a directory of test programs (GridTurtle test programs
(../grid.html)
, contained within the documentation release directory) provides code examples of many of the basic features of the defobj, collections, and activity libraries. These code examples also help indicate the portions of the libraries which are fully implemented and working, since they are run on each new release of these libraries.

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

Last modified: 1998-01-08

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: Advanced Usage Guide

(This section of documentation is not yet available.)

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)
```

```
   /        \         /            \
Stack
 (Stack.html)
    Queue
 (Stack.html)
    Set,
 (Set.html)
         Map,
 (Map.html)

            SetIndex
 (Set.html)
     MapIndex
 (Map.html)

                             |
          OrderedSet
 (OrderedSet.html)
```
(need a graphic with image map, but this will do for now)

Contents:

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

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

Last modified: 1998-01-08

Name

Collection - supertype of all collection types

Synopsis

```
//
// Collection -- supertype of all collection types
//
@deftype Collection <Create, SetInitialValue, Copy, Drop>CREATING
- (void) setMemberType: aDataType;
- (void) setIndexSafety: indexSafety;
- (void) setIndexUpdateHandler: (func_t)handler : (void *)arg;
- (void) setReplaceOnly: (BOOL)replaceOnly;
USING
- getMemberType;
- getIndexSafety;
- (func_t) getIndexUpdateHandler: (void *)arg;
- (BOOL) getReplaceOnly;

- (int) getCount;

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

- getFirst;
- getLast;

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

- begin: aZone;

- (void) forEach: (SEL)aSelector;
- (void) forEach: (SEL)aSelector : arg1;
- (void) forEach: (SEL)aSelector : arg1 : arg2;
- (void) forEach: (SEL)aSelector : arg1 : arg2 : arg3;
@end
```

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-oriented 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.

Member types

The default member type of any collection is the id type defined by Objective C. All messages which add, remove, or access members declare their member value with the id type. None of the collection types, however, requires that the member value actually point to a valid object. The member values of any collection may contain an arbritrary bit pattern, provided these bits fit in the space available within an id pointer.

> In the few cases when default operations assume that a member points to a valid object, there are alternate ways to avoid these defaults and use an arbritrary bit pattern as a member value instead. When using an arbritrary bit pattern, the member value must be cast to or from the id type when adding or retrieving it from the collection. By global portability assumptions
> (../global-port.html)
> , the number of bits in an id value may be safely assumed to be at least as large as an int (signed or unsigned), but not necessarily as large as a long.
>
> There is also an option to override the default member type so that it can be larger than a id pointer. Use of this option, however, changes the way member values must be passed to or from a collection. More information is given with the MemberType option below.

Inherited behavior

    @deftype Collection <Create, SetInitialValue, Copy, Drop>

(.. SetInitialValue and Copy behavior is currently implemented for Array and List, but not for Set or Map.)

> The Collection type inherits standard messages from the defobj library for creating and customizing objects. These messages are defined as part of the defobj Create supertype. The only collection types which are directly creatable, however, are those directly tagged as such (with the CREATABLE type tag).
>
> In addition to the standard create messages inherited from the defobj Create supertype, Collection inherits the SetIntialValue, Copy, and Drop types (also from defobj). Copying or dropping a collection has no effect on objects which might have be contained in a collection as members; only the references to objects contained in the collection are copied or freed.
>
> SetInitialValue defines special create messages to load the initial members of a newly created collection from an existing collection, and optionally to make the members of a collection read-only after this initialization. If the members of a collection are read-only, all collection messages which add, replace, or remove members will raise an InvalidOperation error. Even if a collection was not initially created with ReadOnly status, this status may also be set later. If a collection is created with the ReadOnly restriction set, however, this restriction cannot later be unset.
>
> A collection used as an initial value need not have the same type as a collection

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.

IndexSafety

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

The IndexSafety option has one of the following values:

```
id &lt;Symbol&gt;  Unsafe, UnsafeAtMember, SafeAlways;
```

If not specified, the default value depends on the specific collection subtype. In typical cases, the default value is Unsafe. A non-default value is required for indexes to have a guarantee of safety when independent modifications to the underlying collection are also performed.

Without a guarantee of index safety, the effect of any operation which modifies the underlying collection (adding, removing, or replacing members) could be to disrupt the position assumed by any existing index. The effect of such disruption is, in general, completely unpredictable: it could be an immediate crash, a wrong sequence of members, or other difficult-to-debug conditions. To avoid such unpredictability, several means for assuring the safety of indexes under independent collection update are provided.

The simplest way to guarantee index safety is to set the IndexSafety option to the value SafeAlways when a collection is created. With the value of SafeAlways, any active index on the collection will be updated automatically to reflect any change to the underlying collection whenever such a change occurs. This automatic update continues to occur throughout the lifetime of any index on the collection; it ceases to occur only after an index is dropped. This is the easiest option to use, but consumes extra resources to accomplish (both time and space within the collection), so is most suitable when the extra resource usage deesn't matter or there's no better way.

With an IndexSafety value of SafeAlways, there is also an additional option to declare an external index update handler. The handler is specified using the setIndexUpdateHandler:: message. If an IndexUpdateHandler is specified, the value for IndexSafety is also automatically set to SafeAlways.

The first argument of setIndexUpdateHandler:: specifies a function which is called whenever a modification to the underlying collection occurs which could require an update to existing indexes. The second argument establishes a pointer value which the handler function may use to identify the indexes which need to be updated, or any other information needed for the update.

(.. Exact call conventions for the IndexUpdateHandler are still being established, but the activity library will soon be converted to use this technique, so the conventions will be resolved during that conversion.)

Besides the extreme values of Unsafe and SafeAlways, another valid value for IndexSafety is UnsafeAtMember. This value indicates that the only modification of the collection which could disrupt existing indexes is to remove the member contained at the current index position, or to change the member on either side of an index positioned between members. For some collection subtypes, this level of safety is automatically assured, and for others it can be requested with less overall resource impact than SafeAlways.

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 OffsetOutOfRange 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 sequentially through all members of the collection until a matching member is found. For other subtypes, some form of direct indexing from the member value may be supported. The message is supported regardless of its speed.

The remove: message removes the first member in the collection with a value matching the value passed as its argument. If there is no such member, a nil value is returned. As with the contains: message, the speed of this operation may vary from very low to linear in the number of members, depending on the collection subtype.

The removeAll message removes all existing members of a collection and sets its member count to zero. The collection then remains valid for further members to be added. This message has no effect on the objects which might be referenced by any removed member values. If resources consumed by these objects also need to be released, such release operations (such as drop messages) can be performed prior to removing the member values.

```
- begin: aZone;
```

The begin: message is the standard method for creating a new index for traversing the elements of a collection. All further information about indexes is documented under the Index type.

```
- (void) forEach: (SEL)aSelector;
- (void) forEach: (SEL)aSelector : arg1;
- (void) forEach: (SEL)aSelector : arg1 : arg2;
- (void) forEach: (SEL)aSelector : arg1 : arg2 : arg3;
```

The forEach messages supply a convenient shorthand for repeatedly performing the same message on all objects contained as members in a collection. The message to be sent is identified by the argument aSelector. This selector must define the same number of arguments as contained in any remaining argument slots of the forEach message. The argument types of the message to be sent must be either the id type, or some other type that will fit in the same number of bits as the id type. By global portability assumptions, the argument type could be as large as an int (signed or unsigned), but not necessarily as large as a long. To use the message, any non-id value must be cast to the id type as part of the call expression.

The forEach: messages are implemented by a simple loop through all members of a collection, using an internal, temporary index. If any operation more complex than a simple message send is required, this operation should just be coded directly using a loop that traverses its own index.

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

Last modified: 1998-01-08

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.

Many indexes provde the ability modify the collection they refer to, in addition to simply traversing members. An index often provides the complete means for maintaining the contents of a collection, more than could otherwise be performed on the collection itself. The position or other status of the index is automatically updated to reflect any changes made through the index itself.

If changes to a collection are made while other indexes exist, those other indexes could be affected in potentially catastrophic ways. Each collection type documents which kinds of changes can be made without affecting or invalidating existing indexes. The IndexSafety option of Collection provides additional ways to protect indexes against possible effects of independent updates.

Inherited behavior

```
@deftype Index <DefinedObject, Copy, Drop>
```

Each index is a stand-alone object allocated within a zone passed as an argument in the message that created it. This zone need not match the zone of a collection. It is common for index lifetimes to be shorter than their collection. For example, indexes can be created in a temporary scratch zone for use only within a local loop.

Because messages to a collection are the only valid way to create an index, create messages and create-time options are not used with index types. All valid processing on an index is determined by characteristics of the collection from which it is created. Index types are typically named after the type of collection they are created from, and serve principally to define the specific messages valid for an index on that type of collection.

Index objects support the universal messages of the top-level DefinedObject supertype, along with the standard drop: and getZone messages. Even though they cannot be created except from a collection, new index objects can be created from an existing index using the standard copy: message. Each copy refers to the same collection as the initial index, and starts at the same position in its enumeration sequence. In all other respects, however, the new copy is an independent index that maintains its own position under any further processing.

Index behavior

```
- getCollection;
```

getCollection returns the collection referred to by an index. This collection never changes during the lifetime of the index.

```
- next;
- prev;
```

The next and prev messages are the basic means to traverse through the members of a 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;
```

The getLoc message returns a symbol constant which indicates the type of location at which an index is currently positioned. This index location symbol has one of the following values:

```
extern id <Symbol>  Start, End, Between, Member;
```

The Start symbol indicates the special position preceding all members in the enumeration sequence for the collection. This is the location at which an index is positioned when it is first created. The End symbol indicates the special position following all members in the collection. This is the location at which an index is positioned just after a next message has returned nil, as a result of moving beyond the last member. The Between symbol indicates the special position between two other members, at which an index is positioned after a current member, between two other members, is removed. The Member symbol indicates that the index is positioned at some current member in the enumeration sequence of a collection.

The getLoc message is needed to traverse a collection which could contain nil values for its members. Without getLoc, there would be no way to distinguish a nil value returned by next as either a valid member value or the special value returned at the end of members. With getLoc, a loop that traverses a collection can test specifically for the end (or start) of members. Following is a simple loop which illustrates such usage: index = [aCollection begin: aZone]; do { member = [index next]; // do something with member ... } while ([index getLoc] == Member); The setLoc: message may be used to reset the current location of an index to either Start or End. It may be used to reprocess a collection using an existing index after some other location has already been reached. It may also be used to position an index at the end of all members prior to traversing members in reverse order using prev.

Besides Start and End, setLoc: accepts the special argument values of BetweenAfter and BetweenBefore, which are also defined symbols. These argument values are only valid if the index is positioned at a current member. They reposition the index to the special location between the current member and its immediately following or preceding member.

```
- (int) getOffset;
- setOffset: (int)offset;
```

Provided there is no major computational cost, an index also maintains the integer offset of its current member within the enumeration sequence of the collection. These integer offset values have the same definition as in the atOffset: messages of Collection. The getOffset message returns this current offset value. If the index is current positioned at the Start or End location, getOffset returns -1. If the index is positioned at a Between location, getOffset returns the offset of the immediately preceding member. If the offset is not currently available from the index, getOffset returns the special value UnknownOffset. This value is defined by a macro as the maximally negative value of a 32-bit, 2's-complement integer.

An offset is always available from an index if its current position has been reached by repeated operations from an initial Start or End position, and there has been no other modification to the underlying collection. Some forms of direct member access operations supported by some index types, however, may result in an integer offset not being available. These restrictions are noted with the individual index type.

Using the setOffset: message, an index may be positioned directly to a member using the offset of the member within its enumeration sequence. The speed of this operation 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

Name

Array – collection supporting access to fixed group of members

Synopsis

```
//
// Array -- collection supporting access to fixed group of members
//
@deftype Array <Collection, CREATABLE>CREATING
+ create: aZone setCount: (int)count;
+ create: aZone setMemberBlock:(id *)members setCount:(int)count;
SETTING
- (void) setCount: (int)count;
- (void) setDefaultMember: memberValue;

- (void) setMemberBlock: (id *)members setCount: (int)count;
USING
- getDefaultMember;
- (id *) getMemberBlock;
@end
```

Description

An array is a collection of members that are all created as members of the collection at the same time. Existing member values may be replaced with new values, but the members themselves are fixed at constant offsets within the collection. The fixed structure of an array permits very fast access to members by integer offset positions, since the location of each member may be directly calculated.

The Array type is one of the simplest collection types in the collections library, and the closest to a data structure directly supported in C. Unlike C arrays, the group of members belonging to the array is not necessarily fixed for the lifetime of the array, but may be dynamically resized to contain a different number of members. When an array is dynamically resized, existing member values are preserved as much as possible.

The Array type adds few messages to the generic messages inherited from Collection. This type is provided partly so that a fixed-structure array can be accessed with the same uniform set of basic messages as any other kind of object collection. It also handles all required memory allocation within the collection. As an option, 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;
```

```
- (void *) getMemberBlock;
```

All the Array create-time options can also be set after the array is already created, subject to restrictions noted below.

Count

The Count option sets the number of members which belong to the collection. Any non-negative value including zero is valid. If the array already exists, the any existing members up to the new count will preserve their existing values. If the new count is greater than the existing count, or a new array is being created, all members will be assigned an initial default value of either nil, or a value previously specified for DefaultMember.

> The inherited getCount message of Collection obtains the current value set for this option. There is no way to add or remove members of an array except using the setCount: message directly.

DefaultMember

When this option is set, the initial value of all new members will be set to the member value given (otherwise the default is nil). This option gives a convenient way to distinguish members which have never been set from any other valid member value, which could include nil. This option may be reset after array creation only if some setting for the option was given at create time. (The initial, explicitly set value can still be the default nil, but a value must be set explicitly for the option to be resettable later). The get message for this option always retrieves the current setting, but this value has no effect except when the count of an array is increased, so that new members need to be initialized.

MemberBlock

This option provides a means to wrap an existing C array for access as an object collection. If this option is given, dynamic resizing is not supported. The current C array being wrapped, however, can be replaced by giving a new setting for MemberBlock after an array already exists. A new setting can be given only if some setting was initially given at create time.

> Even if a setting was not given for MemberBlock at create time, the get message for MemberBlock always returns a pointer to whatever internal memory array is currently being used by the array object. If a MemberBlock setting was given, the pointer returned will be the same as the one previously given. In either case, the pointer returned may be used to manipulate member values in any way desired using native C expressions. After an array has been created, there is no way to determine whether the MemberBlock pointer was established externally or by internal allocation. If the pointer was established by internal allocation, however, the external program must make no attempt to free or otherwise reallocate this memory.
>
> A count must always be supplied with an external member allocation, using the setCount: argument of the compound message. If an external allocation is being used, the only way to reset the count is also to reset MemberBlock; any attempt to use the setCount: message by itself will raise an error. Whenever an external member allocation is

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

Name

List, ListIndex – collection of members in linear sequence

Synopsis

```
//
// List - collection of members in linear sequence
//
@deftype List <Collection, CREATABLE>CREATING
- (void) setEndsOnly: (BOOL)endsOnly;
USING
- (BOOL) getEndsOnly;

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

- removeFirst;
- removeLast;
@end

//
// ListIndex -- index with insertion capability at any point in list
//
@deftype ListIndex <Index>- (void) addAfter: anObject;
- (void) addBefore: anObject;
@end
```

Description

A list is a collection of members that are all maintained at some externally assigned position in a linear sequence of all members. The sequence is established by the position at which members are added: members can be added at the start of list, at the end, or at any point in the middle.

A list is also one of the most dynamic of basic collections, in that the cost of adding and removing members is very low. Members can be removed from any position just as easily as they can be added. A list automatically grows and shrinks to reflect the number of members at any one time, and there is no fixed capacity which limits the size to which a list may grow.

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.

ListIndex behavior

```
//
// ListIndex -- index with insertion capability at any point in list
//
@deftype ListIndex <Index>- (void) addAfter: anObject;
- (void) addBefore: anObject;
@end
```

The addAfter: and addBefore: messages add members at a particular point in the sequence of members maintained by a list. The current location of an index determines the point at which a new member will be added. The addAfter: message adds a member at the list position immediately following the current index location. addBefore: adds a member to the immediately preceding location. Neither message changes the current location of the index, except that an index can change from a Start or End location to a location of Between.

Since an index may be positioned to any location in a list, these messages enable the construction of any desired sequence of members. Since the current index location remains unchanged, multiple members may all be inserted successively at some point in a list; previously added members are just pushed out one-by-one as new members are added.

An index with a location of Start, End, or Between is just as valid a location for addAfter: or addBefore: as an index positioned at a member. In these cases, there is no member at the current location of the index, so the new member is just inserted directly at the current index location, and the index is left positioned between the new member and the member that was previously adjacent in the opposite direction. If the previous location was Start and the message addAfter:, or the location was End and the message addBefore:, the index location remains Start or End.

If either the addAfter: or addBefore: message is requested with the EndsOnly option, and the index location is not Start or End, or if remove is requested on an index that is not positioned at either the first or last member, an invalid operation error is raised.

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

Last modified: 1998-01-08

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>
```

...

Stack behavior

```
- (void) push: anObject;
- pop;
...
```

Queue behavior

```
- (void) enqueue: anObject;
- dequeue;
...
```

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

Last modified: 1998-01-08

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;
```

```
- createIndex: aZone at: aKey;
- createIndex: aZone setMember: aMember;

- (void) getPredecessors: aMember;
- (void) getSuccessors: aMember;
- (void) getCostarts: aMember;
- (void) getCoends: aMember;
@end

//
// KeyedIndex -- index behavior shared by Set and Map types
//
@deftype KeyedIndex <Index>
- (int) getCurrentLevel;
- getIndexAtLevel: (int)level;

- at: aKey;

- setMember: aMember;
@end
```

Description

A keyed collection is a collection in which each member can be compared with some other value that identifies the member. This value is referred to as the member key. The key value may be determined either by the member value itself, which defines a Set, or by external association with the member when the member is first added, which defines a Map.

The keyed collection type establishes the common behavior shared by both Set and Map. Standard options are provided to declare ordering of members in the collection, which can be based either on an inherent ordering of the key values (a sorted collection) or external maintenance of a total or partial order. Additionally, options are provided to declare whether duplicate key values should be permitted, and if so, how.

The options for handling of duplicate keys provide considerable flexibility. Entire structures of nested collections can be created that subdivide the key space through progressive levels. By accepting duplicate keys, and organizing members of the same identity into subgroups, keyed collections support the abstract data type known as a bag or multiset. The index on a keyed collection includes the ability to traverse throughout the contents of any of these nested collections.

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;
```

DupOption

The value of DupOption specifies how duplicate key values are to be handled within a keyed collection. This option has one of the following values:

```
id <Symbol>  DupIsError, DupRejected, KeepDuplicates, KeepCountOnly;
```

If the DupIsError value is specified, any attempt to insert a duplicate key is an error, and effects of such an attempt depend on the level of debug checking. If debug checking for this error is turned on, the collection will check whether any new key duplicates an existing one, and raise a DuplicateKey error if it does. If debug checking is turned off, there is no guarantee that any check will be made, and the effects could be unpredictable in or catastrophic or insidious ways.

With a value of DupRejected, attempts to insert a duplicate key are always detected, and the attempt rejected. (All insert messages return a boolean flag as to whether a duplicate occurred.)

With a value of KeepDuplicates, attempts to insert a duplicate key are always accepted, and all members having the same key are retained in a specially created internal collection containing just the members that all have that same key. This collection then becomes the single member internally associated with the key value.

With a value of KeepCountOnly, duplicate members are not retained. Instead, only a count of the net number of insertions and removals at that key value are retained within the collection. With this value, member values cannot be retrieved from the collection, but only a count of members for a given key value, using different messages. This value is accepted only for the Set subtype of a KeyedCollection.

GroupType

The GroupType option specifies the type of internal collection to be created for members that all have the same key. This option is valid only if DupOption is also set to a value of KeepDuplicates.

Without this option, the type of internal collection created for members with the same key depends on the specific subtype. This option allows an external program to customize the type of internal collection permitted. The argument must be a previously created subtype of Collection. Typically, the argument is a customized version of one of the standard collection types such as List, Set, or Map; the argument would be the type object returned by customizeEnd after a customizeBegin: sequence.

More information about all the ways this option can be interpreted is provided with each subtype.

SingletonGroups

If this option is true, the type of collection created for members with duplicate keys is also created even when there is only member for a particular key. This option can be convenient to assure uniformity of a keyed collection with nested structure. This option is valid only if DupOption is also set to a value of KeepDuplicates.

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.

A bucket function accepts a single key value as its argument, and return an integer that identifies a bucket. As with a compare function, the key value is declared as an id pointer type, but may actually contain whatever kind of value is used within the collection. Following is the type of the bucket function pointer:

```
typedef int (*bucket_t)( id );
```

A new bucket function may be established any time during the lifetime of a collection. A new bucket function can ensure that keys remain suitably distributed to buckets as the total number of keys in a collection grows or shrinks.

BucketCount

A value for MaxBuckets must be given in the same message that sets a bucket function. Its value specifies the number of buckets to which a bucket function assigns its keys. The maximum value returned by a bucket function should be one less than this value.

PartiallyOrdered

A partially ordered collection supports the maintenance of individual ordering relations among its members. Each such relation is an assertion that one member should always precede another in the enumeration sequence of the collection. When an index traverses the collection, it will guarantee that all the predecessors of a member have already been visited before the member itself is reached.

If true, PartiallyOrdered specifies that messages for maintaining a partial order are enabled on the collection. An error will be raised if individual members do not have a unique identity within the innermost collection within which they are contained. This means that if duplicate keys are permitted by the collection, a collection type must also be specified for GroupType which does not itself accept duplicates. Every member in a partial order must have a unique identity so that the member itself is sufficient to identify it within the network of partial orders.

PartialOrderContext: aKeyedCollectionCollection

Ordering relations may be maintained not only between members belonging to the same collection, but between members and other collections, as long as all participants in an ordering are contained somewhere within a larger common collection. An ordering relation with a nested collection specifies that all members in the collection are successors or predecessors of the other member. Ordering relations can also be established between members of a larger, containing collection, and individual members of a nested collection.

The keyed collection specified as the value for PartialOrderContext must specify another collection in which the local collection is maintained. It is an error if the PartialOrderContext collection does not actually contain the local 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 UnknownOffset (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 the number of member entries which are all associated with its key value argument. This number is zero if the key value is not contained in the collection, and will never exceed one one unless a collection accepts duplicate entries for a key.

The removeKey: message removes a member matching a key value from the collection, and returns the member just removed. It returns nil if there is no key value in the collection which matches. If more than one entry was present for the key value, it removes and returns the first member in the internal collection created for duplicate members.

```
- createIndex: aZone at: aKey;
- createIndex: aZone setMember: aMember;
```

(.. Neither of these messages is currently implemented.)

The createIndex:at: message creates a new index on the collection and also sets its current position to the first member matching the key value given as its final argument. It returns the index just created.

The createIndex:at: message creates a new index on the collection and also sets its current position to the member passed as its final argument. This message is valid only if an internal member slot was defined for the collection with the MemberSlot option.

```
- (void) getPredecessors: aMember;
- (void) getSuccessors: aMember;
- (void) getCostarts: aMember;
- (void) getCoends: aMember;
```

(.. None of these messages is currently implemented.) These messages maintain the ordering relations within a partially ordered collection.

KeyedIndex behavior

```
- (int) getCurrentLevel;
- getIndexAtLevel: (int)level;
```

An index to a keyed collection traverses all members of the collection, regardless of whether these members belong to collections of members entered under duplicate key values. Internally, however, an index keeps track of any specific subcollection it is currently processing. The getCurrentLevel message returns the level of nesting, within collections of duplicate entries for a current key value, to which the current position of an index belongs. The level returned is an integer that starts at zero for the top-level collection, and increases by one for each level of nested collection that is currently active.

While processing any nested collections, the index for a keyed collection actually maintains a separate, internal index for each level of collection that is currently active. The index at each level may be retrieved using the getIndexAtLevel: message.

```
- at: aKey;
```

The at: message repositions the index to the first member that matches the key value passed as its argument.

```
- 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)

Name

Set – collection of members each having a defined identity

Synopsis

```
//
// Set -- collection of members each having a defined identity
//
@deftype Set <KeyedCollection, CREATABLE>CREATING
- (void) setKeyFunction: (id (*)(id))aFunction;
USING
- (id (*)(id)) getKeyFunction;

- (BOOL) add: anObject;
- (BOOL) add: anObject setIndex: anIndex;
- replace: anObject;
@end
```

Description

Set is a subtype of KeyedCollection in which the key value associated with each member is determined by the member value itself. The key value may be identical to the member itself, or may be defined as a a function of the member using a create-time option.

Inherited behavior

```
@deftype Set <KeyedCollection, CREATABLE>
```

The Set type inherits most of its interface from the KeyedCollection supertype. Set defines no create-time options beyond those already defined by KeyedCollection. If a custom compare or bucket function is specified, the member value is passed as the key value arguments of these functions. These functions determine what part of the member value is part of the key value, by determining which key values will compare equal to any member.

Set behavior

```
- (BOOL) add: anObject;
- (BOOL) add: anObject setIndex: anIndex;
- replace: anObject;
```

@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

OrderedSet – set of members in an externally assigned linear sequence

```
(.. This type is currently implemented only using the low-level option of an internal
member slot, and the messages for that option do not match the documentation in KeyedCollection.
If you need one of these objects, then either use a List or wait for some other implementation.)
```

Synopsis

```
//
// OrderedSet -- set of members in an externally assigned linear sequence
//
@deftype OrderedSet <KeyedCollection, List, CREATABLE>@end
```

Description

An OrderedSet is a totally ordered collection of members in which every member also has a distinct identity as defined by comparison against a key value.

Inherited behavior

```
@deftype OrderedSet <KeyedCollection, List, CREATABLE>
```

The sequence of members of an OrderedSet is established using the same messages that maintain member sequence in a List. An OrderedSet supports customization and access by key as defined by Set and KeyedCollection. The union of messages from all these sources defines the total interface of an OrderedSet. Members with duplicate keys, however, are not valid for an OrderedSet. Each member must have a unique position within the member sequence

---

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 independent 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.

Inherited behavior

For the Map type, key values are independent of the member values with which they are associated. Map defines two additional options to document information about its key values. Map also defines its own messages to distinguish the key value from member value in any operation which involves both.

```
@deftype Map <KeyedCollection, CREATABLE>
```

Create-time options

```
CREATING
- (void) setKeyType: aDataType;
- (void) setKeySlot: (int)byteOffset;
USING
-         getKeyType;
- (int) getKeySlot;
```

(..None of these options is currently implemented.)

Map behavior

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

- at: aKey replace: anObject;
```

at:insert: inserts an entry into a Map containing the key and member values given as its arguments. It returns true if the key was not previously contained in the collection. If the key was already present, the action of the message depends on the value in effect for the DuplicatesOption. With default options, the attempt to insert a duplicate key is simply rejected and false is returned.

at:insert:setIndex: is similar to at:insert:, except that the index passed as the final argument is repositioned to the member just inserted. If further operations are required on the entry in a map that was just inserted, an index provides faster repeated access to this entry than a message involving further key comparison, such as at:, would require. (.. This message is not currently implemented.)

at:replace replaces an existing member value associated with a key value by a new value given as its final argument. The message returns the member value which was formerly associated with the key value. The message returns nil if more than one member is present with the same key.

```
- createIndex: aZone setKey: aKey;
```

(.. 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

Name

String – character string object

(.. This type will be more fully documented when it is generalized beyond its current status as nothing more than an object wrapper for a null-terminated character array.)

Synopsis

```
//
// String -- character string object
//
@deftype String <Create, Drop, Copy, CREATABLE>CREATING
+ create: aZone setC: (const char *)cstring;
SETTING
- (void) setC: (const char *)cstring;
USING
- (const char *)getC;
- (void) appendC: (const char *)cstring;

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

Description

The String object type packages a null-terminated, C-format character string into an object. All memory allocation needed to hold the string value is handled by the object. This type currently defines only the most rudimentary operations for initializing and appending C-format character strings. These are sufficient for its current limited roles in places that need a uniformity between character strings and other kinds of allocated objects.

Inherited behavior

```
@deftype String <Create, Drop, Copy, CREATABLE>
```

...

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

Name

OutputStream – stream of output bytes

(.. This type will be more fully documented when it is generalized beyond its current status as a placeholder used only to direct debuggging output.)

Synopsis

```
//
// OutputStream -- stream of output bytes
//
@deftype OutputStream <Create, Drop, CREATABLE>CREATING
+ create: aZone setFileStream: (FILE *)fileStream;
- setFileStream: (FILE *)fileStream;
USING
- (FILE *) getFileStream;
- (void) appendC: (const char *)cstring;
@end
```

Description

The OutputStream type currently supports only the writing of a character string to a file. It is a placeholder for more general stream types. A stream is a collection that supports only sequential addition of members (an output stream) or sequential removal of members (an input stream).

Inherited behavior

```
@deftype OutputStream <Create, Drop, CREATABLE>
```

...

Create-time options

```
CREATING
- setFileStream: (FILE *)fileStream;
USING
- (FILE *) getFileStream;
```

```
...
```

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: Subclassing Reference

Until the collections library has been fully implemented, subclassing conventions from collections implementation classes are still in flux. In general, these classes will be among the most complex uses of multiple classes selected to implement an independent object type. (See library definition conventions (../defobj/library.html) for a summary of the distinction between types and classes.) New methods are being developed to simplify subclassing from such implementations. In the meantime, if you need to use to a collection within the implementation of your own class, just put an instance variable in your class and put the collection in that, and pass through the messages of the collection you want to have available on your class to this variable. In many if not most cases, this is better design anyway, because you control all use of the underlying structure.

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

Collections Library: Implementation Notes

(This section of documentation is not yet available.)

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

Last modified: 1998-01-08