Last modified: 1998-01-08

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: 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

```
    /         \          /              \
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:

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
```

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

IndexSafety

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

The IndexSafety option has one of the following values:

```
id <Symbol>  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.

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

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 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

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,

```
- (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

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.

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

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

```
- 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.

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.

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.

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.

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;
```

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

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;
```

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>
```

...

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;
```

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: 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