Chapter

11

Collections

Collections are objects that contain a group of other objects. Collections are among the most useful classes in Smalltalk and you'll find yourself using them constantly — even a string is nothing more than a collection of characters. The top level collection class is *Collection*, which is an abstract superclass that provides the generic behavior common to all collections. For example, Collection implements the methods `size` and `capacity`, the testing methods `isEmpty`, `contains:`, and `includes:`, the converting methods such as `asArray`, `asOrderedCollection`, the powerful enumeration methods such as `do:` and `collect:`, and the basic adding and removing methods `add:`, `remove:`, and `remove:ifAbsent:`. (Many of these methods are inherited by subclasses of Collection, but some act as placeholders to tell subclasses they should implement the method appropriately.)

The main subclasses of Collection that you will use are *Bag*, *Set*, and *SequenceableCollection*. SequenceableCollection is an abstract superclass for all the collections that understand sequencing, such as *Array*, *OrderedCollection*, and *String*. Because it expects all its subclasses to understand the concept of sequencing, SequenceableCollection adds behavior such as `first` and `last`, which is not generic enough to be placed in Collection. As we go further down the hierarchy, each subclass provides increasingly specific functionality, sometimes by adding new methods and sometimes by overriding methods defined in one of its superclasses. The hierarchy of basic collection classes that you will use is shown in Figure 11-1.

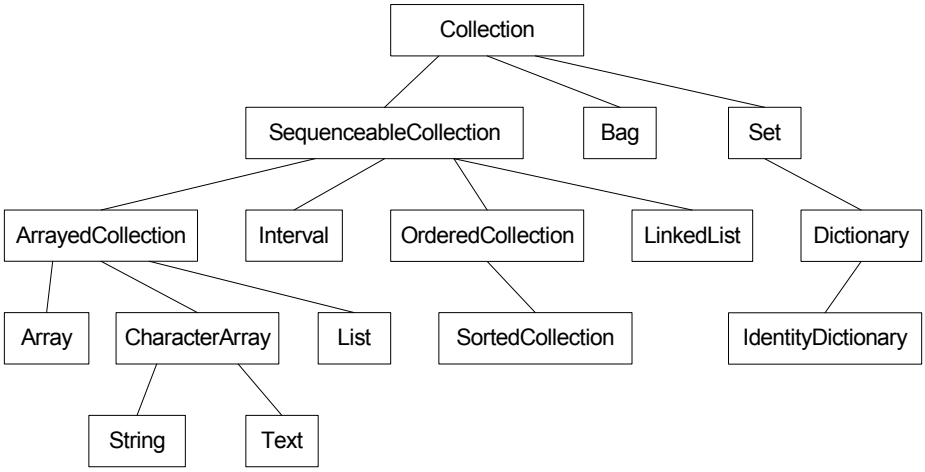

Figure 11-1. The basic collection hierarchy.

Not described here, but worth looking at so you know of their existence are: *IdentitySet*, *WeakDictionary*, *Symbol*, *IntegerArray*, *ByteArray*, *WordArray*, *RunArray*, *WeakArray*, and *KeyedCollection*.

Unfortunately, the hierarchy doesn't tell us as much as we'd like about the operations that can be performed. For example, if we look at removing items from a collection, Bags and Sets allow removal. In the SequenceableCollection hierarchy, Intervals don't allow removal, but LinkedLists and OrderedCollections do. Most ArrayedCollections don't allow removal, but Lists do allow it. So we'll try to generalize where possible, but will largely have to consider each class on its own.

Because there is so much specific functionality associated with collections, we will look at mainly the basic functionality. We'll start by looking at the main types of collection, with a brief explanation of how to use them and when you might use them. Then we'll take a look at the general mechanism for creating a collection, at the general aspects of adding objects and removing objects, at the powerful enumeration methods that are used to loop through collections, and at the general messages that collections understand. However, there is a vast amount of behavior that I've left out so it's well worth spending some time looking through the Collection hierarchy. If you use a Full Browser (from the VisualWorks Advanced Tools) and switch on supers, you will get a better feel for what behavior is inherited and what is implemented by each collection you look at.

Collection types

In what follows, note that collection indexing schemes are 1-based, rather 0-based as in C or C++. If the collection allows indexing, such as an Array or an OrderedCollection, the first element has index 1.

OrderedCollection

OrderedCollection is the workhorse of collection; I use OrderedCollections and Dictionaries more than any other types of collection. OrderedCollection stores objects in the order they were added, and will grow and shrink as you add and remove objects. Because the size is dynamic, OrderedCollections are generally more useful than Arrays, which are not dynamic. Because OrderedCollections are sequenced, you can use them as queues (FIFO) or stacks (LIFO). Here's an example that shows the chronological sequencing of OrderedCollection. You will see the string 'xyz' if you do the following.

```
orderedCollection := OrderedCollection new.
```

```
orderedCollection add: 'xyz'; add: 'abc'.
orderedCollection first inspect
```

The basic operations to add and remove items in an OrderedCollection are `add:`, `remove:`, and `remove:ifAbsent:`. This last message is useful because `remove:` raises an exception if the object is not present. For example,

```
orderedCollection := OrderedCollection new.
orderedCollection add: #here.
orderedCollection remove: #here.
orderedCollection remove: #there ifAbsent: [nil].
```

OrderedCollections also understand a lot of messages related to order of objects in the collection, such as `first`, `last`, `after:` and `before:`.

SortedCollection

If you want the objects sorted according to some collating sequence, use a SortedCollection. SortedCollection is subclassed off OrderedCollection and inherits most of its behavior. The main difference between a SortedCollection and an OrderedCollection is the order in which it keeps its elements. SortedCollection keeps its elements in sorted order by comparing pairs of elements. It's default mechanism is to send the <= message to one element, passing the second element as a parameter. The alternative way is for the programmer to specify a block of code that will compare two objects, in which case the SortedCollection executes the block for pairs of elements.

If you let the SortedCollection send the <= message, the only requirement is that each object you add to the collection responds to the <= message and returns *true* if it is less than or equal to the parameter object, and *false* otherwise. This generally means that SortedCollections will contain homogeneous objects, objects of the same type. Numbers and strings both know how to respond to <= but if you are storing your own objects, you will need to implement <=. Your implementation may compare the values of instance variables, or may do something as simple as

```
MyClass>> <= anObject
   ^self printString <= anObject printString
```

If you choose to specify the sort algorithm, you can create an instance of SortedCollection by sending the `sortBlock:` message to the SortedCollection class. The parameter is the block of code that does the comparison. For example,

```
sortedCollection := SortedCollection sortBlock: [ :x :y | x name < y
name]
```

Alternatively, you can specify a new sort algorithm by sending the `sortBlock: aSortBlock` message to the instance of SortedCollection. This causes the collection to be resorted, and any additions to be put in the correct place. You can create a new instance of SortedCollection, with a different sort sequence, by sending a collection the message `asSortedCollection: aSortBlock`.

Keeping elements in sorted order makes it expensive to add objects. If you spend a lot more time adding elements than reporting on elements, there are three options that are more efficient. The first is to use a List,

which is described below. The second is to use an OrderedCollection, then to ask a *SequenceableCollectionSorter* to sort it in place when required. The easiest way to do this is to send the messages `sort:` or `sort:using:` to the SequenceableCollectionSorter class. For example, the following code sorts the array in place, giving #(1 2 3 4 5).

```
array := #(5 4 3 2 1).
SequenceableCollectionSorter sort: array.
array inspect.
```

The third option is to send the collection `asSortedCollection` or `asSortedCollection:` if you don't mind getting back a new collection.

List

List is a collection that provides the chronological ordering (and indexing) of OrderedCollection, the sorting of SortedCollection, and the dependency notification of Model. Because of their dependency notification, Lists are used by both SelectionInList and MultiSelectionInList in user interface applications. Because of the versatility of Lists, there is a school of thought that says Lists should be used instead of OrderedCollections and SortedCollections. (For a time I did just this, but have now gone back to using the OrderedCollection as my workhorse. I liked having to type in only four characters for the class name, but disliked the seeming lack of elegance of a List. In particular, I like my classes to do one thing well, but a List tries to be all things to all people. Also, I never really liked inspecting a List.)

To sort a List in place, simply send it the `sort` message, or the `sortWith:` message to specify a sort algorithm. Internally, Lists use a SequenceableCollectionSorter to sort themselves.

Array

Arrays are fixed size collections where objects are stored in fixed locations, referenced by integer indexes. Since Arrays are fixed size, you can't add to the end of an array or delete elements from an array. However, Arrays provide faster accessing than other collections and are most useful when there will be lots of accessing. Unlike C and C++, where all elements of an array must be of the same type, in Smalltalk, array elements can be of arbitrary type. The disadvantage of Arrays are that you need to know the size of the array when you create it and they are of fixed size [1].

If you send the `new` message to Array, you get an instance of Array with a size of 0, so the usual way to create an instance of Array is to send the `new:` message and specify the size. For example,

```
array := Array new: 10.
```

The usual way to access an array is to send `at:put:` to put an object in the array and `at:` to retrieve the object at an index. The parameter to `at:` must be an integer. For example,

```
array at: 3 put: anObject.
anObject := array at: 3.
```

[1] Actually, you can grow an array in one of two ways: by sending `changeSizeTo:` or `grow` to it, or by creating a Stream on the Array, the latter being the preferred technique. We'll talk more about Streams in Chapter 13, Streams.

Dictionary

Dictionary is one of my favorite classes. A Dictionary is a collection where you look up values based on a key, and the key can be any object (except *nil*). A Dictionary is like an Array in that you access it using `at:` and `at:put:`. However, whereas Arrays are fixed size and you need an integer index to access elements, a Dictionary can automatically grow, and you can access elements using *any* object as a key. For example,

```
dictionary := Dictionary new.
dictionary at: 'abc' put: 'def'.
```

Dictionaries store Associations, so besides `at:put:`, you can add Associations directly to a Dictionary. The example above could also be written as one of:

```
dictionary := Dictionary new.
dictionary add: 'abc' -> 'def'.

dictionary := Dictionary new.
dictionary add: (Association key: 'abc' value: 'def').
```

To remove an object from a Dictionary, send either the `removeKey:` or `removeKey:ifAbsent:` message.

In the section on centralized error messages in Chapter 20, Error Handling, we'll see the use of Dictionaries with symbols as their keys. If you choose to use objects of your own creation as keys, you will need to implement = and `hash` for your object (whenever you implement =, you need to implement `hash` because two objects that are equal should also generate the same hash value).

IdentityDictionary uses == rather than = to compare keys, so it is a little faster than Dictionary if you have symbols for keys. Since many Dictionaries use symbols for their keys, you'll often find IdentityDictionaries used in preference to Dictionary. However, where Dictionaries keep a collection of Associations, IdentityDictionaries keep two parallel collections for keys and values. Because of the way PoolDictionaries are implemented, you can't use an IdentityDictionary as a PoolDictionary.

Set

A Set is an unordered collection of unique objects, so if you want a collection that guarantees no duplicates, use a Set. There is no structured accessing of a Set because there is no order to the elements. Instead, the way to access the elements of a Set is through the enumeration messages described below. If you try to add an object that already exists, the Set silently ignores it. If you try the following, you'll see a size of 2 since nothing was added when we tried to add the duplicate 7.

```
set := Set new add: 7; add: 8; add: 7; yourself.
set size inspect
```

When adding an object, a Set uses both the `hash` and = messages to determine if it is a duplicate of any other object in the collection. Sets are fairly expensive if you are doing a lot of adding, so use them only when you specifically want the property of uniqueness. One use of Set is to test for duplicates in a collection; for example, the following gives a Boolean that says whether the collection includes duplicate items.

```
includesDuplicates := collection size ~~ collection asSet size.
```

Bag

A Bag is another unordered collection that contains objects of arbitrary type. It is very similar to a Set except that a Bag keeps track of the number of occurrences of each object. Like a Set, it sends both `hash` and = to determine if an object already exists in the collection. One use of a Bag would be to collect duplicate items from another type of collection by doing something like the following. However, I've never seen a Bag used in an application.

```
duplicateList := List new.
collection asBag valuesAndCountsDo:
   [:each :count | count > 1 ifTrue: [duplicateList add: each]].
```

LinkedList

A LinkedList is a collection where each element points to the next element. OrderedCollections can do everything LinkedLists can do, the only difference being that LinkedLists are more efficient at doing large numbers of inserts and deletes. Elements of a LinkedList must be instances of *Link* or one of its subclasses (or at least be polymorphic with Link — respond to the same messages), which means that you will usually create a new subclass of Link in order to use a LinkedList. VisualWorks uses LinkedLists to manage processes, so Process is defined as a subclass of Link. This is the only example of LinkedList I've seen used.

Creating new collections

There are three usual ways to create a collection. One is to send the `new` message to the collection class. The second is to send `new: aCapacity`. The third is to use a message from the `with:` family and pass in the initial elements of the collection.

new:

The underlying instance creation mechanism is to use `new:`, which is implemented as a primitive by the class *Behavior*. Some of the collection classes override `new:`, usually invoking `super new:` then adding class specific code. `new:` creates an instance of a collection with indexed variables, where the parameter to `new:` specifies the number of indexed variables. (You can tell if a class is indexed because the top line of the class definition will say `variableSubclass:` rather than `subclass:`. The other type of variable is the named variable, which is an instance variable with a name.) For fixed size collections such as Array, the parameter to `new:` is both the capacity and the size of the collection. For collections that can grow, such as OrderedCollection, the collection has an initial size of 0, and the parameter specifies the capacity — the number of elements the collection can hold before it has to grow.

Because `new:` is used to specify a capacity, it's poor style to use `new:` to pass in parameters other than the capacity. When writing your own classes, it's better style to use `with:`, or to specify the parameter type in the instance creation message, such as `newLabel:`.

new

The `new` message is also implemented by Behavior but is overridden by the collection classes, and is usually implemented by sending `new: someInitialSize`. For example,

```
OrderedCollection class>>new
   ^self new: 5
```

Many collections will grow automatically as you add to them, so you don't need to specify an initial size and can simply send `new`. However, growing a collection is expensive, so if you know how large the collection will be, it's more efficient to send the `new:` message, specifying the size. In a timing test, it took over three times as long to add a hundred items to an *OrderedCollection* when the OrderedCollection was created using `new` as when it was created using `new:`. The first example took 1253 microseconds while the second example took 372 microseconds.

```
orderedCollection := OrderedCollection new.
1 to: 100 do: [:index | orderedCollection add: index]

orderedCollection:= OrderedCollection new: 100.
1 to: 100 do: [:index | orderedCollection add: index]
```

The with: family

You can create a collection and populate it with some initial elements by sending the class a message from the `with:` family of messages. There are five messages, each specifying a different number of objects to add. If you want to add more than four objects, they must already be in a collection of some type (in which case you could have sent the original collection the `asSomeTypeOfCollection` message; for example, `asOrderedCollection`).

```
with: objectOne
with: objectOne with: objectTwo
with: objectOne with: objectTwo with: objectThree
with: objectOne with: objectTwo with: objectThree with: objectFour
withAll: aCollectionOfObjects
```

For example,

```
collection := OrderedCollection with: 'how' with: 'are' with: 'you'.
collection := List withAll: #('how' 'are' 'you' 'today' 'Alec?').
```

Adding to and removing from Collections

The standard messages to add elements to a collection are `at:put:` for keyed collections such as Array and Dictionary, and `add:` for collections such as Bag, Set, and OrderedCollection. The general behavior when adding to or removing from a collection is to return the object being added or removed. Thus, both `add:` and `at:put:` return the object being stored. When you add to a collection, the collection may have to grow to fit in the new object. OrderedCollections will grow in size automatically when necessary. For more information, see the section on Growing Collections below. An alternative, and efficient approach to adding objects to collections is to create a Stream on the collection. We'll talk more about Streams in Chapter 13, Streams.

A common bug when using `add:` is forgetting that `add:` returns the object added rather than *self*. So, if you are creating a collection and adding to it, the following will *not* give you what you expect.

```
collection := OrderedCollection new
   add: objectOne;
   add: objectTwo.
```

Instead, use one of the following techniques.

```
collection := OrderedCollection new.
collection
   add: objectOne;
   add: objectTwo.

(collection := OrderedCollection new)
   add: objectOne;
   add: objectTwo.

collection := OrderedCollection new
   add: objectOne;
   add: objectTwo;
   yourself.
```

The standard messages to remove elements from a collection are `remove:` and `remove:ifAbsent:`. For keyed collections such as Dictionary, use `removeKey:` and `removeKey:ifAbsent:`. Note that you can't remove elements from an Array. When you send `remove:`, the object that was removed will be returned. If the object can't be found in the collection, an exception is raised. The alternative message `remove:ifAbsent:` will return the object if it is successfully removed. If the object can't be found, it will execute the block of code passed as the parameter to `ifAbsent:`, returning the value of the block.

A subtle `remove:` bug is to iterate over a collection and remove from the same collection. This will cause problems as the collection may be reorganized underneath you when you remove elements. Instead, you should iterate over a copy of the collection and remove from the original collection. For example,

```
collection:= OrderedCollection withAll: #(1 2 3 4 5 6).
collection copy do: [ :each | each even ifTrue: [collection remove:
each]].
```

Sometimes you don't even need to remove objects explicitly, Instead, you can use `select:` or `reject:` to create a new collection. We'll talk more about these messages below, in the section on Enumerating, but here are examples of doing the above operation using these two messages.

```
collection:= OrderedCollection withAll: #(1 2 3 4 5 6).
collection select: [:each | each odd].

collection:= OrderedCollection withAll: #(1 2 3 4 5 6).
collection reject: [:each | each even].
```

Enumerating

One reason it is difficult to have off-by-one errors or array size errors in Smalltalk is that collections provide some very powerful ways of iterating through their elements without having to specify sizes or bounds.

Collections *know* how big they are. Methods that do things over all the elements of a collection live in the `enumerating` protocol.

The enumeration messages go through each element in the collection, executing a block of code for each element. The block of code always takes the element as a parameter. In the examples below, you'll see that instead of getting fancy with the parameter name, I use *each* (I also use *index* for objects that are guaranteed to be index numbers). This has the benefit that if you see *each* or *index* you know it refers to the collection element being processed. You can usually tell what the item is from the collection name or from where it was built.

do:

The `do:` message is the basic enumeration message. It goes through all the items in the collection (in order, if the collection is sequenceable) and performs a programmer-specified block of code for each item in the collection. For example,

```
#(1 2 3 4 5) do: [ :each | Transcript cr; show: each printString].
```

The block is executed for each item in the collection and is passed that item as a parameter. Use `do:` when you want to perform some action for all or some of the elements in the collection. If you just want to create a new collection, use one of the messages below. The method `reverseDo:` iterates over the collection in reverse.

If you send `do:` to a Dictionary, it iterates through the values in the Dictionary. Three other methods, `keysDo:`, `associationsDo:`, and `keysAndValuesDo:` iterate over the keys, the associations, and the key and value pairs.

Another message, `with:do:`, allows you to iterate over two parallel collections, passing corresponding elements from the two collections to the `do:` block.

collect:

The `collect:` message allows you to do something for each item in the collection and puts the result of your action in a new collection. For example, if you wanted to create a new collection of the squares of a collection of numbers you could do either of the following. However, `collect:` is both better style and is more efficient because it doesn't use any variables declared outside the block. A timing test showed the first technique taking 103 microseconds while the second technique took 31 microseconds.

```
squareCollection := List new.
#(1 2 3 4 5) do: [ :each | squareCollection add: each * each].

squareCollection:= #(1 2 3 4 5) collect: [ :each | each * each].
```

It's useful to remember `collect:` because it's easy to find yourself writing code similar to the first example when you should be using `collect:`. Any time you iterate over a collection and add objects to another collection, consider using `collect:` or `select:`.

select:

The `select:` message allows you to create a subcollection where items from the original collection are selected based on some condition being true for them. For example, we get the result `#(1 3 5)` from the following example.

```
#(1 2 3 4 5) select: [ :each | each odd].
```

reject:

The `reject:` message is similar to `select:` but works the opposite way — items are rejected from the new subcollection if the condition is true. Using the same example as for `select:` we get the array `#(2 4)`.

```
#(1 2 3 4 5) reject: [ :each | each odd].
```

detect:

The `detect:` message returns the first item in a collection that satisfies a condition. For example, we get the value `4` from the following example.

```
(1 to: 5) detect: [ :each | each > 3].
```

If no item satisfies the condition, an exception is raised. You can avoid this by using `detect:ifNone:` and providing another block which is executed if no item satisfies the condition. For example, we get the value `99` from this example.

```
(1 to: 5) detect: [ :each | each > 30] ifNone: [99].
```

inject:into:

The `inject:into:` message is a strange message. Some people never use it, while others love it and use it whenever possible. The classic example is subtotaling.

```
(1 to: 5) inject: 0 into: [ :subtotal :each | subtotal + each].
```

When the block is executed for the first item in the collection, the parameter to the `inject:` keyword is passed in as the first parameter. The block returns subtotal + each, which for the first item is 0 + 1. This returned value is then used as the first parameter for the next invocation of the block. Each invocation, the block returns a value which is passed in as the first parameter for the next invocation. In fact, it might be easier to understand if we wrote the above example as:

```
(1 to: 5) inject: 0 into: [ :injectedValue :each | injectedValue +
each].
```

It takes a while to get used to `inject:into:` and to figure out situations where it is useful. To make this process a little shorter, here are some examples of its use. Inspect the results and understand what each example is doing.

```
(1 to: 5) inject: 0 into: [ :max :each | max max: each].
```

```
(1 to: 5) inject: 1 into: [ :factorial :each | factorial * each ].

#('Now' 'is' 'the' 'time' 'for' 'all' 'good' 'men')
   inject: String new
   into: [ :string :each | string, each, ' '].

(#(3 $x 'hello' #mySymbol)
   inject: String new writeStream
   into: [ :stream :each |
      stream
         print: each class;
         nextPutAll: ' value ';
         print: each;
         cr; yourself]) contents.
```

Note that in this last example, we send `yourself` as the last message to make sure that the stream is returned from the block. A common error is to not return the object that will be injected in the next iteration. In this case we didn't actually need `yourself`, but it's good practice to send it just to make sure the correct object is returned.

Enumeration wrap-up

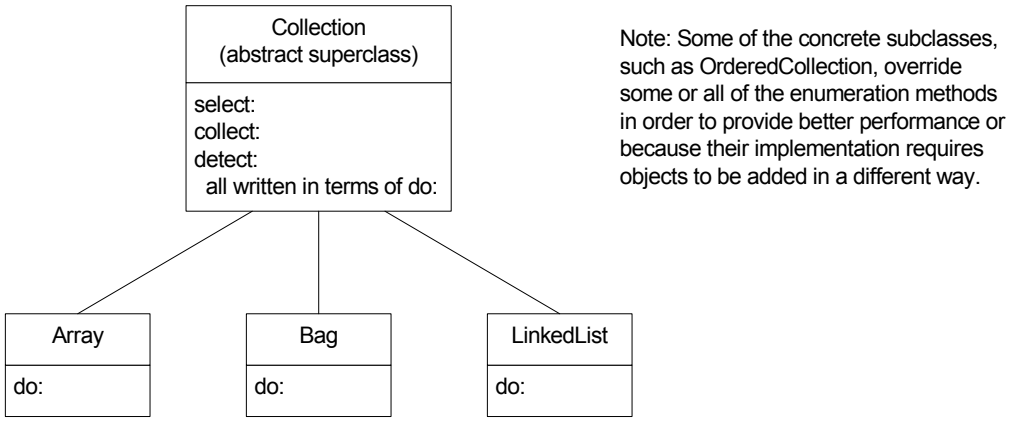

Figure 11-2. Implementing behavior in terms of subclass responsibilities.

The abstract superclass *Collection* implements all the enumeration methods in terms of `do:`. By requiring the subclasses of Collection to implement `do:` in a way that is appropriate for the subclass, the other enumeration methods can then be defined by Collection and inherited by all its subclasses. This is shown in Figure 11-2. It illustrates the useful object-oriented technique of having an abstract superclasses implement behavior in terms of a few undefined methods that are implemented by its subclasses.

General messages for collections

Below are some of the messages that collections respond to. I've listed them here to give you a feel for the types of things you can do. However, not all collections respond to all the messages. For example, some of them only make sense with collections that have a concept of sequence, while others only make sense in collections where you can remove items.

Adding objects

You can add an object with `at:put:` and `add:`, you can add to the front of the collection (`addFirst:`), to the end (`addLast:`, which is the same as `add:`), after another object (`add:after:`) or before another object (`add:before:`). You can add all the objects in another collection to this collection (`addAll:`, `addAll:after:`, `addAll:before:`).

Removing objects

Not all collections allow you to remove things. For example, what does it mean to remove the second element of an array, or to remove the third character in the string 'abcde'? Collections that don't allow you to use `add:` also don't allow you to use `remove:`.

To remove an object from a collection that allows removing, you again have many options. You can remove an object (`remove:`), which raises an exception if the object is not found, or you can remove an object without raising an exception (`remove:ifAbsent:`). You can remove the first object in the collection (`removeFirst`), or the last (`removeLast`), or the first or last so-many objects (`removeFirst:` and `removeLast:`). You can remove all the objects that satisfy a condition (`removeAllSuchThat:`) or all the objects that are contained in another collection (`removeAll:`). To remove objects from Dictionaries send the `removeKey:` or `removeKey:ifAbsent:` messages.

Accessing and locating objects

To access objects in a collection, you again have a rich set of options. If the collection understands indexing or keyed access, you can send `at:` to get the object at that index or key. For collections that understand sequencing, you can get the first object (`first`) or the last object (`last`), and you can find the object before or after another object (`before:` and `after:`). You can find the first position of an object (`indexOf:` or `indexOf:ifAbsent:`), or the last position of an object (`lastIndexOf:` or `lastIndexOf:ifAbsent:`). From there you can find the position of the next or the previous object (`nextIndexOf:from:to:` and `prevIndexOf:from:to:`). You can also find the position of a subcollection of the original collection (`indexOfSubCollection:startingAt:` or `indexOfSubCollection:startingAt:ifAbsent:`).

Replacing objects

You can replace all occurrences of a particular object with another object (`replaceAll:with:`), or do the replacement within a range (`replaceAll:with:from:to:`). Or you can replace all the elements in a range with elements from another collection (`replaceFrom:to:with:` or `replaceFrom:to:with:startingAt:`). Finally, you can copy a collection, adding another object to it as you copy it (`copyWith:`), or copy it and lose all the elements that are equal to a specified object (`copyWithout:`).

Sizes and counts

Collections will tell you how many elements they contain (`size`) and how many objects they are capable of holding (`capacity`). Note that for Arrays, these give the same answer. A collection will also tell you how many times an object occurs in it (`occurrencesOf: anObject`).

Testing

Collections respond with Boolean values (*true* or *false*) to the following messages. You can tell if a collection is empty (`isEmpty`) and you can see if an object is contained in a collection (`includes: anObject`). A similar method, `contains: [some code]`, returns *true* if the code block evaluates to *true* for *any* item in the collection. For example, lines three and four both evaluate to *true*.

```
collection := OrderedCollection new.
collection add: 'abc'; add: 'def'; add: 'xyz'.
collection includes: 'def'.
collection contains: [:each | each includes: $y]
```

Creating other collections from a collection.

Collection provides several methods for creating a collection of one type from a collection of another type. They are located in the `converting` protocol. For example, you can send the following messages to create a new collection of the specified type: `asArray`, `asBag`, `asList`, `asOrderedCollection`, `asSet`, `asSortedCollection`, `asSortedCollection: aSortBlock`. If you do the following you will see an instance of SortedCollection with elements 'abc', 'mno', and 'xyz'.

```
list := List new.
list add: 'xyz'; add: 'abc'; add: 'mno'.
list asSortedCollection inspect.
```

You can also send `reverse` to a sequenceable collection to create a new collection in reverse order.

Growing collections

Many collections can grow in size simply by sending the `add:` message. If the collection is not large enough to add another object, it expands the collection. (It does this by creating a new collection of larger capacity, copying over the collection elements, then *becoming* the new collection.) However, expanding the collection is fairly expensive, so if you know the eventual size of the collection it's more efficient to create it with the `new:` message, passing a value that is your best guess of the eventual size.

If you create a subclass of a collection that can grow and add instance variables to your subclass, they won't be automatically copied when the collection grows. To make sure that the values of your instance variables are retained when the collection grows, you should override the `copyEmpty:` method that is defined for Collection. The best way to do this is something like the following. For more information on `copyEmpty:`, see Chapter 25, Hooks into the System.

```
MyClass>>copyEmpty: size
    | new |
    (new := super copyEmpty: size)
```

```
    instVar1: self instVar1;
    instVar2: self instVar2.
^new
```