Chapter 7 - Introduction to Collections

Overview

Catalogs, inventories, dictionaries, lists of popup menu commands, items in a shopping cart - all these are examples of the fact that objects almost always occur in groups. Because collections of objects are so essential and varied, the Smalltalk library contains many collection classes and numerous useful methods for manipulating them.

In this chapter, we will survey the general properties of the collection class hierarchy and explore arrays, the basic kind of 'ordered' collections whose elements are stored in a fixed order and accessible by an index. We will also examine some extensions of the concept of an array, and the table widget as its application. The remaining collections will be covered in the following chapters.

The last section of this chapter introduces a new GUI widget – the Table. We encourage you to complement this material by reading Appendix 1 which contains a description of additional widgets, explains in more detail how widgets work, and includes several interesting examples.

7.1. Introduction

Although the world consists of individual objects, objects most often occur in groups. Some examples are bicycles standing in a school yard, a sequence of pages in a book, cars following one another on a highway, customers lined up in a bank, a bag of items bought in a store, a team of hockey players, and the seats in an auditorium.

Given this reality, it is not surprising that most programming problems objects also deal with collections of objects. As an example, the text on a page is a string of letters, digits, and punctuation symbols. List widgets, pop up menus, drop down menus, tables, and other user interface components also display collections of items. Other examples include an airport display of flight departures, an inventory program managing a collection of inventory items, and a course-marking program managing a collection of student records, themselves collections of marks and other information. Finally, a computerized encyclopedia provides access to a collection of keywords and their definitions. Several of these examples show that the elements of some collections are themselves collections and it thus makes perfect sense to treat collections themselves as objects.

A closer look shows that collections come in a variety of forms (Figure 7.1):

- Each player on a hockey team has a unique number and the number of players on the team is fixed. We could line the players up in the order of their numbers and we could refer to each by his or her number. If the numbers were consecutive integers, we could put the players into consecutive numbered slots and access them by these numbers. A line up of swimmers on starting blocks is an even better illustration of this kind of collection. In the programming context, this kind of collection is called an *array* and its characteristics are that it has a fixed size and its elements can be accessed by an integer 'index'.
- Cars on a parking lot can also be associated with numbers. The area could be divided into rows and columns, consecutive rows and columns assigned numbers, and each car identified by its row and column numbers. This collection of slots and their contents (cars) is again an array because it has a fixed size and can be accessed by a 'key', but it has two dimensions - rows and columns. A parking lot is an example of a *two-dimensional array*.
- Cars on a highway are a different kind of collection because their number constantly changes as cars arrive on the highway and leave it through exits. In Smalltalk, this kind of collection is called an *ordered collection* and its characteristics are that the number of elements in an ordered collection can change and that the contents are ordered: Assuming a one-lane highway, the cars follow one another. In principle, we can again access the cars by an index but since new cars are added all the time at one end, and leave at the other end, this is not the most natural way to refer to the elements of an ordered collection. The first, the last, and the next or the previous element are usually more important concepts.

- An encyclopedia is yet another kind of collection because its elements themselves have a structure - each consists of a keyword and a value (such as the definition of a word). This kind of collection occurs so often that Smalltalk treats it as a special kind of collection called a *dictionary*. Elements of dictionaries are usually accessed by their key as in 'what does the word *xenophobia* mean?'
- The structure of a collection of items in a shopping bag is very different from the previous examples because its contents are *not* ordered. In Smalltalk, this kind of collection is rather predictably called a *bag*. An interesting property of the shopping bag example is that each item in the bag often is of a different kind such as tooth paste, apple juice, and melon. In fact, most Smalltalk collections allow their elements to be arbitrary objects.
  How are the elements in a bag accessed? Some people just throw them in the bag to settle wherever there is place for them, and when you pull them out without looking, they come out in an unpredictable order.

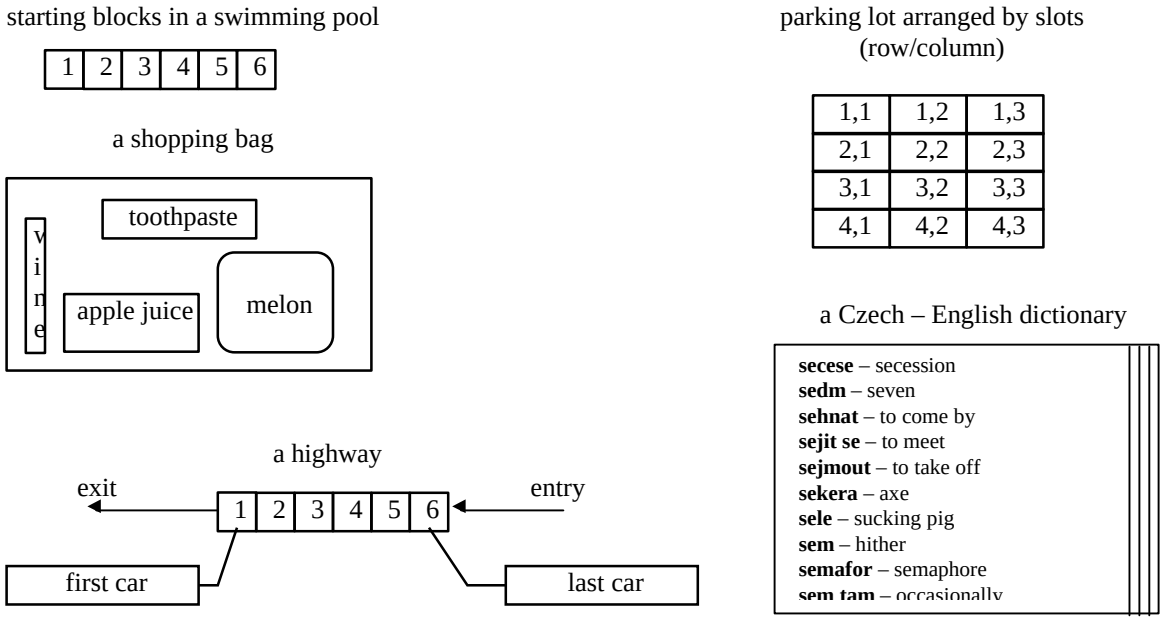

Figure 7.1. Collections may have different structures.

Our programming examples illustrate that collections come in various forms:

- A text stored in a document is a string of letters, digits, and punctuation symbols, in other words an ordered collection of characters.
- List widgets display one string on each line and since strings are ordered collections, a list widget is an ordered collection of ordered collections!
- The contents of a display of flight departures at an airport is an ordered collection of flight descriptions, and flight descriptions themselves are complicated objects with many components such as flight number, flight destination, departure time, and so on. In fact, since flight descriptions are arranged by their departure times, it is better to think of them as *sorted* rather than ordered collections: A *sorted collection* is just like an ordered collection, but its elements are sorted on the basis of some calculation. In our case, the calculation is comparison of departure times.

It is interesting to note that the nature of individual items in the display of departing flights can be interpreted in two ways: One is to view the individual items of the flight display (flight number, flight destination, departure time, and so on) as items of a 'flight object' collection. On the other hand, components of flight items are accessed by name as in 'what is your flight number?' or 'what is your departure time?' and it thus seems more natural to treat each flight item as an instance of a 'flight information' object. The second interpretation seems more convenient because if we treat flight information

as a collection, its elements are nameless and we can only ask questions such as 'what is the value of the third element of the flight information collection' - when we mean 'what is the departure time?'. This shows that the decision whether to treat a complex object as a collection or an instance of a special kind of object may not be obvious until we start thinking about how we want to use this object.

We have just seen that one distinction between a collection and a composite object is that collection elements are accessed anonymously whereas elements of composite objects are accessed by a name. Another distinction between a collection object and a composite object is that a collection can have any number of components but an object consisting of a fixed number of named components can only have the predetermined fixed number of components. As an example, the number of cars on a highway depends on the current situation. Even the number of slots in a parking lot is arbitrary because a parking lot can be designed any size or shape we want. A flight object treated as an instance of a composite object Flight, on the other hand, may only have a flightNumber component, a FlightDestination component, and a DepartureTime component if this is how the object Flight is defined.

The fact that collection elements are not named has the advantage that they can be accessed more easily. As an example, it is easy to describe how to calculate the total price of all elements in the shopping bag: 'Take the elements in the bag one after another and add their prices.' This procedure will work whether the bag contains one element or 100 elements, and whether the items are oranges, dishwasher liquid bottles, soap, or anything else. Similarly, counting all trucks in a satellite snapshot of a highway can be described: 'Start with the first element and count one element after another until you reach the last element.'

In the case of a composite object with named components, enumeration (execution of the same operation on each component) is more complicated and requires explicit naming. Assume, as an example, that we refer to the 11 players on a soccer team as goalie, leftDefence, rightDefence, and so on. If we wanted to calculate the average weight of all players on the team, we would have to calculate

```
averageWeight := (goalie weight + leftDefence weight + etc.) / 11
```

naming each player explicitly. If we put player information into an array, the solution is simply

```
averageWeight := (sum of weights of elements 1 to 11) / 11
```

The obvious need for collections, their open nature, and the generality of enumeration over collections are their essential advantages that make them members of the family of essential Smalltalk classes.

Main lessons learned:

- Objects usually occur in groups. In Smalltalk, such groups are called collections.
- Careful examination shows that different situations require different kinds of collections such as ordered or unordered and fixed or variable sized.
- Performing the same operation on each element of a collection is called enumeration. Enumeration is one of the most important operations on collections.
- Smalltalk objects can be divided in two categories - those that contain zero or more named objects, and those that contain a variable number of nameless objects. Collection items are nameless and this has two major advantages:
    - A collection may contain any number of elements whereas an object with named components contains only the named components.
    - It is much easier to enumerate over unnamed elements of a collection than over named instance variables of a multi-component object.
- Collections belong to the most important programming concepts.

Exercises

1. What kind of collection or named object is most natural for the following situations:
   a. Customers waiting in a bank queue.
   b. Filing cabinet drawer with alphabetically arranged current student files.
   c. Book information including author, title, and catalog number.
   d. Filing cabinet drawer with files of students who graduated last year.
   e. Description of all trees in the city of Halifax.
   f. A list of all rooms on one floor in a student residence.
   g. Student information including first name, last name, home address, home phone number, local address, local phone number, year of first registration, courses taken each past year, courses being taken this year, major area of study, previous degrees, participation in coop program.
   h. A list of rooms in one residence building.
   i. Bank account containing customer name, address, telephone number, and a list of accounts.
   j. A chess board.
   k. A list of 'print jobs' waiting to be printed on a shared printer.
2. What is the nature of the components of the collections and composite objects in Exercise 1?

7.2 Essential collections

Since collections are so useful and their applications so varied, the Smalltalk library contains more than 70 predefined collection classes. Some of them are abstract because different kinds of collections share many properties, but most are concrete. In the library, collections are divided into nine categories according to their abstractness, distinguishing characteristics, and uses. Many collection classes support the Smalltalk environment itself, but others are intended for general use. In the rest of this section, we will identify the essential collections (Figure 7.2) and give a brief overview of each.

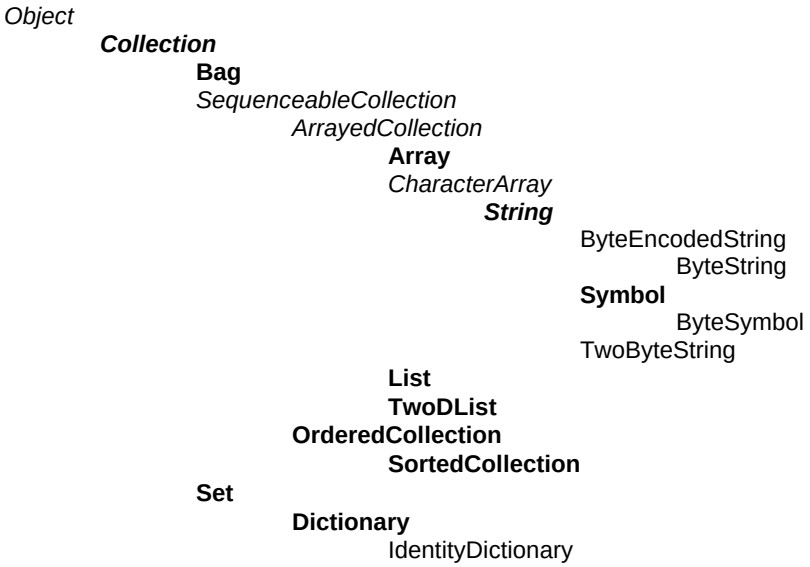

Figure 7.2. The most commonly used classes of the Collection hierarchy. Classes that will be covered in more detail are **boldfaced**, abstract classes are *italicized*.

Collections can be classified according to several criteria. One of them is whether their elements can be accessed by a key or not. A very important part of this group can accesses its elements by an integer number corresponding to the position of the element in the collection. These collections are called *sequenceable* and are implemented on a special branch of the class tree (Figure 7.2). Many of the collections whose elements are accessed by a non-integer key share the abstract class KeyedCollection as

their superclass and their branch is separate from sequenceable collections. These classes, however, are used mainly in Smalltalk implementation rather than general applications. Ironically, the most frequently used members of the family of collections accessed by a non-integer key (various kinds of dictionaries) is completely separate, and resides on the branch of classes that are not accessible by a key at all. This is because of the way they are implemented and because of the limitations of single inheritance.

Some collections (such as bags and sets) cannot be accessed by a key and Smalltalk calls them *unordered*. Although their elements are internally stored in consecutive memory locations, this internal arrangement is hidden from the user.

Another distinguishing feature of collections is that some collections have a *fixed size* which is established when the collection is created, and other collections can grow. Most Smalltalk collections automatically grow when they need more room to accommodate a new element.

Another distinction is that some collections allow *duplicates* whereas other collections throw duplicates away and keep just one copy of each element.

Finally, some sequenceable collections are *sorted* according to a predefined sorting function whereas other sequenceable collections are arranged according to the order in which the elements are added. In some sequenceable collections, the absolute position of an element may change when a new element is added or when an existing element is deleted, in other collections the position of an element never changes.

After this general overview, we will now review the main concrete collections before examining each of them in more detail later.

Arrays have fixed size determined by their creation message which has the form ‘create an array of size n’. Their elements are ordered in sequence and accessed by an integer index. The index of the first element is always 1 (some other languages start at 0) and the following elements have consecutive indexes; as an example, elements of an array with five elements are numbered 1, 2, 3, 4, and 5. Once an object is assigned to a position, its position never changes unless this is explicitly requested by the program. Elements of Smalltalk arrays and most other collections may be any objects, and even a heterogeneous collection of objects of different kinds is acceptable. Because arrays have a fixed size, they are used when we know exactly how many elements the collection will have. Their advantage is that their accessing is very efficient and that they are often easier to create and initialize than other types of collections.

Ordered collections are also sequenceable which means that their elements are also ordered in a sequence and indexed. Unlike arrays, however, ordered collections have a built-in mechanism for growing. Also, ordered collections often add elements at the beginning and this changes the index of elements already in the collection. Programs using ordered collections in this way thus cannot use the index of an element to access it. The price for the flexibility of ordered collections is that some operations on them are not as efficient as operations on arrays. Automatic growing, in particular, which occurs when the currently allocated size becomes too small, is time-consuming. Ordered collections are used when we need to collect objects whose total number is variable or initially unknown and when the order in which they are received matters as when we simulate a queue of customers in a bank.

Sorted collections are a special kind of ordered collections that insert new elements into a position determined by some sorting function, typically comparison by magnitude. For greater flexibility, the sorting function can be specified or changed by the program. The ability to sort is very valuable but its price is that inserting a new element may require considerable processing.

Lists are a relatively recent addition to VisualWorks library and according to the manual, they are intended to supersede ordered and sorted collections. In essence, lists are ordered collections that understand sorting messages and implement the concept of dependency which makes them useful as item holders for widgets such as list views. This is, in fact, the main use of lists.

Two-dimensional lists are a variation on lists and arrays. Like arrays, they have fixed size and their elements are accessed by index or rather a pair of indices, one for row and one for column. They are used to support table widgets because they implement dependency.

Strings are indexed sequences of characters with no information about how they should be displayed. We have already seen that strings are among the most frequently used Smalltalk objects.

Text is an object combining the concept of a string with information about its rendering - the font, style, size, and color in which the string will be displayed on the screen or printed.

Whereas the collections listed above are all sequenceable, the following collections are unordered. We have already explained that this means that although their elements are internally stored in a sequence, the order is determined by the implementation rather than by the accessing message and is hidden from the user.

Sets are unordered collections that don't allow duplication. In other words, if you add an element that is already in the set, the set does not change. This property makes it possible to weed out duplicates from an existing collection simply by converting it to a set.

Bags are unordered just like sets but they keep a tally of the number of occurrences of their elements. If the same element is inserted into a bag five times, for example, the bag knows that its number of occurrences. When an element is removed, its count is decremented by 1 and when the count reaches zero, the element is removed. In contrast, if you remove an element from a set, it is gone, no matter how many times it has been added before.

Dictionaries are collections in which values are accessed by a key. The key may be any object and if we used integers for keys, we would essentially obtain ordered collections, Dictionaries are therefore conceptually generalized ordered collections. In terms of implementation, however, dictionaries are implemented as a subclass of Set. In practice, elements of dictionaries are usually Association objects, key - value pairs similar to entries in a dictionary or an encyclopedia. Being sets, dictionaries eliminate duplication using keys to perform the comparison. This means that adding an association whose key is already in the dictionary replaces the original association. Because the structure of dictionaries is different from the structure of other collections (by relying on key and value access), dictionaries provide a number of specialized messages.

IdentitySet and IdentityDictionary are important subclasses of Set and Dictionary that use == instead of = to eliminate duplication.

Main lessons learned:

- VisualWorks contains many collection classes, some for system needs but many for general use.
- The main differences between collections are whether their size is fixed or not, whether they allow duplication of elements, whether they can be accessed by a key, and whether their elements are ordered and how this ordering is achieved.
- Collections most useful for general use are arrays, ordered and sorted collections, lists, strings, text, symbols, sets, bags, and dictionaries.

Exercises

1. Find the number of library references to Array, OrderedCollection, SortedCollection, List, Set, Bag, and Dictionary.
2. All classes understand message allInstances which returns an array with all currently existing instances of the class. As an example, Bag allInstances returns all instances of active instances of Bag currently in the system. Use this message to find the number of instances of Array, OrderedCollection, SortedCollection, List, Set, Bag, Dictionary. The number, of course, depends on the current state of your session.
3. Extend the previous exercise by calculating the smallest, largest, and average size of collection instances.
4. For each of the concrete collection classes listed in this section, give an example of one situation in which the collection would be appropriate.

7.3 Properties shared by all collections

Before presenting individual collection classes in detail, we will now survey their most important shared characteristics, most of them defined in the abstract class Collection. They include protocols for creating collections, adding and deleting elements, accessing them, converting one kind of collection into another, and enumeration.

Creation of new collections (class methods)

The most common messages for creating collections are new or new:. Both are inherited from class Behavior which defines most of the shared functionality of all class methods across Smalltalk library. The typical uses of new and new: are as follows:

```
Array new: 20          "Creates an instance of Array with 20 uninitialized elements."
Set new                "Creates a Set object with room for a default number of elements (2)."
```

To create a collection with only a few initially known elements, use one of the with: creation messages as in:

```
Array with: 4 factorial                 "Creates an Array with the single element 4 factorial."
Array with: students with: teachers     "Creates a two-element Array with the specified elements."
```

Predefined with: messages have up to four keywords (with:with:with:with: is the last one) but you can easily define your own with: messages with more keywords if you wish.

Another useful creation message is withAll: aCollection which creates a new collection and initializes it to all elements of aCollection. As an example,

```
OrderedCollection wihAll: anArray
```

creates an OrderedCollection containing all elements of anArray. Object anArray is not affected.

Several collection classes have their own specialized creation messages in addition to the ones listed above. Moreover, arrays have the distinction that they can be created as *literal arrays* without any creation message. As an example,

```
#(34 43 78 -22)
```

creates a four-element array containing the four specified numbers. We will say more about this later. Collections are also often created by converting existing collections as explained next.

Converting one kind of collection to another (instance methods)

Almost any kind of collection can be converted into almost any other collection. The process creates a *new collection* containing the elements of the original collection but *the original is not changed*. As we have already seen, conversion messages generally begin with the word as.

Conversion is sometimes used just to perform some operation on the receiver. As an example, if anArray contains some numbers and we want to sort them, message

```
aSortedCollection:= #(34 43 78 -22) asSortedCollection
```

creates a SortedCollection and fills it with the elements of anArray sorted in ascending order of magnitude. When you execute

```
| anArray aSortedCollection |
anArray := #(32 96 -34 89 32 45).
aSortedCollection := anArray asSortedCollection
```

you will get SortedCollection (-34 32 32 45 89 96) and anArray remains changed. (This is a very important property of all conversion messages that is frequently overlooked, causing problems that may be difficult to trace.)
To convert the result back to an array, we could use

```
anArray := anArray asSortedCollection asArray.          "Sort and convert back to Array."
```

The combination of the asSortedCollection and asArray thus sorts the array. Another frequently used conversion message is asSet, as in

```
aCollection asSet
```

which creates a Set with elements from aCollection and removes all duplicates. The receiver is not affected.

```
anArray := anArray asSet asArray
```

removes all duplicates from anArray and stores them in some unpredictable order in a new array.

Another reason for performing conversion is that some operation require a collection of a kind different from the receiver's class. As an example, a popup menu expects its labels to be supplied in an sequenceable collection. If the labels are, for example, in the form of a Set, we must convert them as in

```
listOfLabels := aSet asArray
```

Accessing (instance methods)

One of the main distinctions between different collection classes is how their elements are accessed and so Collection leaves accessing of individual elements to concrete collection classes. Its own accessing protocol is limited to *size* and *capacity* corresponding, respectively, to the number of elements present in the collection, and the space available in the collection.

```
aCollection capacity
```

returns the number of *slots* available in aCollection for 'storing' its elements. Note that if the capacity of a collection is 5, the collection does not necessarily contain five elements; this only means that in its present state, the collection has room for five elements (Figure 7.3).

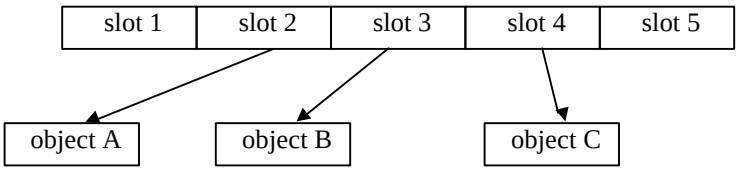

Figure 7.3. A collection with capacity 5 and size 3.

The reason why we put 'storing' in quotes is to emphasize that collections don't really *store* their elements. In other words, the representation of a collection object in computer memory does not *contain* the representation of its elements. It only contains references to these objects, *pointers* to their binary representations - essentially their memory addresses - and each slot is thus only big enough to store a pointer to the corresponding element. This has two important consequences. One is that collections as such require relatively little memory space, and the other is that two or more different collections can share elements. This should not surprise you because *all* Smalltalk objects, not just collections, access their components via pointers. This principle is, however, worth repeating because the potentially large number of elements of collections might lead one to think that collections must occupy a lot of memory space. They don't - but their elements might. We will ignore this technical detail in the future and generally say that a

collection 'contains' or 'stores' its elements and depict collections as if they contained their elements although this is not literally true.
The complement of capacity, the size message, as in

```
aCollection size
```

tells us how many slots of aCollection actually contain pointers to objects, in other words, how many elements the collection really 'has'. For arrays, capacity and size are always the same; for most other collections, capacity and size are different concepts.

Enumeration - doing something with all elements in a collection (instance methods)

One of the most common operations on collections is executing one or more statements with each of its elements. As an example, one might want to print each element in the Transcript, or calculate the square of each element. This is called iteration or enumeration. The most common enumeration messages are illustrated by the following examples in which the block argument is applied to each element of the receiver collection:

```
| aCollection selections rejections squares sum product|
aCollection := #(13 7 89 11 76 4 65 32).
aCollection do: [:element| Transcript cr; show: element printString].
selections := aCollection select: [:element| element > 45].        "Collection of all elements > 45."
rejections := aCollection reject: [:element| element > 45].        "Collection of all elements not > 45."
squares := aCollection collect: [:element| element squared].       "Collection of all element squares."
sum := aCollection inject: 0 into: [:element :tempSum| tempSum + element].       "Sum of all elements."
product := aCollection inject: 1 into: [:element :tempProd| tempProd * element]    "Product of all elements."
```

The do: message executes the block for each element of the receiver. In our example, it displays each element of aCollection in the Transcript. Receiver aCollection is not affected in any way. Message do: is the most important enumeration message and can be used to implement all other types of enumeration. In fact, message do: is the only enumeration message which is left as subclass responsibility and all others are fully defined on the basis of do: in class Collection (some are redefined in a few subclasses). In spite of the power of do:, experienced Smalltalk programmers always use the specialized messages below when appropriate to avoid unnecessary work and possible errors involved in re-inventing the operation.

The next three enumeration messages all *create a new collection* without changing the receiver. As a result, if you don't save the result in a variable or reuse it immediately, the result is immediately lost.

Message select: creates a new collection containing only those elements that satisfy the block, in other words, those elements for which the block evaluates to true (Figure 7.4).

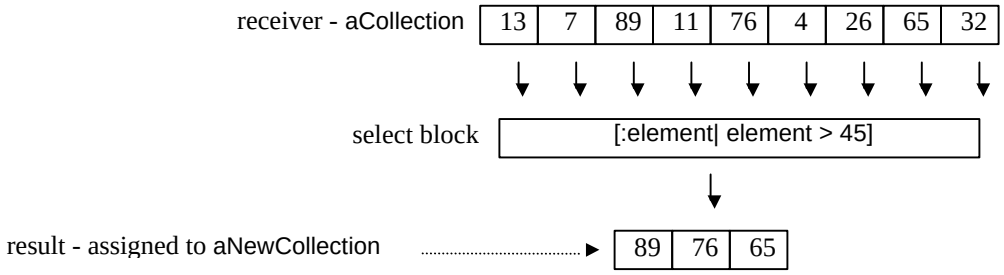

Figure 7.4. aNewCollection := aCollection select: [:element| element > 45].

In our example, the result of the simple test is a new collection containing only those elements of aCollection that are greater than 45. The tests can be much more complex, as long as the block returns true or false for each element, and the whole expression fails if even one element of aCollection cannot execute the block. This is, of course, true for all enumeration messages.

Message reject: is the opposite of select: - it rejects all elements that evaluate the block to true and puts the remaining elements into a new collection. In our case, the new collection will contain all elements of aCollection that are not greater than 45, in other words, all elements smaller or equal to 45.

Message collect: creates a new collection whose size is identical to that of the receiver and whose elements are the objects returned by evaluating the block. In our case,

```
squares := aCollection collect: [:element| element squared]
```

calculates a collection of squares of the original numbers and assigns it to squares.

The inject:into: message uses a block with two arguments. The first argument enumerates over all elements of the collection, the second is the initial value of some calculation performed by the block. In general, the expression in the block takes the element (the first argument) and updates the value of the intermediate result (the second argument). The message returns the last value of the second argument.

Message inject:into: is very powerful but not used much by beginners, perhaps because it is not so easy to understand as other enumerations and because of confusion over which argument is which. To help with the second problem, we suggest that you remember that the first arguments is the receiver's element – and if you read the expression from left to right, the receiver comes first too.

All enumeration messages that create collections, including collect:, select:, and reject:, create a collection of the same *species* as the receiver. In most cases, the species of a class is the class itself (the species of Array is Array, the species of OrderedCollection is OrderedCollection, and so on) because most classes inherit the following definition of species defined in Object:

```
species
"Answer the preferred class for reconstructing the receiver. Species and class are not always the same."
    ^self class
```

In a few classes, the most suitable result is not of the same kind as the receiver and species is then redefined , returning a class different from the class of the receiver.

The handling of species is typical for dealing with messages that behave identically for almost all classes with a few exceptions. Such methods are usually defined in class Object or a suitable superclass in a way that reflects the predominant behavior, and inherited. The few classes that require different behavior override the inherited definition.

The definition of enumeration methods is a classical example of the use of inheritance. All of them are defined in terms of do: and only do: is left to subclasses - as noted prominently in Collection's comment. As an example, the definition of collect: in Collection is as follows:

```
collect: aBlock
"Evaluate aBlock with each of the values of the receiver as the argument. Collect the resulting values into a collection that is like the receiver. Answer the new collection."
    | newCollection |
    newCollection := self species new.
    self do: [:each | newCollection add: (aBlock value: each)].
    ^newCollection
```

Inheriting this definition by all collections, provides a great saving and consistency, and only a few classes redefine collect: because their internal structure allows them to perform it more efficiently.

Testing (instance methods)

The testing protocol of collections allows checking whether a collection contains objects satisfying a block, specific objects, or any objects at all. It includes the following messages:

```
aCollection isEmpty                                 "Returns true or false."
aCollection includes: 'Air Canada'                  "Returns true or false."
aCollection contains: [:element| element > 7]       "Returns true or false."
aCollection occurrencesOf: 'abc'                    "Returns an integer number."
```

As an example of the implementation of these messages, the definition of isEmpty is simply

```
isEmpty
        ^self size = 0
```

Message includes: returns true if at least one of the elements of aCollection is equal to the string 'Air Canada'; it returns false otherwise.

Message contains: returns true if at least one element of aCollection satisfies the *block argument*; it returns false otherwise. Just as in enumeration, it is important to remember that during its execution, contains: may have to execute the block argument for each element of aCollection. Consequently, if even a single element of the collection cannot execute the block, the message may fail. As an example, if all but one of the elements of aCollection in our example are numbers, the message may generate an error because a string cannot compare itself with a number. Ignoring this principle is another common programming mistake.

Finally, occurrencesOf: counts how many occurrences of the collection are equal to the argument. Its definition uses do: as follows:

```
occurrencesOf: anObject
"Answer how many of the receiver's elements are equal to anObject."
        | tally |
        tally := 0.
        self do: [:each | anObject = each ifTrue: [tally := tally + 1]].
        ^tally
```

Adding elements (instance methods)

If a collection can grow - and most collections can - it understands messages add: and addAll:.

Message add: anObject adds anObject in a way consistent with the nature of the collection (ordered or unordered) and 'grows' the collection by increasing its capacity if necessary. Growing can be a very time-consuming because it first creates a new larger collection, and then copies all elements of the original collection into it. It then causes the new collection to 'become' the original collection, changing all existing references to the old version to refer to the new version. (Message become: from class Object performs this operation.) This overhead can be very significant and the only way to avoid it is to try to predict the maximum capacity that the collection will ever need and create it initially with this (or larger) capacity using, for example, the new: message.

Creating a collection whose capacity may never be fully utilized may appear to be wasteful but since a collection is just a bunch of pointers, its memory requirements are small and probably worth the run time saved in growing. The methods that make growing possible also use this strategy - when a collection's capacity must be increased, it is not increased by adding one slot but by adding several slots so that consecutive add: or addAll: messages don't have to grow the collection again.

Message addAll: aCollection copies *each element* from aCollection into the receiver. As an example, if aCollection contains 25 elements, the receiver collection will gain 25 new elements, growing in the process if necessary. (If the receiver is a Set, all duplicates will , of course,be ignored.)

Both add: and addAll: are frequently used and their behavior is slightly but critically different. As an example, consider the following code and the illustration of its effect in Figure 7.5:

```
collection1 add: collection2.       "Adds collection2 to collection1 as a single objects."
collection1 addAll: collection2     "Adds all elements to collection1 one by one."
```

Message add: anObject adds anObject to the receiver collection as a single new element, whereas addAll: anObject assumes that anObject is a collection, 'takes it apart', and adds each element to the

receiver individually. In both cases, the receiver must be able to grow which means that add: and addAll: cannot be used with arrays.

In both add: and addAll: in our example, collection1 and collection2 share elements and if the program later modifies any of these elements, both collections will be affected. If, however, one collection *replaces* one of the elements with a new object, the corresponding pointer will point to the new object but the other collection will not be affected because it still points to the original object. We will clarify this point later.

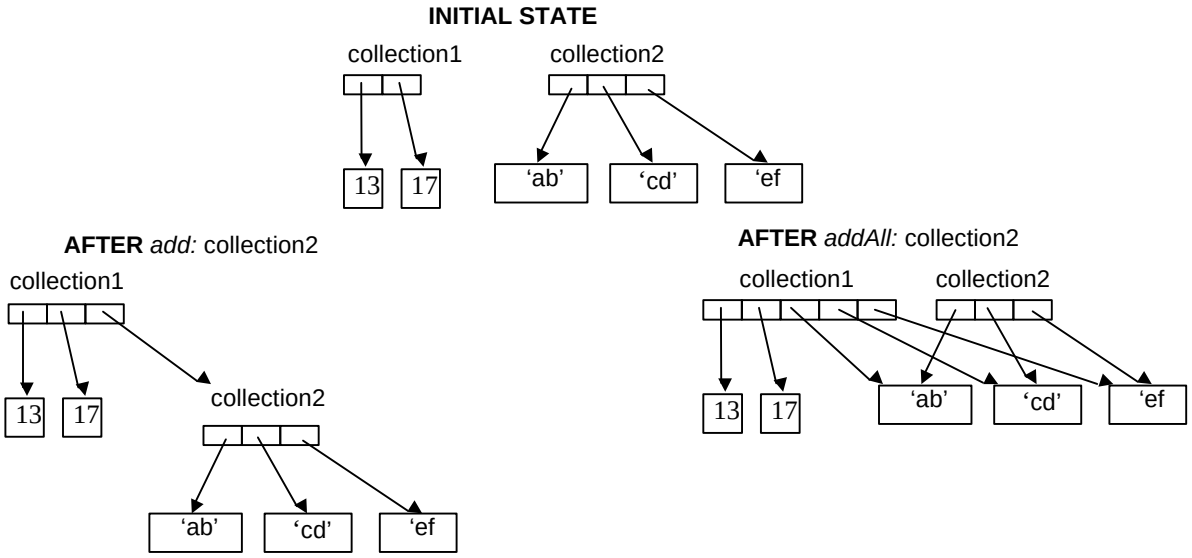

Figure 7.5. Effect of collection1 *add:* collection2 (left) and collection1 *addAll:* collection2 (right).

Class Collection leaves add: as subclass responsibility because the exact place at which the element is added depends on the nature of the receiver. Method addAll:, on the other hand, is fully defined in Collection using add: and subclasses only need to define add:. The shared definition of addAll: uses enumeration:

```
addAll: aCollection
"Include all the elements of aCollection as the receiver's elements.  Answer aCollection."
        aCollection do: [:each | self add: each].
        ^aCollection
```

Unlike other methods (such as conversion) that make a copy of the receiver and work on the copy without changing the receiver, add: and addAll: change the receiver. This is natural but worth mentioning and you can test it on the following code:

```
| orderedCollection |
orderedCollection := OrderedCollection with: 'abc' with: 'xyz'.     "OrderedCollection ('abc' 'xyz')."
orderedCollection add: 'uvw'.          "Changes orderedCollection but returns the 'uvw' object"
orderedCollection                      "Returns OrderedCollection ('abc' 'xyz' 'uvw')"
```

A somewhat unexpected and confusing feature of all add (and remove) messages is that they *return the argument* rather than the changed receiver. As an example,

```
orderedCollection add: 'uvw'
```

changes orderedCollection but returns 'uvw' rather than the modified orderedCollection as one might expect. Failing to realize this is a very frequent cause of errors.

Removing elements (instance methods)

Removing objects from a collection is subject to similar restrictions as adding because collections that cannot grow don't know how to remove elements either. As an example, you can *change* the elements of an array but you cannot *remove* them. Like add, remove messages also return the argument rather than the receiver as one might expect.

If you are certain that the object that you want to remove is in the collection, use remove: as in

```
aCollection remove: 45      "Returns 45."
```

or in

```
aCollection remove: aStudent
```

If the object is not in the collection, this message will raise an error. Consequently, if you are not sure, use the alternative remove:ifAbsent: as in

```
aCollection remove: 45 ifAbsent: []              "Do nothing if the element is not in aCollection."
```

or in

```
aCollection remove: aStudent ifAbsent: [Dialog warn: aStudent printString, ' is not in the collection']
```

The first expression will not do anything if the collection does not contain 45, the second will display a warning if aStudent is not in the collection. These two styles of remove: - one providing an alternative block, and one providing an unprotected operation - are typical for collection messages. Since remove: is a special case of remove:ifAbsent: it is defined as

```
remove: oldObject
"Remove oldObject as one of the receiver's elements.  Answer oldObject unless no element is equal to
oldObject, in which case, provide an error notification. "
        ^self remove: oldObject ifAbsent: [self notFoundError]
```

where notFoundError opens an exception window. Method remove:ifAbsent: itself is left as subclass responsibility.

Example: A new enumeration method

There are occasions when one needs to perform an operation on each element of a collection except a particular one. A method dealing with this problem might be useful for any kind of collection and will thus add it to the enumeration protocol of the abstract class Collection and all other collections will inherit it. In doing this, we must be sure to design the method so that all collections will be able to execute it.

Our method, called do:exceptFor:, first constructs a new collection by removing the special element from it, and then sends do: to the result to enumerate over all remaining elements:

```
do: aBlock exceptFor: anObject
"Evaluate aBlock with each of the receiver's elements except anObject."

        | collection |
        collection := self reject: [:el | el = anObject]].
        collection do: aBlock
```

We tested this approach by printing all elements of a literal array of numbers that are not equal to 3. We used arrays of the following kinds: An array that does not contain 3, an array that contains exactly one element equal to 3, and an array that contains more than one copy of three. As an example, the last of the three tests was

```
#(1 2 3 4 5 3 4 5 3 4 5) do: [:el| Trancript cr; show: el printString] exceptFor: 3
```

We found that all tests worked but we should, of course, test the method for other kinds of collections as well. This is left as an exercise but for now, we will generalize our method somewhat: In some situations, we might want to execute selective enumeration on the basis of a test rather than by specifying the element explicitly. As an example, we might want to do something with all elements that are greater than 3.

In this problem, we must allow the second argument to be either a block or any other object, and treat blocks in a special way. Our approach will be the same as before but the construction of the collection will be performed differently if the second argument is some kind of a block. In that case, we will simply reject all elements of the receiver that satisfy the block. The whole solution is as follows:

```
do: aBlock exceptFor: blockOrAny
"Evaluate aBlock with each of the receiver's elements except those implied by blockOrAny."

    | collection |
    collection := (blockOrAny isKindOf: BlockClosure)
                                ifTrue: [self reject: [:el | blockOrAny value: el]]
                                ifFalse: [self reject: [:el | el = blockOrAny]].
            collection do: aBlock
```

where message isKindOf: checks whether the receiver is an instance of the argument class or its subclass. We tested that our previous tests still work and than added tests of the following kind:

```
#(1 2 3 4 5 3 4 5 3 4 5) do: [:el| Trancript cr; show: el printString] exceptFor: [:el| el > 3]
```

We found that all tests work as expected.

After this general introduction to collections, we will now examine the essential collections in more detail and illustrate them on examples. In the rest of this chapter, we will present sequenceable collections, all of them subclasses of the abstract class SequenceableCollection. SequenceableCollection implements some of the shared protocols, among them the all important do: which it defines as follows:

```
do: aBlock
    "Evaluate aBlock with each of the receiver's elements as the argument."
    1 to: self size do: [:i | aBlock value: (self at: i)]
```

As you can see, the definition is based on the to:do: message defined in number classes and uses the essence of SequenceableCollection - the fact that its elements are accessible by consecutive integer indices starting at 1.

Main lessons learned:

- Abstract class Collection defines many behaviors shared by all collections.
- The main protocols of collections include creation, accessing, testing, adding and removing elements, conversion to other kinds of collections, and enumeration (iteration over collection elements).
- Conversion is often used to perform operations such as elimination of duplicates or sorting.
- When using collection methods, make sure to understand what kind of object the method returns and whether it changes the original or returns a modified copy while leaving the original unchanged.
- Enumeration is among the most frequent collection operations and good Smalltalk programmers always use the specialized enumeration messages when appropriate.
- Messages that add or remove elements always return the argument and modify the receiver.

Exercises

1. Test conversion messages asSet, asSortedCollection, asOrderedCollection on literal arrays #(75 13 254 41 -65 75 75) and #('word' 'symbols' 'digits' 'letters' 'letters') and examine the result and the effect on the receiver.
2. Print the elements of the arrays in the previous exercises in the Transcript in reverse order. (Hint: Check out the enumeration protocol of SequenceableCollection.)
3. Execute
   ```
   | anArray |
   anArray := #(32 96 -34 89 32 45).
   anArray := anArray asSet asArray
   ```
   and comment on the result (type of returned collection, its size, and order of elements).
4. Find how collections grow (Hint: Read the definition of add:.).
5. Classes SequenceableCollection and ArrayedCollection are abstract. Write their brief summaries with a description of one important method from each of three selected instance protocols.
6. Browse class Collection, read the definitions of its enumeration messages, and examine their use by looking at three existing references. Write a summary including one example of each.
7. Find two Collection methods redefined in some of its subclasses and describe the differences.
8. Method add: in Collection should not be used with Array receivers. What will happen if you do send add: to an instance of Array?
9. Find two other examples of the use of the mechanism discovered in the previous exercise.
   Find definitions of new and new:. (See also Appendix 1).
10. Under what conditions contains: will/will not raise an error?
11. We stated that messages that behave identically in most classes are usually defined in class Object in a way that reflects the predominant behavior, and the few classes that requite different behavior override the inherited definition . Browse definitions of species, isNil, and isSymbol as examples of this technique.
12. What is the difference between
    ```
    aCollection remove: 45 ifAbsent: [self]
    ```
    and
    ```
    aCollection remove: 45 ifAbsent: [^self]
    ```
13. Interval is a collection frequently used to represent an arithmetic progression of numbers defined by start, end, and step values, and its main use is for certain types of enumeration. Find Interval in the browser, examine its uses, and write a short summary of your findings.

7.4 Arrays

Among the numerous collection classes, arrays are perhaps conceptually the simplest. They are also among the most frequently used and we will thus start our coverage of collections with them.

Arrays are fixed-size sequenceable collections and their elements are numbered by successive integer indices starting from 1. Unlike some other sequenceable collections, elements of arrays don't change their position during the array's lifetime. Arrays are accessed mostly by enumeration and less frequently by index. Since we have already presented enumeration messages which all apply almost identically to all collections, we will start with element accessing.

Accessing an array element

To obtain the value of the element at a particular index, send the at: index message as in

```
anArray at: 7                "Returns element at index 7. Fails if anArray has fewer than 7 elements."
```

To replace the element at a known index with a new object, send the at:put: message. As an example,

```
anArray at: 7 put: 0.4 sin   "Replaces element at index 7 with 0.4 sin. Fails if anArray size < 7."
```

replaces the seventh element with the object 0.4 sin.

Just like add and remove messages, the at:put: message changes the receiver but *returns the second argument*. As a consequence, the statement

```
anArray := anArray at: 3 put: 5 factorial
```

changes the value of variable anArray to 120 - probably not what was desired (Figure 7.6). If all we wanted was to change the array, we should have written simply

```
anArray at: 3 put: 5 factorial
```

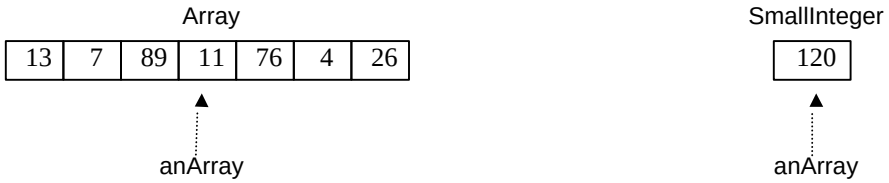

Figure 7.6. Value of anArray before (left) and after (right) executing anArray := anArray at: 3 put: 5 factorial.

An important aspect of array accessing is that Smalltalk always checks whether the index is within the bounds of the interval <1, size>. If the index is out of bounds, the attempted access produces an Exception window. This automatic checking adds overhead time and some people think that bound checking should be optional. The great majority of Smalltalk programmers consider the security provided by bounds checking more important than speed of access and eliminating bounds checking is not an option.

Creating new arrays

Arrays are usually created either as literals, or by a creation message such as new: or with:, or by conversion. We will now illustrate all these possibilities.
The easiest way to create an array is as a *literal* as in

```
array := #(1 3 5 7).                        "Four-element array containing four numbers."
array := #('string 1' 'string 2').          "Two-element array containing two strings."
array := #($a $e $i $o $u)                  "Five-element array of characters."
array := #('string' #symbol).               "Two-element array with a string and a symbol."
array := #(#symbol1 #symbol2 #('abc' 4))    "Three-element array with two symbols and a sub-array."
```

Executing

```
array at: 3
```

returns 5 for the first array, fails with the second and fourth, returns array #('abc' 4) with the fifth, and character $i for the third. If we wanted to obtain the second element of the *nested array* in the fifth example, we would have to extract the sub-array first and then access its second element as in

```
(array at: 3) at: 2          "Returns 4."
```

The limitation of literal arrays is that they can only be created with literal components, in other words, only with the following *literal* objects: nil, true, false, numbers (other than Fraction), characters, strings, symbols, and literal arrays. The following statements thus will not produce what you might expect:

```
array1 := #(3 factorial 5 factorial 7 factorial).   "Same as #(3 #factorial #5 #factorial #7 #factorial)."
array2 := #(3/4 (5/6 7/8)).                         "Same as #(3 #/ 4 #(5 #/ 6 7 #/ 8))."
array3 := #(student1 student2)                      "Same as #(#student1 #student2)."
```

When an array cannot be created as a literal array, the following methods are the most common:

1. Use the new: message and assign elements by enumeration or by accessing individual indices.
2. Use one of the with: messages.
3. Obtain the array from another collection by conversion.
4. Calculate the array from another array by enumeration.

Method 1: Using new: to create an uninitialized array and calculating individual elements

Let's construct an array containing the squares of ten consecutive numbers starting with 37. Since the values of the elements must be calculated, we create an array of size 10 and use enumeration over the index as follows:

```
array := Array new: 10.                    "Create an array of size 10 with all elements initially nil."
1 to: 10 do
        [:index| array at: index put: (index + 36) squared]        "Calculate elements."
```

As another example, the following code fragment shows how we could prompt the user to fill an array of size 5 with numbers. We again use the index as the basis of the solution:

```
| array |
array := Array new: 5.
1 to: 5 do:
        [:index| | string |
                string :=  Dialog request: 'Enter a number'.
                array at: index put: string asNumber]
```

A simpler solution that eliminates the temporary block variable is

```
| array |
array := Array new: 5.
1 to: 5 do:
        [:index| array at: index put: (Dialog request: 'Enter a number ') asNumber]
```

Sometimes the elements of an array cannot be obtained as neatly as this. As an example, assume that we have a class StudentRecord with instance variables studentID, firstName, lastName, middleInitial, streetAddress, city, province, postalCode, coursesTaken, registrationYear, and degree. Assume that our application requires an array containing studentID, firstName, lastName, province, and postalCode, and that StudentRecord provides accessing methods to access these components. In this case, we must create the array as follows:

```
selectedComponents := Array new: 5.          "Returns #(nil nil nil nil nil)."
selectedComponents at: 1 put: aStudentRecord studentID.
selectedComponents at: 2 put: aStudentRecord firstName.
selectedComponents at: 3 put: aStudentRecord lastName.
selectedComponents at: 4 put: aStudentRecord province.
selectedComponents at: 5 put: aStudentRecord postalCode
```

Finally, note that Array also understands the new message but this message is almost useless because it creates an array with size 0.

Method 2: Creating an array using one of the with: messages

This approach is useful when we need to create an array with up to four known elements and cannot use the literal form, as in

```
array1 := Array with: 5 factorial with: aNumber with: 3
array2 := Array with: Student new with: Address new with: Marks new
```

Neither of these arrays could be constructed as a literal array because all their elements are not literals.

Although the maximum number of with: keywords in the Array protocol is four (message with:with:with:with:) you could define a method with more arguments by creating an uninitialized array and then assigning the individual arguments as in

```
with: firstObject with: secondObject with: thirdObject with: fourthObject with: fifthObject
    | newCollection |
    newCollection := self new: 5.
    newCollection at: 1 put: firstObject.
    newCollection at: 2 put: secondObject.
    newCollection at: 3 put: thirdObject.
    newCollection at: 4 put: fourthObject.
    newCollection at: 4 put: fourthObject.
    newCollection at: 5 put: fifthObject.
    ^newCollection
```

This style is used by the predefined with messages.

Method 3: Creating an array by converting another type of collection

It occasionally happens that we need an array but don't know how many elements it will have. In this case, we start with a suitable type of collection to collect the elements, and convert it to an array as in

```
"Create variable size collection shoppingList."
...
"Convert it to an Array."
shoppingList := shoppingList asArray
```

Note that simply sending the conversion message would not give the desired result

```
"Create collection shoppingList"
...
"Convert it to an Array"
shoppingList asArray
```

because the last statement creates an array from collection shoppingList but does not change the value of variable shoppingList itself. The creating array is thrown away because it is not referenced by any object.

Method 4: Creating an array by enumeration

Creating a collection by enumeration is very common. As an example, the following code fragment uses an array of strings to create another array called shortStrings containing only those strings that contain at most 15 characters:

```
| strings shortStrings |
strings := #( 'short string 1' 'in our context, this is a  long string' 'short string 2').
"Get a new array by selection."
shortStrings := strings select: [:string| string size <= 15].
etc.
```

Note that we did *not* have to convert the result to an array even though its size is different from that of the original, because the species of receiver strings is Array.

As another example, assume that we want to calculate an array of cosines of arguments stored in another array. To do this, we use collect: as follows:

```
| argArray cosArray |
argArray := #(0.1 0.15 0.25 0.4 0.45 0.5).
```

```
cosArray := argArray collect: [:arg| arg cos].
etc.
```

Note that we could not use to:by:do: because the arguments stored in argArray are not regularly spaced.

Main lessons learned:

- An array is a fixed-size sequenceable collection whose elements are accessed by consecutive integer indices numbered from 1.
- Arrays, like most other Smalltalk collections, allow any objects to be their elements.
- Arrays are usually created as literals or by conversion from other types of collections, using new: followed by calculation of individual elements, or using one of the with: messages.
- When an array is created, it is assigned a fixed size and all its elements are initialized to nil.
- Each array access checks whether the index is within the array's index bounds.
- The main advantage of arrays is their efficient accessing.

Exercises

1. Array's definition of collect: is not inherited from Collection. Find where it comes from and what saving it provides compared to the Collection definition.
3. Inspect #(3/4 5/6 7/8).
4. What happens when you execute #(1 2 3 4) at: 7?
5. Since any Smalltalk programs can fail and produce an Exception window, what is so bad about failures that could occur if Smalltalk did not have automatic bounds checking?
6. One of the following expressions succeeds and the other fails. Explain why.
   #(13 53 21 87 9 'abc') contains: [:n| n>200]
   a. #(13 53 21 87 9 'abc') contains: [:n| n>20]
7. Write a code fragment to print
   a. successive elements of an array separated by carriage returns (use do:separatedBy:)
   b. elements of an array with their indices (use keysAndValuesDo:)
   c. elements of an array in reverse order (use reverseDo:)
8. When you try to print a large array such as Smalltalk classNames asArray, you will only get some of its elements. Which method controls this behavior?
9. Use the inject:into: message to solve the following tasks where appropriate.
   a. Find the sum of all elements of an array of numbers.
   b. Find the product of all elements of an array of numbers.
   c. Repeat the previous problem without assuming that all elements of the array are numbers.
   d. Find the product of all non-zero numbers in an array whose elements are all numbers.
   e. Find the largest number in an array whose elements are all numbers.
   f. Find the smallest rectangle containing all rectangles in an array of rectangles. (Hint: Check the protocols of Rectangle and create rectangles using originPoint corner: cornerPoint as in 1@1 corner: 10@15.)
   g. Find the largest rectangle contained in all rectangles in an array of rectangles.
10. Calculate the following collections using enumeration messages:
   a. The string containing all vowels in 'abcdefg'. (Hint: Use message isVowel.)
   b. The string containing all characters that are not vowels in 'abcdefg'.
   c. The string of characters in 'abcdefg' shifted by one position – in other words, 'bcdefgh'. (Hint: Convert a character to its integer code using asInteger, then increment the code, and convert to Character again.)
   d. String entered by the user, encoded by shifting characters as in the previous exercise.
   e. All numbers between 1 and 5000 that are divisible by 7 and 11 but not by 13. (Hint: The species of an Interval is an Array and an expression such as (1 to: 5000) select: aBlock thus produces an array.)
11. Calculate the sum of 10,000 numbers using do: and then using inject:into: and compare the speed.

12. Perform the following tasks using enumeration messages:
    a. Create an array of three rectangles and use the current window to display their outlines. (Hint: To display the circumference of aRectangle in the current window, use expression aRectangle displayStrokedOn: Window currentWindow component graphicsContex. Message displayFilledOn: displays the rectangle filled. We will talk about graphics in detail in Chapter 12.)
    b. Ask user for the side s of a square and for a displacement d, construct an array containing five squares with the specified side whose origins are displaced by d@d, and display the filled squares in the current window. (Hint: See class Rectangle for rectangle operations, and the previous exercise for information about display.)

7.5 Examples of uses of arrays

Smalltalk programmers use arrays extensively but not as universally as programmers in some other languages. One reason for this is that if you really need a collection, Smalltalk provides many other types of collections which are often more suitable because they can grow, because they sort their elements, eliminate duplication, and so on. And if you need an object whose components have identifiable roles, the only correct solution in most cases is to define a class with named instance variables.

When you browse the Smalltalk library, you will find many references to arrays. Most of them use an array because they require enumeration and enumeration is particularly efficient with arrays.

After this introduction, we will now do three more examples showing the use of arrays.

Example 1. Multiple choice dialogs

A typical use of arrays is for creating menus and multiple choice dialogs. As an example, the following statement produces the dialog window in Figure 7.7 and returns the selected ice cream flavor if the user clicks *OK*, or an empty string if the user clicks the *Cancel* button.

```
Dialog    choose: 'Which kind of ice cream do you prefer?'
          fromList: #('vanilla' 'strawberry' 'raspberry' 'banana' 'pecan' 'pistachio')
          values: #('vanilla' 'strawberry' 'raspberry' 'banana' 'pecan' 'pistachio')
          lines: 6
          cancel: ['']
```

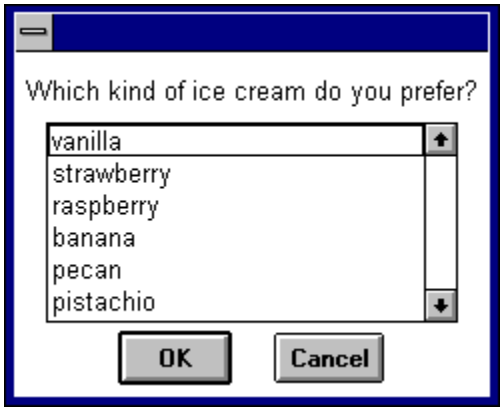

Figure 7.7. Multiple choice dialog produced with literal arrays.

The message choose:fromList:values:lines:cancel: is one of several very useful multiple choice dialogs in class Dialog. The argument of choose: is the prompt line, the argument of fromList: is an array of displayed strings, the values: argument is an array of objects that are returned when the corresponding label is selected (its elements may be anything you want but we chose strings identical to the labels), lines:

specifies how many lines should be displayed at a time (the rest can be scrolled), and the last argument is a block that will be executed if the user clicks *Cancel* (returns an empty string in our example).

Example 2. Using arrays for mapping

A interesting use of arrays is for converting one collection of objects into another. Assume, for example, that we want to convert some text into unreadable strings. In a very simple scheme, each letter in the message could be replaced by some other fixed letter, preferably without any easily discernible pattern. As an example, letter $a could be replaced by letter $f, letter $b by letter $z, $c by $v, $d by $r, and so on. With this scheme, a string such as 'acdc' would be encrypted into 'fzrz'. This task can be nicely implemented by storing the encryption scheme in a literal array such as

```
encryption := #($f $z $v $r etc.)
```

and extracting the encoding from the array, using the original character's position in the alphabet to calculate the index. As an example, $a is the first letter of the alphabet and so we use the first letter in the encryption array to encrypt it. Character $b is the second letter in the alphabet and its encryption character is the second character in the encryption array, and so on (Figure 7.8).

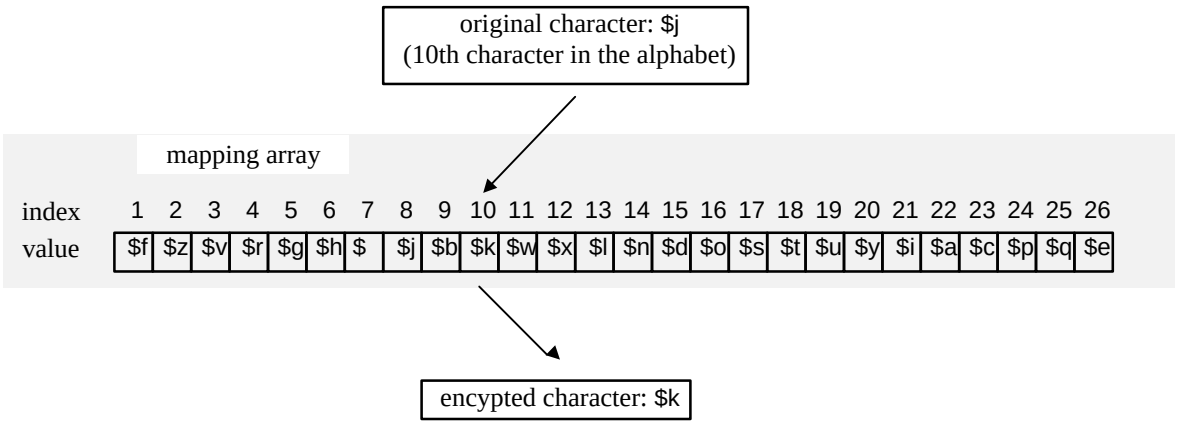

Figure 7.8. Encrypting secret message using a mapping array.

Since each character has a numeric code (the ASCII code[1]) which is related to its position in the alphabet (the code of $a is 97, the code of $b is 98, and so on), and since the code of a character can be obtained by sending message asInteger, the formula for finding the index of a character is

```
(character asInteger) – ($a asInteger) + 1      "Returns 1 for $a, 2, for $b, and so on."
```

We will now write a code fragment to ask the user for a string, print it, encrypt it using the mapping in Figure 7.8, and print the encrypted string. As an example, the program will encode the string 'a multi-word string' into 'lixyb-cdtr uytbnm'. The basis of the program is the use of collect: to calculate the encrypted string from the original string:

```
| encrypted encryption original shift |
"Initialize an encryption array, making sure that each letter appears exactly once."
encryption := #($f $z $v $r g $h $m $j $b $k $w $x $l $n $d $o $s $t $u $y $i $a $c $p $q $e).
"Pre-calculate and cache index shift to avoid the need to recalculate it for each character."
shift := $a asInteger - 1.
"Get string from user."
original := Dialog request: 'Enter a string' initialAnswer: ''.
```

[1] ASCII (American Standard Code for Information Interchange) is a widely used set of codes of printable characters and control characters such as Line Feed and Escape.

```
"Display original."
Transcript clear; show: original; cr.
"Encrypt and display."
encrypted := original collect: [:char| encryption at: char asInteger - shift].
Transcript show: encrypted
```

Unfortunately, if the character is not a lowercase letter, the index calculated in the collect: block falls outside of the limits of the array and causes an exception. We leave it to you to correct this shortcoming.

Example 3. Creating the encrypting array automatically

Instead of having to create an encrypting array by hand as we did in Example 2, we can create it randomly as follows: Start with an array containing all 26 letters of the alphabet. Create a random number between 1 and 26 and insert the corresponding letter at the start of the encryption array; then remove the letter from the original array. In the next iteration, create a random number between 1 and 25, extract the corresponding letter, put it at index 2 of the encryption array, and so on. The principle is as follows:

```
“Create a string with all letters as a startt for the mapping array.”
alphabet := 'abcdefghijklmnopqurstuvwxyz'.
“Create a random number generator, create a mapping array with 26 elements, initialize index.”
“Calculate encryption array.”
26 to: 1 by: -1 do: whileTrue:
        [:max| “Get a random number between 1 and max, extract corresponding character from
        alphabet, put it in the mapping array, remove it from alphabet].
```

This is all relatively simple but we must find how to remove an element from a String, in other words, how to make a copy of the original without the removed character. Browsing the collection classes (a string is a collection of characters), we find that there is a useful message called copyWithout: anElement defined in SequenceableCollection and therefore applicable to strings. When we test it with

```
‘abc’ copyWithout: ‘b’
```

it does not work because the elements of strings are characters. When we try

```
‘abc’ copyWithout: $b
```

we get the correct result ‘ac’. We can now write our code for creating the encoding array as follows:

```
| rg map alphabet |
"Create alphabet string and random generator, initialize index for accessing mapping array."
alphabet := 'abcdefghijklmnopqrstuvwxyz'.
rg := Random new.
map := Array new: 26.
1 to: 26 do: [ :index | | ch |
"Extract characters at random locations, deleting the selected character from the alphabet each time."
        ch := alphabet at: (rg next * alphabet size) truncated + 1.
        map at: index put: ch.
        alphabet copyWithout: ch].
"Print result in Transcript to see if it works."
Transcript show: map printString
```

The code works correctly.

Example 4. Using arrays to execute a list of messages

A clever and common use of arrays is for execution of one of several alternative messages in a list or all the messages in a list. We will use this idea to print the values of sin, cos, ln, exp, and sqrt for x = 1 to 10 in increments of 0.5 in the Transcript.

Solution: We will use enumeration over #(#sin #cos #ln #exp #sqrt) and the powerful perform: message. The perform: message requires a Symbol argument, treats it as a method, and executes it. As an example,

```
5 perform: #factorial
```

has the same effect as

```
5 factorial
```

We would, of course, never use perform: if we knew exactly which message we want to send (as in 5 perform: #factorial) but if the message may be any one of several messages, as in our example, or is supplied as a variable or an argument, perform: is the only way to go. With the perform: message, the solution of our problem is simply

```
1 to: 10 by: 1/2 do: [:x |
        Transcript cr; show: x printString.
        #(#sin #cos #ln #exp #sqrt) do:
                [:message | | value |
                value := x perform: message.
                Transcript tab; show: value printString]]
```

Example 5. Inappropriate use of arrays

As an example of an *inappropriate* use of arrays, assume that we need an object containing an address with components including country, state, city, street name, and postal code. Each of these components has an identifiable role and we should define a class with named instance variables to hold the individual components. If we used an array to represent this object and put the city name, for example, into the fourth element, we would have to access city names as in

```
address at: 4 put: 'Halifax'
```

which is not intuitive because the index in expression at: 4 put: 'Halifax' does not provide any hint that we are dealing with a nameOfCity object. This is dangerous because it makes it too easy to use the wrong index with the result of storing the name of the city in the wrong element. Similarly, accessing a component of an array by an expression such as

```
address at: 4
```

does not make any sense and requires that you remember the nature of element 4. A very unpleasant possibility is that somebody (maybe even you) changes the usage of the array so that city name is now the third rather than the fourth element, and does not change all code that depends on it. Our code will then give an incorrect result.

For all these reasons, arrays should not be employed in this way. And if they are, access to their elements should at least be explicit, for example by special accessing methods such as

```
city
"Return the value of city."
        ^array at: 4

city: aString
"Change the value of city"
        array at: 4 put: aString
```

With these methods, we can replace the ugly

```
address at: 4 put: 'Halifax'
```

with

```
address city: 'Halifax'
```

which is both more readable and safer. If somebody changes the meaning of individual elements, he or she only needs to change the definition of accessing messages and all code using the array will continue working as before. If the elements were accessed by index, *all* uses of the index would have to be found and corrected. Nevertheless, as we already mentioned, the best solution usually is to gather the components as named instance variables in a new class called perhaps Address.

Main lessons learned:

- Like other collections, arrays are used to hold and process nameless elements. Their advantages over other collections are greater efficiency of access and smaller memory requirements.
- Arrays are not used as much in Smalltalk as in other languages because Smalltalk provides many other types of collections and because a class with named components is often preferable.
- Literal arrays are often used for multiple choice dialogs and popup menus.
- Arrays are useful to map one set of objects into another.

Exercises

1. Modify the multiple choice dialog example so that if the user clicks *Cancel*, the program opens a dialog window showing: We regret that you don't like our ice creams. To display the apostrophe, repeat it as follows: 'We regret that you don''t like our ice creams.'
2. Check out other multiple choice dialogs. Use choose: fromList: values: buttons: values: lines: cancel: to create a multiple choice dialog as in Figure 7.9. Making a selection in the list should return the same string as the label, clicking *'Cancel'* should return nil, clicking *'Copy all'* should return #copyAll, and clicking *'Delete all'* should return #deleteAll.
3. Combine the encryption program and random generation of encryption and add code to decrypt the encrypted string.
4. The idea of mapping with a mapping array looks like a potentially useful operation. Define a class called Mapper that will encrypt and decrypt a string. Decide the details of desired behaviors and implementation and use your class to reimplement the encrypting example.

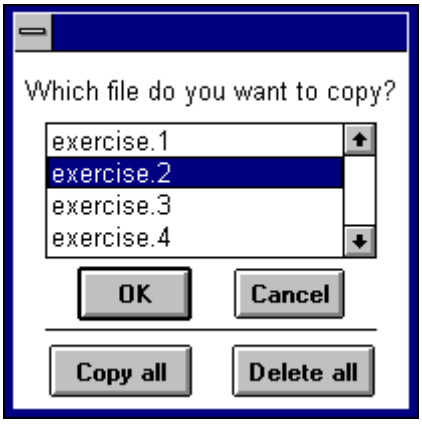

Figure 7.9. Desired user interface for Exercise 2.

7.6 Two-dimensional arrays - tables and matrices

A table is a collection of elements accessible by row and column numbers. It is thus closely related to an array and most programming languages would call it a two-dimensional array; mathematicians would call it a two-dimensional matrix. More generally, an n-dimensional array or matrix is a collection whose elements can be accessed by a set of n indices. An array is thus a one-dimensional matrix, and a table is a two-dimensional matrix.

Two-dimensional matrices are very useful because they capture the substance of a table widget and because they have applications in two-dimensional geometry. While the VisualWorks library does not include a class implementing general n-dimensional matrices, it does have a class called TwoDList which implements two-dimensional arrays and uses it as the value holder for the table widget. In this section, we will present class TwoDList and show how it could be extended to include mathematical behavior of matrices; the use of TwoDList in the table widget will be illustrated in the next section.

Class TwoDList

Class TwoDList is a subclass of ArrayedCollection and this means that its size is fixed. Its instance variables are dependents, collection, rows, columns, and transposed. Variable dependents makes it possible to use TwoDList as a table widget model which is its intended function. Dependency is implemented so that the change of any of its elements sends an update message to all dependents. Variable collection holds the elements of the table in a linear array. Variables rows and columns hold the number of rows and columns of the table, and variable transposed holds true or false depending on whether the table elements should be viewed in their original arrangement or transposed - with rows and columns interchanged.

A table is an interesting example of information hiding because the concept can be implemented in at least four ways and the user does not need to know which one is used (Figure 7.10). One possibility is to implement a table as a one-dimensional array of elements and access it by converting row and column indices into a single linear index. (*VisualWorks uses the first index as column number and the second as row number, treating horizontal axis as the x coordinate of a point and vertical axis as y*. This is the opposite of the usual mathematical interpretation.) This can be done in two ways: row-wise (all elements of row 1 followed by all elements of row 2, and so on), or column-wise (all elements of column 1 followed by all elements of column 2, and so on). Another possible implementation is as an array whose elements are rows which are themselves arrays or as an array whose elements are columns which are arrays. As indicated below, TwoDList uses one-dimensional row-wise storage.

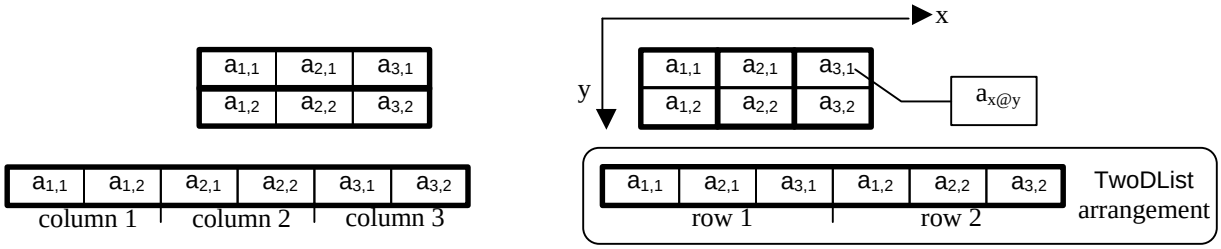

Figure 7.10. Tables can be stored as arrays of columns or rows (top) or as one-dimensional arrays (bottom). Class TwoDList uses the bottom right arrangement.

Creating a TwoDList

The two main TwoDList creation messages are on: aCollection columns: c rows: r and columns: c rows: r. The first creates a TwoDList with elements initialized to aCollection, the second creates an uninitialized TwoDList with specified shape and size. As an example,

```
TwoDList on: #(12 14 16 21 42 24) columns: 3 rows: 2
```

returns TwoDList (12 14 16 21 42 24) with rows #(12 14 16) and #(21 42 24)

whereas

TwoDList columns: 3 rows: 5

returns TwoDList (nil nil nil nil nil nil nil nil nil nil nil nil nil nil nil).

Accessing a TwoDList

Just like Array, TwoDList is accessed by at: and at:put:. The difference is that the at: argument is normally a Point because a position in a table requires two coordinates. The x part of the point is the column number, and the y part is the row number. The following example illustrates the principle:

```
| table |
table := TwoDList on: #(12 14 16 21 42 24) columns: 3 rows: 2.
Transcript clear; cr; show: (table at: 1@2) printString.          "Prints 21 – element in column 1, row 2."
table at: 2@2 put: 77.                "Changes table to TwoDList (12 14 16 21 77 24) and returns 77."
```

The internal conversion from the Point argument to the position in the underlying array uses the following method:

```
integerIndexFor: aPoint
        ^(columns* (aPoint y - 1)) + aPoint  x
```

which confirms that elements are stored row-wise because 1@1 is converted to index 1, 2@1 is converted to index 2, and so on.

The at: message can also be used with an integer argument, in which case the table is treated as a one-dimensional array and accessed directly.

Example. Implementing a two-dimensional matrix with TwoDList

As we already mentioned, matrices are very important for computer graphics because they are the basis of transformations of geometric objects such as shifting, scaling, rotation, and mirroring. All these operations can be performed by a combination of matrix scaling, multiplication, addition, and subtraction and implementation of a class capable of these operations would thus be very useful. Creating a framework with a truly sophisticated set of classes supporting matrices and related concepts is beyond the available space and expected mathematical prerequisites and we will thus limit ourselves to designing a simple Matrix class supporting scaling, addition, and transposition.

The first question is where to put Matrix in the class hierarchy Since Matrix has use for much of TwoDList behavior, we will make Matrix a subclass of TwoDList. Its basic functionality - creation, accessing, and even transposition - are inherited from TwoDList and all we have to do is define methods for scaling and addition.

Figure 7.11. Matrix scaling.

To *scale* a matrix by a number, all its elements are multiplied by the number (Figure 7.11). This can be implemented by enumeration via the collect:, multiplying all elements by the scaling factor:

```
scaledBy: aNumber
"Create a new matrix whose elements are elements of self multiplied by aNumber."
        ^self collect: [:el | el * aNumber]
```

Unfortunately, when we test this on an example such as

```
| matrix |
matrix := Matrix on: #(12 14 16 21 42 24) columns: 3 rows: 2.
matrix scaledBy: 3
```

we fail miserably, getting a walkback (Exception window) which says that "this class does not support variable size allocation" and indicates that it failed in the new: message which was sent by collect:. The problem is that message new: cannot be used if all parts of the class are not accessed by index. We could have anticipated a problem had we compared the definition of Array

```
ArrayedCollection variableSubclass: #Array
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Collections-Arrayed'
```

with the definition of Matrix which is

```
TwoDList subclass: #Matrix
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Book'
```

This is an important lesson because it tells us that some collections don't understand the collect: message and even the new: message! If collect: does not work with Matrix because it is not just a collection, it should work with the collection component of Matrix. We will thus rewrite our method to create a new scaled collection from the receiver's collection and then create a new Matrix from it as follows:

```
scaledBy: aNumber
"Return a new matrix whose elements are elements of self multiplied by aNumber."
| newCollection |
"Modify existing collection."
newCollection := collection collect: [:el | el * aNumber].
"Create new instance with the same size and shape and with the new collection."
^self class
        on: newCollection
        columns: self columnSize
        rows: self rowSize
```

This indeed works as it should as you can test by executing the following example:

```
| matrix |
matrix := Matrix on: #(12 14 16 21 42 24) columns: 3 rows: 2.
matrix scaledBy: 3
```

As an alternative solution, we will now implement the same operation by enumeration over indices. There are two ways to do this: We can either calculate over the two-dimensional coordinates, or we can access the elements of the underlying array directly. The first approach requires two nested loops (over rows and columns), the second approach depends on the knowledge of the internal representation of TwoDList. The second approach is dangerous because it depends on internal representation and if the implementation of TwoDList changed, our method would stop working. We will thus use the first approach:

```
 scaledBy: aNumber
"Return a new matrix whose elements are elements of the receiver multiplied by aNumber."
| scaled |
"Create a new Matrix object with the appropriate size and shape."
scaled := self class columns: self columnSize rows: self rowSize.
"Calculate its elements from the elements of the receiver."
```

```
1 to: self rowSize do: [:rowIndex |
        1 to: columnSize do:
                [:columnIndex | |point| point := columnIndex @ rowIndex.
                        scaled at: point put: (self at: point) * aNumber]].
^scaled
```

This definition is more complicated than the first one but it works.
To *add* two matrices together, elements from the corresponding cells are added together (Figure 7.12).

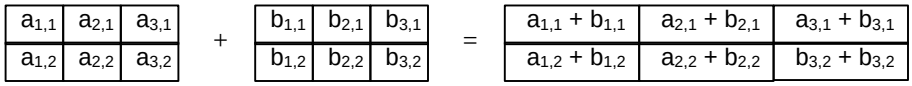

Figure 7.12. Matrix sum. Both matrices must have the same number of rows and columns.

We can thus define matrix addition as follows:

1. Create a new uninitialized matrix with the same size and shape as the receiver.
2. For each point in the range calculate the sum of the corresponding elements of the receiver and the argument and put this value at the corresponding position of the sum.
3. Return the sum.

The implementation is

```
+ aMatrix
"Calculate a new matrix whose elements are sums of corresponding receiver and argument elements."
| sum |
sum := self class columns: self columnSize rows: self rowSize.
1 to: self rowSize do: [:rowIndex | "Enumerate over rows first."
        1 to: self columnSize do: [:columnIndex | | point |
                                point := columnIndex @ rowIndex.
                                sum at: point put: (self at: point) + (aMatrix at: point)]].
^sum
```

and execution of the following program fragment with *inspect* confirms that it works:

```
| matrix1 matrix2 |
matrix1 := Matrix on: #(12 14 16 21 42 24) columns: 3 rows: 2.
matrix2 := Matrix on: #(6 3 8 5 9 7) columns: 3 rows: 2.
matrix1 + matrix2
```

Main lessons learned:

- Class TwoDList implements a two-dimensional array with dependency.
- The main purpose of TwoDList is to serve as the value holder for table widgets but its behavior can be extended to implement the mathematical concept of a matrix.
- Some collections don't understand all enumeration messages.

Exercises

1. How can you determine whether a particular collection class has fixed size?
2. Define and test Matrix method multipliedBy: implementing matrix multiplication. Add printing.
3. The concept of a mathematical matrix is different from the intended use of TwoDList. Redefine Matrix independently of TwoDList. Provide the following protocols: creation, accessing, arithmetic (addition, subtraction, negation, scaling, multiplication), and printing.
4. Define a three-dimensional matrix class called ThreeDArray with the protocols listed in Exercise 3.
5. We found that collect: does not work for Matrix. This raises several questions:
   a. Does do: work?

b. Which other basic Collection messages don't work with Matrix?
c. Which other collection classes suffer from the same problem as Matrix?
d. How can we redefine collect: for Matrix?
e. What is the advantage of putting TwoDList in the Collection hierarchy if it does not understand collect:?

6. Implement matrix addition using with:do:.
7. Our matrix addition does not check whether the two matrices can be added. Add such a check.

7.7 Implementing an n-dimensional array

Although most applications use one- or two-dimensional arrays, three-dimensional arrays are also useful (for example in three-dimensional graphics) and higher dimensional arrays have their uses as well. It is thus natural to extend our definitions and define a class that can implement arrays of *any* dimension. We will now implement such a class under the name NDArray in two different ways.
Class NDArray will have the following responsibilities:

- *Creation* - create an uninitialized NDArray with the specified number of dimensions and 'shape' (size of individual dimensions). This will be implemented with message dimensions: anArray where anArray is an array of integers specifying sizes along individual axes. As an example

  ```
  NDArray dimensions: #(3 4)
  ```

  will create a two-dimensional array with 3 by 4 elements.
- *Accessing* - get or set an element specified by its indices. The two messages will be atIndexArray: anArray and atIndexArray: anArray put: anObject. As an example, if x is an instance of NDArray,

  ```
  x atIndexArray: #(3 4)
  ```

  will return the element with indices 3 and 4.
- *Arithmetic* - negation, addition, subtraction, and multiplication (messages negated, +, -, and *)
- *Printing* (implemented by printOn: aStream). The style of printing will be 'change the first index first'. As an example, a three-dimensional array with dimension sizes #(2 3 4) will be printed as follows:

```
element at index 1 1 1
element at index 2 1 1
element at index 1 2 1
element at index 2 2 1
element at index 1 2 1
element at index 2 2 1
element at index 1 3 1
element at index 2 3 1
element at index 3 3 1
element at index 1 1 2
element at index 2 1 2
etc.
```

- Any auxiliary protocols required to implement the above functionality.

In the following, we will consider two approaches to this problem. One is based on the fact that all collections are internally stored as sequential collections, and the other on the fact that an n-dimensional collection is an array of n-1 dimensional collections.

Solution 1: n-dimensional array as a mapping to one dimension

In this arrangement, the n indices of the n-dimensional array are mapped into a single index of the corresponding one-dimensional array using some mathematical formula. Since the formula is not obvious, we will feel our way through to it by starting with a solution for small dimensions and generalizing the experience to n dimensions.

In the *one-dimensional case*, no mapping is needed – a one-dimensional array is stored as a one-dimensional array and the counterpart of element a_i in the original array is element a_i in the 'storage' array. Index i maps into index i.

In the *two-dimensional case*, we already have experience with TwoDList and we know how it is mapped. For a TwoDList with m columns, element col@row is stored in location (row-1)*m + col. We conclude, that when we are traversing the storage, 'the first index changes first'. We will apply this principle to any number of dimensions.

Consider now the *three-dimensional array* in Figure 7.14. To access the shaded element with indices 4,3,1 (column, row, 'plane'), we must first skip all elements of the first two-dimensional plane in the foreground (1*(4*3) elements), then the first two rows of the background plane (2*4 elements), and then access the fourth element in the row. In general, if the number of elements along the column, row, plane dimensions is m, n, and k, the formula to access an element with indices (a, b, c) is (c-1)*m*n + (b-1)*m + a or a + (b-1)*m + (c-1)*m*n. It is useful to realize that the first element of the sum corresponds to the first dimension, the second to the second dimension, and the last to the last dimension.

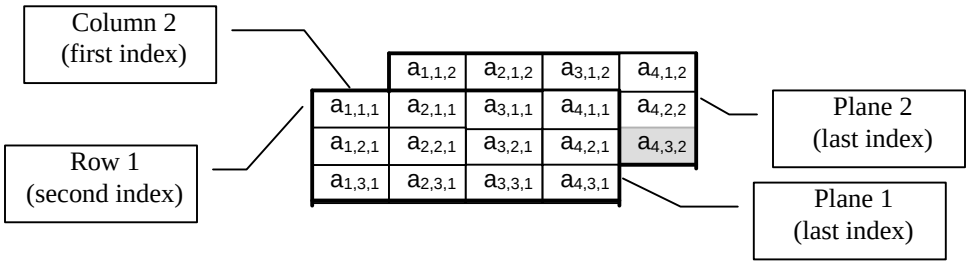

Figure 7.14. A three-dimensional array with dimensions 4, 3, 2 (columns, rows, planes).

We can now deduce that if a an n-dimensional array has dimension sizes a_1, a_2, a_3, ..., a_n, the formula for converting an n-dimensional index i_1, i_2, i_3, ..., i_n, into a one-dimensional index i is

$$i = (i_n-1)*a_{n-1}*a_{n-2}*\ a_{n-3}*...*a_1 + (i_{n-1}-1)*a_{n-2}*a_{n-3}*...*a_1 + (i_{n-2}-1)*a_{n-3}*...*a_1 + ... + (i_3-1)*a_2*a_1 + (i_2-1)*a_1 + i_1$$

To confirm, at least partially, that this formula is correct, check that it gives the correct result for the cases that we investigated above - one-, two-, and three-dimensional arrays.

After finding the general formula, we will now improve it to speed up the calculation of the index. To do this, we will multiply out the parenthesized expressions, separate the constants, and keep them in instance variables of each instance of NDArray. We find that

$$\begin{aligned} i &= (i_n-1)*a_{n-1}*a_{n-2}*\ a_{n-3}*...*a_1 + (i_{n-1}-1)*a_{n-2}*a_{n-3}*...*a_1 + (i_{n-2}-1)*a_{n-3}*...*a_1 + ... + (i_3-1)*a_2*a_1 + (i_2-1)*a_1 + i_1 \\ &= (i_n*\ a_{n-1}*a_{n-2}*\ a_{n-3}*...*a_1 + i_{n-1}*\ a_{n-2}*a_{n-3}*...*a_1 + ... + i_2*a_1 + i_1) - (a_{n-1}*a_{n-2}*\ a_{n-3}*...*a_1) - (a_{n-2}*...*a_1) - ... - a_1 \\ &= (i_n*c_n + i_{n-1}*c_{n-1} + i_{n-2}*c_{n-2} + ... + i_2*c_2 + i_1*c_1) - C \end{aligned}$$

where

$c_n = a_{n-1}*a_{n-2}*\ a_{n-3}*...*a_1$
$c_{n-1} = a_{n-2}*a_{n-3}*...*a_1$
etc.
$c_3 = a_2*a_1$
$c_2 = a_1$
$c_1 = 1$

and

$C = c_2 + c_3 + ... + c_n$ "Does not include c_1"

After establishing this theoretical background, it remains to identify the internal attributes of NDArray. An NDArray needs to know its elements and we will keep them in instance variables elements - an Array of sufficient size. We also need to know the sizes of the individual axes (instance variable sizes - an Array of integers), and the constants calculated according to the formulas derived above. These constants will be held in instance variables multipliers (an Array of c_i constants) and constant (holding the value of C). Finally, we will find it useful to have an instance variable to hold the number of dimensions (dimensions). With this, we can now proceed to implementation.

Implementation

Creation protocol. Method dimensions: anArray will return an NDArray with properly initialized instance variables but no elements. We will define it in the usual way:

```
dimensions: anArayOfSizes
^self new dimensions: anArayOfSizes
```

where instance method dimensions: performs all initialization as follows:

```
dimensions: anArayOfSizes
"Initialize, including calculation of internal constants from specification."
        | temp |
        dimensions := anArayOfSizes size.
        sizes := anArayOfSizes.
        "The total number of elements is equal to the product of all dimensions."
        elements := Array new: (anArayOfSizes inject: 1 into: [:size :el | el * size]).
        multipliers := Array new: dimensions.
        multipliers at: 1 put: (temp := 1).
        2 to: dimensions
                do:
                        [:index |
                        temp := temp * (anArayOfSizes at: index - 1).
                        multipliers at: index put: temp.].
        constant := multipliers inject: -1 into: [:sum :el | el + sum]
```

Test that the method works correctly.

Accessing protocol. To identify an element, we must specify all its indices. The argument of both the *get* and the *put* method must therefore be an n-dimensional array of indices. If we called our methods at: and at:put:, this could cause some problems because conventional at: and at:put: methods expect an integer argument. To avoid this, we will thus use names atIndices: and atIndices:put:. The definition of atIndices: is a simple implementation of the formulas derived above:

```
atIndices: anArray
“Return element specified by its n indices.”
| index |
index := 0.
multipliers with: anArray do: [:el1 :el2 | index := index + (el1 * el2)].
^elements at: index - constant
```

We took advantage of the with:do: enumeration method which operates on two collections simultaneously. Method atIndices:put: is a simple extension of this definition. Since the calculation of the index is the same in both accessing method, it would be advantageous to define it as a special method and share it. When you are finished, create an NDArray and test its accessing protocol.

Arithmetic protocol. Given our accessing protocol, arithmetic is easy and we will leave it as an exercise.

Printing. Assume that we want to print the contents of a NDArray with dimensions #(3 4 6) as

```
1 1 1: e  “Element at index 1 1 1”
2 1 1: e  “Element at index 2 1 1”
3 2 1: e  “Element at index 3 2 1”
1 3 1: e  “Element at index 1 3 1”
2 3 1: e  “Element at index 2 3 1”
3 3 1: e  “Element at index 3 3 1”
1 1 2: e  “Element at index 1 1 2”
2 1 2: e  “Element at index 2 1 2”
3 1 2: e  “Element at index 3 1 2”
```

```
1 2 2: e  “Element at index 1 2 2”
etc. (comments added only for explanation)
```

in other words, we again use the principle ‘first index changes first’.

To obtain the proper behavior of printString, we must redefine the printOn: method. In our example, the desired output consists of a pattern of indices followed by the element at that index. We will implement this by constructing and continuously updating the index pattern, printing it, printing the element, and updating a pointer into the elements array. Pattern updating will be implemented with a method that returns nil when the last pattern has been printed:

```
printOn: aStream
"Append to the argument, aStream, the elements of the Array enclosed by parentheses."
| pattern position |
position := 1.
aStream nextPutAll: 'NDArray'; nextPutAll: ' ('; cr.
pattern := (Array new: dimensions) atAllPut: 1.
[pattern isNil] whileFalse:
        [1 to: pattern size do: [:index | aStream nextPutAll: (pattern at: index) printString , ' ']].
        aStream nextPutAll: ':   ', (self atIndices: pattern) printString; cr.
        position := position + 1.
        pattern := self nextPattern: pattern].
aStream nextPut: $)
```

The process of updating the pattern is as follows: Start from the left end of the current pattern and go right, looking for a number whose value is smaller than the value of the corresponding dimension. If such an element exists, increment it by 1 and reset all elements to the left to 1; return the resulting new pattern. If such an element does not exist, return nil to indicate the end of the process. This algorithm is implemented by the following private method:

```
nextPattern: anArray
"Find the next pattern of indices, return nil if there is none."
| nextPattern |
nextPattern := anArray copy.
“Look for the first element that is smaller than the corresponding dimension.”
1 to: dimensions do:
        [:index | | value |
                value := anArray at: index.
                value < (sizes at: index)
                        ifTrue: [ “Increment the element, reset the preceding, return result.”
                                nextPattern at: index put: value + 1.
                                nextPattern atAll: ( 1 to: index - 1) put: 1.
                                ^nextPattern]].
"All patterns printed, we are finished."
^nil
```

Solution 2: n-dimensional array as an array of n-1 dimensional arrays

A completely different view of an n-dimensional array is as a one-dimensional array of n-1 dimensional arrays (Figure 7.14). This is a recursive view in which a structure contains objects of similar structure but simpler and the bottom object in the hierarchy is a one-dimensional array which contains the actual elements. Although this idea appears complicated, its implementation is simpler than Solution 1. To distinguish our two approaches, we will call this class NDArrayRecursive.

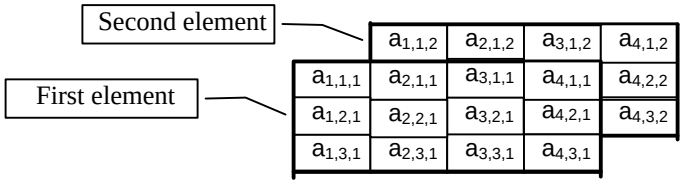

Figure 7.15. Three-dimensional array (three indices) as an array of two-dimensional arrays.

The class will have similar protocol as in Solution 1 and its instance variables will also be similar. They will include dimensions (integer number of dimensions), sizes (array of sizes of individual dimensions), elements (array of n-1 dimensional arrays for n > 1, array of elements for n = 1). Our printing strategy will again be based on a pattern array holding indices and this array will be held in instance variable pattern. Since several of the methods will be recursive, it will be useful to keep track of the number of dimensions of the whole matrix and we will use instance variable globalSize for this purpose.

Implementatio of protocols

Creation. We need two creation messages: One to create the topmost matrix, and another to create the matrices inside it. The reason for this is that we want all enclosed matrices know the size of the outermost matrix.
The topmost creation method will be called newWithDimensions: anArray where anArray is the array of dimensions as in Solution 1. This method recursively creates lower level matrices by message dimensions: anArray globalSize: anInteger. The definitions of the top level class creation method is

```
newWithDimensions: anArray
"Return new instance with axe dimensions as specified"
    ^self new dimensions: anArray globalSize: anArray size
```

and the corresponding class method for creating the enclosed lower level matrices is

```
dimensions: anArray globalSize: anInteger
"Return new instance with axe dimensions as specified"
    ^self new dimensions: anArray globalSize: anInteger
```

This method sends the instance message dimensions: globalSize: which initializes the created object:

```
dimensions: anArray globalSize: anInteger
"Initialize elements from specification."
| size |
sizes := anArray.
dimensions := anArray size.
globalSize := anInteger.
size := anArray at: 1.
elements := Array new: size.
dimensions > 1 ifTrue: “Create the lower level matrix.”
                [1 to: size do:
                        [:index | elements at: index put: (NDArrayRecursive
                                        dimensions: (anArray copyFrom: 2 to: dimensions)
                                                globalSize: anInteger)]]
```

The block statement creates the appropriate number of lower dimensionality arrays if dimension is greater than 1 and passes the remaining dimensions down. As an example, if we want to create a three-dimensional array with sizes #(4 6 2), the creation method creates an NDArrayRecursive with dimension =3, size = 4, and three elements, each of them a two-dimensional NDArrayRecursive. Each of these arrays has dimension 2 (it is two-dimensional), size = 6, and contains six one-dimensional arrays. Each of these one-dimensional arrays has two uninitialized elements. All share the same globalSize = 3.

When we tried to test the creation mechanism with inspect, we got an exception because inspecting elements requires the do: message which class Collection leaves as subclass responsibility. We thus added do: as follows:

```
do: aBlock
"Evaluate aBlock with each of the receiver's elements as the argument."
dimensions = 1    ifTrue: [elements do: [:element | aBlock value: element]]
      ifFalse: [elements do: [:element | element do: aBlock]]
```

We then inspected

```
NDArrayRecursive dimensions: #(2 3 4)
```

and the result was as expected.

Accessing. Here again, we must distinguish higher dimension arrays whose elements are lower dimensionality arrays, and the actual data stored in the NDArray:

```
atIndices: anArray
^dimensions = 1   ifTrue: [elements at: (anArray at: 1)]
                  ifFalse: [(elements at: (anArray at: 1))
                                    atIndices: (anArray copyFrom: 2 to: dimensions)]
```

Note that we must use message atIndices: on the last line - message at: would fail because the argument is an array rather than an integer. Note also that we remove the first index from the elements array as we pass it to the n-1 dimension array. Method atIndices:put: is similar and we leave it as an exercise. We tested accessing by executing the following test program with *inspect*

```
| nda |
nda := NDArrayRecursive dimensions: #(2 3 4).
1 to: 2 do:
      [:col| 1 to: 3 do:
            [:row| 1 to: 4 do:
                  [:plane| nda atIndices: (Array with: col with: row with: plane)
                                    put: col*row*plane]]].
nda
```

and everything works.

Printing. For this implementation, we will write code to output the following format (produced for the array from the test code above):

```
NDArrayRecursive (
1 1 1 : 1
1 1 2 : 2
1 1 3 : 3
1 2 1 : 2
1 2 2 : 4
1 2 3 : 6
1 3 1 : 3
1 3 2 : 6
1 3 3 : 9
2 1 1 : 2
2 1 2 : 4
2 1 3 : 6
2 2 1 : 4
2 2 2 : 8
2 2 3 : 12
2 3 1 : 6
```

```
2 3 2 : 12
2 3 3 : 18
)
```

Our solution will again be based on an pattern of indices but we will leave all printing to the lowermost NDArray level - the one-dimensional array. However, all levels in the hierarchy must contribute to the pattern as they increment their indices and we will implement this as follows:

- When a request to execute printOn: arrives to the uppermost n-dimensional NDArray, this object initializes pattern to an array consisting of all 1s. It then passes pattern down to its n-1 dimensional elements which pass it down to their components, and so on. The one-dimensional at the bottom NDArray prints the indices and the values.
- During each iteration through the printing loop of a k-dimensional NDArray, the NDArray object updates pattern by incrementing its k-th component, and resetting the elements to the right to 1. This is done recursively and all indices are thus correctly updated.

The implementation of this strategy is as follows:

```
printOn: aStream
"Append to the argument, aStream, the elements of the Array enclosed by parentheses."
“The top level array prints the name of the class, etc.”
dimensions = globalSize
                ifTrue: [aStream nextPutAll: 'NDArrayRecursive'; nextPutAll: ' ('; cr.
                        “Initialize index pattern and pass it down to lower level arrays.”
                        pattern := (Array new: dimensions) atAllPut: 1.
                        elements do: [:element | element pattern: pattern]].
dimensions > 1
        ifTrue: [elements do: [:element |
                                aStream nextPutAll: element printString.
                                self updatePattern]]
        ifFalse: [pattern size timesRepeat:
                [1 to: pattern size do: [:index | aStream nextPutAll: (pattern at: index) printString , ' ']].
                aStream nextPutAll: ': ' , (elements at: pattern last) printString; cr.
                self updatePattern]].
“Top level array prints the closing bracket.”
dimensions = globalSize ifTrue: [aStream nextPut: $)]
```

where

```
pattern: anArray
“Initialize this pattern and pass it down to lower level patterns.”
pattern := anArray.
dimensions > 1 ifTrue: [elements do: [:element | element pattern: anArray]]
```

Finally, method updatePattern simply assigns a new pattern to pattern, and updating is done as follows:

```
updatePattern
“Increment my pattern index by 1 and reset those to the right to 1.”
        | myIndex |
        myIndex := globalSize - dimensions + 1.
        pattern at: myIndex put: (pattern at: myIndex) + 1.
        pattern atAll: (myIndex + 1 to: globalSize) put: 1
```

This completes the implementation. We conclude that the solution based on recursion is simpler – we did not even have to figure out how to calculate the index from the index array. It would be interesting to compare the speed and memory requirements of the two solutions.

Main lessons learned:

- Recursion may be applied to structure, not only to algorithms.
- Recursive implementations of algorithms and objects are often simpler.

Exercises

1 . Compare the number of calculation required to calculate the index before and after the simplification of the formula.
2 . Implement the missing protocols in each implementation of NDArray.
3 . Implement printing of the first solution by accessing elements directly. Compare speed.
4 . Re-implement printing for the second solution so that the first index changes first.
5 . Compare the speed of element accessing of both implementations.

7.8 Use of TwoDList in the Table widget

To conclude the chapter, we will now present the table widget, both as an example of the use of sequenceable collections and as a useful GUI component.

The table is a read only widget (you cannot change the contents of its cells) for displaying objects in two-dimensional form. A closely related widget is a data set which also displays data in the form of a table but allows the user to modify cell values. Also, while a table can display any mixture of objects, a data set is used to display rows of related multi-item objects such as student records, or book information. In both the table and the data set, the user can select a cell, a row, or a column, and the program can monitor selection changes and respond appropriately.

The *Aspect* of a table is a TableInterface object (Figure 7.16) with

- information about the data displayed in the table, and the current selection (combined into a SelectionInTable object),
- a description of the table's row and column labels and its other visual aspects.

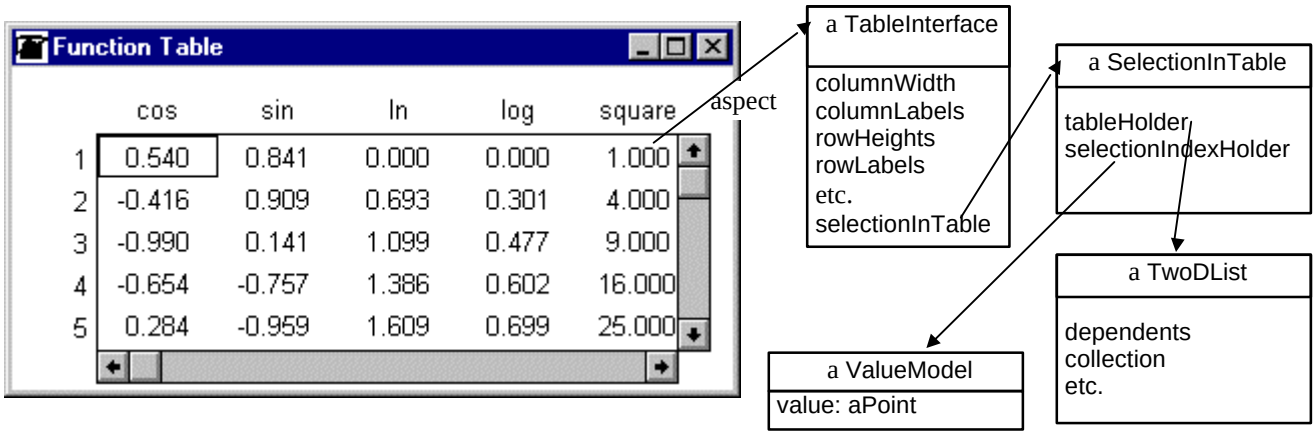

Figure 7.16. The structure of table widget support.

The following slightly edited comment of TableInterface captures its main properties.

"A TableInterface is used to hold the information needed for controlling the format and content of a TableView. The tabular structure of the underlying data is usually captured in terms of a SelectionInTable. TableInterface is also a point at which many operations may be done to the table component. Any operations that are needed to be done to either the row labels or column labels of a table should go through the TableInterface.

Operations that have to deal with the visual properties of a table (color and emphasis) on a column, row, or element basis should either be done in the same way that the TableInterface does it, or through the TableInterface.

Instance Variables:

columnWidths	<ArrayedCollection of <Integer> | Integer | nil> The widths of the columns
columnFormats	<Array | Symbol | nil> #left, #right, and #centered are format symbols
columnLabels	<ValueHolder with: <TableAdaptor> | nil> Labels for the columns of the Table
columnLabelsFormats	<Array | Symbol | nil> #left, #right, and #centered are format symbols
rowLabelsWidth	<Integer | nil> The width of the row label display
rowLabels	<ValueHolder with: TableAdaptor | nil> Labels for the rows of the Table
rowLabelsFormat	<Symbol | nil> #left, #right, or #centered
selectionInTable	<SelectionInTable> The underlying data in tabular form
columnLabelsAndSelection	<SelectionInTable> Models the selection of an entire column
rowLabelsAndSelection	<SelectionInTable> Models the selection of an entire row

A TableInterface allows you to control the width of columns in pixels, and the format (left justified, right justified, or centered) of the contents of the cells and the row and column labels. It also allows the user to select either an individual cell or a whole row or column.

Example: Table of functions

Problem: Write an application displaying a table of values of sin, cos, ln, log, and square for all values from 1 to 10 in steps of 1. The desired format is shown in Figure 7.16.

Solution: To solve this problem, we must paint the user interface, define its parameters, and define the required methods.

Painting the table and defining its parameters. This step is routine. Draw the outline of the table, enter the name of the *Aspect* (name of method returning a TableInterface - we called it tableInterface) and specify any *Details* you want to override the defaults. *Define* the *Aspect* of the table. The labels and the number of columns and cells will be assigned by the initialize method.

Defining the methods. Since our application is so simple, we only need an initialize method to create the all-important TableInterface object from its pieces: A TwoDList with appropriate size and values, and the TableInterface object itself with a SelectionInTable containing the TwoDList object as its underlying collection value holder. Finally, we will specify visual aspects of the table including column widths, text format, and row and label column:

```
initialize
"Create objects underlying the Table widget and calculate its parameters."
| functions values columnWidths |
"Create a TwoDList to hold the data and provide dependency based binding to the widget."
values := TwoDList columns: 5 rows: 10.
"Calculate the data to be displayed, format it, and insert it into the TwoDList."
functions := #(#cos #sin #ln #log #squared).
1 to: values columnSize do:
        [:column | 1 to: values rowSize do:
                [:row | values atPoint: column @ row
                put: (PrintConverter print: (row perform: (functions at: column))
                                        formattedBy: '####.###')]].
"Create the TableInterface and associate it with the list of values."
tableInterface := TableInterface new selectionInTable: (SelectionInTable with: values).
"Define labels and column widths of the TableInterface."
columnWidths := 60. "All columns have the same width."
tableInterface columnLabelsArray: #('cos' 'sin' 'ln' 'log' 'square');
                rowLabelsArray: #(1 2 3 4 5 6 7 8 9 10);
                columnWidths: columnWidths
```

This definition requires several comments:

1. Since we access the TwoDList via rows and columns, we use accessing method atPoint: aPoint put: anObject. Be careful to specify aPoint correctly as column@row. In our first attempt, we accidentally reversed the order of row and column and obtained a strange result.
2. To calculate cell values, we stored the names of the necessary methods as Symbols in an array called functions. We then sent the messages by using the column number to extract the appropriate function from the array, and sending perform: with the value of the row as the argument.
3. After calculating a cell value, we used the PrintConverter to convert the number to a suitable format - the default format is not satisfactory.
4. The default display of the table is awkward and does not have any labels. To obtain the desired look, we ask the TableInterface to set suitable column widths, and define appropriate labels.

Main lessons learned:

- A table is a read-only widget.
- The *Aspect* of a table is an instance of TableInterface .
- TableInterface has two functions: It holds the displayed data and the current selection, and contains information about the visual aspects of the table such as column width and row and column labels.

Conclusion

Most computer applications work with collections of objects. Collections can be divided according to several classification criteria and VisualWorks library contains many collection classes, some for system needs, others for general use.

Abstract class Collection at the top of the collection hierarchy defines several protocols shared by all or most collections. Besides creation, the most important of these is probably enumeration - iteration over all elements of the collection. Enumeration messages are used in most iterations over collections and experienced programmers use their specialized forms whenever opportunity arises.

The collections most useful for general use are array, ordered and sorted collection, list, string, symbol, set, bag, and dictionary. The main differences between these collections are whether they are restricted to a fixed size, whether they allow duplication, and whether their elements are ordered and how this ordering works. The class hierarchy uses ordering as the distinguishing feature to divide collections into two main groups - sequenceable and non-sequenceable.

The main protocols shared by all collections include creation, enumeration, accessing, testing, adding and removing elements (if allowed), and conversion to another kind of collection. Conversion is often used to perform an operation such as sorting or elimination of duplication.

Beginners usually encounter problems when using certain collection messages. For proper use, make sure to know which object the message returns (some methods return the argument when one might expect the receiver), and whether the method changes the receiver or returns its modified copy while leaving the receiver unchanged.

In this chapter, we started our presentation of collections with class Array. An array is a fixed-size sequenceable collection which means that its size cannot grow or shrink, and that its elements are accessed by consecutive integer indices numbered from 1. Arrays are most commonly created as literals, by conversion from other types of collections, by new: followed by calculation of individual elements, or by one of the with: messages. When an array is created by the new: creation message, all its slots are filled with nil objects. Arrays - like most other Smalltalk collections - allow their elements to be any objects or even a mixture of different kinds of objects. The main advantages of arrays over other sequenceable collections are their ease of creation, compactness in terms of internal representation, and fast access.

Each array access in Smalltalk checks whether the specified index is within the array's bounds. This checking adds some execution overhead but makes use of arrays safe.

Arrays are not used as much in Smalltalk as in other languages because Smalltalk provides many other types of collections which may be more suitable for the task at hand, and because it is often preferable to define a class with named components to hold the values instead of using collections. The test that determines whether a multi-component object should be represented by a collection or a class with named

variables is whether the components have identifiable, 'name-able' roles or whether they are just a collection of objects of the same kind.

A TwoDList is a collection designed to support table widgets. It can be considered an extension of Array in two ways: Its elements are accessed by a pair of indices representing a row and a column number, and it implements dependency. We showed how TwoDList could be used to create a new class implementing mathematical matrices with minimal effort.

The table widget is a read-only two-dimensional GUI component equipped with vertical and horizontal scroll bars and visual features including row and column labels, programmable cell width and height, and formatting of labels and cell contents. Its underlying *Aspect* object is a TableInterface. A TableInterface holds information related to the appearance of the table such as labels and column widths, and a SelectionInTable object containing the displayed data and the current selection. The data itself is represented by a TwoDList.

Important classes introduced in this chapter

Classes whose names are **boldfaced** are very important, classes whose names are printed in *italics* are less important, classes whose names are printed in regular font are not of much interest.

Array, *Association*, *Bag*, **Collection**, **Dictionary**, **List**, **OrderedCollection**, *SelectionInTable*, **Set**, **SortedCollection**, **String**, **Symbol**, **TableInterface**, **Text**, **TwoDList**.

Widgets introduced in this chapter

Table.

Terms introduced in this chapter

array - fixed-size collection whose elements are indexed by consecutive integers starting at 1
bag - unordered collection that keeps track of the number of occurrences of its elements
bounds checking - testing whether an index of a sequenceable collection falls within acceptable limits
collection - group of nameless elements with creation, accessing, testing, adding, deletion, and enumeration protocols
dictionary - collection of key-value pairs
enumeration - iteration over collection elements
index - integer number used to access elements of sequenceable collections
nested collection - collection used as an element of another collection
ordered collection - collection whose elements are stored in a fixed externally accessible order; in a more restrictive sense, instance of class OrderedCollection
set - unordered collection that eliminates duplication of elements
sort block - block used by sorted collection to determine the order of its elements
sorted collection - a sequenceable collection whose elements are ordered according to a sort block
sequenceable collection - a collection whose elements are arranged in a fixed order defined by integer indices
string - a sequence of character codes with no information about their rendition
table -a fixed size read-only two-dimensional widget for displaying heterogeneous objects in tabular form
text - string with emphasis prescribing how the string should display itself
unordered collection - collection whose internal ordering is determined by the implementation and inaccessible to the user