Chapter 9 - Sets, bags, and dictionaries

Overview

This chapter presents the remaining frequently used collections - sets, bags, and dictionaries. All of them are unordered which means that they don't provide access to their elements in a user-defined order.

Sets are unordered collections that don't allow duplication so if you add an element to a set several times, the result is the same as if you added the element once. And if you delete an element, it is gone, no matter how many times it has been previously added. Sets are important and frequently used.

Bags are unordered collections that keep track of the number of occurrences of their elements. The Smalltalk library does not use bags and although this does not mean that they are useless, applications of bags are rare.

Dictionaries are sets of special kinds of objects called associations which are key - value pairs. For the purpose of preventing duplication, dictionaries compare association keys. Because the structure of dictionaries is substantially different from that of other collections, their protocols include a number of specialized methods. Dictionaries are used very often.

As a by-product of the discussion of sets, we will deal with the subject of copying objects. We introduce the concepts of shallow and deep copy, and point out the dangers of copying.

9.1 Sets

Sets are an important and frequently used kind of collection. Their capacity automatically grows when needed, and their elements are unordered which means that although they are internally stored as an indexed collection, there is no accessing method that allows accessing sets by index. Most often, sets are accessed by enumeration. The most important part of Set hierarchy is shown below with classes covered in this chapter in bold face.

```
Object ()
        Collection ()

                Set ('tally')
                        Dictionary ()
                                IdentityDictionary ('valueArray')
                                PoolDictionary ('definingClass')
                                        SystemDictionary ('organization')
                        IdentitySet ()
```

Sets eliminate duplication. This means that adding an element that is equal to an element already in the set has no effect. Since a set contains only one copy of each element, removing an element has the same effect no matter how many times the element has been added to the set before - a removed element is gone.

Creating sets

Sets are usually created with new and new:. Message new creates a new empty set with room for two elements:

```
new
        ^self new: 2        "Capacity 2, size 0."
```

Although sets automatically grow when their capacity is filled, the best way to create them is to try to estimate the maximum required capacity beforehand and use new: to create the set with a capacity about *50% larger* for better performance of element accessing.

Another creation message that is occasionally used to create sets is with: and its multi-keyword variants as in

```
Set with: ‘a string’            “Returns the one element set Set ('a string')”
```

The withAll: aCollection creation message can be used to create a set containing the elements of aCollection. The following code fragment uses this approach to create a collection of all elements of array that are divisible by 6 without duplication:

```
| array divisible |
array := #(13 13 12 12 24 24 56 56 54 54).
divisible := Set withAll: (array select: [:number | (number rem: 6) = 0])
```

It is instructive to see the definition of withAll:

```
withAll: aCollection
| newCollection |
        newCollection := self new: aCollection size * 3 // 2.
        newCollection addAll: aCollection.
        ^newCollection
```

The method first creates a new set (self is class Set) whose size is 50% bigger than the size of the collection, adds all the elements of the original collection one after another, and returns the result - the new set. The new set thus shares its elements with the original collection and if either the set or the original modifies one of them, the other collection is affected too.

Sets can also be created with asSet which is understood by all other collections, and this method is often used to eliminate duplication. With asSet, our previous code fragment could be rewritten as follows, giving exactly the same result:

```
| array divisible |
array := #(13 13 12 12 24 24 56 56 54 54).
divisible := (array select: [:number | (number rem: 6) = 0]) asSet
```

Note that if our only goal was to eliminate duplication in the original values, we could proceed as follows:

```
| array |
array := #(13 13 12 12 24 24 56 56 54 54).
array := (array select: [:number | (number rem: 6) = 0]) asSet asArray
```

Adding, testing, and removing elements

Set messages for *adding* new elements include the usual add: anObject and addAll: aCollection. Message add: adds a new element without duplication, addAll: adds all elements of aCollection individually, again without duplication. To illustrate the difference between add: and addAll:, compare the results of executing the following two expressions:

```
Set new addAll: #(1 2 3 4); yourself.        “Returns Set (1 2 3 4) - a set with four elements.”
Set new add: #(1 2 3 4); yourself.           “Returns Set (#(1 2 3 4)) - a set with a single element.”
```

Note that we had to use yourself to obtain the resulting set because both add: and addAll: return their argument rather than the modified receiver. This behavior is, of course, identical to the behavior of add: and addAll: in all other collections.

To *test* whether an object is a member of the set use includes: anElement or contains: aBlock. Both messages return true or false. As an example,

```
| array set |
array := #(13 13 12 12 24 24 56 56 54 54).
set := (array select: [:number | (number rem: 6) = 0]) asSet.     “Returns Set (12 24 54).”
set includes: 12.                                       “Returns true.”
set includes: [:element| (element rem: 6) ~= 0]        “Returns false.”
```

Removing methods include remove: anElement and remove: anElement ifAbsent: aBlock. Both return the argument, just like add: and addAll:

```
| set |
set := Set new addAll: #(1 2 3 4); yourself.    "Returns Set (1 2 3 4)."
set remove: 1.                                  "Returns 1."
set                                             "Returns Set (2 3 4)."
```

Another message that removes set elements is the binary message - (minus) which calculates *set difference*. The difference of two sets is the set of those elements that are in the receiver set but not in the argument set. In other words, the difference is a set obtained by removing from the receiver those elements that are in the argument. As an example,

```
#(1 2 3 4 5) asSet - #(1 3 5) asSet
```

returns Set (2 4).
After this overview, we will now look at several short examples.

<u>Example 1: Is there a difference between asSet and withAll:?</u>
Problem: asSet and withAll: seem to have the same effect. Is there any difference between them?
Solution: There are two definitions of asSet, one in Collection and one in Bag. The definition in Collection is the one that most collections inherit and its listing is as follows:

```
asSet
"Answer a new instance of Set whose elements are the unique elements of the receiver."
	| aSet |
	aSet := Set withAll: self.
	^aSet
```

This shows that asSet and withAll: have exactly the same effect and that addAll: aCollection is more direct.

<u>Example 2: How does a Set eliminate duplication when adding a new element?</u>
Solution: The definition of add: in Set is as follows:

```
add: newObject
"Include newObject as one of the receiver's elements.  Answer newObject."
	| index |
	newObject == nil ifTrue: [^newObject].          “Ignore nil objects.”
	index := self findElementOrNil: newObject.    “Find index where newObject should be.”
	(self basicAt: index) == nil                    “If newObject is not at this location, add it.”
		ifTrue: [self atNewIndex: index put: newObject].
	^newObject
```

The definition reveals that sets are stored as indexed collections even though the Set protocol makes the index invisible. The definition is based on findElementOrNil: which is defined as follows:

```
findElementOrNil: anObject
"Answer the index of anObject, if present, or the index of a nil entry where anObject would be placed."
	| index length probe pass |
	length := self basicSize.
	pass := 1.
	index := self initialIndexFor: anObject hash boundedBy: length.
	[(probe := self basicAt: index) == nil or: [probe = anObject]]
		whileFalse: [(index := index + 1) > length
```

```
                                        ifTrue:  [index := 1.
                                                  pass := pass + 1.
                                                  pass > 2 ifTrue: [^self grow findElementOrNil: anObject]]].
        ^index
```

Method findElementOrNil: tries to find the object in the collection, and if it does not find it, it returns the index where it should be stored. The principle of the search is as follows (Figure 9.1):

1. Ask the object for its hash value and scales it to the size of the set. This is the index at which the object would *ideally* be stored.
2. If the corresponding slot is empty or if it already contains anObject, we are finished.
3. If the slot is not empty but does not contain anObject, anObject could be stored further down. The method checks the next index, and the next, and so on, until the an empty slot or a slot containing anObject is found, or until the method reaches the last index. If the method succeeds, we are finished.
4. If the method reaches the last index without finding an empty slot or anObject, it starts pass 2 from index = 1. Search continues until the condition succeeds (it then exits and returns index) or until it fails by reaching the last available index.
5. If search in Step 4 failed, the collection is full and does not contain anObject. The method thus grows the collection and sends findElementOrNil: again. This time, the message will succeed because we have just created empty slots.

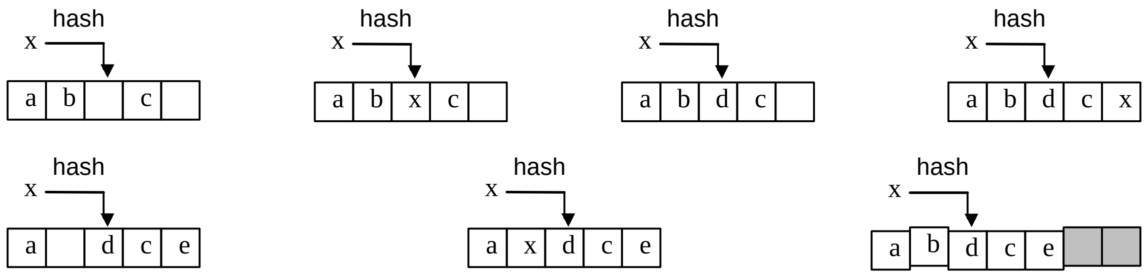


Figure 9.1. Situations that can occur while adding an element to a set. Top (left to right): Slot at hash index is empty, slot contains the argument, slot is occupied by another element but another slot further down is available, slot is occupied by another element but the element is stored further down. Bottom (left to right): No success in the first pass but available slot is found on second pass, element is found on second pass, two passes fail because collection is full and does not contain element - enlarge capacity (gray squares) and try again, this time finding an empty slot.

Our analysis of findElementOrNil: explains the major the role of hash - it calculates an number characterizing an object. In this case, the number is used as the first candidate for an index to store the object. The same principle is used to find an object as follows:

```
includes: anObject
        "Answer whether anObject is one of the receiver's elements."

        ^(self basicAt: (self findElementOrNil: anObject)) notNil
```

and hashing is also used to remove an element from the set. If the hash method is well designed and if there are enough available slots, the scaled hash values calculated for different objects will be spread out and the number of clashes between different objects producing the same hash value and competing for the same slot will be minimal. An element inserted into a set will then be accessed very fast and this is a major advantage of sets.

Since all objects must respond to the hash message, hash is defined in class Object. Since equality and hashing are closely related – two objects that are equal (message =) must have the same hash value -

hash should be redefined in all classes that redefine =. For efficiency of accessing in sets, two objects that are not equal should have a different hash value. However, this is not required and often not guaranteed.

The main use for hashing is fast access of set elements and its calculation must therefore be simple and fast. The principle of the hash method usually is that if only one component of a multi-component object is used to determine equality, its hash value is used for the hash value of the object. As an example, if class Person has instance variables firstName and lastName and if we compare Person objects by their last name only

```
= aPerson
^lastName = aPerson lastName
```

the definition of hash in class Person should be

```
hash
	^lastName hash
```

If equality depends on more components, their hash values should be combined to get the hash function of the object and the usual way of doing this is to use exclusive or. As an example, consider a LibraryEntry class with instance variables author, title, publisher, and year. If we define equality on the basis of author and title

```
= anEntry
^author = anEntry author) and: [title = anEntry author]
```

we will probably define hash in LibraryEntry as follows:

```
hash
	^(author hash) bitXor: (title hash)
```

With this definition, two entries that are equal will also have the same hash value, and if the hash methods of author and title are good, the hash method of LibraryEntry will also behave well.

As a final note on hashing, the principle of converting an object to a representative integer is the principle of hash tables, a very efficient technique for storing performance is better when the capacity of the set is a prime number. More detailed coverage of hashing is beyond the scope of this book.

Example 3: Set union.

Problem: Set difference (the - message) is only one of several mathematical set operations. The other common operations are set union, intersection, and symmetric difference but these operations are not defined in the Set class. Define a method for *set union* as a binary message + using the definition that the union of two sets is the set containing all elements that are in at least one of the two sets.

Solution: The obvious solution of this problem is to take the first set and add all the elements of the second set to it as in

```
+ aCollection
"Calculate union of receiver and aCollection. Return the argument as in all add and remove messages."
	^self addAll: aSet
```

which could be used as in

```
|set1 set2 set3 |
set1 := #(1 3 5) asSet.
set2 := (#(1 2 3 4) asSet).
set3 := set1 + set2                    “Returns Set (1 2 3 4 5).”
etc.
```

Although this solution works, it changes the receiver set1 and this is undesirable. To calculate the union without changing the receiver, we can use a *copy* of the receiver rather than the receiver itself as in

```
+ aSet
"Calculate the union of the receiver and the argument."
| temp |
        temp := self copy.  "Make a copy so that we don't change the receiver."
        temp addAll: aSet.
        ^temp
```

Note that the argument may be any collection, not just a Set.

Example 4: Finding all text fonts available on the current platform

Problem: An application such as a word processor needs to know which text fonts are available on the current platform. Determine how to find their names.

Solution: Hoping that there are messages with the word Font in them, we opened the *implementors of* .. browser on *Font messages, and found one called availableFonts. It turned out to be an instance message in class FontPolicy.

To be able to use the method, we need an instance of FontPolicy. After some further searching we found a FontPolicy used as a component of class GraphicsDevice responsible for display devices. FontPolicy has an instance variable called defaultFontPolicy which provides access to defaultFonts. The next question is how to get an instance of GraphicsDevice. After looking at subclasses of GraphicsDevice, we found class Screen and we thus inspected

```
Screen default defaultFontPolicy availableFonts
```

and this indeed returned a large array of FontDescription objects. When we inspected one of them, we found that FontDescription has many components and one of them called family contains a string with the name of the font. To collect the names of all fonts, we thus inspected

```
Screen default defaultFontPolicy availableFonts collect: [:font| font family]
```

This returned a large array of strings and when we inspected them, we found that many of them are duplicates. The reason for this is turned out to be that there are usually several instance of the same family differing in various parameters such as size. To eliminate this duplication we converted the array of family names to a set by

```
 (Screen default defaultFontPolicy availableFonts collect: [:font| font family]) asSet
```

This indeed returns the desired result.

Example 5. The 'lost object' paradox

Consider the following code fragment:

```
| rectangle |
rectangle := 10@10 corner: 100 @ 100.
Transcript clear; show: rectangle hash printString; cr.
rectangle origin: 20 @ 20.
Transcript show: rectangle hash printString
```

It will not surprise you that the two values of hash output to Transcript are different because the rectangle object, while maintaining its identity, changed its structure and thus its hash value. (When we say that "object x maintains its identity after a change" we mean that all objects that referred to x before the change still refer to it after the change. Just like John Doe remains John Doe when he moves from one address to another or loses weight, the x object retains its identity even when it changes its state.) Now assume that we insert the rectangle into a set, change it, and then test the set for its presence as in

```
| rectangle set |
…
“some statements calculating set”
rectangle := 10@10 corner: 100 @ 100.
set add: rectangle.
…
“some more statements”
…
rectangle origin: 20 @ 20
“some more statements”
set includes: rectangle
```

The amazing result is that although rectangle is still in the set when we send includes:, the last statement *may* return false! The reason for this is that when we execute includes:, its findElementOrNil: method will recalculate hash (getting a different result than when it inserted rectangle into set) and the starting index for its search will thus be different from what it was when we inserted rectangle into set. findElementOrNil: will thus look for rectangle in a slot different from the one where it was initially put.

One of the bad things that can now happen is that the slot corresponding to the new index is still empty, the findElementOrNil: method concludes that the element is not in the set, and includes: returns false.

In essence, the reason for this abnormality is that the set element rectangle changed its hash value during its lifetime. To eliminate this possibility, the library contains another kind of set called Identityset which uses identityHash (and ==) instead of hash (and =) in its findElementOrNil: method. The difference between identityHash and hash is that the value of identityHash never changes during the lifetime of an object, even if the object itself changes. We leave the study of Identityset and identityHash as an exercise.

Main lessons learned:

- Sets (and other unordered collections) store their elements in a sequential fashion but don’t allow index-based access.
- The most common set-related protocols are creation, conversion, enumeration, adding, and removing. All behave consistently with other collections.
- To access an element, sets calculate its hash value, a SmallInteger, using the hash method. The default definition of hash is in class Object.
- Because of their shared use in sets, hash and = are related and when = is redefined, hash should be redefined as well.
- The value of hash may change during an object’s lifetime even though the identity of the object does not. This may have unexpected results when testing sets. Class IdentitySet eliminates this problem.

Exercises

1. Explain the purpose of instance variable tally in the definition of Set.
2. To create a collection of elements without duplication, you can create a set and add the elements to it, or create an OrderedCollection with the elements and convert it to a set. Which approach is faster?
3. Define Set methods to calculate symmetric difference (set of all elements that are in one set but not in the other), and intersection (set of all elements that are in both sets). The receiver should not be affected.
4. Is there a measurable difference in speed between withAll: and asSet? What do you conclude about the penalty of extra message sends?
5. Implement a new set class using equality instead of hashing to eliminate duplication and compare its performance with the existing implementation.
6. Method findElementOrNil: is the basis of accessing objects in a set. Explain how its implementation creates a close relationship between hash and = messages.

7. How does remove:ifAbsent: work? What is the major problem that it must resolve?
8. Explain why inspecting a Set often shows many nil elements.
9. The size of a set should always be considerably smaller than its capacity. Why?
10. We explained that if a set element changes its state during its lifetime, the test for its presence in the set may fail. This begs two questions:
    a. Under what conditions will the test succeed, and when will it fail?
    b. What can we do to prevent this from happening without using IdentitySet? (Hint: Consider what is the cause of the problem and examine the protocols of Set for a solution.)
11. Demonstrate the 'lost object' paradox in a code fragment. (Hint: Under what conditions is it more likely to occur?)
12. Experiment with efficiency of hashing as a function of the relation between set capacity and its size, and the choice of the capacity of the set as a prime number.
13. How does class IdentitySet differ from Set?
14. Examine ten definitions of hash in the library and check if each has a corresponding definition of =.
15. Explain the difference between hash and identityHash and illustrate it on several examples.
16. Explain the reason for the word identity in the name of identityHash.
17. The comment of identityHash contains the following sentence: When two objects are not ==, their identityHash values may or may not be the same. Explain it.
18. Are there other uses of hashing in the class library beyond the one discussed in this section?

9.2 Copying objects

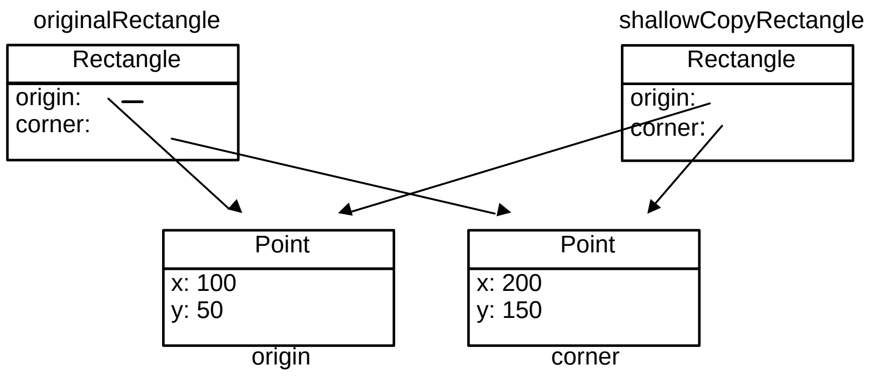
How does the copy method that we used in the previous section work? Does it create a new object that shares its components with the original, or is it a completely new object with new components? Or is it something in between? This is a very important question because if copy shares instance variables with the original then changes of the original affect the copy and vice versa, and this may not be desirable. As an example, if you wanted to make a new version of a document by making and editing a copy, changing the copy would change the original as well.

As a specific example, consider a Smalltalk Rectangle - an object with instance variables origin and corner, each of them an instance of Point. What does the following program fragment do?

```
| aRectangle aRectangleCopy |
aRectangle:= Rectangle origin: 10@10 corner: 20@20.
aRectangleCopy := aRectangle copy
```

The two ways in which copy could be implemented are as follows (Figure 9.2):

- The copy message could create a new Rectangle object and make its two instance variables share its values with the instance variables of the original. This kind of copy is called a *shallow copy*.
- The message could create a new Rectangle and assign its instance variables copies of instance variables of the original. This kind of copy is called a *deep copy*.



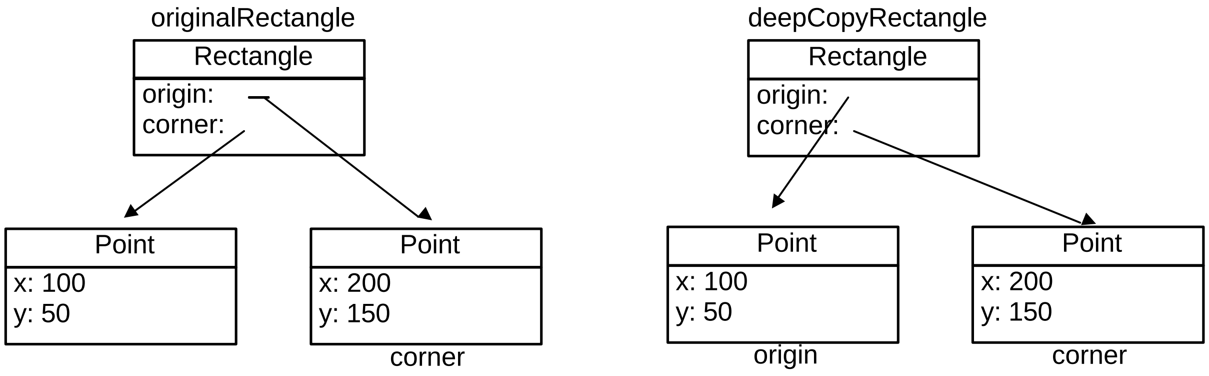


Figure 9.2. Shallow copy (top) and deep copy (bottom).

Using these new terms, we can rephrase our original question as follows: Does Smalltalk make shallow copies, deep copies, or some other kind of copies? The answer is that Smalltalk generally makes shallow copies but provides a mechanism to extend the copy operation, allowing the programmer to create a deep copy if it is needed. We will now explain how this works.
The default version of copy is defined in Object as follows:

```
copy
" Answer another instance just like the receiver."
	^self shallowCopy postCopy
```

The method first makes a shallow copy and then sends postCopy to the result. The default definition of postCopy does not do anything and its only purpose is to allow other classes to perform any additional work after the shallow copy has been made. In other words, postCopy is a hook.

The overwhelming majority of classes don't redefine postCopy and for most classes, a copy is thus a shallow copy. Method shallowCopy itself is almost never redefined and the only exceptions are classes that do not allow copies at all such as Symbol (each Symbol is unique) or UndefinedObject (there is only one instance - nil). As an example, the definition of shallowCopy in Symbol is

```
shallowCopy
"Answer the receiver because Symbols are unique."
	^self
```

As we have already mentioned, few classes redefine postCopy and the default definition is to do nothing. The following example shows how postCopy is *redefined* in class Rectangle:

```
postCopy
	super postCopy.
	origin := origin copy.
	corner := corner copy
```

The definition creates two new Point objects whose values are the same as those of the original and a copy of a Rectangle is thus a deep copy as in the bottom part of Figure 9.2.

Although the difference between the effects of shallow and deep copy should now be clear, we will illustrate it on a small example. The following fragment first creates a Rectangle and makes its shallow copy using shallowCopy (not the usual way of making copies) and a deep copy using copy. It then modifies the origin of the original rectangle and shows how this change affects the shallow and the deep copy:

```
| deepCopyRectangle shallowCopyRectangle originalRectangle corner |
“Create a rectangle.”
originalRectangle := Rectangle origin: 10@10 corner: 20@20.
“Make shallow and deep copies.”
shallowCopyRectangle := originalRectangle shallowCopy.
deepCopyRectangle := originalRectangle copy.
“Show whether instance variables are the same objects or just copies.”
Transcript show: 'shallow copy corner == original corner ?',
            (shallowCopyRectangle corner == originalRectangle corner) printString; cr.
Transcript show: ' deep copy corner == original corner ?',
            (deepCopyRectangle corner == originalRectangle corner) printString; cr.
“Get the corner of the original and change it.”
corner := originalRectangle corner.
corner x: 0; y: 0.
“Show how this affected the original, the shallow copy, and the deep copy.”
Transcript cr; show: 'Rectangles: '; cr;
            tab; show: 'original rectangle: ', originalRectangle printString; cr;
            tab; show: 'shallow copy: ', shallowCopyRectangle printString; cr;
            tab; show: 'deep copy: ', deepCopyRectangle printString
```

The program produces the following output:

```
shallow copy corner == original corner?  true
deep copy corner == original corner?  false

Rectangles:
            original rectangle: 10@10 corner: 0@0
            shallow copy: 10@10 corner: 0@0
            deep copy: 10@10 corner: 20@20
```

The result is as expected: Changing the corner object in the original affects the shallow copy (and the original would be similarly affected by changing the shallow copy), but not the deep copy.

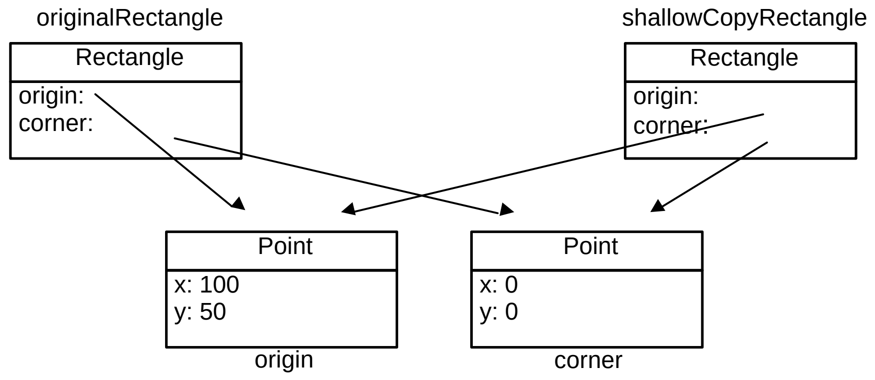
Note that the side effect produced by shallow copy and illustrated by the above example occurs only if we change the *state* of a shared component but not its *identity*. If we *replace* a component rather than *change* it, we change the identity (the two objects now refer to different components) and there is no side effect (Figure 9.3). To see this, replace

```
corner := originalRectangle corner.
corner x: 0; y: 0.
```

in the example above with

```
originalRectangle corner: (Point x: 0 y: 0)
```

and re-execute it. This time, the original will be different even from its shallow copy.



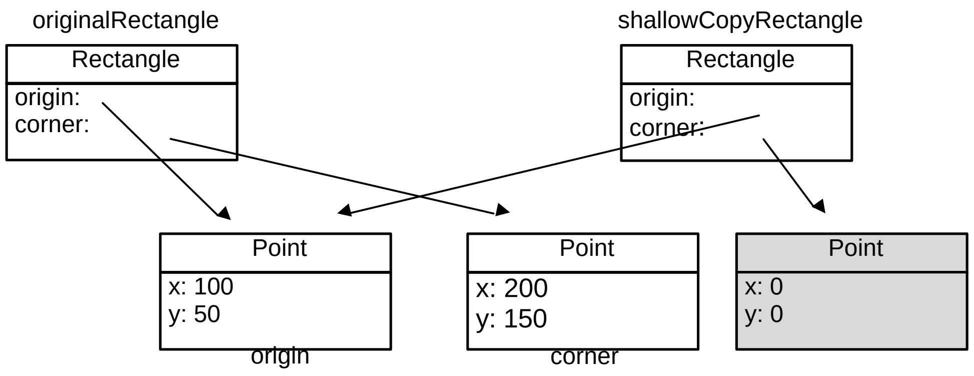


Figure 9.3. Original and shallow copy from Figure 9.2 after changing the value of corner (top) and after assigning a new Point to corner (bottom).

Up until recently, Smalltalk contained a deepCopy method but the method was removed because it is difficult to implement for *circular (recursive) structures* - objects, whose instance variables point to objects whose instance variables point to the original or to some other object that points to the original. A deep copy can then enter into an infinite loop, copying and recopying all objects in the component chain. Since many situations don't really require a deep copy, the method has been dropped in favor of postCopy.

Circular structures are not only of academic interest and occur frequently in natural situations. As an example, a Student object could have a variable called major whose value is an instance of class Major representing the field in which the student majors. Major, in turn, could have a variable whose value is the collection of all students with this major. By following the pointers, we thus very quickly return to the starting object (Figure 9.4) and deep copy will have to be more sophisticated than just following references and making copies of referenced objects until no more references can be found.

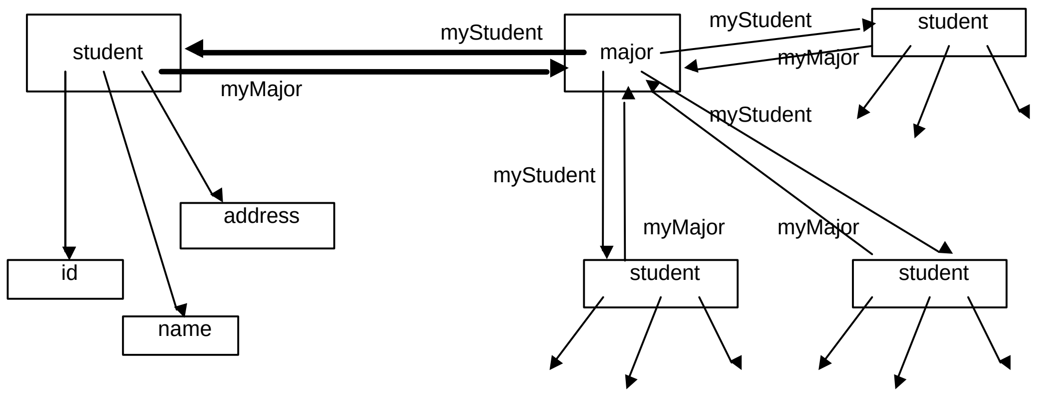


Figure 9.4. Example of a circular structure.

Main lessons learned:

- The copy operation comes in two flavors - shallow copy and deep copy.
- A shallow copy creates a new object which shares its components with the original.
- A deep copy creates a new object whose components are distinct from the corresponding components of the original but equal in value.
- Changing the state of a component of a shallow copy or its original affects both the original and the copy. This does not happen with deep copy.
- *Replacing* the component of a shallow copy or its original with a new object breaks sharing.
- VisualWorks default copy is a shallow copy followed by postCopy. The default behavior of postCopy is to do nothing but the method is redefined in several classes to obtain non-default behavior.
- The definition of shallowCopy is shared by all classes except those that don't allow a copy. These classes reimplement shallowCopy to return the receiver.

Exercises

1. Define class Student with instance variables name, a String. To test how instances of Student behave with respect to the default copy operation, create a Student object with name 'John' and its copy and observe the effect of changing the value of name in the original on the copy and vice versa. Change the name by a message such as student name at: 2 put: $x.
2. Modify class Student to eliminate the side effect resulting from shallow copy.
3. Add class Major implementing the essence of the example in this section and making make deep copies of student-major combinations.

9.3 Bags

A bag is like a set in that it is an unordered collection. Unlike sets, however, it keeps track of the number of occurrences of each of its elements. As an example, if the same element is added to a bag five times and one copy is then removed, the gas knows that it still has four copies of the element.

Bags can be useful when we collect objects and don't care about keeping individual copies of each and the order in which they are stored, but want to know how many copies of each element we have. A typical example is, of course, putting items in a shopping bag and we will illustrate bags in this context.

Example: A better shopping minder

Modify the shopping minder program from Section 8.2 according to the following new specification:

Problem: The program allows the user (presumably a store employee) to initialize a list of items and prices via a sequence of dialogs. The program then asks the user (now the customer) to select items from a multiple choice dialog window (Figure 9.5) repeated as many times as necessary. Note that the items are displayed in alphabetical order.

When the user indicates the end of selections by clicking *Cancel*, the program prints the selected items with their counts, the total price, and all selected items costing more than ten dollars in the Transcript. The form of output is similar to that in Chapter 8. (This is still a preliminary implementation of a program with a nice user interface and file storage, but the final implementation is left as an exercise.)

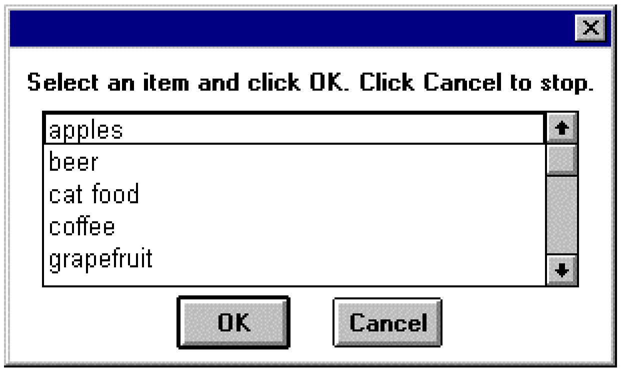


Figure 9.5. Example: Desired user interface for item selections.

Solution: Our previous implementation Item in Section 8.2 used classes ShoppingMinder. The details of ShoppingMinder will clearly change but the nature of class Item remains unchanged. We will thus restrict our attention to analyzing and modifying ShoppingMinder.

The existing ShoppingMinder stores items in an OrderedCollection called items. In the new solution, we will use this collection for the original list and add a new variable called selectedItems to hold the items selected by the customer; this variable will be used as the shopping Bag. The new ShoppingMinder thus has two instance variables called items and selectedItems and the new initialization method now has two instance variables to initialize:

```
initialize
"Initialize items to a suitably sized ordered collection."
	items := (SortedCollection new: 20) sortBlock: [:item1 :item2| item1 name < item2 name].
	selections := Bag new: 20
```

Execution of the program is the responsibility of the execute method, as before. Its essence remains valid for the new formulation except that we have now separated item entry and item selection:

```
execute
"Let store person add items, let the user select, and display the results."
	[self addItem] whileTrue.
	[self selectItem] whileTrue.
	self displayItems.
	self displayTotal.
	self displayExpensiveItems
```

It now remains to update the components of this definition. Method addItem simply asks the user to confirm that another item is to be added, requests the information, and adds it to the items variable. The old definition is

```
addItem
"Obtain an Item object from the user or an indication that no more items are needed. Add it to items."
	| name price |
	name := Dialog request: 'Enter item name' initialAnswer: ''.
```

```
        name isEmpty ifTrue: [^false].        "Exit and terminate loop - user indicated end of entry."
        price := (Dialog request: 'Enter item price' initialAnswer: '') asNumber.
        items add: (Item name: name price: price).
        ^true
```

and although the nature of variable items has changed from OrderedCollection to SortedCollection, this does not affect the validity of the original code thanks to polymorphism - OrderedCollection and SortedCollection both respond to add: appropriately. This method thus remains unchanged.

Method selectItem is new. It opens a multiple choice dialog window with items, and allows the customer to make a selection. The definition is

```
selectItems
"Open window with available items and allow customer to make selections."
        | labels selectedItem |
        labels := (items collect: [:anItem | anItem name]).
        selectedItem := Dialog     choose: 'Select an item and click OK. Click Cancel to stop.'
                                   fromList: labels
                                   values: items
                                   lines: 5
                                   cancel: [nil].
        selectedItem isNil
                ifTrue:   [^false]
                ifFalse:  [selectedItems add: selectedItem.
                          ^true]
```

The definition of displayItems is simple, we only add the display of the number of occurrences of each item:

```
displayItems
"Display alphabetically all items selected by the customer along with their prices. Use alphabetical order."
        Transcript clear.
        selections asSortedCollection do: [:item | Transcript show: item displayString; tab;
                show: (selections occurrencesOf: item) printString; cr]
```

The code illustrates the power and flexibility of collections - the bag is easily converted into a sorted list and easily displayed alphabetically in the Transcript. The definition of displayTotal is almost the same aqs before except that we must take care of the nuber of occurrences of individual items:

```
displayTotal
"Add together prices of all selected items and display them in Transcript."
Transcript cr; show: 'Total price: '; tab; show: (selectedItems inject: (Currency cents: 0)
        into: [:total :item | total + (item price * (selectedItems occurrencesOf: item))]) displayString
```

Finally, the only change in displayExpensiveItems is that it operates on selectedItems rather than items:

```
displayExpensiveItems
"Display a heading and all items that cost more than 10 dollars."
        Transcript cr; cr; show: 'Expensive items:'; cr.
        selectedItems do: [:item | item isExpensive ifTrue: [Transcript show: item displayString; cr]]
```

We are now finished and the last step is to test that everything works. And sure enough, when we test, the program does not do quite what we expected. The problem is that the total price seems to be incorrect as in

```
apple juice        price: 1 dollars   90 cents     number of units: 2
cat food           price: 24 dollars   50 cents    number of units: 1
mineral water      price: 2 dollars   50 cents     number of units: 3

Total price:       54 dollars, 60 cents
```

```
Expensive items:
cat food	price: 24 dollars   50 cents
```

Something is wrong with displayTotal, and the only thing that can be wrong is the calculation in inject:into:. This method is defined in Collection in terms of do: and we should thus check whether we are not misunderstanding do:. Checking implementors of do:, we find that Bag has its own definition

```
do: aBlock
	contents keysAndValuesDo:		"Use block providing access to objects and their counts."
		[:obj :count | count timesRepeat: [aBlock value: obj]]
```

which shows that for each element of the bag, the block statements are executed as many times as there are occurrences of the element and this is the cause of our problem. We must eliminate multiplication by the number of occurrences as follows:

```
displayTotal
"Add together prices of all selected items and display them in Transcript."
	Transcript cr; show: 'Total price: '; tab; show: (selectedItems inject: (Currency cents: 0)
				into: [:total :item | total + item price]) displayString
```

After this change, everything works fine and the output for the same items is

```
apple juice		price: 1 dollars   90 cents   number of units: 2
cat food	price: 24 dollars   50 cents   number of units: 1
mineral water		price: 2 dollars   50 cents   number of units: 3

Total price:		35 dollars, 80 cents

Expensive items:
cat food	price: 24 dollars   50 cents
```

Main lessons learned:

- Bags are unordered collections that don't eliminate duplication.
- If the number of occurrences of an element is n, the do: message in class Bag executes the block statements with this element n times.

Exercises

1. Although bags are conceptually related to sets, this relationship is not reflected in their place in the Collection hierarchy. Examine the definitions of Set and Bag and write a brief comparison.
2. How does Bag know the number of occurrences of its elements?
3. Is Bag's treatment of new elements based on equality or equivalence?
4. Write a summary of all definitions of occurrencesOf:.
5. Create an array containing 500 random integer numbers between 1 and 1,000 and calculate how many different integers it contains and how many copies of each. Use Bag.

9.4 Associations and dictionaries

Many applications involve operations on key-value pairs. Arrays and other kinds of sequenceable collections provide a specialized form of this structure if we consider consecutive integer indices to be keys, but in many cases we need a non-integer object for the key. For situations such as these, Smalltalk provides class Dictionary.

An obvious example of the need for dictionaries is an assembler program which translates a source program into machine instructions, replacing symbolic instruction names with their binary opcodes. This process is based on associating symbolic names with binary opcodes which could be described as

```
'add' -> 2r10010001          "The prefix 2r means that the code is a base 2 number. r stands for radix."
'sub' -> 2r10010010
etc.
```

Another possible use of dictionaries is an English-French dictionary consisting of pairs of English words and their French equivalents as in

```
'father'   -> Set ('pere' , 'papa')
'mother' -> Set ('mere', 'maman')
```

As yet another example, one could represent a font as a set of associations such as

```
'Arial'                      -> Dictionary ('a' -> shape of 'a', 'b' -> shape of 'b', etc.)
'New Times Roman'            -> Dictionary ('a' -> shape of 'a', 'b' -> shape of 'b', etc.)
```

A library of fonts would then be a dictionary whose values are other dictionaries (Figure 9.6).

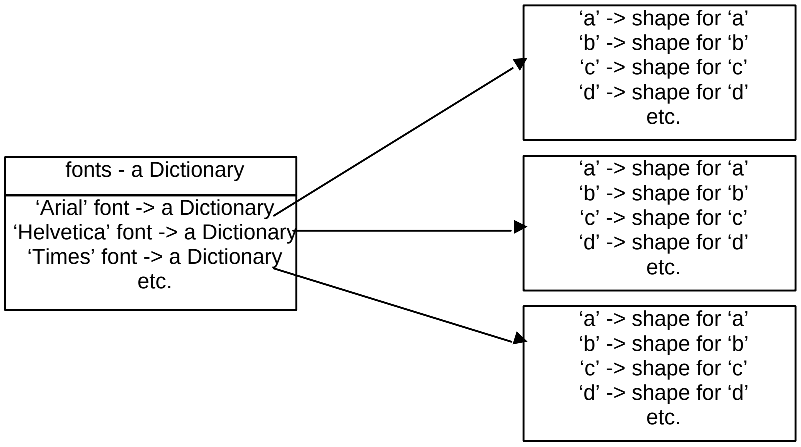


Figure 9.6. Font library as a dictionary of dictionaries.

<u>Class Dictionary</u>

According to its comment, Dictionary is a Set of Association objects, key->value pairs. It is a computer analogy of the conventional dictionary except that it is not ordered. Another difference is that while a conventional dictionary often include entries with the same keys, a Smalltalk dictionary does not allow two different associations with the same key because it is a Set and its comparison for equality uses = on keys. Associating a new value with a key already present in a Dictionary thus replaces the old association with the new one. As an example, if a Dictionary contains association[1]

```
'father' -> 'dad'
```

and you add a new association

```
'father' -> 'daddy'
```

[1] Binary message -> which creates a new association is declared in class Object because any object may be used as the key of an Association.

the second association replaces the first one because its key is equal to the first key.

IdentityDictionary is an important subclass of Dictionary that uses equivalence instead of equality, and is internally represented as a collection of keys and a collection of values rather than as a set of associations. An IdentityDictionary is often used as a kind of a knapsack to carry around a variety of identifiable objects that could not be predicted when a class was designed and consequently could not be declared as instance variables. To put this differently, an IdentityDictionary can be used as an on-the-fly collection of instance variables. As an example, when a user interface implements drag and drop functionality (Appendix 8), the object in charge of following the mouse cursor around the screen is an instance of DragDropManager. One of the functions of this object is to carry around information about the source widget and the target widgets that might be useful in the operation. Since the nature of some of these objects is unpredictable, they can be stored in an IdentityDictionary as associations identifyingSymbol -> object. Any object that needs them may then access them by their symbolic name, almost as if they were instance variables. As another example, a software agent wandering around the Web might carry along and gather all kinds of unpredictable objects, and these objects, along with identifying symbols, could be stored in an IdentityDictionary. An ordinary Dictionary could be used as well but Symbols are unique and identity comparison is thus appropriate.

The idea of making the dictionary class a subclass of set seems strange. After all, real dictionaries are arranged alphabetically by their keys and subclassing to SortedCollection would seem more logical. The reason for this choice is efficiency: Set elements are accessed by hashing and this is a very fast operation.

After this general introduction, we will now examine the main protocols of class Dictionary and give several short examples of their use.

Creation

Dictionaries are usually created with message new which creates an uninitialized dictionary with a default capacity. If you can estimate the size, use the new: message to eliminate growing. You can also create an initialized dictionary using message withKeysAndValues: whose argument is an array of alternating keys and values as in

```
|array |
array := #('a' 1 'b' 2 'c' 3).
Dictionary withKeysAndValues: array          "Returns Dictionary ('a'->1 'b'->2 'c'->3)."
```

This method is rarely used.

Adding and changing elements

Message at: key put: value adds a new association or replaces an existing association that has the same (equal) key. This message is frequently used as in the following example which creates a table of natural logarithms:

```
|logarithms |
logarithms := Dictionary new: 10.
1 to: 10 do: [:key | logarithms at: key put: key log].
etc.
```

Storing values in a Dictionary for easy retrieval is one of its typical uses. It is useful when we need frequent access to objects that require a long time to calculate and when we are willing to trade memory space for execution time. This approach to increasing program speed is called *caching*.

Note that at:put: returns the value of the second argument rather than the dictionary itself. This behavior is, of course, in line with behavior common to all collections.
Another way to add associations to a dictionary or to change them is to use add: anAssociation as in

```
dictionary add: key -> value
```

which has the same effect as

```
dictionary at: key put: value
```

Removing associations

To remove an association from a dictionary, use remove*Key*: or remove*Key*:ifAbsent:. The standard Collection messages remove: and remove:ifAbsent: are illegal with dictionaries because they are redefined as follows:

```
remove: anObject ifAbsent: exceptionBlock
"Provide an error notification that Dictionaries can not respond to remove: messages."
	self shouldNotImplement
```

and similarly for remove:.

Accessing

To obtain the value associated with a given key, use the at: message if you know that the key is present, or the at:ifAbsent: message if you are not sure. The following example allows the user to enter an English word and displays the French equivalent if the pair is in the dictionary. If the English word is not in the dictionary, it displays a 'not in the dictionary' warning:

```
| answer array dictionary key |
"Construct a dictionary."
array := #('mother' 'mere' 'father' 'pere' 'sister' 'soeur' 'father' 'papa').
dictionary := Dictionary withKeysAndValues: array.
"Prompt user for key word."
key := Dialog request: 'Enter an English word present in the dictionary' initialAnswer: ''.
"Check whether the key is in the dictionary and construct a message."
answer := dictionary
	at: key
	ifAbsent: ['not in the dictionary'].
"Display result."
Dialog warn: 'The French equivalent of ', key, ' is ', answer
```

Note that we put two associations with keyword ‘father’ in the dictionary. Check what is the result.

Two other useful accessing messages are keys and values. Message keys returns the *set* of all keys, message values returns an *ordered collection* of values as in

```
|array itemsAndPrices |
array := #(sugar 15 salt 8 coffee 35 eggplant 20 kiwi 25).
itemsAndPrices := Dictionary withKeysAndValues: array.
itemsAndPrices keys.		"Returns  Set (#eggplant #kiwi #sugar #salt #coffee)."
itemsAndPrices values.	"Returns OrderedCollection (20 25 15 8 35)."
```

Enumeration

Dictionaries can be enumerated with do: and other enumeration methods just like other collections. Remember, however, that do: work son *values*, not on associations as one might expect. As an example of the use of enumeration, the total price of grocery items stored in a dictionary containing item-price pairs could be calculated with inject:into: because its definition uses do: and therefore works on values:

```
| array itemsAndPrices total |
"Create an Array with items and prices."
array := #(sugar 15 salt 8 coffee 35 eggplant 20 kiwi 25).
```

```
"Convert it to a Dictionary of symbol -> integer values."
itemsAndPrices := Dictionary withKeysAndValues: array.
"Calculate the total."
total := itemsAndPrices inject: 0 into: [:subTotal :price | subTotal + price].
total "Returns 103."
```

The reason why enumeration works on values is that Dictionary is conceptually a keyed collection, a generalization of the concepts of Array and OrderedCollection. Just as enumeration messages for Array and OrderedCollection enumerate over values rather than indices, enumeration over dictionary elements thus ignores keys. In fact, this analysis suggests that Dictionary is conceptually misplaced in the class hierarchy and that its position is dictated by implementation reasons. In other Smalltalk dialects, Dictionary is indeed located on a different branch of the Collection tree.

In addition to enumeration over values, it is often necessary to enumerate over *associations*. For this, use message keysAndValues:. As the name suggests, keysAndValues: requires a two-argument block ranging over the keys and values of individual association. The first block argument is the key and the second is the value as in the following example which uses keysAndValues: to display the pairs of item names and prices from a shopping list in the Transcript in the following format:

```
eggplant costs 20
kiwi costs 25
```

The program is as follows:

```
|array itemsAndPrices |
“Construct a dictionary of names and prices.”
array := #(sugar 15 salt 8 coffee 35 eggplant 20 kiwi 25).
itemsAndPrices := Dictionary withKeysAndValues: array.
“Display the name-price pairs.”
Transcript clear.
itemsAndPrices keysAndValuesDo: [:item :price | Transcript show: item, ' costs ', price printString; cr]
```

Since a dictionary is a set, the order in which the associations are listed is unpredictable. It is interesting to note that keysAndValuesDo: is also understood by all sequenceable collections where it treats keys as integer indices.

Testing

Message contains: aBlock is defined in Collection on the basis of detect: which, in turn, uses do:. As a consequence, Dictionary inherits contains: and applies it to values as in

```
itemsAndPrices contains: [: aValue | aValue > 10]
```

Message includes: aValue tests whether the dictionary contains an association with value aValue. To check for a *key*, use includesKey: aKey. All testing messages return true or false. As you can see, many dictionary messages access values by default.

Main lessons learned:

- A dictionary is a set whose elements are associations.
- An association is a pair of objects called key and value.
- Uniqueness in dictionaries is based on equality of keys.
- The most frequently used Dictionary messages include creation, adding, removing, accessing, and enumeration.
- Most dictionary methods work with values rather than associations.
- One of the uses of dictionaries is for caching frequently needed objects that would require considerable time to calculate. The price for improved speed is increased memory requirements.

Exercises

1. Test whether caching logarithms in a dictionary really saves time.
2. What does the add: anAssociation message return?
3. Write and test method withKeys: array1 withValues: array2 to create a new dictionary from two arrays.
4. Implement two-dimensional arrays using dictionaries. Each association's key is a row number and the value is an array of elements of the corresponding row.
5. Reimplement the two dimensional array by using an array of indices for the key and the element value as the value of the association.
6. Reimplement NDArray from Chapter 7 using a dictionary.
7. Write a description of IdentityDictionary.

9.5 Dictionary with multiple values

In many situations requiring dictionaries, the value part of each association is a collection. As an example, each English word in an English-French dictionary may have multiple French counterparts. When we add an English-French pair to such a dictionary, the message should create a new pair if the dictionary does not yet contain the English word, but if the English word already is in the dictionary the French word should be added to the existing collection of meanings. Assuming that the dictionary is stored in instance variable dictionary, this can be done as follows:

```
addEnglish: string1 french: string2
"Add string2 to the collection for string1 if it exists, otherwise create new string->collection association."
	dictionary includesKey: string1
		ifTrue: [dictionary at: string1 add: string2]
		ifFalse: [dictionary at: string1 put: (Set with: string2)]
```

Checking whether a French word is among the values of such a dictionary could be done by obtaining all the values (a collection of Set objects) and checking whether any of them includes the word:

```
includesFrench: aString
"Check whether French word aString is in the dictionary."
	^dictionary values contains: [:collection| collection contains: [:el| el = aString]]
```

Deletion is also possible but again not trivial.

Dealing with collection-type values is thus possible but not completely elementary. Since one of the main principles of Smalltalk is that programming should be made as simple as possible by extending the built-in facilities, it makes sense to create a new kind of dictionary whose values are collections and where manipulations of the above kind are implemented as methods. We will create such a class and call it MultiValueDictionary. The new class is a specialization of Dictionary and all methods available in Dictionary should remain executable. We will thus make Dictionary its superclass.

If we want to make our new class really useful, it should allow any type of collection as the value of its associations, not just sets as in the above example. However, it seems natural that all collections of a particular instance of MultiValueDictionary should be collections of the same kind. As an example, the methods in the example above were based on the assumption that the value is a Set and used Set with:. To make it possible to create new associations with the proper kind of value collection, each MultiValueDictionary must know what kind of collection its values should be. We will keep this information in a new instance variable called collectionClass and define the class as follows:

```
Dictionary variableSubclass: #MultiValueDictionary
	instanceVariableNames: 'collectionClass '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Chapter 9'
```

The next question is which protocols inherited from Dictionary must be redefined. The place to start is, of course, *creation*. Since our new class has a new instance variable which is essential for the operation of some of the other messages, none of the existing Dictionary creation messages is appropriate. We will thus disable all inherited creation messages (^self shouldNotImplement) and define new ones as follows:

```
newOnClass: aClass
"Create new MultiValueDictionary with default capacity on the specified class."
^(self new: 2) collectionClass: aClass

new: size onClass: aClass
"Create new MultiValueDictionary with specified capacity on the specified class."
^(super new: size) collectionClass: aClass
```

where collectionClass: is an instance accessing method of instance variable collectionClass. Time to test! We executed the following expression with *inspect*

```
MultiValueDictionary newOnClass: OrderedCollection
```

and obtained the inspector in Figure 9.7. There is obviously something wrong here - where is our collectionClass instance variable?

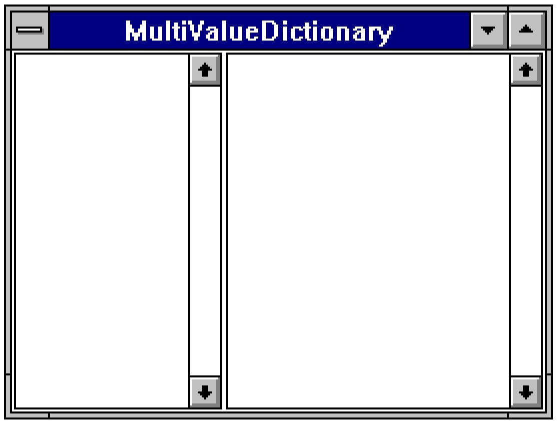


Figure 9.7. Inspector obtained for a MultiValueDictionary.

There are two possible explanations - either the inspector is 'wrong' or our definition is wrong. Since our definition is very simple and does not seem to leave room for error, we check whether Dictionary happens to have its own definition of inspect - and it turns out that it does:

```
inspect
	DictionaryInspector openOn: self
```

where DictionaryInspector is a subclass of ApplicationModel. We will leave writing a new inspector for MultiValueDictionary to you as an exercise and restrict ourselves to redefining printString and using it for testing instead of the inspector. As we know, printString is based on printOn: and we will thus redefine printOn:. We will reuse the definition inherited from Dictionary to display the associations, and append the name of the collectionClass:

```
printOn: aStream
"Print dictionary associations followed by name of value class."
	super printOn: aStream.
	aStream nextPutAll: ', '; print: collectionClass
```

To test, we execute

```
MultiValueDictionary newOnClass: OrderedCollection
```

with *print it* and obtain

```
MultiValueDictionary (), OrderedCollection
```

as expected.

The next protocol that we must re-examine is *accessing* of keys, values, and associations. Dictionary has numerous accessing methods and we will restrict ourselves to at: put:, leaving the rest to you. Our new at: put: must first check whether an association with the specified key already exists and if it does not, it must create one with the appropriate collection as its value. If the key already exists, it will add the new value:

```
at: key put: anObject
"If key already exists, add anObject to its value collection, otherwise create new collection with anObject."
^(self includesKey: key)
	ifTrue: [(self at: key) add: anObject]
	ifFalse: [super at: key put: (collectionClass new add: anObject; yourself)]
```

Note that we used super to reuse the definition of at:put: inherited from Dictionary, and yourself to assign the collection as the value. Note also that we returned the argument to follow the general convention used by at:put: methods. To test the new method, we executed

```
((MultiValueDictionary newOnClass: OrderedCollection)
	at: ‘Science’ put: ‘Computer Science’;
	at: ‘Science’ put: ‘Physics’;
	at: ‘Art’ put: ‘Music’;
	at: ‘Art’ put: ‘Drama’) printString;
```

with *print it* and obtained

```
'MultiValueDictionary ("Art"->OrderedCollection ("Music" "Drama") "Science"->OrderedCollection
("Computer Science" "Physics") ), OrderedCollection'
```

Perfect! We will leave the rest to you and conclude by showing how much our new class simplifies the definition of an English-French dictionary. In the original implementation, adding a new pair of words required

```
addEnglish: string1 french: string2
	^dictionary includesKey: string1
		ifTrue: [dictionary at: string1 add: string2]
		ifFalse: [dictionary at: string1 put: (Set new add: string2; yourself)]
```

Assuming that dictionary is a MultiValueDictionary, the new definition is simply

```
addEnglish: string1 french: string2
	^dictionary at: string1 put: string2
```

<u>Exercises</u>

1. Complete the definition of MultiValueDictionary so that its protocols match the protocols of Dictionary. Make sure to redefine only those methods that must be redefined.
2. Create an inspector for MultiValueDictionary. (Hint: An inspector is an application.)

3. When the collection class of a MultiValueDictionary is SortedCollection, we should be able to specify its sort block. To do this, we could modify the definition of MultiValueDictionary or create a subclass of MultiValueDictionary called, for example, SortedMultiValueDictionary. Compare these two approaches and implement the one that you consider better.
4. Compare our definition of MultiValueDictionary with MultiValueDictionary defined as a subclass of Object with instance variables collection and collectionClass.
5. What requirements must a class satisfy so that it can be used as the argument of the new:onClass: message?

9.6 Example - A Two-Way Dictionary

In this section, we will design and implement a computerized dictionary for two languages such as English and French. This application, an example of the use of dictionaries, will be designed according to the following specification.

Specification

The program provides access to two complementary dictionaries using the interface in Figure 9.8. The information kept in the two dictionaries is complementary in that one dictionary can at any time be constructed from the other dictionary and any change made in one dictionary is automatically reflected in the other dictionary.

Each word in each language may have multiple equivalents in the other language. Selecting a word in the list of 'originals' in the user interface displays all its equivalents in the associated list of 'translations' and selections in the two lists of originals are independent. Control over the dictionaries is via pop up menus in the list of 'originals' according to details described below.

The application is opened by executing a Smalltalk opening message which identifies the languages to be used to label the user interface.

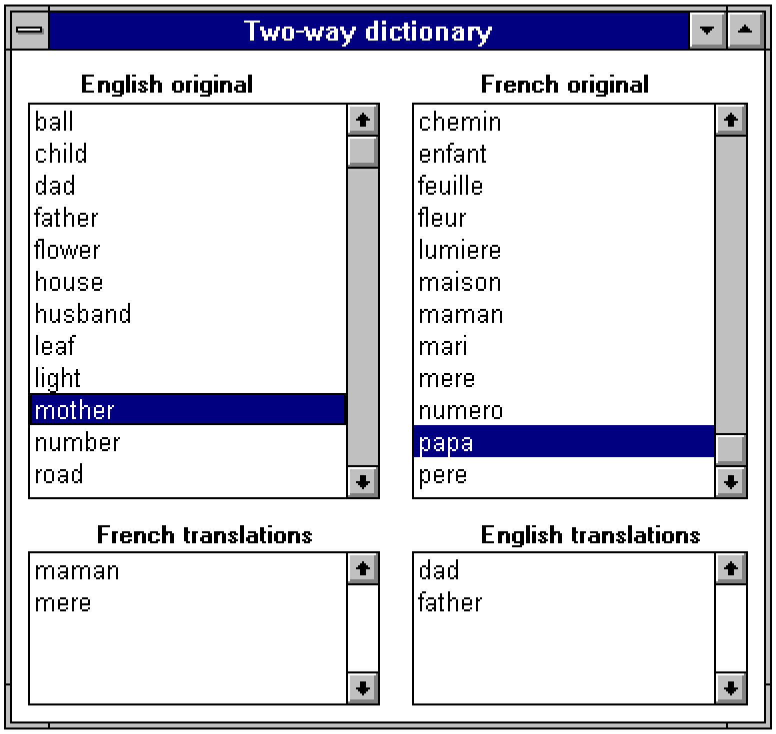


Figure 9.8. Desired interface of the two-way dictionary.

The pop up menu of both lists of original words is as in Figure 9.9. Its commands allow the user to add a new pair of words, add a new meaning to a word selected in the list, delete the selected word, and correct its spelling.

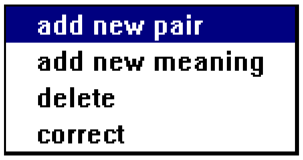


Figure 9.9. Pop up menu of the 'original' list widgets.

Solution: The problem is conceptually simple and we will implement it using an application model class, and a domain model class called TwoWayDictionary. We will leave the application model class as an exercise and concentrate on the domain model class.

Class TwoWayDictionary

Class TwoWayDictionary holds two complementary dictionaries[2], one from language 1 to language 2, the other from language 2 to language 1. Its responsibilities include instance creation, and methods implementing the menu commands in Figure 9.9.

The class will hold the names of the two languages in instance variables lang1 and lang2. The two complementary one-way dictionaries will be stored in instance variables lang1lang2 and lang2lang1. Instance variable lang1lang2 will hold the language 1 $\rightarrow$ language 2 dictionary, lang2lang1 will map language 2 to language 1. The key of each association will be a string and the value will be a sorted collection of strings (a not so accidental use for class MultiValueDictionary from the previous section). Figure 9.10 is an example of a possible state of the two dictionaries; note that all words appear in both dictionaries.

French - English part

'pere'	'dad' 'daddy' 'father'
'mere'	'mother' 'mummy'
'papa'	'dad' 'daddy' 'father'
'maman'	'mother' 'mummy'

English - French part

'dad'	'papa' 'pere'
'father'	'papa' 'pere'
'daddy'	'papa' 'pere'
'mother'	'maman' 'mere'
'mummy'	'maman' 'mere'

Figure 9.10. Typical dictionary components of a two-way dictionary for English and French.

Class TwoWayDictionary is a domain model, therefore not a subclass of ApplicationModel. Since the library does not contain any related class, it will be a subclass of Object. Its definition is

```
Object subclass: #TwoWayDictionary
        instanceVariableNames: 'lang1 lang2 lang1lang2 lang2lang1'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Book'
```

and its comment is

Class TwoWayDictionary implements two complementary dictionaries such as English-French and French-English. In each of the two dictionaries, one key may have any number of meanings (values). Entering a key-value pair into one of the two dictionaries automatically updates the other dictionary.

[2] Note that we are using delegation to implement a complex dictionary with simpler dictionaries.

Instance variables are

<lang1>	- name of the first language, a String
<lang2>	- name of the second language, a String
< lang1lang2 >	- one language dictionary, a MultiValueDictionary on SortedCollection
< lang2lang1>	- the other language dictionary, a MultiValueDictionary on SortedCollection

In both dictionaries, keys are strings (words in one language) and values are sorted collections of strings - equivalents of the key word in the other language.

Having decided these basic features, we will now list the selectors of some of the methods and write the corresponding short methods or their descriptions. The remaining methods are left as an exercise.

Creation will be implemented by method on: lang1String and: lang2String. This class method creates a new TwoWayDictionary on languages named lang1String and lang2String. Example of use:

```
TwoWayDictionary on: ‘English’ and: ‘French’
```

The underlying dictionaries will be two multiway dictionaries using SortedCollection as their value. The creation message is

```
on: lang1String and: lang2String
	^(self new) initialize; lang1: lang1String; lang2: lang2String
```

and the initialization method creates the appropriate MultiValueDictionary objects:

```
initialize
	lang1lang2 := MultiValueDictionary newOnClass: SortedCollection.
	lang2lang1 := MultiValueDictionary newOnClass: SortedCollection
```

We test our definition with the above statement using *inspect* and everything works fine.

Addition will be implemented by instance method addLang1:lang2 which adds a pair of words, one to each, without damaging the information already in the dictionary. Example of use:

```
addLang1: 'dad' lang2: 'pere'
```

The problem is simple because the MultiValueDictionary does all the hard work, and the corresponding definition is

```
addLang1: lang1String lang2: lang2String
"Update both dictionaries."
	lang1lang2 at: lang1String put: lang2String.
	lang2lang1 at: lang2String put: lang1String.
	^lang1String -> lang2String
```

Note that we return the argument as all add methods do. A test with

```
| dict |
dict := TwoWayDictionary on: 'English' and: 'French'.
dict addLang1: 'dad' lang2: 'papa';
	addLang1: 'dad' lang2: 'pere';
	addLang1: 'father' lang2: 'papa';
	yourself
```

confirms that everything works.

Deletion will be implemented by instance method language1Delete: aString (and a symmetric method language2Delete: aString). The method deletes the whole language1 association with key aString, and propagates the change to language2. Since this and similar messages are sent by the pop up menu after the user has made a selection in the user interface, we can assume that aString is present in the set of keys of language1. We can thus use removeKey: and similar messages without worrying about the ifAbsent: part. The algorithm for responding to pop up command *delete* when a word in language 1 is selected is as follows:

1. Obtain all language 2 meanings of the association whose language 1 key is to be deleted.
2. Delete the language1-language2 association.
3. Find all language 2 -> language 1 associations corresponding to the words found in Step 1 and delete the language 1 word from their value collection. (The language 1 word is guaranteed to be there.)

and the corresponding definition is as follows:

```
language1Delete: aString
"Remove language1 word aString from both dictionaries."
	| meanings |
	"Get all language2 meanings for aString."
	meanings := lang1lang2 at: aString.
	"Remove aString from all associations in language 2 containing them."
	meanings do: [:meaning| (lang2lang1 at: meaning) remove: aString].
	"Remove association with key aString from language 1. We know that aString is an existing key."
	lang1lang2 removeKey: aString
```

The definition of language2Delete: is similar and we leave it to you to test that both methods work.

Exercises

1. Complete the definition of class TwoWayDictionary and test it.
2. Create the application model of the dictionary application and test it.
3. Our popup menus are in English. Modify the design to make it possible to translate the commands to another language. The technical term for such an extension is internationalization and VisualWorks provides several tools to simplify it. These tools are not included in the basic library but you can solve the problem without them.

9.7 A Finite State Automaton

As an example of the use of dictionaries, we will now develop a class implementing a finite state automaton (FSA). A finite state automaton is a very important theoretical concept used in several areas of computer science including hardware design, compiler design, and software engineering.

The principle of an FSA can be formulated as follows (Figure 9.11): An FSA consists of a collection of *states*, each performing some *actions*. When an FSA enters a state, it executes all actions associated with this state and makes a transition to a new state depending on the outcome of these actions. Typically, one of the actions executed in each state is a response to an event such as user input. FSA operation starts in a well-defined *initial* state and continues until the FSA reaches one of its *terminal* states where it stops.

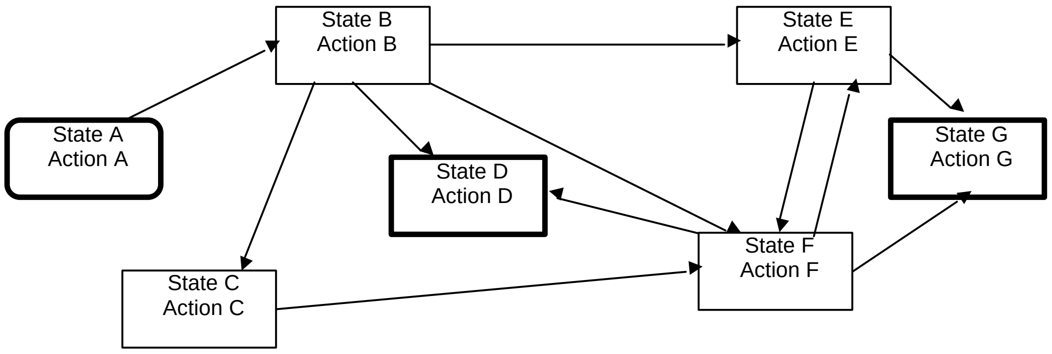


Figure 9.11. An FSA with states represented as rectangles and transitions represented as links between states. The initial state is shown as an oval, final states are thick line rectangles.

Since all FSAs work on the same principle, we can implement an FSA as a class whose instances are FSAs with specific behavior defined at the time of their creation. Each FSA must know

- all its states and their associated actions,
- transitions corresponding to action outcomes,
- the initial state,
- the terminal states,
- the current state

For practical reasons, we will also require that an FSA know what to do if the requested destination state does not exist - because the creation message did not provide a proper FSA description.

To describe state-action pairs, we will use a dictionary whose keys are state objetcs (for example integers or symbols) and whose values are blocks defining the actions executed in this state. Each block must return the next state and thus define a transition.

To illustrate this approach, the state behavior of the FSA depicted in Figure 9.12 can be described by a dictionary containing the following associations which ignore the failure mechanism:

key	value
1	[Transcript show: ‘state 1’. ^(DialogView request: 'Enter name of next state (1 or 2 to continue, 3 or 4 to quit)') asNumber]
2	[Transcript show: 'state 2'. ^(DialogView request: 'Enter name of next state (1 to continue, 3 to quit)') asNumber]
3	[Transcript show: 'state 3 - end'. ^3]
4	[Transcript show: 'state 4 - end’. ^4]

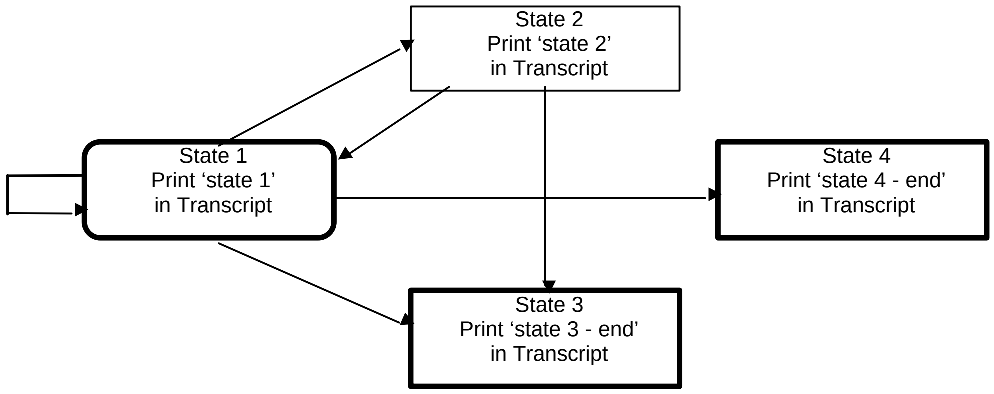


Figure 9.12. Example FSA with transitions obtained from user.

The FSA works in a loop that can be described as follows:

```
current state := initial state.
[current state := value of block corresponding to current state.
currentState is valid ifFalse: [currentState :=  failBlock value]]
current state is an end state] whileFalse.
self executeState               “To execute action in terminal state.”
```

After this analysis, we can now implement class FSA. Its definition is as follows:

```
Object subclass: #FSA
	instanceVariableNames: ‘ currentState endStates initialState stateDictionary failBlock ‘
	classVariableNames: ‘’
	poolDictionaries: ''
	category: 'Book'
```

The comment is

I implement an abstract finite state automaton.

Instance variables:

stateDictionary	<Dictionary>	- keys are states, values are blocks; each block returns a next state
endStates	<Set>	- a Set of terminal states, subset of stateDictionary keys
current state	<anObject>	- current state object - a key in stateDictionary
initialState	<anObject>	- initial state - a key in stateDictionary
failBlock	<aBlockClosure>	- block executed on any attempt to make an illegal transition; must return a next state

As usual, we will start with a class method to create a fully specified FSA. The definition is

onDictionary: aStateDictionary initialState: anInitialObject endStates: aSet failBlock: aBlock
“Create an initialized FSA.”

```
        ^self new
                onDictionary: aStateDictionary
                initialState: anInitialObject
                endStates: aSet
                failBlock: aBlock
```

where the instance method initializing the created FSA is simply

```
onDictionary: aDictionary initialState: anInitialState endStates: aSet failBlock: aBlock
"Initialize FSA."
        stateDictionary := aDictionary.
        initialState := anInitialState.
        currentState := anInitialState.
        endStates := aSet.
        failBlock := aBlock
```

The operation of the FSA follows our algorithms and is defined by the following method:

```
run
"Iterate until FSA enters one of the terminal states."
        currentState := initialState.
        [currentState := self executeState.
        (stateDictionary includesKey: currentState)
                ifFalse: [currentState := failBlock value].
        endStates includes: currentState] whileFalse.
        self executeState "Execute terminal state."
```

Method executeState evaluates the action block of the current block and returns the result, presumably the next state. The definition is as follows:

```
executeState
"Execute action and make transition."
        ^(stateDictionary at: currentState) value
```

We will now test FSA on the example in Figure 9.11. To do this, we must create the complete description of the state graph, use it to create the FSA, and start FSA operation. The operation is controlled by the user via dialogs:

```
| dictionary fsa |
"Create state dictionary."
dictionary := Dictionary new.
dictionary at: '1'
        put:    [Transcript show: 'state 1'; cr.
                Dialog request: 'This is state 1.\Enter name of next state (1 or 2  to continue, 3 or 4 to
                                quit)' withCRs initialAnswer: '1'];
        at: '2'
        put:     [Transcript show: 'state 2'; cr.
                Dialog request: 'This is state 2.\Enter name of next state (1 or 2  to continue, 3 or 4 to
                                quit)' withCRs initialAnswer: '2'];
        at: '3'
        put:    [Transcript show: 'state 3'; cr. Dialog warn: 'This is state 3 - a terminal state'. '3'];
        at: '4'
        put:    [Transcript show: 'state 4'; cr. Dialog warn: 'This is state 4 - a terminal state'. '4'].
"Create FSA."
fsa := FSA      onDictionary: dictionary
                initialState: '1'
                endStates: #('3' '4') asSet
                failBlock: [Dialog request: 'Illegal choice; try again' initialAnswer: ''].
Transcript clear; show: 'Sequence of states:'; cr; cr.
"Run."
```

```
fsa run
```

The program runs correctly.

Exercises

1. Use FSA to simulate the JK flip-flop. The JK flip-flop is a binary storage element that stores 0 or 1 and changes state depending on its current state and two inputs called J and K. Its transitions are shown in Figure 9.13. Assume initial state 0, have user enter the two inputs.
2. Use FSA to implement a recognizer that reads and calculates non-negative integer numbers such as 13, 0, and 383888495000101 digit-by-digit. The digits are entered by the user with a Dialog request: initialAnswer: message. The machine returns the calculated value when it encounters a non-digit character.
3. The classical application of FSAs is to 'recognize' input sequences Use the FSA developed in this section to
   a. recognize legal Smalltalk identifiers (letter followed by a sequence of letters or digits). Input is via a letter-by-letter dialog, output is to the Transcript. The output will show the entered string followed by the word 'legal' or 'illegal'. This kind of operation is performed by compilers.
   b. unlock a door when the user enters the combination '2', '7', '9', '3', '9'.
4. Find how the Smalltalk compiler recognizes identifiers and relate this to Exercise 5.
5. A frequent programming task is reading text, processing it according to some fixed scheme (*filtering* it), and printing the result. Implement a filter defined as follows:
   - Replace every blank line preceded and followed by a line of text with a line of asterisks.
   - Replace every block of two or more blank lines between two text lines with one line of dots.
   - Leave lines with text unchanged.

   As an example of the desired behavior,

   ```
   Line 1.
   Line 2.

   Line 3.



   Line 4.
   ```

   should be replaced with

   ```
   Line 1.
   Line 2.
   ********************
   Line 3.
   ...........................
   Line 4.
   ```

   This behavior can be described as in Figure 9.14. Implement the filter using FSA, extending our definition if necessary.

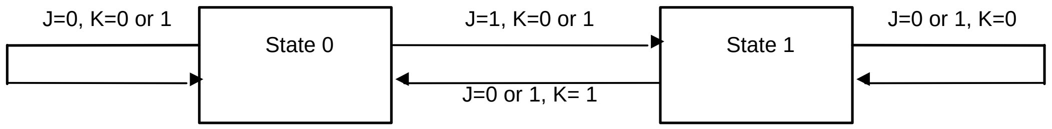


Figure 9.13**.** State transition table of a JK flip-flop.

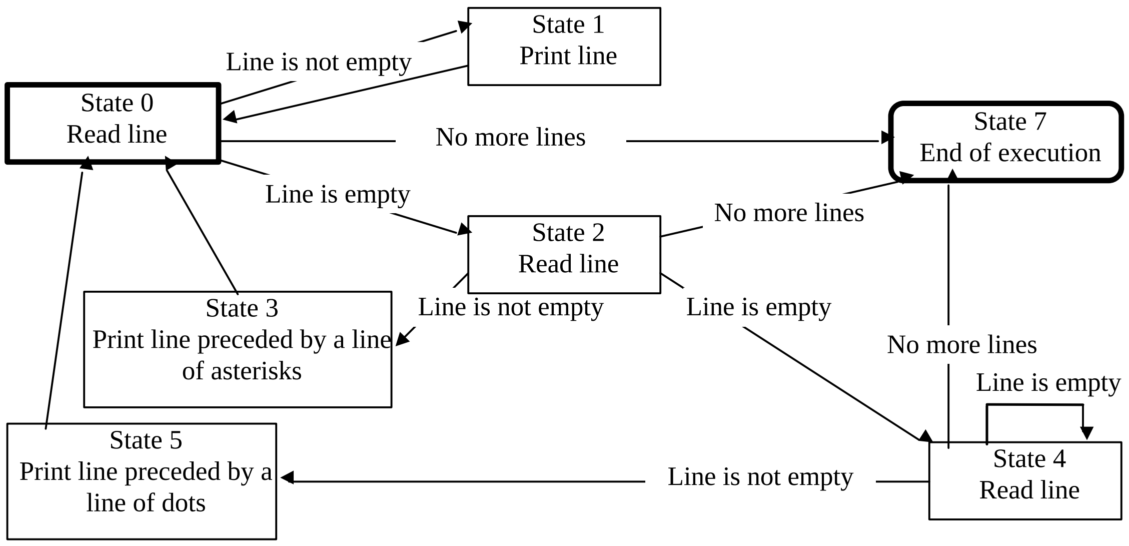


Figure 9.14. FSA for processing lines of text.

Conclusion

Sets, bags, and dictionaries are unordered collections which means is that their protocol does not allow element accessing by an index. Although the external interface is non-sequenceable, internal implementation is based on sequential storage and uses the basicAt: method inherited from Object.

Sets eliminate duplication: No matter how many times a particular object is added to a set, the set contains only one occurrence and if this occurrence is deleted no other copies of the element remain in the set. One of the main uses of sets is to eliminate duplication (often by the conversion message asSet). Testing whether an element is already in the set is based on hashing - calculation of an integer characterizing the object. A subclass of Set called IdentitySet uses identity hash for comparison to eliminate the possibility of changed hash values during an object's lifetime. If the hash function is well designed and the set is not too full, hashing allows very fast element access and this is another reason while sets and their subclasses are very popular.

Bags keep track of the number of occurrences of their elements both during addition and during removal. They are little used but their ability to keep the count of the number of copies of individual elements while eliminating their storage is useful.

Dictionaries are sets whose elements are associations, in other words pairs of key-value objects. Because of their special structure, Dictionary protocols contain a number of specialized messages and even the messages shared with other collections sometimes have specialized behaviors. Uniqueness of dictionary elements (the essential property of sets) is implemented by comparison of keys. A frequently used subclass of Dictionary called IdentityDictionary uses identity hashing and equivalence instead of hashing and equality for comparison. A very important subclass of Dictionary is SystemDictionary which holds information about the structure of the system and its shared objects. Its only instance is Smalltalk.

When using an object, it is often essential to decide whether we want to use the object itself or its copy. In many cases, we must use a copy because the original must not be affected by subsequent operations. A copy may be shallow (new object which shares instance variables with the original), deep (completely separate from the original), or intermediate, obtained in a special way. When using a shallow copy, remember that state changes of the components of the original affect the copy and vice versa.

The difficulty with deep copy is that it is not easy to implement when the copied objects have circular structure. In VisualWorks, copy is implemented as a combination of shallow copy and a post-copy operation which makes it possible to implement any form of copying be redefining the postCopy method. The default behavior is equivalent to shallow copy and it is used by most classes.

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.

Association, *Bag*, **Dictionary**, IdentityDictionary, IdentitySet, **Set**, SystemDictionary.

Terms introduced in this chapter

finite state automaton - an abstraction of a machine mechanically changing its state in response to actions associated with states
filter - a process that mechanically transforms an input (usually text) into a new form
hash - an integer characterizing an object; used, for example, to place or find an object in a set efficiently; depends on object's state and may change during its lifetime
deep copy - a copy completely disjoint from the original
identity hash – hash that does not change hash value during an object's lifetime
post copy - method executed by the copy method after a shallow copy is made; allows customization of copying mechanism
shallow copy - a copy in which the top level object is new but its components are shared with the original; the basis of the copy operation in Smalltalk