Chapter 8 - More Sequenceable Collections, List widgets

Overview

In this chapter, we conclude our exploration of sequenceable collections with OrderedCollection, SortedCollection, List, String, Symbol, and Text. We then present single- and multiple-selection lists. Unordered collections are the subject of the next chapter, and additional widgets including Dataset, Subcanvas, Notebook, Dialog Window, and Menus are covered in Appendix 2.

8.1 Class OrderedCollection

Class OrderedCollection is a very frequently used subclass of SequenceableCollection. Its instances have the following properties:

- Elements are indexed starting with 1 (as in all sequenceable collection).
- Size (number of elements in collection) may be smaller than capacity (number of slots).
- Capacity automatically grows when size = capacity and a new element is added.
- Elements can be removed, changing size but not capacity.
- Elements are usually accessed by enumeration, at the start, or the end rather than by index.

Ordered collections are used mainly to gather elements whose number is not known beforehand or whose number changes. One example of both situations is a program that collects labels for a list widget when the number of labels is initially unknown. Two common situations of the second kind are stacks and queues. Quoting from the comment of OrderedCollection 'a stack is a sequential list for which all additions and deletions are made at one end of the list; a queue is a sequential list for which all additions are made at one end, but all deletions are made from the other end'. You will immediately recognize that a pile of stacked trays in a cafeteria is an example of a stack, and that a line of customers in a bank is an example of a queue. Stacks and queues are very important in computer applications and we will deal with them in more detail in Chapter 11.

After this general introduction, we will now briefly examine OrderedCollection protocols and then give several examples of their use.

Creation

Ordered collections can be created in the same ways as arrays except that there is no concept of a literal OrderedCollection. In practice, ordered collections are most often created with new (default capacity 5) because the eventual maximum size of the collection is usually unknown. When the approximate size is known beforehand, one should always use new: to eliminate time-consuming growing. When a new OrderedCollection is created, it is allocated a number of slots equal to the requested capacity but its size is 0 and its response to isEmpty is true.

Ordered collections are also often created by the conversion message asOrderedCollection, and the withAll: aCollection message is occasionally used with the same effect.

Accessing

Although elements of OrderedCollection may be accessed by at: and at:put:, they are mostly accessed by enumeration, at the start, at the end, or by locating the desired element by its value. Some of the messages used for accessing are

first	"Returns the first element without changing the collection."
last	"Returns the last element without changing the collection."
addFirst: anObject	"Makes anObject the first element, shifts rest to the right, grows if necessary."

addLast: anObject	"Adds anObject at end, grows if necessary."
add: anObject	"Same as addLast: anObject."
add: newObject after: oldObject	"Inserts newObject after the first occurrence of oldObject."
removeFirst	"Removes the first element."
removeLast	"Removes the last element."
remove: anObject	"Removes the first occurrence of anObject."
removeAllSuchThat: aBlock	"Removes all elements that satisfy the block."

Note that adding an element at the beginning of an OrderedCollection or inside it changes the index of all the remaining elements. As the collection evolves, the index of an element may thus change and this is one of the reasons why at: and at:put: messages are not much used with ordered collections.

To understand ordered collections and how they differ from arrays, it is useful to know how they are implemented. When an element is removed, it is replaced with nil and the OrderedCollection adjusts its instance variables firstIndex and lastIndex which it uses to keep track of its first and last *valid* elements (Figure 8.1). This means that the concept of a position and an index are not equivalent and to deal with this, OrderedCollection redefines accessing messages at: and at:put: on the basis of firstIndex and lastIndex.

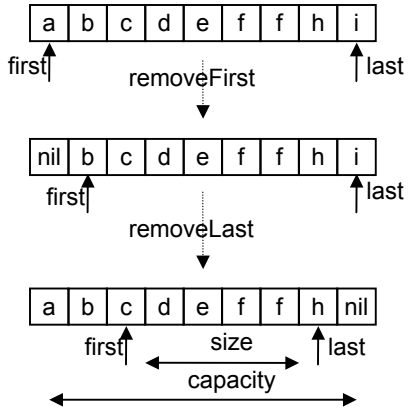


Figure 8.1. Operation of removeFirst and removeLast.

Here is the definition of removeFirst:

```
removeFirst
"Remove the first element of the receiver.  If the receiver is empty, provide an error notification."
        | firstObject |
        self emptyCheck.
        firstObject := self basicAt: firstIndex.
        self basicAt: firstIndex put: nil.
        firstIndex := firstIndex + 1.
        ^firstObject
```

Message emptyCheck first tests whether the collection is empty. If it is, emptyCheck causes a *non-proceedable error*, opening an Exception notifier with the *Proceed* button disabled. Clicking *Terminate* exits the execution of the method and avoids getting into further trouble by attempting to access an object at a non-existent index. The *Debug* button remains enabled and you can proceed in the Debugger - only to run into other error conditions.

Note that removeFirst does not use the at: message, but rather basicAt:. The reason for this is that OrderedCollection defines its own at: and this definition is inappropriate in this context. What we want to use is the definition that works on the basis of the 'regular' index rather than firstIndex and basicAt: allows us to do that. Method basicAt: is defined in Object and is equivalent to the usual at: message.

To see why we need methods such as basicAt:, consider the situation in Figure 8.2 and assume that class C needs the top level definition of at:. If we used super at: we would access the definition in class A, presumably different from that in class Object. The only way to deal with this situation is to define a *synonym* of at: in Object and make a gentlemen's agreement with all Smalltalk programmers that they will

never redefine it in any subclass. Method basicAt: has exactly this role. If everybody plays by the rules, basicAt: thus has a guaranteed single definition that we can always rely on. When you examine the library, you will find that there are quite a few basic... methods.

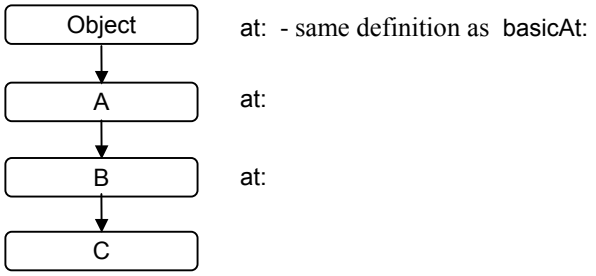


Figure 8.2. The need for basicAt:. Class A redefines at: but class C needs at: from Object.

How much are ordered collections used and how?

To find how much Smalltalk relies on ordered collections, we checked how many methods in the library use them and we found several hundred references to class OrderedCollection. This shows that OrderedCollection is one of the most heavily used classes. Browsing shows that ordered collections are usually created with new and used mainly to gather a collection of elements whose number is initially unknown. The resulting collection is often processed in some way, for example sorted, converted to an array for more efficient processing, or to a set to eliminate multiple copies of duplicated elements.
We also checked how many instances of OrderedCollection our running session has by executing

```
OrderedCollection allInstances size
```

We found over 200 references, a large number that will fluctuate greatly as you run an application[1]. These finding confirm that ordered collections are very popular and that they are a very important part of the Smalltalk environment.

Main lessons learned:

- Ordered collections are indexed but the index is rarely explicitly used for access. Instead, elements are usually added and retrieved at the end or at the start, preserving the order in which elements are added.
- The size of an ordered collection may be smaller than its capacity.
- Adding a new element to an ordered collection automatically grows the capacity if necessary.
- The growth of an ordered collection is relatively time-consuming and the only way to avoid it is to create the ordered collection with capacity equal to the maximum anticipated need or greater.
- Making an ordered collection larger than necessary wastes a negligible amount of memory but making it too small may cost considerable execution time.
- Removing an element from the beginning of an ordered collection changes the index of remaining elements. It does not shrink the capacity.
- A new ordered collection contains slots but no elements.
- Ordered collections are used mainly to collect elements when their exact final number is unknown and as stacks or queues.

Exercises

[1] Method allInstances does not give an accurate picture of all instances of a class because it counts even unreferenced objects that have not yet been garbage-collected.

1. When a new ordered collection is created, it contains slots but no elements. What are these 'slots'? Is something stored in them? (Hint: Check the implementation of at: in OrderedCollection and see if you can examine the unoccupied slots.)
2. Find how to obtain the number of references to a given class. (Hint: The browser knows how to find all references and returns them as a collection.)
3. List at least two useful methods inherited by OrderedCollection from SequenceableCollection and give examples of their use.
4. OrderedCollection and other variable-size collections automatically grow but don't shrink. Define a method to shrink an ordered collection. Comment on the usefulness of such a method.
5. Why is add: in OrderedCollection equivalent to addLast:? (Hint: Which of addFirst and addLast is faster? Why?)
6. Find two other basic... methods and explain their purpose.
7. Write a code fragment to create an OrderedCollection with capacity 5 and three elements. Add four new elements, then delete one at the beginning and one at the end. Print all slots of the collection before and after each step. (Hint: Use basicAt:.)
8. Since removeFirst moves the firstIndex pointer, the 'removed' element cannot be accessed so why bother changing it to nil?
9. Are the slots vacated at the beginning of an OrderedCollection by removeFirst ever reclaimed? If so, when and how?

8.2 Several examples with ordered collections

In this section, we present several examples using ordered collections. They include code from the library, a brief code fragment, and a small application using two classes.

Example 1: Selecting book entries from a library catalog

Problem: Assume that we have a library catalog application with information about books. All book information is stored in a collection accessible via variable books. The collection is an OrderedCollection because it must be able to grow and shrink, and its elements are instances of class Book. Class Book has instance variables title, author, publisher, year, and number (Figure 8.3) and accessing methods for accessing them. A typical task for the application is to extract all books published by 'Addison-Wesley' in 1997.

	title	author	publisher	year	number
book 1	'The personal computer ...'	'Sargent ...'	'Addison-Wesley'	1995	11
book 2	'Principles of computer science'	'Schaffer'	'Prentice-Hall'	1988	22
book 3	'Programmer's guide to the IBM PC'	'Norton'	'Microsoft Press'	1985	13
book 4	'Programming languages'	'Pratt'	'Prentice-Hall'	1965	54
book 5	'Data structures ...'	'Kruse'	'Prentice-Hall'	1994	52
book 6	'Introduction to AI'	'Charniak ...'	'Addison-Wesley'	1986	36
book 7	etc.	etc.	etc.	etc.	etc.

Figure 8.3. Structure of ordered collection books.

Solution: To solve this problem, the code must check the publisher and year components of each instance of Book and select all books that satisfy the given condition. This calculation can be performed very easily with select: as follows:

```
selectedBooks := books select: [:book| (book publisher = 'Prentice Hall') and: [book year = 1997]]
```

Example 2: Conversion to OrderedCollection - how does it work?

Class Collection contains many conversion messages and one of them - asOrderedCollection - converts its receiver - an arbitrary collection - to an OrderedCollection. Here is its definition :

```
asOrderedCollection
        | anOrderedCollection |
        anOrderedCollection := OrderedCollection new: self size.
        self do: [:each | anOrderedCollection addLast: each].
        ^anOrderedCollection
```

The method first creates a new ordered collection whose capacity is equal to the size of the receiver collection. It then adds all elements of the receiver at the end of the new collection and returns the new collection. The original collection is not affected but shares all its elements with the returned ordered collection.

Example 3: A simple shopping minder application

Problem: This problem is a test of an idea that will be explored in more detail in the next chapter. In the present version, our task is to write a program allowing the user to enter a list of store items and prices and display it in the Transcript one item per line followed by the total price. Items are sorted alphabetically by name and all items that cost more than $10 are listed behind the total. An example of the desired output is

```
cat food  price: 24 dollars   54 cents
flour     price: 4 dollars   75 cents
sugar     price: 3 dollars   15 cents

Total price:        32 dollars, 44 cents

Expensive items:
cat food  price: 24 dollars   54 cents
```

Solution: We will define two classes, one to hold item information, and the other the future application model. Since the problem is quite simple, we can write class specifications without constructing scenarios and performing other preliminary work:

ShoppingMinder: Holds a collection of items, knows how to obtain item information from user and how to display summary information in the Transcript.
Superclass: Object. In the final implementation, Shopping minder will be a subclass of ApplicationModel.
Components:

- items - OrderedCollection of Item objects – the number of items is unpredictable

Contracts and behaviors:	*Collaborators*
• Creation	
• Create and initialize items to a reasonably large OrderedCollection (capacity 20)	
• Accessing	
• Get item information from user via Dialog and create an Item	Item
• Adding	
• Add new item to items	
• Arithmetic	
• Calculate totals	
• Printing	
• Display items	Item
• Display expensive items	Item
• Display total	
• Execution	
• Execute shopping process including entry of items and output of results	

Item: Holds name and price of item.
Superclass: Object
Components:

- name - name of item (a String)

- price - dollars and cents (a Currency)

Contracts and behaviors: *Collaborators*

- Creation
  - Create instance with given name and price
- Accessing
  - Get and set methods
- Printing
  - Return name and price as a suitably formatted String

On the basis of this plan, it is now easy to write the implmentation. We will leave most of the work to you and confine ourselves to a few elements.

Class ShoppingMinder

We will start at the execution level. To run the application, the user will execute

```
ShoppingMinder new execute
```

This method creates a new instance of ShoppingMinder and initializes its items instance variable. Its definition is

```
new
	^super new initialize
```

and the initialize method is

```
initialize
"Initialize items to a suitably sized ordered collection."
	items := OrderedCollection new: 20.
```

Method execute obtains items from the user and displays the results. Its definition is simply

```
execute
"Get items from the user and display the results."
	[self addItem] whileTrue: [self addItem].
	self displayItems.
	self displayTotal.
	self displayExpensiveItems
```

Note that the formulation implies that addItem returns true when an item is added, and false when the user indicates that there are no more items. Its definition is as follows:

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

where name:price: is the creation method for Item, to be defined later.

```
displayTotal
"Add prices of all items and display the total in Transcript."
	Transcript cr; show: 'Total price: '; tab; show:
```

```
(items inject: (Currency cents: 0)
        into: [:total :item | total + item price]) displayString
```

Note the use of inject:into: with Currency objects, and the displayString message - you may have expected printString. displayString is one of the two standard methods that convert an object to a String, the other being the familiar printString. As you know, printString is used mainly to display information during debugging and in the inspector ; in other words, it is intended primarily for the *code developer*. Method displayString, on the other hand, is intended for the *user interface*, for example, when displaying an object in a list widget. Its essence is the same as that of printString (it creates a String or Text describing an object) but the contents may and the form (for example the font) may be different.

The default definition of displayString in Object is simply printString and most classes don't redefine this definition.

To follow the spirit of the distinction of the two messages, we will expect them to produce different strings. Message printString will produce text such as

'Item: Apple, a Currency: 13 dollars, 27 cents'

taking advantage of printString in Currency, whereas displayString will produce

sugar price: 3 dollars 15 cents

Both methods will be defined in class Item. The definition of displayItems is

```
displayItems
"Display alphabetically all items selected by the customer along with their prices. Use alphabetical order."
    Transcript clear.
    (items asSet asSortedCollection: [:item1 :item2 | item1 name < item2 name])
        do: [:item | Transcript show: item displayString; cr]
```

The notable thing about this definition is that the conversion to a sorted collection uses a special sort block instead of the simple asSortedCollection message. The definition of displayExpensiveItems is

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

We leave the other methods in ShoppingMinder to you as an exercise.

Class Item

The name:price: creation method that we used in addItem expects a String and a Number which it converts to a Currency object. Its definition is as follows:

```
name: aString price: aNumber
"Create new instance using aString for name and a aFixedPoint for currency."
    | dollars cents |
    dollars := aNumber asInteger.                    "Get the integer part of the price."
    cents := ((aNumber - dollars) * 100) asInteger.  "Get the fractional part."
    ^self new
        name: aString;
        price: (Currency dollars: dollars cents: cents)
```

where we assume that accessing messages name: and price: have already been defined. The definition of displayString is as follows:

```
displayString
```

```
| tabString |
tabString := String with: Character tab.
^name , tabString , 'price: ' , price dollars printString , ' dollars   ' ,
                    (price cents rem: 100) printString , ' cents'
```

We leave the rest, including the printString method, as an exercise. By the way, we are not happy with the strategy that we used to implement our application. It would have been better to implement the domain class first and test it, and then go to the application model and test it with the already existing domain class. Most developers proceed in this order.

What happens when a collection grows?

When you add a new element to a collection filled to capacity, the following steps take place:

1. A new collection of the same kind as the receiver is created with double the capacity of the original.
2. All elements of the receiver are copied to the new collection. (In reality, only the pointers are copied, of course.)
3. All references from the system to the receiver are switched to point to the new collection (Figure 8.4). This step uses message become: which is understood by all objects and may be time consuming.
4. Since there are now no references to the old collection, it is garbage collected[2].

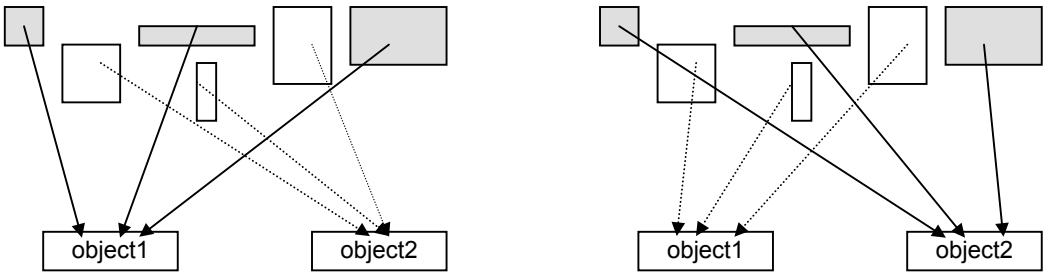


Figure 8.4. Message object1 become: object2 switches references.

The heart of the process is the following method:

```
changeCapacityTo: newCapacity
"Change the capacity of the collection to be newCapacity."
        | newSelf |
        “Make an empty copy whose capacity is obtained elsewhere.”
        newSelf := self copyEmpty: newCapacity.
        “Initialize value of firstIndex.”
        newSelf setIndicesFrom: 1.
        “Add all elements of existing collection fast - with no checks.”
        firstIndex to: lastIndex do:
                [ :index | newSelf addLastNoCheck: (self basicAt: index)].
        “Switch references.”
        self become: newSelf
```

The nature of collections - variable sized classes

The definition of OrderedCollection in the browser

```
SequenceableCollection variableSubclass: #OrderedCollection
        instanceVariableNames: 'firstIndex lastIndex '
```

[2] See Appendix 8 for a discussion of garbage collection.

```
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Collections-Sequenceable'
```

uses a creation message different from most other classes such as

```
Object subclass: #Document
        instanceVariableNames: 'entities currentMargins '
        classVariableNames: 'DefaultMargins DefaultTabStops '
        poolDictionaries: ''
        category: 'System-Printing'
```

You will find that several other collection class definitions use the keyword *variableSubclass:* instead of *subclass:*. This is because Smalltalk recognizes essentially two types of classes - those whose components are accessed by an index, and those that don't have index-based access. Both types of classes may or may not have named instance variables. The standard browser template for declaring a new class assumes the second kind of class because most classes have only named variables, and if you want to define a class with indexed variables (a *variable* class), you must modify the definition template in the browser.

A class such as OrderedCollection thus has the structure shown in Figure 8.5 and contains *unnamed* index elements and (possibly) *named* instance variables. Named instance variables are accessed via an accessing method, but indexed elements are accessed using a mechanism such as self at: index and self at: index put: object. The fact that there is an indexed component can only be recognized from the definition of the class because the indexed components don't have any variable name associated with them.

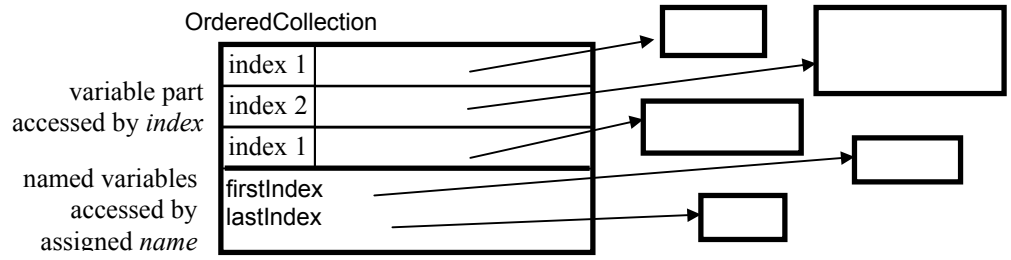


Figure 8.5. Indexed elements are accessed by index as in self at: index rather than by name.

Main lessons learned:

- VisualWorks has two forms of messages converting an object to a String. Message printString should be used for debugging, displayString is for output to the user interface.
- Smalltalk recognizes two main types of classes - those that have index-accessed elements and those that don't. The standard browser template for creating a new class creates a class without access by index unless the superclass is a variable size class itself. To create a new variable-size class that is not a subclass of a variable-size class, modify the class creation template.

Exercises

1. Complete the shopping minder example. Add printString to both new classes for testing.
2. Write a code fragment to request a list of labels from the user and use them in a multiple choice dialog. Repeat with both forms of Dialog's multiple choice messages.
3. Read Appendix 3 and write an expression to find all variable class in the library.
4. ClassOrderedCollection defines method removeAllSuchThat: aBlock. Compare it with reject:.

8.3 Ordered collections as the basis of dependency

In this section, we will take a closer look at an important application of ordered collection - the dependency mechanism - and illustrate it on an example. Let us start by recapitulating what we know about dependency.

Any object may have any number of dependents. The basic dependency mechanism is defined in class Object and redefined in class Model. Model's implementation is preferable because it holds dependents in instance variable called dependents (an OrderedCollection) rather than a global variable as Object does. As a consequence, when a Model object ceases to exist, its dependents are no longer referenced and may be garbage collected, whereas Object's dependents remain referenced until explicitly removed from the global variable that references them. If Object's dependents are forgotten, they will thus stay in the image forever.

The protocol for adding new dependents and removing existing ones is defined in class Object and only the accessing method for the dependents collection is redefined in Model[3]. The dependency mechanism is based on three changed methods called changed, changed:, and changed:with:. All three are defined in class Object, exhibit the same basic behavior (Figure 8.6), and differ only in their number of arguments.

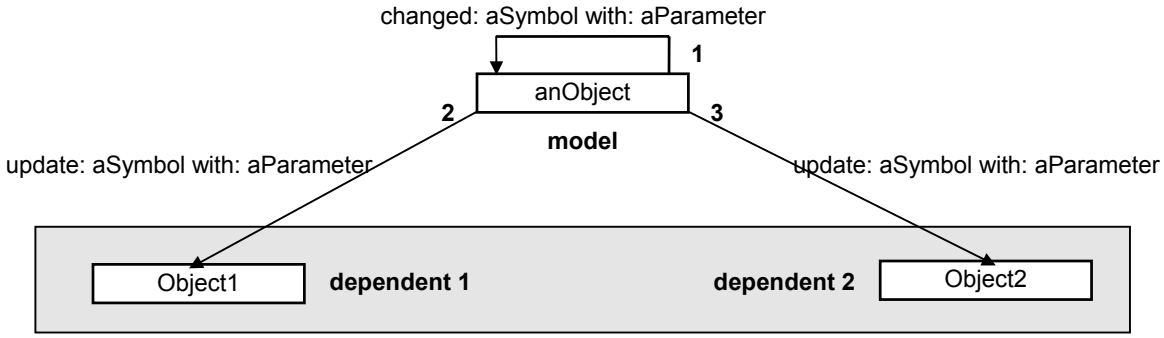


Figure 8.6. A model with two dependents and its response to self changed:with:. Numbers indicate order of message execution.

Dependency works as follows: When the model object needs to notify its dependents that it has changed in a way that might affect them, it sends *itself* an appropriate changed message as in

```
self changed
```

or

```
self changed: #value
```

or

```
self changed: #value with: 153
```

In response to the last message, for example, the definition of changed messages in Object then sends update: #value with: 153 to each of its dependents:

changed: anAspectSymbol with: aParameter
"The receiver changed. The change is denoted by the argument anAspectSymbol. Usually the argument is a Symbol that is part of the dependent's change protocol, that is, some aspect of the object's behavior, and aParameter is additional information. Inform all of the dependents."

```
self myDependents update: anAspectSymbol with: aParameter from: self
```

[3] The ease with which Model can redefine its inherited behavior by redefining only one accessing method illustrates the advantage of using accessor methods instead of direct access to instance and class variables.

where myDependents returns the dependents collection. Each dependent then responds according to its own definition of update:with:. Typically, the definition of update:with: examines the argument anAspectSymbol, decides whether it cares to do anything for the kind of change indicated by the argument and if it does, it uses aParameter to execute the appropriate response. The response of individual dependents may range from identical for each dependent to different for each dependent, to none for some or all dependents. A default implementation of all update messages is defined in Object and does not do anything. As a consequence, if a dependent does not have an update method, it simply does not do anything when the model changes (unless update is defined in a superclass).

The implementation of changed messages is interesting. The simplest one - the changed method with no arguments - is as follows:

```
changed
        self changed: nil
```

This definition thus does not send any update message and depends on changed: to do so. The changed: message is defined as

```
changed: anAspectSymbol
        self changed: anAspectSymbol with: nil
```

It again passes the real work to a more complete version of changed which is defined as follows:

```
changed: anAspectSymbol with: aParameter
        self myDependents update: anAspectSymbol with: aParameter from: self
```

This method finally sends an update message to the dependents. This update message contains an extra argument - the sender of the message. This is essential in some situations because it allows the dependent to ask the model for additional information.

We conclude that no matter which changed message the model sends, the result is always the same update message, namely update:with:from:. The question now arises how a dependent that does not define this rich version of update responds. As an example, how does a dependent who defines only update: aSymbol respond to update:with:from:?

The answer is obvious: If an object gets a message that it does not understand, Smalltalk looks for the method in the method dictionary of its superclass. If this class does not contain the definition, it searches the next superclass, and so on until it reaches class Object. To find how the update mechanism works, we must thus look at the definition of update: in Object where we find, for example,

```
update: anAspectSymbol with: aParameter from: aSender
"Receive a change notice from an object, denoted by aSender, of whom the receiver is a dependent. The
argument anAspectSymbol is typically a Symbol that indicates what change has occurred and aParameter is
additional information. The default behavior defined here is to do nothing;  a subclass might want to change
itself in some way. Note that this implementation assumes that the object does not respond to this protocol
and an attempt is made to try a simpler message."
        ^self update: anAspectSymbol with: aParameter
```

This shows that if the dependent does not understand the 'rich' update message, Object downgrades the message to a simpler update and asks the dependent to execute it. If the dependent does not have that definition, update again goes up the superclass chain and if it reaches Object, it is downgraded again, and so on, until either a suitable update definition is found in the receiver's class or its superclass or until execution eventually passes to update in Object - which does nothing.

We have already seen the most conspicuous use of dependency in VisualWorks in its implementation of the user interface: A widget is a dependent of its ValueHolder (a subclass of Model) and when the value holder gets a value: message, it informs its dependent widget via changed and update, and the widget then gets the necessary information from the ValueHolder and redraws itself.

Although UI implementation is the most prominent use of dependency, there are other uses as we will illustrate in the following example. Note that this example (and all other uses of dependency) could be implemented without the dependency mechanism. The use of dependency just makes the implementation more logical and neater because it gives the model object a uniform way to deal with all its dependents, whatever their nature may be.

Example. Dependence of object state on weather

The most common use of dependency is to send changed: aValue and use the value in the update: method of the dependent to decide exactly what to do. We will now present a very simple example that uses this form of dependency.

Problem: Objects on Earth respond to weather in various ways. When the sun is out, people sit in the shade, talk, and smile, birds sing, and stones get warm. When it rains, people get umbrellas, stones get wet, and birds don't do anything. When it snows, people ski, and birds and stones disappear. We will now write a program that asks the user to select current weather (Figure 8.7) and prints the state of people, birds, and stones in the Transcript. The program terminates when the user clicks *Stop*.

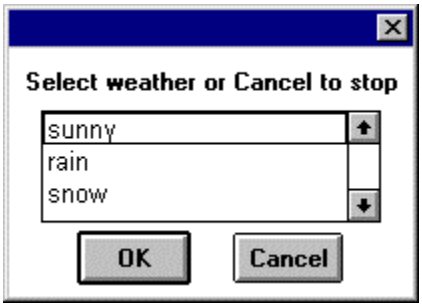


Figure 8.7. User interface for text example.

Solution. We will define a class for each part of the problem - Weather, Person, Bird, and Stone. Class Weather will be responsible for running the program and displaying the user interface, Person, Bird, and Stone will be its dependents and will respond to weather changes by printing their state to the Transcript.

Class Person

The only responsibility of Person is to respond to an update message. This message must specify the new weather, a Symbol, one of #sunny, #rainy, or #snow. This is the 'aspect' of change and we will thus use the update: anAspect message. According to the specification, the definition is simply

```
update: aSymbol
"Display appropriate string according to weather described by aSymbol."
        (aSymbol == #sunny)
                ifTrue: [^Transcript show: 'People talk and sit in the shade.' ; cr].
        (aSymbol == #rainy)
                ifTrue: [^Transcript show:'People get umbrellas.'; cr].
        (aSymbol == #snow)          "This test is not really necessary."
                fTrue: [^Transcript show:'People go skiing.'; cr].
```

No other methods need to be defined because instances can be created with the new message.

Class Bird

This class is essentially identical to Person but the details of update: are different because birds behave differently:

```
update: aSymbol
"Display appropriate string according to weather described by aSymbol."
        (aSymbol == #sunny)
                ifTrue: [^Transcript show: 'Birds sing.' ; cr].
        (aSymbol == #snow)
                fTrue: [^Transcript show:'Birds disappear.'; cr].
```

Note that Bird does not respond when the value of the update: argument is #rainy. No other methods need to be defined because instances can be created with the new message.

Class Stone

Similar to the two classes above and left as an exercise.

Class Weather.

Since Weather has dependents it will be a subclass of Model. Since it is also an application model, we will make it a subclass of ApplicationModel (which is a subclass of Model). Its initialization method must assign new instances of Person, Bird, and Stone as it dependents:

```
initialize
"Create new instance and add Person, Bird, and Stone as it dependents."
self dependents
        addDependent: Person new;
        addDependent: Bird new;
        addDependent: Stone new.
self run  "We don't know how to create lists yet so we will use this intermediate implementation for now."
```

and the instance method run is

```
run
"Repeatedly ask user to select weather and stop when the selectoion is Cancel. Notify dependents."
repeat [selection := Dialog choose: 'Select weather or Cancel to stop'
                                fromList: #('sunny' 'rain' 'snow')
                                values: #(#sunny #rainy #snow)
                                lines: 3
                                cancel: [].
        selection isNil ifTrue: [^self].
        self changed: selection]
```

Now assume that we decided to add another object, such as a river. We must define its class and modify the initialize method to add a River dependent, but the handling of changes does not change because the Model does not need to be explicitly aware of its individual dependents and does not care what they are and what are the details of their behavior as long as they know how to respond to an appropriate update message. This is dependency's main claim to fame.

Main lessons learned:

- Dependency is defined by specifying the dependents of a model object.
- The model object is normally a direct or indirect subclass of Model although any object can have dependents via the mechanism defined in Object.
- Dependents in Model subclasses are stored in an ordered collection.
- When the model changes in a way that may affect its dependents, it sends itself one of the changed messages.
- The changed message sends a matching update message to each dependent.
- There are several varieties of changed and update and they differ in the number of keywords.
- The update method defined in the dependent's classes does not have to match the changed message

sent by the model but it normally does.
- Dependency is used mainly by the user interface but there are many other possibilities for its use.
- All problems can be solved without dependency but the use of dependency makes the solution cleaner and more flexible.

Exercises

1. Browse and describe implementation of dependency in class Object.
2. Complete the weather example and test it.
3. Add a river object to the example. When it is sunny, the river gets warm, when it snows, the river freezes. Rain does not have any effect.

8.4. Tennis - another example of dependency

Have you ever watched the spectators at a tennis game? As the ball moves across the court, their heads turn in unison as if enchanted. In this example, we want to simulate this effect. Consider a very simple implementation of the problem, assuming a collection of spectators arranged in a straight line in parallel with the axis of the court and a ball moving along the axis from one end of the court to the other (Figure 8.8).

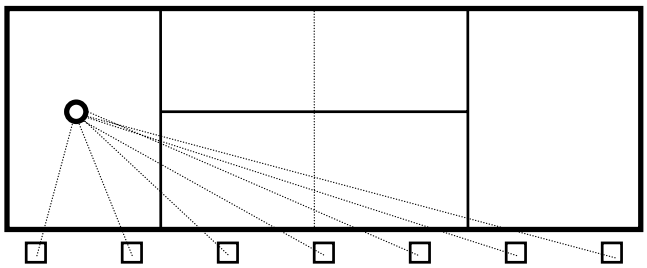


Figure 8.8. Tennis court with spectators following the ball's motion.

In this section, we will not represent the court and the moving ball graphically because we don't know how to do that yet; we refer you to Chapter 12 for information necessary for a graphical implementation. Our implementation will simply print the position of the ball and spectator information to the Transcript in the following form:

```
Move number 1
Ball position: 10 direction: left.
Spectator location: 1 angle: 1.25
Spectator location: 4 angle: 1.11
Spectator location: 7 angle: 0.79

Move number 2
Ball position: 9 direction: left.
Spectator location: 1 angle: 1.21
Spectator location: 4 angle: 1.03
Spectator location: 7 angle: 0.59
```

and so on for each consecutive ball displacement. Angles are expressed in radians.

To make the program more interesting, we will provide the user interface in Figure 8.9 allowing selection of court parameters, the number of spectators, and the number of successive changes of ball positions (*Number of moves*). The location of spectators along the court will be calculated from the assumption that spectators are uniformly distributed.

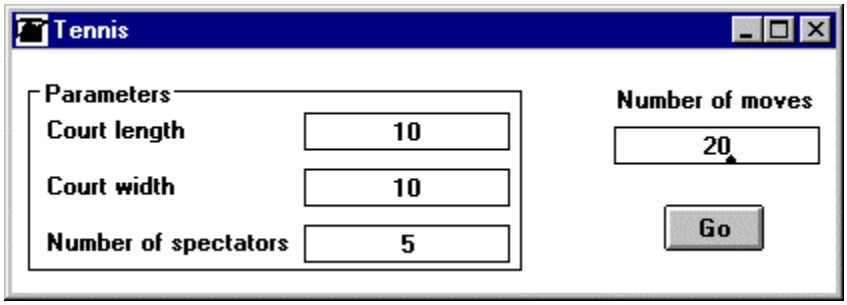


Figure 8.9. User interface for Tennis program with initial default parameters.

Specification

The statement of the problem is clear and there are only two interesting scenarios - opening the application and clicking *Go*.

Scenario 1: Starting the program
1. *User* executes expression such as Tennis open
2. *Program* opens the window in Figure 8.7.

Scenario 2: Starting simulation by clicking *Go*.
1. User clicks *Go*.
2. Program places the ball at the right end of the court and starts simulation, performing a series of ball displacements accompanied by adjustments of spectators' viewing angles.

Preliminary design

The specification suggests three classes - Tennis, Ball, and Spectator. Their initial description is as follows:

Tennis: I maintain simulation parameters, perform the simulation, and display the user interface.

Responsibilities	Collaborators
• interface specs - define UI window	
• creation - open user interface, initialize default values	Ball, Spectator
• action - response to *Go*. Reset ball and perform the required number of steps, printing simulation results in Transcript	Ball, Spectator

Ball: I represent the tennis ball moving across the tennis court.

Responsibilities	Collaborators
• creation - create a ball with initial position and direction	
• moving - calculate new position and notify spectators	Spectator

Spectator: I represent a spectator who turns his or her head while watching the moving ball.

Responsibilities	Collaborators
• creation - create a spectator at given location	
• watching - calculate new head position	Ball

We will now check whether these three candidate classes can support our two scenarios and how.

Scenario 1: Starting the program
1. *User* executes expression Tennis open
2. Tennis initializes user interface parameters and opens the interface in Figure 8.7.

Scenario 2: Starting simulation by clicking *Go*.

1. User clicks *Go*.
2. Tennis creates a new Ball and a new collection of Spectator objects according to the values of user interface parameters.
3. Tennis executes simulation by asking the ball to move the specified number of times, displaying the required output for each step. Ball moves by incrementing or decrementing its position depending on its current direction of motion, notifies all spectators of its new position, and checks whether it has reached the end of the court and should change direction. Each Spectator calculates its viewing angle from its location and the current position of the ball.

We conclude that our three classes should be able to handle the problem.

Design refinement

Let's start with a preliminary discussion of the place of the three classes in the class hierarchy. Tennis is a subclass of ApplicationModel because it is responsible for the user interface, Ball and Spectator have no class relatives and this suggests superclass Object.

As the next step, we will take a closer look at class descriptions starting with Ball and Spectator. Clearly, the state of Spectator depends directly on the position of the ball: Whenever the ball moves, every spectator is notified and responds by changing its viewing angle. Spectators are thus dependents of the ball. This has several consequences. First, Ball should thus be a subclass of Model. Second, the action responsibility of Tennis now requires only the collaboration of Ball, because Ball will take care of Spectator. With this insight, we can now rewrite the class descriptions as shown below. Note that we changed the descriptions of some of the responsibilities to reflect our better understanding of the situation.

Tennis: I perform the simulation and display the user interface.
Concrete class.
Superclass: ApplicationModel
Components: ball (Ball), spectators (Array), and aspect variables on integer values courtWidth, courtLength, numberOfMoves, and numberOfSpectators

Responsibilities	Collaborators
• interface specs - define UI window	
• creation - open user interface, initialize default values	
• initialize - initialize aspect variables, create a Ball	Ball
• action - response to *Go*.	
• go - reset everything, create spectators and make them dependents of ball, perform the required number of steps, printing simulation steps in Transcript	Ball, Spectator
• printing - supply your description	
printOn: - return descriptive string according to output specification	

Ball: I represent the tennis ball moving back and forth across the tennis court.
Concrete class.
Superclass: Model
Components: court (Tennis), direction (Symbol) direction of motion - #left or #right, position (Integer) - rightmost position is 0, increases towards left

Responsibilities	Collaborators
• creation - create a ball	
• onCourt: aTennis - Ball needs access to Tennis to get simulation parameters at the start of each run	
• moving - move the ball by one step	
• move - increment or decrement position depending on direction, notify dependent spectators, check for end of court and change direction if necessary	Spectator
• printing - supply your description	
• printOn: - return descriptive string according to output specification	

Spectator: I represent a spectator turning his or her head while watching the moving ball.
Concrete class.
Superclass: Object
Components: angle (Float) - viewing angle, distance (Integer) distance to ball line (court axis), location (Integer) - location measured from the right end of the court

Responsibilities	Collaborators
• creation - create a spectator	
• distance:location: - create new spectator at given distance from ball line and at given location from right end of court	
• updating - respond to update request from Ball	
• update:with: - calculate new viewing angle from ball position	
• printing - supply your description	
• printOn: - return descriptive string according to output specification	

Figure 8.10 shows the Hierarchy Diagram and the Object Model Diagram.

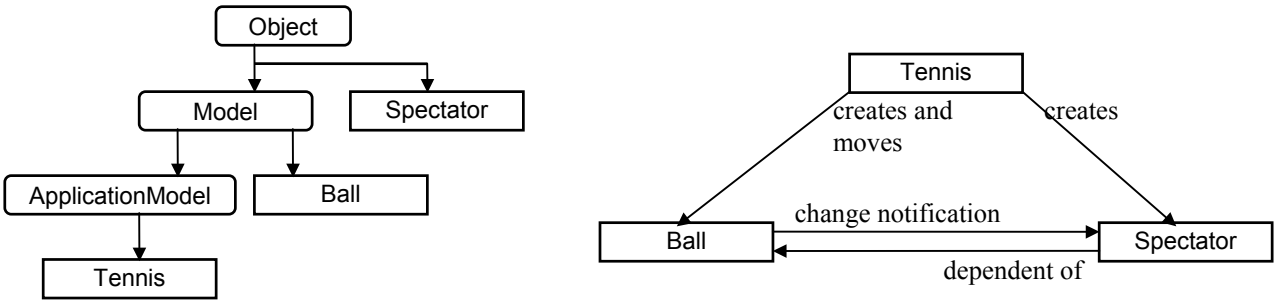


Figure 8.10. Hierarchy Diagram (left) and Object Model Diagram (right) of Tennis.

Implementation

We will now show some selected methods class by class.

Class **Tennis**

The definition of the class is

```
ApplicationModel subclass: #Tennis
        instanceVariableNames: 'ball spectators courtWidth courtLength numberOfMoves
                                numberOfSpectators '
```

```
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Tennis'
```

Initialization defines default values for value holders and creates a Ball:

```
initialize
"Initialize the opening values of simulation parameters, create a ball."
        courtLength := 10 asValue.
        courtWidth := 10 asValue.
        numberOfSpectators := 5 asValue.
        numberOfMoves := 20 asValue.
        ball := Ball for: self                  "Create tennis ball on this court."
```

The basis of the operation of the whole program is response to the *Go* button which is defined as follows:

```
go
"Reinitialize parameters according to current settings, execute the simulation the required number of times,
and display the result."
        self reset.
        Transcript clear.
        1 to: numberOfMoves value
                do: [:number | Transcript cr; cr; show: 'Move number ' , number printString; cr;
                                show: self printString.
                        self ball move]
```

Method reset reads the current settings of user interface value holders and uses them to create spectators and assign them to ball as its dependents:

```
reset
"Remove existing spectators, if any, create equally spaced new spectators and make them dependents of
the ball. Used to update parameters before the start of each simulation run."
        | distance |
        1 to: spectators size do: [:index | self ball removeDependent: (spectators at: index)].
        spectators := Array new: self numberOfSpectators value.
        distance := (courtLength value / spectators size) truncated.
        1 to: spectators size do: [:index | spectators at: index put:
                                (Spectator distance: distance location: 1 + (distance * (index - 1)))].
        spectators do: [:aSpectator | self ball addDependent: aSpectator]
```

Note that we had to remove existing spectators before creating new ones; we assume that simulation parameters have changed. Printing of results uses the printOn: method because this is only a test. The definitive user interface will be implemented when we learn about graphics.

```
printOn: aStream
"Display ball and spectator information."
        aStream nextPutAll: self ball printString , '. '; cr.
        self ball dependents do: [:aSpectator | aStream nextPutAll: aSpectator printString , ' '; cr]
```

The operation of this method depends on printOn: methods defined in Ball and Spectator.

Class **Ball**

The definition of the class is

```
Model subclass: #Ball
        instanceVariableNames: 'court direction position '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Tennis'
```

Operation of Ball requires access to Tennis to obtain court parameters and we thus define the creation message as follows:

```
for: aTennis
"Create a ball at the right end of the court."
        ^(self new) court: aTennis; reset
```

Initialization depends on method reset which sets the ball's initial position and direction of motion:

```
reset
"Assign initial position (right end of court) and direction of motion (left)."
        self position: self court courtLength value.
        self direction: #left
```

The essence of the whole simulation is method move which moves the ball by one step and changes its direction when it reaches the end of the court:

```
move
"Make a one-pixel move in the current direction. Check for end of court and change direction if necessary."
        self direction == #left
                ifTrue: [self position: self position - 1]
                ifFalse: [self position: self position + 1].
        self changed: #position with: self position.            "This triggers the dependency mechanism."
        self endOfCourt ifTrue: [self reverseDirection]
```

where the test for the end of the court is

```
endOfTrack
"Did I reach the end of the court? Return true or false."
        ^position = 1 or: [position = self court courtLength value]
```

and the change of direction is implemented as follows:

```
reverseDirection
"I reached the end of the course and direction of motion must be reversed."
        self direction: (self direction == #left
                                ifTrue: [#right]
                                ifFalse: [#left])
```

Finally, printing uses the printOn: method which is defined as follows:

```
printOn: aStream
        aStream nextPutAll: 'Ball position: ' , self position printString , ' direction: ' , self direction asString
```

Class **Spectator**

The definition of the class is

```
Object subclass: #Spectator
        instanceVariableNames: 'angle distance location '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Tennis'
```

To create a Spectator, we must initialize its distance from the right end of the court and its distance from the axis of ball motion:

```
distance: distanceInteger location: locationInteger
        ^self new distance: distanceInteger location: locationInteger
```

This definition depends on the following creation method defined on the instance side of Spectator protocol:

```
distance: distanceInteger location: locationInteger
"Complete initialization and calculate initial head angle for ball in position 1."
        self distance: distanceInteger.
        self location: locationInteger.
        self calculateAngleFor: 1
```

Angle calculation uses some simple trigonometry and its definition is

```
calculateAngleFor: anInteger
"Find angle for this spectator from current ball position anInteger."
        | xDistance |
        xDistance := anInteger - location.
        self angle: (xDistance / self distance) arcTan
```

The temporary variable xDistance is used only to make the code more readable.

Since spectators are dependents of Ball, they must understand an update message. The only parameter we need from the model is the ball's current position but to follow established conventions, the update: argument is a Symbol even though the method does not use it, and the second argument will be the ball's position:

```
update: aSymbol with: anInteger
"The ball has moved to position anInteger - recalculate head angle."
        self calculateAngleFor: anInteger
```

Finally, the printOn: method is as follows:

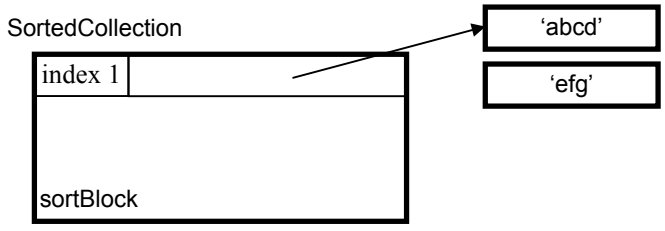
```
printOn: aStream
        aStream nextPutAll: 'Spectator location: ' , self location printString , ' angle: ' ,
                                (self angle asFixedPoint: 2) printString
```

Exercises

1. Complete and test the program from this section.
2. Dependency may be mutual as in the following simulation problem: Consider a collection of 5 nuclear particles distributed uniformly along a line. Each particle initially has some random amount of energy between 1 and 5. In each simulation step, the energy of a particle grows by a random integer amount between 1 and 3. When a particle's energy reaches or exceeds 10, it explodes, loses all its energy, and transfers it to the other particles in equal amounts. Implement the problem with a user interface allowing the user to control simulation parameters.

8.5 Class SortedCollection

SortedCollection is a subclass of OrderedCollection that automatically inserts new elements in their proper place among elements already in the collection. Insertion is based on a *sort block* stored in instance variable sortBlock (Figure 8.11). Most methods inherited from OrderedCollection work for sorted collections as well but some, such as addAfter:, are invalid because sorted collections determine where to put new elements themselves.



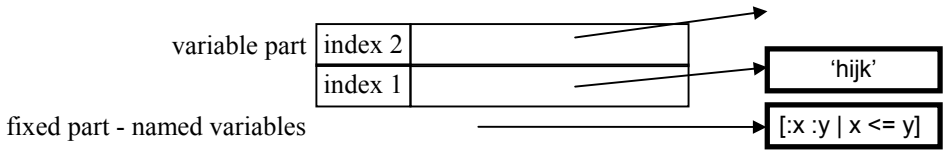


Figure 8.11. SortedCollection is a variable-size collection with one named instance variable whose value is a two-argument block. Its default value is [:x :y | x <= y].

Since each SortedCollection keeps its sort block in instance variable sortBlock, its value can be assigned when the SortedCollection is being created or even during its lifetime. Most sorted collections are created with new or new:, both of which assign sortBlock the default value stored in class variable DefaultSortBlock. *This* value is assigned by the SortedCollection initialize method which is

```
initialize
        DefaultSortBlock := [:x :y | x <= y]
```

This definition deserves several comments:

- None of the references to SortedCollection sends message initialize. What is its purpose then? This is executed to initialize SortedCollection when the class is first defined and the result is saved in the Smalltalk dictionary as a part of SortedCollection. You can change the default sort block by redefining initialize and re-executing SortedCollection initialize or by assigning a new value to DefaultSortBlock directly, and you could make the change permanent by saving the image. This, however, would change the effect of those existing uses of sorted collections that assume the current value of the sort block.
- The definition of initialize shows that default comparison uses message <=. This means that if you create a sorted collection with its default sort block, all elements that will ever be added to the collection must be able to compare themselves with all objects already in the collection using <=. Collections of strings and collections of numbers satisfy this condition. We can also define <= for other objects and compare, for example, rectangles by their size, Smalltalk methods by the amount of time that they take to execute, colors by the value of their hue or redness, and so on, and all these objects will now be able to sort themselves. The mechanism is very powerful.

The assignment of the default sort block in new: happens as follows:

```
new: anInteger
        ^(super new: anInteger) initialize
```

Here super new: refers to new: in OrderedCollection and creates a new uninitialized instance of SortedCollection. This SortedCollection object then executes the instance message initialize whose definition is

```
initialize
        sortBlock := DefaultSortBlock          "Instance method accessing a class variable."
```

We have already mentioned that you can assign a different sortBlock when creating an instance of SortedCollection or during its lifetime. In the second case, the sorted collection automatically resorts itself according to the new sort block. The following code fragment shows both possibilities. It first creates a sorted collection with a non-default sort block and then changes the sort block to the default value:

```
| sortedCollection |
sortedCollection := SortedCollection sortBlock: [:x :y | x > y].    "Non-default sort block - descending order."
sortedCollection addAll: #(13 53 56 21 98 73 56 89).                "Elements added one by one and sorted."
Transcript clear; show: sortedCollection printString; cr.
sortedCollection sortBlock: [:x :y | x <= y].                       "Switch to a different sort block and resort."
Transcript show: sortedCollection printString
```

Execution produces

```
SortedCollection (98 89 73 56 56 53 21 13)
SortedCollection (13 21 53 56 56 73 89 98)
```

Although the code sends sortBlock: twice, the two messages are different: The first is a *class* method and creates a SortedCollection with a non-default sort block, the second is an *instance* method and changes the sort block of the existing SortedCollection and resorts it.

Example 1. Sorting a collection of random numbers
Problem: Generate ten random numbers, display them in the Transcript in the order in which they were generated, and then again in the ascending order of their magnitudes. Print the numbers in the format #.### where # is a digit. An example of the desired output is

```
0.030  0.762  0.421  0.988  0.299  0.048  0.786  0.921  0.718  0.300
0.030  0.048  0.299  0.300  0.421  0.718  0.762  0.786  0.921  0.988
```

Solution: The principle is simple:

1. Create a SortedCollection with the default sort block.
2. Create a random number generator.
3. Send next to the random number generator ten times and each time display the generated number and add it to the SortedCollection.
4. Display the resulting SortedCollection in the desired format.

The problem of formatting the output is new. There may already be a method to do this and we thus search the library using the *Implementers Of ...* browser for something like print.... We find many matching selectors and one of them is called print:formattedBy: which seems promising. It is defined as a class method in PrintConverter and the definition shows that it indeed does what we want. As an example,

```
PrintConverter print: 3.14567 formattedBy: '#.##'
```

returns the string '3.14'. With this information, we can solve our problem as follows:

```
| generator number randomNumbers |
"Create a sorted collection and a random number generator."
randomNumbers := SortedCollection new: 10.          "Use default sort block."
generator := Random new.
Transcript clear.
"Generate random numbers, print them, and save in sorted collection."
10 timesRepeat:   [number := generator next.
                                Transcript show: (PrintConverter print: number  formattedBy: '#.###'), '  '.
                                randomNumbers add: number].
Transcript cr.
"Display all elements of the collection in the desired format."
randomNumbers do: [:element| Transcript show: (PrintConverter print: element  formattedBy: '#.###'), '  ']
```

We ran the program several times getting a different output each time as expected.

Example 2: Sorting shopping minder items.
Problem: We are to modify the shopping minder from the previous section to display the purchased items in alphabetical order.
Solution: We only need to use a SortedCollection in initialize instead of an OrderedCollection to store the items, and the sort block must make comparisons on the basis of the name of the item:

initialize

```
"Initialize items to a suitably sized sorted collection. Sort alphabetically by name."
        items := SortedCollection sortBlock: [:item1 :item2| (item1 name) < (item2 name)].
        self execute
```

Main lessons learned:

- Sorted collections are ordered collections that automatically sort their elements according to a sort block.
- The default sort block uses <= but a different sort block can be assigned by a class or an instance message.
- As elements are added to a sorted collection, they are automatically inserted into the proper place according to the sort block.
- Every pair of elements must be able to execute the sort block.

Exercises

1. Find and examine all references to sortBlock: and determine which of them are instance messages: and why they are used.
2. Study and describe how SortedCollection adds a new element.
3. Browse SortedCollectionWithPolicy and explain how it differs from SortedCollection.
4. Formulate the sort block for storing entries of the library catalog from the Section 8.3 in a SortedCollection using the alphabetical order of author names. Make up any necessary accessing methods.
5. Write a code fragment to create ten random rectangles and sort them in the order of increasing area.
6. Items are frequently sorted by more than one property. As an example, listings in telephone directories are by last name, and if last names are the same then by first name. Extend the previous exercise by using the area as the primary comparison key, and the circumference as the second comparison key, used when areas are the same.
7. Write a program to let the user compare the speed of execution of selected unary mathematical messages such as squared, ln, sin, and cos. Message names are entered by the user. Use SortedCollection.
8. Find forms of perform: that can be used when the method has one or more arguments and use them to write a more general version of the previous exercise that accepts any mathematical messages. (Hint: Use method keywords to obtain the number of keywords in the selector entered by the user.)
9. Modify the shopping minder to sort items in decreasing order of price.
10. Modify the System Browser to display names of categories in alphabetical order.
11. Ask the user to enter a string pattern and a series of strings, and sort the strings by their similarity to the pattern. As an example, if the user enters pattern 'abcd' and then strings 'abcd', 'abbd', 'aaad', and 'xxxx', one would expect that ordering by similarity will produce 'abcd', 'abbd', 'aaad', and 'xxxx'. (Hint: VisualWorks has a method for measuring the similarity of two strings. It is used by command c*orrect* displayed when you misspell the name of a variable. Find it.)
12. Ask the user to enter several labels and display them in alphabetical order in a multiple choice dialog.

8.6 The List collection

Class List combines properties of ordered and sorted collections and adds dependency. According to the manual, List is intended to supersede ordered and sorted collections but its main use in the library is as an item holder in multi-item GUI widgets. This is consistent with the placement of List in the UIBasics - Collections category in the library.

In terms of implementation, List is not on the OrderedCollection branch of the Collection subtree, but a subclass of ArrayedCollection. Its instance variables are collection, limit, collectionSize, and dependents.

According to the class comment, collection holds the list's elements and the creation methods initializes it to an Array. This means that List is not itself a collection but contains a collection as one of its components. The role of limit is to hold the index of the last element in the collection that holds a valid element. This is because List makes a distinction between size and capacity and must keep track of how many of its slots are valid elements. collectionSize holds the size of the collection. dependents is nil, one object, or a collection of objects and implements the dependency mechanism: When an object is added or deleted, all dependents are notified.

Lists can be used as sorted collections because they can sort their elements. However, List does *not* have a sort block and does not automatically insert new elements. Instead, new elements are added using the adding protocol of OrderedCollection, for example addLast: and addFirst:. To sort a List, send it sortWith: aSortBlock or the sort message which uses the default sort block [:a :b | a <= b].

If List does not have a sort block, how can it sort itself? The answer is that a List does *not* sort itself but asks class SequenceableCollectionSorter to do it. The details are obvious from the following definition s of List's sorting methods:

```
sort
        ^SequenceableCollectionSorter sort: self

sortWith: aBlock
        ^SequenceableCollectionSorter sort: self using: aBlock
```

SequenceableCollectionSorter, a class that knows how to do sorting, is a nice example of object-oriented thinking - an object that can perform a service for its clients.

We will use List when we present widgets that depend on it later in this chapter.

Main lessons learned:

- List objects resemble ordered and sorted collections and add dependency handling.
- For accessing, lists behave like ordered collections.
- Unlike sorted collections, List does not sort itself automatically but must be explicitly asked to do so.
- A List sorts itself by sending a message to SequenceableCollectionSorter.
- The most common use of lists is as item holders of multi-item GUI widgets.

Exercises

1. List initializes collection to an Array which is a fixed size collection. What happens when you add a new item? What happens when you remove an element?
2. Write a program to test how sorting time depends on the number of elements in the list. The program should create a list with 10, 100, 1,000, and 10,000 random numbers and send them the sort message. Plot the numbers as a function of the size of the list.
3. Sorted collections insert a new element into its proper place as soon as it is added, lists sort their elements only when explicitly requested to do so. Write a program to compare the speed of the two approaches by creating a sorted collection containing a set of 100, 1,000, and 10,000 random numbers with a List and with a SortedCollection. Create the two collections so that they don't need to grow.

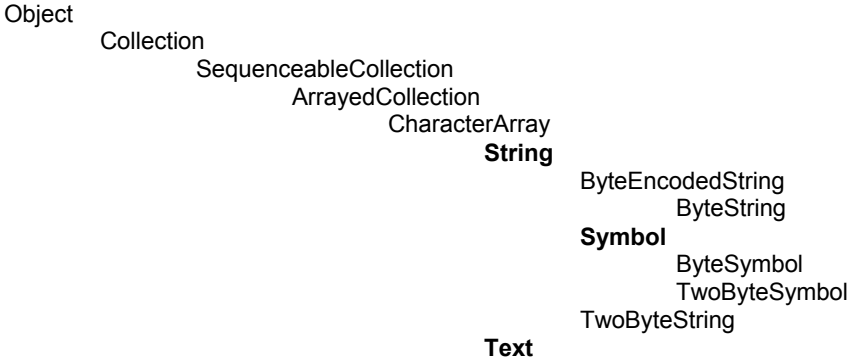
4. Class SequenceableCollectionSorter can be used with any sequenceable collection but its main purpose is to sort List objects, for example to display items in a multi-item widget in a particular order. Write a short description of its behavior and instance variables.
5. Write a comparison of List and OrderedCollection highlighting the similarities and differences in their behavior and implementation.
6. List is one of the few collection classes that are not variable-size classes and that store their elements in an instance variable. What does List gain from being in the Collection tree?

8.7 String, Text, and Symbol - an introduction

Classes String, Text, and Symbol are all in essence collections of characters such as letters, digits, and punctuation symbols. A String is a sequence of characters and its functionality includes searches, deletions, and replacements. Strings are concerned with contents and don't contain any information about rendering - font, size, boldness, and so on. When they are displayed ('rendered'), they use default system representation.

A Symbol is also a sequence of characters but the rules for its formation are slightly different and its main use is for selectors - names of methods; this is reflected in its different protocol. Unlike strings, symbols are unique in that only one instance of a particular sequence of characters can exist at a time. This is their most important defining property which guarantees very fast access. Finally, instances of Text are aggregate objects containing a string and information about its emphasis - font, size, color, and other rendering information.

The internal natures of String and Symbol are closely related - both are essentially collections of characters - whereas Text objects are related conceptually but their internal structure is different. This is reflected in the class hierarchy: Classes String and Symbol are on the same branch but Text is separate:



All three classes inherit many useful methods from the abstract class CharacterArray. We used strings and symbols informally before and we will now present some new information and examples of both. Text will be covered in the next section.

Class String

String is an *abstract class* that understands messages such as finding substrings and replacing substrings with other strings. Its superclass is the abstract superclass CharacterArray which means that its elements are characters (instances of class Character). *Concrete* strings are implemented by concrete subclasses of String, each designed for a different kind of character encoding. In most applications, strings are instances of ByteString which encodes its characters as one-byte ASCII codes. For large character sets requiring more codes, strings can be implemented as TwoByteString objects.

The fact that String is an abstract class and that string objects are instances of one of its concrete subclasses is hidden and you don't have to think about it. This is because when you create a new string as in

```
String new: 20      “String with room for 20 characters”
```

the definition of new: in class String automatically determines which concrete subclass to use as follows:

```
new: size
        ^self defaultImplementor new: size
```

Here defaultImplementor is a class method defined as follows:

```
defaultImplementor
"Answer a class that usually represents strings."
        ^ByteString
```

As a consequence,

```
String new: 20
```

is equivalent to

```
ByteString new: 20
```

and all messages to the resulting string object are handled by class ByteString. This trick of using an abstract class as if it were concrete and allowing it to select the most suitable concrete class is used in several places and we will encounter it again later. Its advantage is that applications can be made independent of the concrete class and the same program may be, for example, executed on different platforms that use different defaults.

Most of the interesting protocol of String is defined in CharacterArray. Since String is a collection, it also inherits the rich collection protocol of all its superclasses up to Collection. The most useful methods defined in String include comparison and conversion to text via asText which allows strings to be used where Text objects are expected.

The protocols inherited from CharacterArray include accessing (replacement and search), converting (to number, text, symbol, and other related objects), copying, comparing (alphabetical relations), and displaying on the screen. One example of an accessing message is changeFrom: start to: stop with: replacement which takes the receiver string, removes characters indexed from start to stop, and inserts the string provided as the third argument starting at index start. The replacement string may contain fewer, more or the same number of characters as the specified range of indices and when start > stop, the argument string is simply inserted into the original. Another useful accessing message is findString: subString startingAt: start which searches for the first occurrence of subString starting from start and returns the index of its first character or 0 if the substring is not found. We will now develop a new method using both messages.

Example 1: Replacing substrings

Problem: Write a method with three string arguments: a start string, a match string, and a replacement string. The method finds all occurrences of the match substring in the start string and substitutes the replacement string for them. The result is returned.

As an example, if the start string is ‘a secret 1 containing encoded sub1s’ and the match string is ‘1’, and the replacement string is ‘string’, the program should replace every occurrence of 1 with ‘string’ and produce ‘a secret string containing substrings’.

Solution: The basis of the method is the following algorithm:

1. Set starting position to 1.
2. Look for the next occurrence of the match string. If a match is found, make the replacement and repeat this step.

The algorithm is easy to implement because all steps can be implemented with built-in messages. We will put the method in the *accessing* protocol in CharacterArray because it is related to other methods in this protocol and because it will be useful in subclasses. The definition is as follows:

```
replace: matchString with: replacementString
"Replace all occurrence of matchString in string with replacementString."
        | index newString matchSize |
        newString := self copy.
        matchSize := matchString size.
        "Find all occurrences and do replacement."
         [index := newString findString: matchString startingAt: 1. index = 0]
                whileFalse:
                [newString changeFrom: index to: index + matchSize - 1 with: replacementString].
        ^ newString
```

Unfortunately, when you try to execute the method, it enters into an infinite loop! The problem is due to our misunderstanding of the exact operation of changeFrom:to:with:. We leave it to you to find and correct this common error.

Closing comments

Before we close this very brief introduction to String, here is a useful piece of information. As you know, the most common way of concatenating two short strings (and many other collections) is with the, message as in

```
'abc' , 'def'          "Returns 'abcdef'."
```

However, what if you wanted to concatenate a string with a single character?

```
'abc' , $d
```

does not work because one cannot concatenate a string with a character. One could, of course, use

```
'abc' , 'd'
```

but what if the character is calculated or stored in a variable? In this case, you can use the copyWith: message from SequenceableCollection which creates a copy of a collection and appends the argument at the end as in

```
'abc' copyWith: aCharacter
```

This works because a string is a collection of characters.

Another comment related to concatenation is that the , method is very inefficient. When speed is important, concatenate strings using streams, a topic covered in Chapter 10. Since concatenation is usually an essential part of creating a string, streams are essential for *efficient* use of strings in general.

Finally, the concatenation method is defined in SequenceableCollection and its use is not limited to strings. As an example, you can execute

```
orderedCollection1 , orderedCollection2
```

Class Symbol

Symbol is a special kind of String whose literal form is a sequence of characters preceded by the # character as in #aWidget or #aMethod or even #'abc def'. A Symbol must not include characters such as blank space, tab, line feed or the # character itself, unless the string following # is a literal string. As an example, #ab cd is not a single Symbol but #'ab cd' is - and the apostrophes are *not* a part of it.

Unlike instances of String, Symbols are implemented as *unique* objects. This means that if you create two or more Symbols consisting of identical characters, they will be stored in memory only once. On

the other hand, if you create two *strings* consisting of identical characters, they will be stored as two separate objects. As a consequence, two different instances of String may be *equal* (message =) but not *equivalent* (message ==), whereas for symbols, equality and equivalence are the same:

```
'characters' = 'characters'          "Returns true."
'characters' == 'characters'"Returns false."
#characters = #characters            "Returns true."
#characters == #characters           "Returns true."
```

Because Symbols are unique their comparison uses *equivalence* which, as we know, is fast because it requires only comparison of pointers to internal memory representations. Strings, on the other hand, are normally compared for equality which may be very time consuming. Note that although Symbol defines == to be the same as = and one could thus use = with the same effect, most programmers use == because it is faster, eliminating the = message send.

Symbols are used mainly for names of methods as in our comparison of speed of number messages, as widget IDs and *Action* and *Aspect* methods (see any windowSpec method for an example), and in similar situations in which uniqueness makes sense (such as days of the week, male and female gender, and other non-ambiguous characteristics) and in which the speed of comparison is important. Because one of their main uses is to represent method selectors, class Symbol contains a protocol dedicated to recognizing unary, binary, and keyword messages and similar aspects of selectors. The following example illustrates this protocol.

Example 2: Get information about messages

Problem: Write a program to count how many unary, binary, and keyword methods are defined in the instance protocols of Object. In addition, find how many of the keyword messages have one argument, how many have two arguments, and so on. Print the result in Transcript in the following format:

```
Total number of methods: 161

Number of unary methods: 81
Number of binary methods: 5
Number of keyword methods: 75

Number of arguments/Number of methods
1:  53
2:  15
3:  5
4:  2
```

Solution: The problem is quite specialized and we will implement it as a code fragment. The implementation will use the following algorithm:

1. Create an ordered collection keywordMethods to hold the selectors of all keyword methods.
2. Enumerate over all methods defined in Object as follows:
   a. If the method is unary, increment the count of unary messages.
   b. If the method is binary, increment the count of binary messages.
   c. If the method is a keyword method, increment the count and add its selector to keywordMethods.
3. Enumerate over all selectors in keywordMethods and count the number of methods with one argument, two arguments, and so on. Store the number of keyword messages with one argument in element 1 of array counts, the number of keyword messages with two argument in element 2, and so on.
4. Create an array called keywordCounts whose size is equal to the maximum number of keywords in keywordMethods. Store the number of all keyword messages of size n in the n-th element of this array.
5. Print results.

To implement this algorithm, we must know how to find all methods defined in a class, how to recognize whether they are unary, binary, or keyword methods and determine how many arguments they have. As we have already mentioned, Symbol is mainly used for selectors and it contains the selector-

related methods. But how about finding all methods defined in a class? One way to find out is to think of a Smalltalk tool that performs a similar function, and as usual, this tool is the Browser; its class view <operate> menu includes command *find method* which lists all instance or class methods. Its implementation in class Browser must know how to find the methods! When you examine Browser, you will find that the following definition contains the desired information:

```
findMethodAndSelectAlphabetic
"Show a menu of the methods implemented by this class.  Select the chosen one."
| chosenSelector selectorCollection |
self changeRequest ifFalse: [^self].
(selectorCollection := self selectedClass organization elements asSortedCollection) size = 0
etc. - the rest is irrelevant for our needs
```

It seems that the part organization elements does what we need. To test it, we execute

```
Object organization elements
```

with *inspect* and find that it returns an array containing all instance methods defined in Object[4].

The next task is to find Symbol methods that recognize various types of selectors and number of arguments. We find that instance protocol *system primitives* contains methods isInfix (recognizes binary messages), isKeyword (recognizes keyword messages), and numArgs (returns the number of arguments of a keyword message). There is no special method for unary messages but unary messages are those that are not keyword or binary. With this information, we can now convert our algorithm into the following code:

```
|unaryCount binaryCount keywordCounts keywordMethods methods |
unaryCount := 0.
binaryCount := 0.
keywordMethods := OrderedCollection new.

"Enumerate over all selectors defined in Object and store their counts in unaryCount, binaryCount, and
keywordCount."
(methods := Object organization elements) do:
	[:method| method isInfix
		ifTrue: [binaryCount := binaryCount + 1]
		ifFalse:[method isKeyword
			ifTrue:[keywordMethods add: method]
			ifFalse: [unaryCount := unaryCount + 1]]].
Transcript clear; show: 'Total number of methods: ', methods size printString; cr; cr;
		show: 'Number of unaryCount methods: ', unaryCount printString; cr;
		show: 'Number of binaryCount methods: ', binaryCount printString; cr;
		show: 'Number of keywordCount methods: ', keywordMethods size printString.

"Create an array to hold counts of keywordCount messages with the same number of arguments.
Find its size as the maximum number of arguments over all keywordMethods elements."
keywordCounts := Array new:
			(keywordMethods inject: 0 into: [:max :method | method numArgs max: max]).
keywordCounts atAllPut: 0.
"Calculate the counts and store them in the array."
keywordMethods do:
	[:method| | size | size := method numArgs.
		keywordCounts at: size put: (keywordCounts at: size) + 1].

"Output results."
Transcript cr; cr; show: 'Number of arguments/Number of methods'; cr.
1 to: keywordCounts size do:
	[:number| Transcript show: number printString, ':'; tab;
			show: (keywordCounts at: number) printString; cr]
```

[4] To find more about organization, read Appendix 1.

Note that we calculated the maximum number of arguments over keyword selectors using the inject:into: message.

Main lessons learned:

- String is an abstract sequenceable collection whose elements are character codes.
- Concrete strings are instances of String subclasses, ByteString by default. Selection of the concrete class is transparent and the programmer does not need to pay attention to it unless a larger character set is needed. In that case, a TwoByteString may be used or a new subclass defined.
- The most useful protocols of String include messages to search for substrings and to replace substrings, copy, and compare strings.
- Symbol is a subclass of String that does not allow duplication.
- The main uses of symbols are for method selectors and as widget IDs in GUI specifications.
- Binary messages are also called infix messages.

Exercises

1. Why didn't we use self instead of newString in replace:with:?
2. Does our replace:with: method work in all subclasses of CharacterArray?
3. Write a method to remove all occurrence of a substring from a given string.
4. List all collections that understand the , (comma) concatenation message.
5. Explain the definition of = in class String.
6. Study the definition of changeFrom:to:with: and show how to use it to insert 'xyz' behind the first occurrence of letter $a in an existing string, and how to append a substring at the end.
7. One of the frequently used String methods is withCRs. Read its definition and find how it is used. Use withCRs to display a dialog request window whose prompt has 'Enter your first and last names.' on the first line, and 'Capitalize the first letter of each name' as the second line. Write the message with and without using withCRs and comment on the difference between the two styles.
8. Modify the example from the Symbol section to gather statistics on *all* classes in the library. The program should print the total number of unary, binary, and keyword messages in the library, and the selectors of all methods with the largest number of arguments. (Hint: To enumerate over all classes, use message allClassesDo: from SystemDictionary.)
9. Repeat the previous exercise for class methods. (Hint: Class methods are instance methods of the class's class and to ask an object what is its class, send it the class message. See Appendix 3 for more.)
10. Executing 'aSymbol' asSymbol returns #aSymbol. What is ('Symbol1 ', 'Symbol2') asSymbol?
11. What is the difference between asString, displayString, and printString?
12. How many different characters can be implemented with ByteString? How many with TwoByteString?

8.8 Text - its nature and use

The purpose of Text objects is to gather all information necessary to display characters on a display medium such as the computer screen or a printer. Although you can display a String without converting it to Text because it automatically converts itself to a text object when asked to display itself, using a Text is necessary if you want to display the string with a non-default font or color, underlined, large or small, and so on.

Text has two instance variables called string and runs. Variable string holds the characters to be displayed (a String), and runs is an instance of RunArray which holds ranges of indices and encoded information about their emphasis. As an example, runs might say that the first characters use default style, the next 15 are underlined, and so on. Conversion from styles to actual fonts is performed with a TextAttributes object which indicates, for each style, the rendering parameters to be used. The concept of TextAttributes is not simple and we will not deal with it in this book.

As an example of a Text object, if we specify that string 'abcdefgh' should use emphasis #bold for elements 3 to 6, in other words should be displayed as 'ab**cdef**gh', the internal variables are

```
runs = RunArray (nil nil #bold #bold #bold #bold nil nil)
string = 'abcdefgh'
```

where nil means default emphasis, in other words default boldness, font, color, and size. Style #bold is a code name for a certain font converted to an actual graphical representation when the character is displayed.

The protocol of Text contains *accessing*, *emphasis*, *displaying*, *comparison*, *copying*, and other protocols and most of them are *character oriented*. As an example, *accessing* method at: anIndex returns the character at location index but not its emphasis.

Truly sophisticated display of text requires the understanding of TextAttributes and other display characteristics. As we already mentioned, these are detail beyond our scope and we will restrict ourselves to explaining how to create Text objects with simple built-in emphases and how to control text color.

The basic message to control emphasis is emphasizeFrom: start to: stop with: emphasis where start and stop are the indices of the endpoints of the substring to be emphasized, and emphasis is a Symbol representing emphasis - an association specifying color, or an array containing emphasis and possibly color information. Standard emphases symbols include #bold, #italic, #large, #normal, #serif, #small, #strikeout, and #underline. As the simplest example,

```
'Bold text' asText emphasizeFrom: 1 to: 4 with: #bold
```

changes the emphasis of the word 'Bold' to #bold. Note that we had to convert the string to text before we could send the emphasis message.

When you want to change the color of a string, you must define it as an Association object consisting of the symbol #color and a value specifying the desired color expressed as a ColorValue object. We will have more to say about associations later but for now, use them as in the following example:

```
'Bold text' asText emphasizeFrom: 1 to: 4 with: #color -> ColorValue red
```

The binary message -> creates an Association object with the specified key (here #color) and value (here ColorValue red). An important point about specifying emphasis is that every message that defines new emphasis overrides the emphasis assigned previously. As an example,

```
| text |
text := 'Bold text' asText.
text emphasizeFrom: 1 to: 4 with: #bold.
text emphasizeFrom: 1 to: 2 with: #italics
```

ends up overriding the #bold emphasis for the first two characters with #italics. If you want to assign multiple emphasis to a character, you must specify it as an array, as in

```
| text |
text := 'Bold italicized text' asText.
text emphasizeFrom: 1 to: 15 with: #(#bold #italics).
text emphasizeFrom: 5 to: 15 with: #bold
```

or

```
| emphases text |
text := 'Bold italicized red text' asText.
emphases := Array with: #bold with: #italics with: #color -> ColorValue red.
text emphasizeFrom: 1 to: 19 with: emphases
```

As a matter of style, using specific numbers for the start and end index as we did is not usually a good idea because your indices may be wrong and must be recalculated if you change the string.

Example 1: Experiment with emphasis and labels

Label widgets in user interfaces use text and this example is an experiment with labels, their change at execution time, and emphasis.

Problem: Design an application with the user interface in Figure 8.12. When the user clicks *New label*, a series of prompts requests a string, an emphasis (using multiple choice dialog), and the starting and end index of a substring to which the selected emphasis should be applied. The emphasis selection dialog is repeated until the user clicks *Cancel*. The program then displays the text with the specified emphases in place of the original text 'Label'.

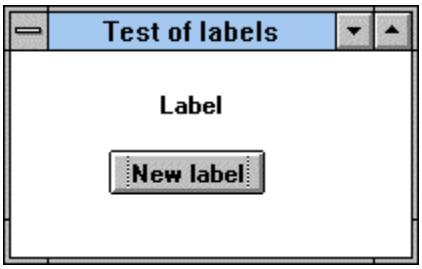


Figure 8.12. Desired user interface for Text example.

Solution: This very simple application requires only an application model class. After painting the user interface, assign #label as the ID to the label widget so that it can be accessed and modified at run time, and install the user interface on an application model. The only method that needs to be written is the *Action* method for the button:

```
newLabel
"Get text and emphases and display the label."
|text |
        text := (Dialog request: 'Enter new text' initialAnswer: '') asText.
        "Get emphases with ranges and apply them."
        [| emphasis noSelection |
                emphasis := self getEmphasis.
                (noSelection := emphasis == #noChoice)
                        ifFalse: [text := self emphasizeText: text with: emphasis].
                noSelection] whileFalse.
                "Now send the new label to the builder for immediate display."
                (self builder componentAt: #label) widget label: (Label with: text)
```

The two auxiliary methods used in the definition are

```
getEmphasis
"Obtain emphasis through dialog."
        Dialog  choose: 'Select emphasis for ' , text
                fromList: #('bold' 'italic' 'underlined')
                values: #(#bold #italic #underlined)
                lines: 3
                cancel: [#noChoice
```

```
emphasizeText: text with: emphasis
"Apply specified emphasis to the text."
[| start end |
        start := (Dialog request: 'Start index (' , text , ')' initialAnswer: '1') asNumber.
        end := (Dialog request: 'End index (' , text , ')' initialAnswer: text size printString) asNumber.
        ^text copy "We cannot change the argument so we must work on the copy."
                emphasizeFrom: start
                to: end
                with: emphasis
```

Printing text

There are two ways to print text. The more primitive one is to use the hardcopy message, the more sophisticated and more flexible approach uses class Document. We will now describe both.

Method hardcopy is defined in class ComposedText whose instances hold information about the display of text, indentation, spatial properties such as the box in which the text is displayed, and so on. Text objects themselves don't understand hardcopy and to display them with hardcopy, we must first convert them into ComposedText. Altogether, to display text, execute

```
text asComposedText hardcopy
```

The limitation of hardcopy is that it prints the whole text in the default style and ignores emphasis. To print text with its emphases, use the Document class.

Class Document has all the functionality to create documents containing text and graphics on PostScript printers. Its message toPrinter sends an instance of Document to class DocumentRenderer which takes care of the details of transmitting the document to the printer via class Printer. Both Document and Printer contain several interesting examples illustrating their functionality. In the following, we will limit ourselves to showing how to print a Text object with its emphasis, and how to print a window displayed on the screen (not relevant to Text but useful).
The general procedure for creating a text document and printing it is as follows:

1. Create a new Document object using the new message.
2. Use messages addParagraph and addString: text to define the contents of the document.
3. Close the document with the close message.
4. Send the document to the printer using toPrinter.

Example 2: Printing a Text object.
Problem: Create and print a document containing two consecutive paragraphs. The first paragraph consists of the text

Comment for class ***Document***:

which is followed by an empty line and the class comment for Document. The comment is printed using italics.

Solution: Following the general algorithm for creating and printing a document, the code is as follows:

```
| document string text start stop |
"Construct a Text object for the first line of document."
string := 'Comment for class Document:'.
start := string findString: 'Document' startingAt: 1.
stop := start + 'Document' size - 1.
text := string asText emphasizeFrom: start to: stop with: #(#bold #italic).
"Now go into a block that creates a Document and prints it, waiting until the process has been completed."
Cursor wait          "Display a special cursor while the following block is executing."
        showWhile:
                [document := Document new.          "Create a Document object."
                document startParagraph.            "Do the first paragraph."
                document addString: text.
                document startParagraph.            "Do the second paragraph with class comment."
                document addString: (Document comment asText emphasizeAllWith: #italic).
                "Close Document and send it to the default printer."
                document close.
                document toPrinter]
```

Expression Cursor wait showWhile: aBlock displays the 'wait' cursor while until the block executes, in our case until the printing process gets under way.

Example 3: Printing a graphics object

The example method doTest2 in class Document lets the user select a rectangle on the screen and prints it on the printer preceded with a short text. Its definition is as follows:

```
doTest2
"Make a simple PostScript file... a screen dump scaled in a device independent way."
	| document |
	Cursor wait
		showWhile:
			[document := Document new.
			document startParagraph.
			document addItalicHeader: 'Screen dump:' pixelSize: 24.
			document addImage: Image fromUser.
			document close.
			document toPrinter]
```

The general procedure is the same as in the previous example but a new message called addImage: displays the graphics. The image itself is obtained from the user who selects it using crosshair cursors. Note that this method is based on class Image.

Example 4: Extend Example 1 in Appendix 1 to print the window.

Problem: Write a method to print the window from Example 1 in Appendix 1 on the printer and add a new button to the window to do this.

Solution: We have seen how to print an Image using Document and the only remaining problem is how to make an Image out of a window. First, we need to know how to get hold of the window object. Since the window that we want to print is active when the user clicks the button, we will need to find the currently active window. Realizing that a Window object is a collection of widgets, labels, measurements, and other things - and not an Image - we must then find how to specify the rectangle occupied on the screen by the window, and how to convert it to an image.

We will start by determining how to identify a rectangle on the screen. We note that our previous example obtains the Image by sending fromUser to class Image. The definition of fromUser is as follows:

```
fromUser
"Answer an instance with the same contents as a user-specified rectangle of the default screen."
	^Screen default contentsFromUser
```

As we already know, Screen default provides access to the screen, and contentsFromUser returns the Image drawn by the user on the screen. Its definition is

```
contentsFromUser
"Answer an Image with the contents of a user-specified area on my screen."
        ^self completeContentsOfArea: Rectangle fromUser
```

This shows that if we have a window rectangle (call it windowRectangle) we can obtain its rectangle by

```
Screen default completeContentsOfArea: windowRectangle
```

The next step is to determine how to find the active window and its screen coordinates. Since the window is the active window of our application, we can get it by sending window to the builder. When we try this and *inspect* the window, we find that the returned object is an instance of ApplicationWindow. Finally, ApplicationWindow's protocol includes message getDisplayBox which returns the window's rectangle. With this last piece of information, we can now write the complete solution of our problem:

```
printActiveWindow
 "Print currently active window on printer."
        | document window windowRectangle |
        "Find window rectangle."
        windowRectangle := self builder window getDisplayBox.
        window := Screen default completeContentsOfArea: windowRectangle
        "Create document with window and print it."
        Cursor wait showWhile:      "Display special cursor while the following block is evaluating."
                [document := Document new.
                document addImage: window.
                document close.
                document toPrinter]
```

Add a *Print* button to the original interface in the example, add this method to the application model's class as the *Action* of the *Print* button, and test it.

Main lessons learned:

- Class Text adds emphasis to strings, allowing their display in different fonts, sizes, and colors.
- Text objects consist of a string and a run array describing character fonts and other rendering parameters.
- Text objects are usually obtained by converting a string with asText.
- For simple output of text to the printer, convert text to ComposedText and send hardcopy. This message ignores emphasis.
- For sophisticated output, possibly including graphics, use class Document.
- The builder of an application window provides access to its window.
- Windows know about properties such as screen coordinates of their bounding rectangle.

Exercises

1. Add button *Color* to Example 1. Clicking *Color* should open another multiple choice dialog listing all available colors. (Hint: See the class protocol of ColorValue.)
2. Define Text message allItalic to italicize the receiver.
3. Explain the operation of message asText in class String.
4. We simplified the nature of RunArray somewhat. Study its class definition and write a more accurate description. Explain why the description displayed by the inspector does not exactly correspond to the definition . Explain why Text uses RunArray rather than Array.
5. Write a description of class Cursor.
6. Class Screen and its superclasses provide access to some very interesting behaviors and information. Write their description.
7. Implement Example 4.

8. The method printActiveWindow does not contain any specific reference to the application example and can be used with any application based on the UIBuilder. Move it to a more general class where it could be used by any application, modify it as necessary, and test it.
9. Add a *print* command to the <window> menu so that the printActiveWindow method can be executed on any window. (Hint: Find currently active window using Window currentWindow.)

8.9 List widgets

In this section, we will introduce the list widget, both for its own sake and as an illustration of the use of collections. List widgets are available in two varieties - those that allow only one selection at a time, and those that allow multiple selections (Figure 8.13)

Single selection lists are much more common and they are used, for example, in the System Browser. Their purpose is similar to that of menus - making a selection - and their advantage over menus is that they allow the user to see the selections at all times – menus must be popped up or dropped down. Menus have the advantage that they don't take up space in the window. *Multiple selection lists* are used, for example, in the Definer in the UI Painter

In addition to the similarity of the purpose of lists and menus, lists are also related to check boxes and radio buttons: Single selection lists allow one selection at a time, just as radio buttons, except that in a set of radio buttons one is normally always selected which is not required in lists. Multiple lists allow any number of selections, just like check boxes. One would use radio buttons or check boxes when the number of options is small and fixed, lists would be better when the list can change or when the number of choices is large. We will now cover both types of lists starting with single selection lists.

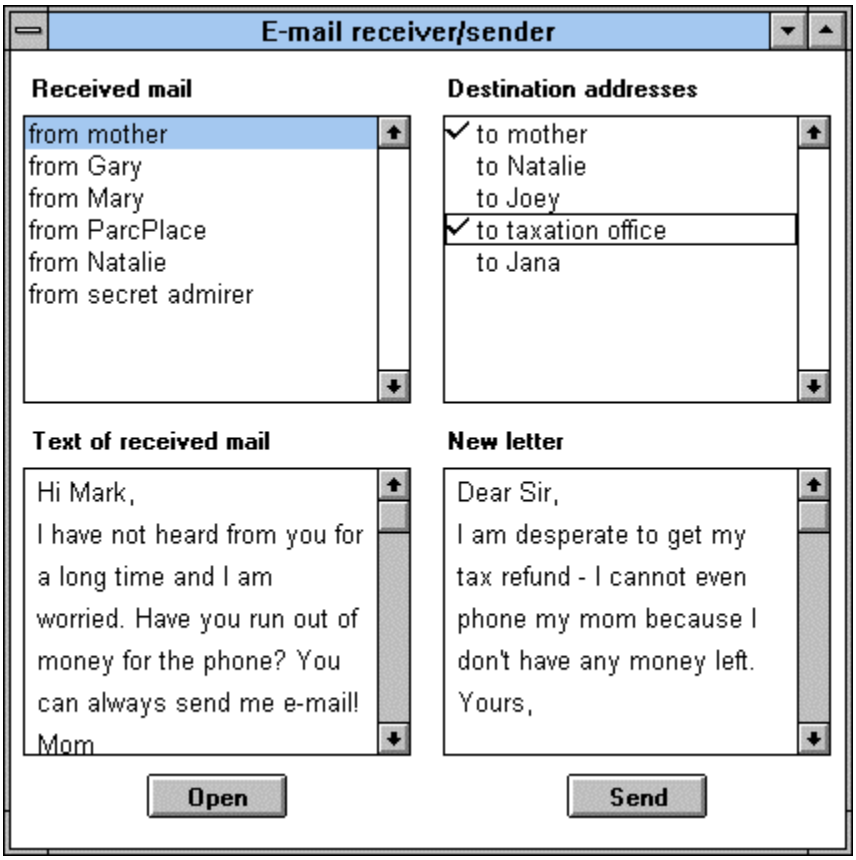


Figure 8.13. Single selection list (left) and multiple selection list (right).

Single selection lists

The basic behaviors of list widgets are: create a list widget with labels, add a new label, remove an existing label, sense change of selection, get current selection, and access labels.

The model of a single selection list (the object on which the list widget is dependent) is normally an instance of SelectionInList. This class has two instance variables called listHolder and selectionIndexHolder, both instances of ValueHolder. Variable listHolder holds the list of objects displayed by the widget (a List object), and selectionIndexHolder holds the index of the currently selected item or 0 if no item is selected. To be able to use single selection lists you need to know that the *Aspect* of the list widget is a SelectionInList and that this SelectionInList can be created either with message new as in

```
SelectionInList new
```

if the list is not available in a sequenceable, or using with: as in

```
SelectionInList with: aSequenceableCollection
```

where the argument contains the items to be displayed.

To display some information in the list when the window opens, initialize the list holder in the initialize method. To obtain the value of the currently selected item, send selection (returns currently selected object or nil) or selectionIndex (returns index of selection or 0) to the list holder. To change the selection programmatically, send selection: aValue or selectionIndex: anInteger to the list holder. To obtain the collection of items displayed in the widget, send list to the list holder. And to change the displayed list, send list: aSequenceableCollection with any sequenceable collection containing the objects for its argument. The elements of the collection should be strings. As an example,

```
listHolder list: #('line 1' 'line 2' 'line3')
```

will erase the current labels and replace them with 'line 1' 'line 2' 'line3'.

To ensure that the list notifies the application model when the selection changes, send onChangeSend:to: to the selectionIndexHolder during initialization, as in

```
classList selectionIndexHolder onChangeSend: #changedClass to: self
```

Example 1: Basic list behaviors

Problem: Implement an application with the interface in Figure 8.14 such that the window opens with labels 'label 1', 'label 2', 'label 3'. The *Add* button will display a prompt asking the user for a new label and add it to the list, the *Delete* button removes the currently selected label, and the *Print* button displays all labels in the Transcript. Every change of selection displays a notification in the Transcript. The labels must be alphabetically sorted at all times.

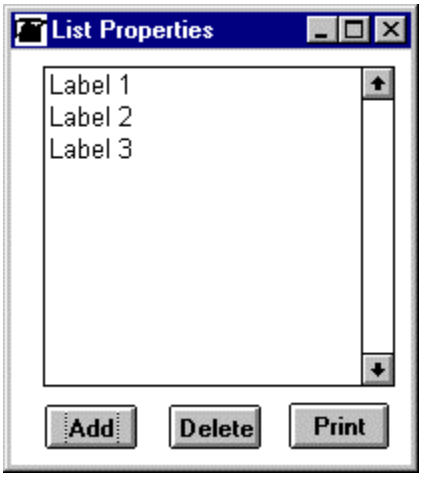


Figure 8.14. User interface for Example 1.

Solution: Paint and install the window and the widgets, and name the *Aspect* of the list (we used the name #labels). The *Action* methods will be called add, delete, and print. Since we want the window to open with three labels, we must initialize the SelectionInList and we will do this in the initialization method. The specification requires that every change in list selection trigger a message to the Transcript; we will register this interest in selection changes via the onChangeSend:to: message. The whole definition is as follows:

```
initialize
"Initialize labels in list and register interest in change in list selection."
        labels := SelectionInList with: #('label 1' 'label 2' 'label 3').
        labels selectionIndexHolder onChangeSend: #printChange to: self
```

If you now open the application, it will look exactly like Figure 8.14 but the buttons will not work and any attempt to select a label in the list will open an exception window because we have not defined printChange specified as the change message. This method, which is executed whenever the selection changes, simply asks the list for the current selection and sends appropriate output messages to the Transcript:

```
printChange
"List selection has changed, print appropriate message to Transcript."
        labels selection isNil
                ifTrue: [Transcript cr; show: 'no selection']
                ifFalse: [Transcript cr; show: 'selection is ', labels selection]
```

The delete method first checks whether we have a selection, and if we do, it deletes the selected label from the list. To do this, we must ask the list for its current labels (message labels listHolder value), remove the current selection from it, and assign this modified list as the new list using the list: message. It is essential to use list: because if we did not, the notification of change would not propagate to the widget and the binding between the Aspect and the list could be destroyed. The whole definition is as follows:

```
delete
"Check if there is a selection. If so, delete it from the list."
        labels selection isNil
                ifFalse:  [| newList |
                        Transcript cr; show: 'deleted ' , labels selection.
                        newList := (labels listHolder value) remove: labels selection; yourself.
                        labels list: newList]
```

Note the use of yourself after remove: - it returns the modified receiver list. This is necessary because remove: modifies its receiver but returns the argument. Without yourself, the value of newList will be the string containing the deleted label, the widget will treat it as a sequenceable collection of objects, and display its individual characters as strings - instead of displaying the labels. Forgetting yourself is a very common mistake.

We used some unnecessary parentheses and an extra temporary variable to make the code more readable. Unfortunately, the method does not work because the remove: message is illegal for arrays which is the collection that we specified as underlying the list. To correct this, we convert the array to an ordered collection, changing initialization as follows:

```
initialize
"Initialize labels in list and register interest in change in list selection."
        labels := SelectionInList with: #('label 1' 'label 2' 'label 3') asOrderedCollection.
        labels selectionIndexHolder onChangeSend: #printChange to: self
```

Deletion now works and we can proceed to the add method. This method requests a new label from the user and continues much like delete except that it adds a new element rather than removing it:

```
add
"Request a new label. If one is supplied, add it to the list holder."
| string |
        string := Dialog request: 'Enter new item' initialAnswer: ''.
        string isEmpty
                ifFalse:  [Transcript cr; show: 'added ' , string.
                        labels list: ((labels listHolder value) add: string; yourself)]
```

Finally the method to print all label when the print button is clicked. This method simply obtains the list from the list holder and displays its elements one after another in the Transcript:

```
print
"Display all labels in the Transcript."
        Transcript cr.
        labels listHolder value do: [:item | Transcript show: item; cr]
```

Example 2: Class definition browser

Problem: Write an application with the user interface in Figure 8.15. The single selection list on the top lists all classes in the library and when the user selects one, its definition appears in the text view at the bottom.

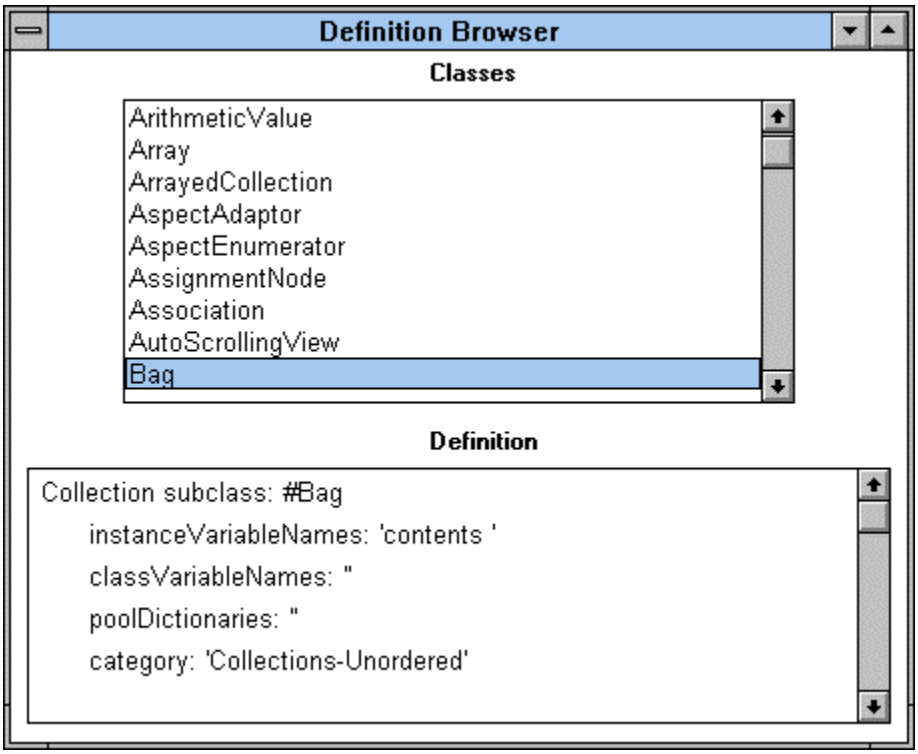


Figure 8.15. Definition browser.

Solution: As we know, class information is stored in the Smalltalk dictionary, an instance of SystemDictionary. Among its class methods is one called classNames which returns an ordered collection of symbols, the names of all classes in the library. Once we have a class, Smalltalk at: nameSymbol as in

```
Smalltalk at: #Array
```

returns the class corresponding to the symbol.

The next question is how to get the when we have the class. One way to find out is to look at the definition of Browser because the browser knows how to display a definition . Another possibility is to examine Class and other classes in its category because we know that all class information comes from this group. After a bit of searching (alternatively, you could read Appendix 3), we find that class ClassDescription has method definition which returns the definition of the class. Since ClassDescription is a superclass of Class, we can send the message directly to the class. As an example,

```
(Smalltalk at: #Array) definition
```

returns the definition of class Array in the form of a String.

We are now ready to implement our task. We paint the interface, *Install* it, and specify and *Define* the properties. The methods that we need include initialization (the window should open with the class list and changes in the selection list must cause changes in the text view), and the change method which responds to a new list selection and displays the definition of the selected class in the text view.

```
initialize
"Define the list holder for the list widget and register interest in selection changes."
        classList := SelectionInList with: Smalltalk classNames.
        classList selectionIndexHolder onChangeSend: #changedClass to: self
```

where method changedClass responds to a changed class selection in the list, and changes the text displayed in the text view (*Aspect* definition). The same effect can be achieved by specifying *Notification* properties in the Properties of the List widget in the UI Painter. The definition of the message is

```
changedClass
"Class selection has changed. Blank the text view or display the definition of the selected class."
        classList selection isNil
                ifTrue: [definition value: ''] "Nothing selected in the list view - blank the text editor."
                ifFalse: [definition value: (Smalltalk at: classList selection asSymbol) definition]
```

Multiple selection lists

Single and multiple selection lists are essentially the same and the differences between them are as follows:

- To paint a multiple selection list, paint the List widget and click the *Multiple Selections* check box on the *Details* page of the *Properties* tool.
- The model of a multiple selection list is an instance of MultiSelectionInList.
- The methods accessing selections are called selections and selectionIndexes, and selections: and selectionIndexes: respectively. Message selections returns an ordered collection containing all selected objects, message selectionIndexes returns a value holder containing a set of indices. The argument of the corresponding selections: and selectionIndexes: messages must also be a collection and a set respectively.

Example 3: Function calculator

Problem: Write an application to generate and display tables of mathematical functions selected by the user from a list (Figure 8.16). If at least one function is selected, clicking *Print* prints the values of all selected functions for arguments 1 to 10 in steps of 1 in the Transcript using the following format:

```
arg     cos             ln              log

1       0.540302        0.0             0.0
2       -0.416147       0.693147        0.30103
3       -0.989993       1.09861         0.477121
4       -0.653644       1.38629         0.60206
5       0.283662        1.60944         0.69897
6       0.96017         1.79176         0.778151
7       0.753902        1.94591         0.845098
```

```
8       -0.1455         2.07944         0.90309
9       -0.911132.19722         0.954242
10      -0.839071       2.30259         1.0
```

When no function is selected, clicking *Print* opens a warning with appropriate text.

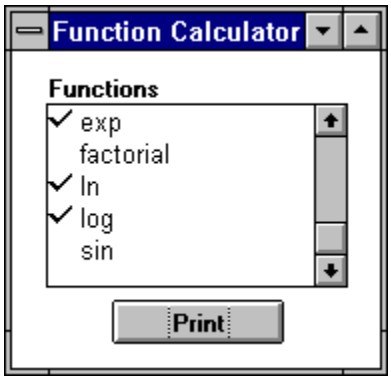


Figure 8.16. Selection window for Example 2.

Solution: The initialization method initializes the list holder variable (list *Aspect* called functions) with the names of the functions:

```
initialize
"Create list holder."
functions := MultiSelectionInList with: #( 'cos' 'exp' 'factorial' 'ln' 'log' 'sin' ).
```

When the user clicks *Print*, we enumerate over the whole interval, and for each value of the argument and for each selected function we let the argument execute the message:

```
print
"Display selected functions in Transcript."
functions selectionIndexes isEmpty ifTrue: [^Dialog warn: 'You must select at least one function.'].
“We have at least one selection. Clear Transcript and display heading.”
Transcript clear; show: 'arg'; tab.
functions selections do: [:selection | Transcript show: selection asString; tab].
Transcript cr; cr.
“Calculate value for each argument and function and print it in Transcript.”
1 to: 10 do: [:arg |
        Transcript show: arg printString; tab.
        functions selections do:
                [:function | Transcript show: (arg perform: function) printString; tab].
        Transcript cr]
```

Our examples show why message onChangeSend:to: is frequently used with single selection lists but rarely used with multiple selection lists. The reason is that it does not make sense to force the program to respond automatically to every new selection in a multiple selection list - presumably, the user will make several selections and then indicate that he or she is finished. With multiple selection lists, action is thus usually triggered by some other event such as clicking an action button.

With this example, we conclude our coverage of list widgets. They are among the most useful widgets and we will see many more examples of their use later.

Main lessons learned:

- List widgets display a list of labels and allow the user to make selections.
- Single selection lists allow one selection at a time, multiple selection lists allow any number of

selections at a time.

- The underlying data models of selection lists (their list holders) are instances of SelectionInList and MultiSelectionInList. Both hold the list of displayed items - an instance of List - and a value holder on the current selection or selections.

Exercises

1. Write tutorial describing how to create a list widget.
2. Extend the single selection list example into a simple browser that displays class names, definition of the selected class, and an alphabetized list of the class's instance methods.
3. In the previous exercise, add a pair of radio buttons labeled *instance* and *class* to allow the user to choose between class and instance methods.
4. Extend the previous exercise to display also the definition of the selected method.
5. Modify the multi-selection list example by adding function cubed.
6. Extend Example 3 to allow the user to enter the start and end values of the argument and the step.
7. In Example 3, we used variables functions and messages to hold essentially the same arrays. Remove one of them and modify the solution appropriately.
8. If you register interest in selection change, a new list selection sends a change message. Is the message sent also when the contents of the list changes?

Conclusion

In the first part of this chapter, we concluded our coverage of sequenceable (indexed) collections with ordered and sorted collections, lists, strings, text, and symbols.

Ordered collections can grow without limits and shrink (their elements can be removed) and distinguish between size and capacity. They are created with a default or specified capacity and when all slots are filled, new slots are automatically added. Growing an ordered collection is time-consuming and should be avoided by creating the ordered collection with a capacity at least equal to the maximum anticipated need. The memory space that may be wasted is generally negligible.

Ordered collections are used mainly when the number of elements is not known beforehand, or when the number of elements changes during the lifetime of the collection. They are usually accessed at the end or at the start. An important use of ordered collections is in the implementation of dependency. The major dependency protocols are defined in Object but the nature of the collection of dependents is redefined in Model which should be used as the superclass of all model objects under ordinary circumstances.

Sorted collections are ordered collections that sort their elements on the basis of a test defined by a sort block. The sort block compares two arbitrary elements of the collection and returns true if the first elements precedes the second element, and false otherwise. The built-in default sort block uses ≤ comparison but can be redefined via a class or instance message. New elements are automatically inserted into the proper place according to the sort block. Every pair of elements in the collection must be able to execute the sort block.

Class List combines the accessing mechanism of ordered collections with sorting of sorted collections, and adds dependency handling. Unlike sorted collections, lists do not sort themselves automatically but must be explicitly asked to do so. They sort themselves by sending a message to SequenceableCollectionSorter. Lists are used mainly to hold elements of multi-item GUI widgets.

Class String is an abstract sequenceable collection whose elements are character codes. Concrete strings are instances of String subclasses, normally class ByteString, the default string implementer. The concrete implementation of strings as ByteString objects is transparent and the programmer does not need to pay attention to it unless a larger character set is needed. In that case, a TwoByteString may be used. The most common protocols of String include methods for finding and replacing substrings, copying, and comparison. Much of this protocol is inherited from the abstract class CharacterArray.

Class Text adds emphasis to strings, allowing their display in different fonts, styles, and colors.

Class Symbol is a subclass of String that does not allow duplication, making its processing, in particular comparison, more efficient. The main uses of symbols are for method selectors and as names of widgets in GUI specifications.

In the second part of this chapter, we introduced two new types of selection widgets. Single-selection lists allow only one selection at a time, multiple-selection lists allow any number of selections at a time. Their basis is MultiSelectionInList for single-selection widgets, and SelectionInList for multi-selection lists. They hold a list of items and the current selection or selections. The list component of the selection in list object is an instance of List.

We recommend that you now read Appendix 2 which covers additional widgets.

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, *ByteSymbol*, *CharacterArray*, *ComposedText*, Cursor, *Document*, DocumentRenderer, **List**, **MultiSelectionInList, OrderedCollection**, Printer, RunArray, SequenceableCollectionSorter, *Screen*, **SelectionInList, SortedCollection**, **String**, **Symbol**, **Text**, TwoByteString.

Widgets introduced in this chapter

Multiple selection list, *Single selection list*.

Terms introduced in this chapter

association - object consisting of a key - value pair, instance of Association
emphasis - specification of font, style, or color to be used when displaying a string
multiple-selection list - list widget allowing any number of selections
ordered collection - used either to refer to any collection whose elements are stored in a fixed externally accessible order, or in the more restrictive sense of an instance of class OrderedCollection
single-selection list - list widget allowing at most one selection at a time
sort block - a block defining how to compare two elements when sorting a collection
sorted collection - an ordered collection whose elements are ordered according to a comparison function called the sort block
symbol - a string with unique instantiation; used mainly for method selectors
text - string with emphasis prescribing how the string should display itself