Cincom®

Cincom Smalltalk™

Basic Libraries Guide

P46-0146-03

SIMPLIFICATION THROUGH INNOVATION®

Part Number: P46-0146-03

Software Release 7.7

This document is subject to change without notice.

Cincom Systems, Inc.

55 Merchant Street

Cincinnati, Ohio 45246

Phone: (513) 612-2300

Fax: (513) 612-2000

World Wide Web: http://www.cincom.com

Contents

Chapter 6 Working with Geometric Objects 6-1

Chapter 7 Working with Text 7-1

About This Book

Overview

This document, the VisualWorks *Basic Libraries Guide*, provides an introduction to the content and use of several of the core class hierarchies provided standard with VisualWorks. The descriptions provide more than reference documentation, and are actually incomplete in that regard. Instead, they introduce the main features supported by the libraries and their use, providing a foundation for further explorations.

For complete reference style documentation, use SmalltalkDoc and browse the hierarchies.

The libraries currently covered in this document are:

- Collections
- Streams
- Numbers
- Graphics
- Geometrics
- Colors
- Text and Fonts
- Sockets
- XML Framework
- The Parcer and Compiler

Additional libraries are covered in other documents.

Audience

This guide assumes that you have at least a beginning familiarity with object-oriented programming, Smalltalk, and the VisualWorks environment. For most purposes, this background information is provided by the Application Developer's Guide.

Conventions

We have followed a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

Example	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
`cover.doc`	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).
`filename.xwd`	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File ☨ New	Indicates the name of an item (New) on a menu (File).
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.

Examples	Description
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.

Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

<Select> button	*Select* (or choose) a window location or a menu item, position the text cursor, or highlight text.
<Operate> button	Bring up a menu of *operations* that are appropriate for the current view or selection. The menu that is displayed is referred to as the *<Operate> menu*.
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks *window* (except dialogs), such as **move** and **close**. The menu that is displayed is referred to as the *<Window> menu*.

These buttons correspond to the following mouse buttons or combinations:

	3-Button	2-Button	1-Button
<Select>	Left button	Left button	Button
<Operate>	Right button	Right button	<Option>+<Select>
<Window>	Middle button	<Ctrl> + <Select>	<Command>+<Select>

Getting Help

There are many sources of technical help available to users of VisualWorks. Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and non-commercial license holders.

Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support. Cincom provides all customers with help on product installation. For other problems there are several service plans available. For more information, send email to supportweb@cincom.com.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- The *version id,* which indicates the version of the product you are using. Choose **Help** ✝ **About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id:**.
- Any modifications (*patch files*) distributed by Cincom that you have imported into the standard image. Choose **Help** ✝ **About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches:**.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **copy stack** in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

Contacting Technical Support

Cincom Technical Support provides assistance by:

Electronic Mail

To get technical assistance on VisualWorks products, send email to supportweb@cincom.com.

Web

In addition to product and company information, technical support information is available on the Cincom website:

http://supportweb.cincom.com

Telephone

Within North America, you can call Cincom Technical Support at (800) 727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time.

Outside North America, you must contact the local authorized reseller of Cincom products to find out the telephone numbers and hours for technical support.

Non-Commercial Licensees

VisualWorks Non-Commercial is provided "as is," without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are *not* alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides several resources on VisualWorks and Smalltalk:

- A mailing list for users of VisualWorks Non-Commercial, which serves a growing community of VisualWorks Non-Commercial users. To subscribe or unsubscribe, send a message to:

  vwnc-request@cs.uiuc.edu

  with the SUBJECT of "subscribe" or "unsubscribe". You can then address emails to vwnc@cs.uiuc.edu.

- A Wiki (a user-editable web site) for discussing any and all things VisualWorks related at:

  http://www.cincomsmalltalk.com/CincomSmalltalkWiki

The Usenet Smalltalk news group, comp.lang.smalltalk, carries on active discussions about Smalltalk and VisualWorks, and is a good source for advice.

Additional Sources of Information

This is but one manual in the VisualWorks library. The Cincom Smalltalk publications website:

http://www.cincomsmalltalk.com/documentation/

is a resource for the most up to date versions of VisualWorks manuals and additional information pertaining to Cincom Smalltalk.

Online Help

VisualWorks includes an online help system.

To display the online documentation browser, open the **Help** pull-down menu from the VisualWorks main menu bar and select one of the help options.

News Groups

The Smalltalk community is actively present on the internet, and willing to offer helpful advice. A common meeting place is the comp.lang.smalltalk news group. Discussion of VisualWorks and solutions to programming issues are common.

VisualWorks Wiki

A wiki server for VisualWorks is running and can be accessed at:

http://brain.cs.uiuc.edu:8080/VisualWorks.1

This is becoming an active place for exchanges of information about VisualWorks. You can ask questions and, in most cases, get a reply in a couple of days.

Commercial Publications

Smalltalk in general, and VisualWorks in particular, is supported by a large library of documents published by major publishing houses. Check your favorite technical bookstore or online book seller.

1

Collections

VisualWorks provides a wide variety of classes for operations involving collections of objects. In addition to the conventional arrays, there are bags, dictionaries, sets, linked lists, and more. These classes and operations involving them classes are discussed in this chapter.

The first section describes several main collection classes. The variety of collections is far richer than is covered here, however. Use a System Browser to explore the collection classes when you need a special kind of collection.

Iterative operations involving collections are discussed in detail in the Application Developer's Guide. A string of characters is also a collection and shares much of the behavior of other collections.

Choosing the Appropriate Class

There are nine main kinds of collections. Three of them have specialized variations. A brief description of each collection class follows, proceeding from the simplest to the more complex. As a rule of thumb, choose the simplest class that suits your purpose.

Collection class	Distinguishing features
Set	Discards duplicate elements
Bag	Tallies duplicates
Array	Integer index (and fastest access)
Interval	Integer elements in progression
OrderedCollection	Integer index; preserves the order in which elements are added
SortedCollection	Integer index; elements are sorted by user-defined algorithm (ascending order is default)
LinkedList	Each element points to the next element, for maximum efficiency of dynamic lists
Dictionary	Noninteger index; each element consists of a key-value pair for dictionary-like lookups

Set

A Set is about as close to a generic collection as you can get. No index. No sorting. It does discard duplicates, which is often useful. The fact that an instance of Set has only one special capability should not distract you from the fact that the generic behavior it inherits, as described in later sections of this chapter, includes powerful mechanisms for manipulating elements of a data set.

An IdentitySet is identical in all respects, except that it uses == for comparisons instead of =.

Bag

An instance of Bag is like a Set, except that it counts the duplicate. For each element in a Bag there is also a tally of the occurrences of that object. If each character in the word collection were an element in a Bag, for example, the tally for the element $c would be 2. Bag does not create a new element for a duplicate, but increments the counter the item.

Array

Array allows you to maintain relative positions of elements, via an integer index. In our collection example, $e can be identified by its external key, the integer 5. (In a Set or a Bag, by contrast, the position of $e is unpredictable.) As another example, if a customer name were to be stored as a collection of three elements—first, middle, and last names—it would make sense to use an Array rather than a Set because the relative positions of the elements must be preserved.

A RunArray provides efficient storage for situations in which a value is repeated consecutively over long stretches of an array. For example, the font information for a block of text is a likely candidate—a roman font would be used for many sequences of elements in the array (letters in the text), with occasional bursts of italic, bold, etc. Although RunArray responds to the same messages as Array, its internal representation avoids waste by storing an element only if it differs from the preceding element, along with a tally of that element's repetitions.

A ByteArray provides space-efficient storage for bytes. Its elements are restricted to the set of SmallIntegers from 0 to 255. WordArray is for manipulating 16-bit words; its elements can be integers from 0 to 65535.

Interval

An Interval is a finite arithmetic progression, such as the series 2 4 6 8. It is typically used to control an iterative loop, as described in the Application Developer's Guide.

OrderedCollection

An OrderedCollection, like an Array, has an integer index and accepts any object as an element. Unlike Array, however, an OrderedCollection permits elements to be added and removed freely. It is frequently used as a stack (the last element in is the first one removed) or a queue (first in, first out). However, its uses extend farther because there are so many situations in which ordering must be preserved as an arbitrary number of elements are added.

SortedCollection

When elements are not added in the desired order, sorting is required. SortedCollection provides that extra capability. By default, elements are sorted in ascending order. You can override this default by specifying an alternative sort algorithm enclosed in a block. For example, the expression:

```
SortedCollection sortBlock: [:x :y | x >= y]
```

creates a new collection whose elements will be sorted in descending order.

List

A List represents a collection of elements explicitly ordered by the sequence in which objects are added and removed. Elements are accessible by their indices. Instances of List continue to extend the valid range that can be indexed as elements are added. Lists propagate change notices to their dependents. A List is generally used with GUI widgets.

LinkedList

As its name suggests, a LinkedList is a collection in which each element points to the next element. An OrderedCollection can accomplish the same thing, but is less efficient in circumstances involving large numbers of additions and deletions. For example, the ProcessorScheduler class makes use of LinkedList to track the highly dynamic list of processes. LinkedList achieves its efficiency in a way that prohibits its elements from belonging to other collections at the same time.

Dictionary

The Dictionary class, instead of imposing an integer index on each element, permits any object to be the external key. The result, as in the familiar Webster's dictionary, is a collection of key-value pairs. For example, an element might consist of the word 'object' with the associated definition 'something solid that can be seen or touched'. Thus, each element in a Dictionary is typically an instance of Association, which is a key-value pair. The nil object is specifically excluded as a valid element.

An IdentityDictionary is similar, except that it uses == for comparisons instead of =. That is, the values in an IdentityDictionary are expected to be literals or other unique objects that can be compared with the more efficient identity operator (==)

Creating a Collection

Typically, you create an empty collection, and then add elements to it. All collections respond to the new message, as shown here for OrderedCollection.

```
| list |
list := OrderedCollection new.
list add: 'Leonardo';
   add: 'Michelangelo';
   add: 'Donatello';
   add: 'Raphael'.
^list.
```

Note that add: returns the new element. Consequently, you do not want to cascade the add: messages directly from the new message, as you might be inclined to do. Or, if you do, conclude the cascade with yourself.

For an Array, which cannot add elements, it is necessary to specify the size of the array. Each element is nil until replaced with another object.

```
| array |
array := Array new: 4.
array at: 1 put: 'Leonardo';
   at: 2 put: 'Michelangelo';
   at: 3 put: 'Donatello';
   at: 4 put: 'Raphael'.
^array.
```

Other collections can be created with an initial size as well.

To create a collection filled with a filler object, send a new:withAll: message to the desired collection class:

```
^Array new: 16 withAll: 0.
```

You can also create a collection by specifying up to four elements. This approach is typically used to create a small array. Variations of the with: message, for up to four elements, are provided in VisualWorks:

```
| array |
array := Array
   with: 'Leonardo'
   with: 'Michelangelo'
   with: 'Donatello'
   with: 'Raphael'.
```

When an array contains only literal elements, such as numbers and strings, you can also create the array using its literal form:

```
| array1 array2 |
array1 := #( 'Leonardo' 'Michelangelo' 'Donatello' 'Raphael' ).
array2 := #( 1 2 3 4 )
```

Notice the use of # to indicate that a literal is being created.

Sometimes a new collection needs to be created from an existing collection. For example, a non-growing array might need to be expanded to accommodate more elements. Or a dictionary's keys might be placed in a list for sorting.

Send a withAll: message to the desired collection class, with an expression yielding the elements of the old collection, for example:

```
OrderedCollection withAll: Smalltalk keys
```

Adding Elements

Different kinds of collections add elements in different ways. Most collections will add an element when sent an add: message with an element to add. Arrays are the exception, since they are restricted to the number of elements with which they are created. A Dictionary always adds a key-value pair.

Because the elements of a Set are each unique, adding an element that already exists in the set results in no change; duplicates are omitted. A Bag, on the other hand, adds duplicates without limit.

By default, an OrderedCollection adds new elements to the end of the collection. You can also position the additional element at the beginning of the collection, before a particular element, or before a particular index. (A Set and a Dictionary do not keep their elements in an externally visible order, so the notion of inserting a new element does not apply.)

Adding an Element to a Collection

You can add an element to most collections by sending an add: message to the collection with an object as the argument. For ordered collections, the default is to add the object at the end of the ordering. For classes such as Set, there is no meaningful position.

```
| list |
list := OrderedCollection new.
list add: 'Leonardo';
   add: 'Michelangelo';
   add: 'Donatello';
   add: 'Raphael'.
^list
```

To add an element to a Dictionary, send an at:put: message to the dictionary. The first argument is the lookup key (typically but not necessarily a Symbol). The second argument is the object to be associated with the key.

```
| dict |
dict := Dictionary new.
dict at: #Leader put: 'Leonardo';
    at: #Member1 put: 'Michelangelo';
    at: #Member2 put: 'Donatello';
    at: #Member3 put: 'Raphael'.
^dict
```

Inserting an Element at a Specific Location

Collection classes which preserve order, such as OrderedCollection, support protocol for inserting elements at specific positions.

The default position, where an object is added using the add: message, is the end of the collection. It is sometimes helpful to make this position explicit, in which case you can use the addLast: message.

To insert an element at the beginning of an ordered collection, send an addFirst: message, which the new element as the argument.

To insert an element before or after a specific element already in the collection, send an add:before: message or add:after: message to the collection. The first argument is the element to be inserted. The second argument is the element relative to which the insertion is to take place.

To insert an element at a numbered position, send an add:beforeIndex: message to the collection. The first argument is the element to be inserted. The second argument is the index of the element before which the insertion is to take place.

```
| list |
list := OrderedCollection new.
list add: 'Raphael';
    addFirst: 'Leonardo';
    add: 'Michelangelo' before: 'Raphael';
    add: 'Donatello' beforeIndex: 3.
^list
```

Adding a Collection of Elements

When a collection is used to accumulate the contents of other collections, additions can be made in batches by adding an entire collection. For ordered collections, each batch can be inserted at a specific location.

To add all members of a collection to a collection, send an addAll: message to the collection, with the collection of elements to be added as argument. The receiving collection will determine any specific behavior. For example, a Set will discard duplicate elements, and an OrderedCollection will add all elements to the end of the collection.

For an ordered collection, the addAllFirst: message inserts all members of the argument collection at the start of the collection. Similarly, List, which is a subclass of OrderedCollection used primarily with widgets, defines the addAll:beforeIndex: message inserts the collection before the position specified by the second argument.

```
| sizes totalElements |
sizes := List new: 10000.
sizes addAll: (List allInstances collect: [ :list | list size]).
sizes addAllFirst: (Dictionary allInstances collect: [ :dict | dict size]).
sizes
   addAll: (Array allInstances collect: [ :array | array size])
   beforeIndex: 2.
totalElements := 0.
sizes do: [ :sz | totalElements := totalElements + sz].
^totalElements
```

Expanding an Array

Although an Array can contain only the number of elements with which it was created, you can expand an array by creating a copy that has a new element appended to it. The copy can then be substituted for the original.

To create the copy, send a copyWith: message to the Array. The argument is the object that is to be appended to the end of the new array.

```
| array copy |
array := #( 1 2 3 4 5 6 7 8 9 ).
copy := array copyWith: 10.
array := copy.
^array
```

Removing Elements

The basic method for removing an object from a collection is to send a remove: message to the collection, with the object to be removed as argument:

```
| list |
list := OrderedCollection withAll: ColorValue constantNames.
list remove: #red.
^list
```

If the specified object is not an element in the collection, an error results. To supply an alternative action (including doing nothing) when the object is not found, send a remove:ifAbsent: message to the collection. The first argument is the object to be removed. The second argument is a block containing the action or actions. An empty block is an effective means of taking no action, so the process can continue without an error message or other action.

```
| list |
list := OrderedCollection withAll: ColorValue constantNames.
list remove: #brickRed
   ifAbsent: [Dialog warn: 'You must be kidding -- brickRed?'].
list remove: #moonbeam
   ifAbsent: [ ].
^list
```

Removing a Subcollection

The removeAll: message allows you to remove all members of one collection from a target collection. Send removeAll: to the collection from which you want elements removed. The argument is a collection containing the elements to be removed.

```
| list |
list := OrderedCollection withAll: ColorValue constantNames.
list removeAll: #( #red #green #blue ).
^list
```

If an element is not found, an error is reported.

Because removeAll: is defined in Collection, it can be used with any collections as receiver and argument.

Removing an Element or Range of Elements by Index

Ordered collections provide several messages for removing a single element at a specified position or a range of elements:

removeFirst	Removes the first element in the collection.
removeFirst:	Removes the number of elements specified by the argument from the beginning of the list.
removeLast	Removes the last element.
removeLast:	Removes the number of elements specified by the argument from the end of the list.
removeFrom:to:	Returns an Array containing only elements removed from the collection, from the starting index (first argument) to the ending index (second argument). Defined only for List.
removeFrom:to: returnElements:	Same as removeFrom:to:, except that if the third argument is false, nil is returned. This is used for efficiency if the array is not needed.

For example:

```
| list |
list := List new: 25.
1 to: 25 do: [ :i | list add: i].
list removeFirst.         "Removes 1"
list removeFirst: 5.      "Removes 2 3 4 5 6"
list removeLast.          "Removes 25"
list removeLast: 5.       "Removes 20 21 22 23 24"
list removeFrom: 8 to: 12."Removes 14 15 16 17 18"
^list
```

Removing All Elements That Pass a Test

You can remove elements from any ordered collection based on a test, by sending a removeAllSuchThat: message to the collection. The argument is a block containing the test. The block must declare one argument variable for the element to be tested.

```
| list |
list := OrderedCollection withAll: ColorValue constantNames.
list removeAllSuchThat: [ :name | name first == $r].
^list
```

Removing an Association from a Dictionary

To remove elements from a Dictionary, you remove the entire association by sending a removeKey: message to the dictionary. The argument is the key of the association that you want to remove. The removed value is returned.

```
| dict |
dict := Dictionary new.
dict at: #Leader put: 'Leonardo';
    at: #Member1 put: 'Michelangelo';
    at: #Member2 put: 'Donatello';
    at: #Member3 put: 'Raphael'.
dict removeKey: #Member2.
dict removeKey: #Villain ifAbsent: [ ].
^dict
```

If the key is not found, an error results. To provide an alternative response to the key-not-found condition, send a removeKey:ifAbsent: message to the dictionary, with a block that specifies the action to take if the key is not found. An empty block causes no action, which is the same as silently ignoring the condition.

Removing an Element from an Array

To remove occurrences of an object from an array, you create a copy of the array, omitting each occurrence of a specified object. Send a copyWithout: message to the Array. The argument is the object to be removed. The copy can then be substituted for the original array.

The copyWithout: message works for all ordered collections as well as arrays.

```
| array copy |
array := #( 1 8 3 4 5 6 7 8 9 ).
copy := array copyWithout: 8.
array := copy.
^array
```

Replacing Elements

Replacing elements in a collection is useful when the collection has sufficient structure so that its elements have a position. Indexed collections, such as List and Array, have the right structure, as do keyed collections, such as Dictionary. Unordered collections, such as a Set, do not support replacing of elements, because there is no corresponding notion of a location at which to make the replacement.

Replacing Individual Elements

Both keyed and indexed collections support an at:put: message for replacing elements. For keyed collections, such as Dictionary, the first argument is the lookup key. For indexed collections, such as List and Array, the first argument is the index of the element to be replaced. For both kinds of collection, the second argument is the object that is to replace the old element.

```
| list dict |
dict := Dictionary new.
dict at: #Leader put: 'Leonardo';
    at: #Member1 put: 'Michelangelo';
    at: #Member2 put: 'Donatello';
    at: #Member3 put: 'Raphael'.
list := List withAll: dict values.
list sort.
dict at: #Leader put: 'Rembrandt'.
list at: 1 put: 'Rembrandt'.
```

Replacing All Elements

Sequenced collections, such as List, Array, and OrderedCollection, allow you to replace all elements with a single object by sending an atAllPut: message to the collection. The argument is the object that is to replace all existing elements. This is useful, for example, in reinitializing the collection.

```
| list |
list := List new.
1 to: 10 do: [ :number | list add: number ].
list atAllPut: 0.
^list
```

Replacing Specified Elements

Sequenced collections, such as List, Array, and OrderedCollection, allow replacing several specified elements with a single object by sending an atAll:put: message to the collection. The first argument is a collection containing the index numbers of the elements to be replaced. The second argument is the object to be placed in those slots.

```
| list |
list := List new.
list
    add: 'red';
    add: 'ghoulishGreen';
    add: 'red';
    add: 'blackAndBlue'.
list atAll: #( 1 3) put: 'bloodRed'.
^list
```

Replacing All Occurrences of an Object

Sequenced collections, such as List, Array, and OrderedCollection, allow replacing of all occurrences of a specified object with another object by sending a replaceAll:with: message to the collection. The first argument is the object whose occurrences you want to replace. The second argument is the replacement object.

```
| list |
list := List new.
list
    add: 'red';
    add: 'ghoulishGreen';
    add: 'red';
    add: 'blackAndBlue'.
list replaceAll: 'red' with: 'bloodRed'.
^list
```

Replacing a Subcollection

Sequenced collections, such as List, Array, and OrderedCollection, allow replacing an interval of objects with objects from another sequenced collection by sending a replaceFrom:to:with:startingAt: message to the collection. The first and second arguments are index numbers identifying the replacement range. The with: argument is a collection containing the new elements. The startingAt: argument is the index number in the new collection at which to begin copying the replacement elements.

```
| mainList replacements |
mainList := #( 1 2 3 4 5 6 7 8 9 ).
replacements := #( 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 ).
mainList
    replaceFrom: 1
    to: mainList size
    with: replacements
    startingAt: 7.
^mainList
```

Copying Elements

A collection, like any other object, can provide a copy of itself in response to being sent a copy message. The result is a new object which is a complete copy of the original.

```
| dict1 dict2 |
dict1 := Dictionary new.
dict1 at: #Leader put: 'Leonardo';
    at: #Member1 put: 'Michelangelo';
    at: #Member2 put: 'Donatello';
    at: #Member3 put: 'Raphael'.
dict2 := dict1 copy.
```

You can then modify the copy without affecting the original.

Note, however, that the effect of making changes to the elements of the collections, rather than to the collections themselves, is different for literal and non-literal elements. Literal elements, such as numbers and strings, can be modified in one collection without affecting the other.

For a non-literal element, however, the collections hold the same object, not copies. Any changes to the object in one collection are reflected in the other as well. If you do not want this effect of the copy, you can replace each element with a copy of itself. Since this is a change to the collection itself, the change will not affect the copy.

Copying a Subcollection

For sequenced collections, such as List, Array, and OrderedCollection, send a copyFrom:to: message to copy a segment of the collection. The first argument is the starting index of the range you want to copy, and the second argument is the ending index.

```
| list copy |
list := List new.
1 to: 10 do: [ :number | list add: number ].
copy := list copyFrom: 1 to: 3.
^copy
```

Concatenating Two Collections

Like strings, sequenced collections, such as List, Array, and OrderedCollection, can be concatenated using the , (comma) message. The argument is another sequenced collection. A new collection is returned, of the same type as the first collection, containing the elements of both collections.

```
| list array combinedList |
list := List withAll: ColorValue constantNames.
array := #( #bloodRed #ghoulishGreen #blackAndBlue ).
combinedList := list, array.
^combinedList
```

Subtracting One Set from Another

Instances of Set (and its subclasses) understand subtraction. Send a -– (minus) message to the set with another set as the argument. A similar type of collection is returned, containing the elements that occur in the first set but not the second.

```
| set1 set2 |
set1 := Set withAll: ColorValue constantNames.
set2 := set1 select: [ :name |
    (name indexOfSubCollection: 'light' startingAt: 1) > 0].
^set1 - set2
```

Testing Collections

It is useful to be able to test collections for a variety of properties. The following sections describe a number of useful tests. For others, browse the collection classes.

Equality and Identity

One collection is equal (=) to another collection if it is the same type of collection, has the same number of elements, and all of the elements are equal.

This example shows that a copy is equal, but a copy with one changed element is not equal.

```
| list1 list2 test1 test2 |
list1 := List withAll: ColorValue constantNames.
list2 := list1 copy.
test1 := list1 = list2.     "true"
list2 at: 1 put: #burntOrange.
test2 := list1 = list2.     "false"
```

Testing for identity (==) determines whether two collections are the same object. While this is a very fast test, it is seldom used since two distinct collections will fail the test even if they are of the same type, have the same number of elements, and all of their elements are the same.

Getting the Number of Elements

To get the number of elements in an collection, send a size message to the collection. The return value is an integer.

```
| array |
array := ColorValue constantNames.
^array size
```

To get the number of occurences of a specific object, send an occurencesOf: message:

```
'This is a test' occurrencesOf: $e
```

Getting the Capacity

Each position in which an element can be stored is known as a slot. A collection often has more slots than elements to avoid having to expand the collection each time a new element is added. To get the number of slots in a collection, send a capacity message to the collection. The return value is an integer.

```
| set |
set := Set withAll: ColorValue constantNames.
^set capacity
```

Testing for Emptiness

Frequently, it is useful to know whether a collection is empty of elements. To test for emptiness, send an isEmpty message to the collection. The response is true when the collection has no elements and false otherwise.

```
| list |
list := List allInstances.
list isEmpty
    ifFalse: [^list first]
```

Similarly, you can test whether the collection is not empty by sending a notEmpty message.

Testing for the Presence of an Object

Any collection will answer whether it includes a specific object in response to the includes: message. It will answer true if it includes the object, and false otherwise. A Dictionary will respond to the more specific includesKey: and includesAssociation: messages.

A collection will also answer the number of instances of an object in response to an occurrencesOf: message. The returned value is an integer, zero if the object is not found.

```
| list found1 found2 |
list := List withAll: #( #red #green #blue #red #yellow #blue).
found1 := list includes: #red.
found2 := list occurrencesOf: #red.
^Array with: found1 with: found2
```

Additional messages for testing the presence of an object are contains:, allSatisfy:, and anySatisfy:.

Looping through the Elements (Iterating)

It is common for an application to perform a set of actions for each element in a collection. For example, a sales processing application might want to generate a packing slip for each element in a list of sales orders. To create a loop that repeats a series of steps for each element in a collection, send a do: message to a collection. The argument is a block that performs a series of operations on an element. The block declares one argument variable to hold the element being processed.

```
| list color |
list := List withAll: ColorValue constantNames.
list sort.
list do: [ :colorName |
    Transcript show: colorName asString; cr.
    color := ColorValue perform: colorName.
    Transcript
        show: color red printString;
        tab;
        show: color green printString;
        tab;
        show: color blue printString;
        cr; cr].
```

Occasionally the elements in a collection need to be processed in reverse order, starting with the final element and proceeding toward the first element. To do this, use the reverseDo: message instead of do:.

Additional variations of do: are available, as are other special-purpose enumerator methods.

Looping by Index or Key

For indexed collections (such as List and Array) and keyed collections (Dictionary), it is common to loop on the index or key instead of the values. This is especially useful with dictionaries, whose values are sometimes meaningless without the associated keys.

To loop on the index or key, send a keysDo: message to the collection. The argument is a block that performs a series of operations on each element. The block is expected to declare one argument variable to hold the element to be processed.

To loop on the collection and process using both the key or index and the value, send a keysAndValuesDo: message to the collection. The argument is a two-argument block that performs a series of operations on the key and associated value for each element.

```
| dict randomGenerator gc randomX randomY colorValue |
randomGenerator := Random new.
gc := (ExamplesBrowser prepareScratchWindowOfSize: 300@400)
   graphicsContext.
dict := Dictionary new.
ColorValue constantNames do: [ :colorName |
   colorValue := ColorValue perform: colorName.
   dict at: colorName put: colorValue].
dict keysDo: [ :colorName |
   randomX := randomGenerator next * 300.
   randomY := randomGenerator next * 300.
   colorName displayOn: gc at: (randomX @ randomY)].
dict keysAndValuesDo: [ :colorName :color |
   randomX := randomGenerator next * 300.
   randomY := randomGenerator next * 300.
   gc paint: color.
   colorName displayOn: gc at: (randomX @ randomY)].
```

Collecting the Results of the Processing

Frequently the results of iterating on a collection create related objects that need to be collected in a new collection. The collect: message is a shorthand way of doing this. The effect is the same as iterating with do: and explicitly creating the new collection.

```
| list capitalizedName initial |
list := List withAll: ColorValue constantNames.
list sort.
list collect: [ :colorName |
    capitalizedName := colorName asString.
    initial := (capitalizedName at: 1) asUppercase.
    capitalizedName at: 1 put: initial.
    capitalizedName].
```

Looping through Two Parallel Collections

Often two collections need to be processed in tandem. The with:do message passes corresponding elements from two ordered collections into a two-argument block. The first argument is a second ordered collection. The second argument is a two-argument block that performs a series of operations on a pair of elements, one from each of the two collections. (The example creates key-value pairs for a dictionary, taking the keys from one array and the associated values from a second array.)

```
| array1 array2 dict |
array1 := #( #Leader #Member1 #Member2 #Member3).
array2 := #( 'Leonardo' 'Michelangelo' 'Donatello' 'Raphael' ).
dict := Dictionary new.
array1 with: array2 do: [ :array1Element :array2Element |
    dict at: array1Element put: array2Element].
^dict
```

Sorting a Collection

Sorted collections can rearrange themselves either in ascending order or according to a specified sort criterion. A List has a simplified form of the sorting messages.

The sort messages assume that the elements respond to < and = messages, which are used to compare elements during the sorting.

Sort criteria are specified in a block containing the test for determining whether one element comes before another. The block is given two elements to compare, and is expected to answer true when the first element should precede the second element.

Arbitrary collections are sorted by first being converted to an instance of SortedCollection.

asSortedCollection	Returns a new collection as an instance of SortedCollection, with the collection's elements in ascending order.
asSortedCollection:	Returns a new collection as an instance of SortedCollection, with the collection's elements sorted according to the specified sort criteria.
sort	Defined for List, and returns a list with the elements sorted into ascending order.
sortWith:	Defined for List, and returns a new list with the elements sorted according to the specified sort criteria.
reverse	Returns a new collection of the same kind, but with the elements in reversed order.

For example:

```
| array1 sort1 array2 sort2 |
array1 := #( 'Leonardo' 'Michelangelo' 'Donatello' 'Raphael').
sort1 := array1 asSortedCollection.
array2 := #( 'Leonardo' 'Michelangelo' 'Donatello' 'Raphael').
sort2 := array2 asSortedCollection: [ :name1 :name2 | name1 > name2].
^Array with: sort1 with: sort2.
```

Converting Collection Types

The Collection class defines several methods for creating a specific kind of collection from any other kind of collection. The result is a new collection of the specified kind. The original collection remains unchanged. Since these conversion methods are defined in Collection, they work for all collection types.

```
| array list |
array := ColorValue constantNames.
list := array asList.
^list.
```

When converting an unordered collection, such as a `Set` or `Dictionary`, to an ordered collection, an order is imposed. One practical implication of this is that a later conversion of the same collection may return a collection with the elements in a different order, making it unequal to the first conversion.

When a `Dictionary` is converted, its keys are ignored and the new collection contains only its values.

The following are a few of the conversion methods. Browse the `Collection` class `converting` protocol for additional methods.

asArray	returns an Array
asBag	returns a Bag
asList	returns a List
asOrderedCollection	returns an OrderedCollection
asSet	returns a Set

2

Streams

Streams provide a general access mechanism for any sequencable data, regardless of the source of that data. Streams are used to read from and write to both internal and external data structures.

Stream Class Hierarchy

The Stream hierarchy contains several abstract classes. In the following partial list, only classes shown in **bold** are actually instantiated.

- Object
  - Stream
    - PeekableStream
      - **EncodedStream**
      - PositionableStream
        - ExternalStream
          - BufferedExternalStream
            - **ExternalReadStream**
              - **ExternalReadAppendStream**
              - **ExternalReadWriteStream**
            - **ExternalWriteStream**
        - InternalStream
          - **ReadStream**
          - **WriteStream**
            - **ReadWriteStream**
            - **TextStream**
  - Random
    - **FastRandom**
    - **LaggedFibonacciRandom**
    - **ParkMillerRandom**
      - **MinumumStandardRandom**

The major division of functionality is between internal and external streams. Internal streams operate on collections that are purely internal to VisualWorks. External streams operate on collections from an external data sources, such as files and network connections. ExternalStream and InternalStream are implemented as subclasses of PositionableStream, which provides the ability to maintain a position within the stream.

Within those major divisions are classes providing read and write access to the data source. Write access is not permitted to all data sources, or is limited to appending data.

EncodedStream is a wrapper class for streams on which an "encoding" has been specified. Encodings specify how characters are represented as byte values. Because of the wide variety of sources for external data, external streams are almost always wrapped in EncodedStream by the system upon creation. Internal streams, on the other hand, hardly ever need to deal with encodings. See Encoded Streams for more information.

TextStream is useful for writing text with emphases on a stream.

Random and its subclasses provide pseudo-random numeric values on a stream.

Basic Protocol

Specific behavior of streams depends on the data. However, the Stream hierarchy polymorphically defines a consistent protocol for basic stream operations.

Instance Creation

Streams are not created with the usual new message, but instead using on: and similar messages that identify the collection over which they stream. The basic messages are:

on: *aCollection*

Returns a new stream of the receiver class type on *aCollection* as the data source.

on: *aCollection* **from:** *firstIndex* **to:** *lastIndex*

Returns a new stream of the receiver class type on a copy of *aCollection* from *firstIndex* to *lastIndex*.

The pointer is positioned at the beginning of the stream (position 0), so a write operation will overwrite data starting at that point. In general, the collection is not initialized, and no assumptions are made about the availability of data, so the content of the collection is not reliable until the first read operation.

For internal streams for which it can be assumed that the collection is already full, there are these additional creation messages:

with: *aCollection*

Returns a new stream of the receiver class type on *aCollection* as the data source.

with: *aCollection* **from:** *firstIndex* **to:** *lastIndex*

Returns a new stream of the receiver class type on a copy of *aCollection* from *firstIndex* to *lastIndex*.

The pointer is positioned at the end of the stream, past the last byte of data.

Most often, however, you create the appropriate stream type by sending one of the following messages to the collection or data source:

readStream

Returns an appropriate read-only stream type on the receiver.

writeStream

Returns an appropriate write-only stream type on the receiver.

readWriteStream

Returns an appropriate read-write stream type on the receiver.

readAppendStream

Returns an appropriate read-append stream type on the receiver.

For example, to open a read stream on a file, you would send a readStream message to a Filename:

```
('..\fileList.txt' asFilename) readStream
```

which creates an ExternalReadStream on the file.

Positioning

When first created, the stream position pointer is at the beginning of the stream collection, which is position 0. Read and write operations advance the pointer, as described below.

Any PositionableStream, of which the read and write streams are all subclasses, reports its current position in response to this message:

position

Returns the current pointer position.

In a read or read-write stream, the pointer's position in the stream can also be set by sending position:. The position specified must be within the stream's current collection, or an error notification is invoked.

position: *anInteger*

Set the position pointer to *anInteger* as long as *anInteger* is within the bounds of the receiver's contents. If it is not, issue an error notification.

For a read or read-write stream, you can position by reading through an object, but without returning the contents.

reset

Set the position of the receiver to the beginning of its stream of elements.

setToEnd

Set the position of the receiver to the end of its stream of elements.

skip: *anInteger*

Move the pointer *anInteger* positions from the current position. *anInteger* may be positive or negative (but not less than -1 for encoded streams).

skipThrough: *anObject*

Skips forward through the occurrence of *anObject*, leaving the position following *anObject*. If successful, the stream itself is returned. If the object is not found the stream is positioned at the end and nil is returned.

skipThroughAll: *aCollection*

Skip forward to the next occurrence of *aCollection*, leaving the stream positioned following aCollection, and answers the receiver. If *aCollection* is not found the stream is positioned at the end and nil is returned.

skipUpTo: *anObject*

Skip forward to the next occurrence, if any, of *anObject*. If not found, answer nil. Leaves the stream positioned before *anObject*.

For non-read streams, these messages do not invoke an error, but neither do they advance the pointer, because they do not have read access to the stream.

Reading

The basic "read" message for streams is next, which returns the next available object on the stream. Additional messages provide read options.

When reading from a stream, the object at the current position is returned, then the position pointer advances to the next object.

contents

Returns a copy of the receiver's collection from 1 to readLimit.

next

Returns the object at the current position, then advances the pointer.

next: *anInteger*

Returns the next *anInteger* objects from the stream.

nextAvailable: *anInteger*

Returns the next *anInteger* elements of the receiver. If there are not enough elements available, returns as many as are available.

through: *anObject*

Returns a subcollection of the receiver from the current position to and including the first occurrence of *anObject*. If there are no occurrences, then returns through the end of the receiver.

peekFor: *anObject*

Returns a Boolean indicating whether the next object on the stream is the same as (=) *anObject*. If false, does not advance the pointer; if true, advances the pointer.

throughAll: *aCollection*

Returns a subcollection of the receiver from the current position to and including the first occurrence of *aCollection* in the receiver. If there are no occurrences, then returns through the end of the receiver.

upTo: *anObject*

Returns a subcollection of the receiver from the current position to the occurrence, if any, of *anObject*. The stream is left positioned *after anObject*. If *anObject* is not found, returns the entire remaining stream contents, and leave the stream positioned at the end.

upToAndSkipThroughAll: *aCollection*

Returns a subcollection of the receiver from the current position up to the occurrence, if any, of *aCollection*. The stream is left positioned *after* the occurrence. If no occurrence is found, returns the entire remaining stream contents, and leave the stream positioned at the end.

upToEnd

Returns the entire remaining stream contents from the current position up to the end of the stream.

Writing

The basic "write" message for streams is nextPut:, which writes its argument value onto the stream at the current position.

nextPut: *anObject*

Put *anObject* at the next position in the receiver, and return *anObject*.

nextPutAll: *aCollection*

Put each of the elements of *aCollection* starting at the current position of the receiver and return *aCollection*.

next: *anInteger* **put:** *anObject*

Put *anObject* into the next *anInteger* elements of the receiver, and return *anObject*.

When writing to a buffered external stream, such as a file stream, you should send a commit message to make sure the buffer is flushed before closing the stream, or any time you rely on written data to be available on the stream.

The following messages insert text and control characters into a stream:

cr

Insert a carriage return.

crtab

Insert a carriage return and a single tab.

crtab: *anInteger*

Insert a carriage return followed by *anInterger* number of tabs.

space

Insert a space character.

tab

Insert a tab character.

lf

Insert a linefeed character.

print: *anObject*

Writes the printString representation of *anObject* on the stream.

Closing a Stream

For internal streams, there is no need to close the stream when you are done with it. You can send a close message, but it does nothing.

For external streams, however, you should close the stream. This also releases any external resource. Then, once the references to the stream have all died, the resources can be reclaimed.

Internal Streams

Internal streams provide read/write access to collections within VisualWorks, without any dependency on an external connection. For example, Arrays or Strings whose elements you need to access sequentially or randomly by position, can be read and written using an internal stream.

Because the collection is internal, it can be assumed that the entire collection is available upon creation of the stream. Access is not buffered, so writes are immediate and flushing is not necessary. Also, internal streams do not include encoding information. These conditions make internal streams very simple to use.

Creating an Internal Stream

You can create a read, write, or read-write stream on any SequenceableCollection by sending a readStream, writeStream, or readWriteStream message to the collection. For example, assuming you both want to be able to position in a stream, which requires reading, and write to the stream, create a read-write stream on the collection:

```
array := Array with: $a with: $b with: $d with: $d.
readStrm := array readWriteStream.
readStrm position: 2.
readStrm nextPut: $c
```

You can have multiple read and/or write streams on a collection. This is useful if you are reading and writing at different positions.

```
coll := 'This is a test' copy.
readStrm := coll readStream.
writeStrm := coll writeStream.
   [readStrm atEnd]
      whileFalse:
         [ | char |
         char := readStrm next.
         writeStrm nextPut: char asUppercase].
^coll
```

In this example, using two streams avoids the need to reposition before each write.

The above creation messages leave it to the collection to determine what type of stream is appropriate for the object, which is generally the preferred approach. However, two other instance creation methods, on: and with:, are also useful.

Using on: produces the same effect as readStream, writeStream, and readWriteStream. For example, repeating the above:

```
coll := 'This is a test' copy.
readStrm := ReadStream on: coll.
writeStrm := WriteStream on: coll.
```

The pointer is set to the beginning of the collection.

Using with: differs by positioning the pointer at the end of the stream, which is useful for appending data on a write stream.

```
coll := 'This is a test' copy.
writeStrm := WriteStream with: coll.
writeStrm nextPutAll: ' of the emergency broadcast system.'.
^writeStrm contents.
```

Reading and Writing Internal Data

The ability to read from or write to a stream depends on the kind of stream. Read streams only allow reading, write streams only allow writing, and read-write streams allow both.

Since positioning in a stream requires the ability to read from the stream, the positioning messages are only supported by read and read-write streams. So, if you need to position for write operations, use a read-write stream.

The messages for reading and writing are described under Basic Protocol. For internal streams, the use of these messages is very straight-forward. A few further illustrations will suffice.

To read the next object on a read or read-write stream, send a next message, which returns the object, if any, and moves the pointer ahead. To read a number of successive objects, send a next: message with the number of objects to return. The objects are returned in a Collection, and the pointer is advanced past the last object.

```
| strm |
strm := ReadStream on: #(eliot dave sam bruce vassili tamara bob).
Transcript cr;
   show: strm next printString; cr;
   show: (strm next: 3) printString
```

Similarly, to write an object to the next position, overwriting any object currently in that place, send a nextPut: message to the write or read-write stream. To write a number of successive objects, send a nextPutAll: message with the object to write as a collection.

```
| strm |
strm := WriteStream with:
   #(eliot dave sam bruce vassili tamara bob) copy.
strm nextPut: #kevin.
strm nextPutAll: #(alan sherry sean martin).
Transcript cr; show: strm contents printString; flush.
```

To ensure that your data is written, send a flush message to the internal stream, as illustrated above.

Reading and Writing Past the End of Data

When reading through a collection with next, you eventually reach the end. As illustrated above, you can test whether the position is at the end by sending an atEnd message to the stream. The message returns true if the position is at the end of the collection, and false otherwise.

Note that if you read past the end of the collection, the returned value is nil. Because nil can be a legitimate member of a collection, testing for nil is not suitable for testing the end of the stream.

When writing past the current end of a collection, the collection grows to accommodate additional values. Note that, due to the growth algorithm, the collection returned by sending collection to the stream might be padded with nil, and so not be the collection you want. For example:

```
str := 'This is a test' copy.
rwStream := str readWriteStream.
rwStream setToEnd.
rwStream nextPutAll: ' of the emergency broadcast system.'.
^rwStream collection
```

results in a string of 78 characters in length, rather than the 49 actually required by the string. To get the correct result, you want the contents of the stream, not the string.

```
str := 'This is a test' copy.
rwStream := str readWriteStream.
rwStream setToEnd.
rwStream nextPutAll: ' of the emergency broadcast system.'.
rwStream flush.
^rwStream contents
```

Writing and Immutable Objects

Some collections are declared by VisualWorks to be “immutable,” as described in the Application Developer’s Guide. When this is the case, attempting to write to the collection will trigger a NoModificationError.

For example, a literal String is an immutable object, so attempting to write a character to it will evoke the exception:

```
str := 'This is a test'.
str writeStream nextPut: $D.
```

If you need to write to the object, create a copy of the original object. Copies are always mutable:

```
str := 'This is a test' copy.
str writeStream nextPut: $D.
```

External Streams

External streams provide read/write access to data external to VisualWorks, such as data from a file or a socket connection. While the basic read/write operations are the same as for internal streams, in some cases the behavior differs in important ways. Because you are reading and writing an external resource, a variety of considerations need to be taken into account.

For accessing connections such as for databases, sockets, or Internet connections, refer to the specific documentation for that subsystem. In this section we will use files for examples.

Note that additional protocol is added by special purpose components, such as Net Clients. Refer to the specific documentation for functionality added by these components.

Creating an External Stream

Read, write, or read-write stream creation messages are available for all I/O sources supported by VisualWorks. In addition to the usual readStream, writeStream, and readWriteStream messages, some data sources also accept appendStream and readAppendStream messages, indicating that writes go only to the end of the stream. For instance, when using a socket connection (such as HTTP), it only makes sense to write to the end of a stream, and so you would use one of these messages to create the stream.

For example, to append text to a file, create the stream by sending appendStream to a Filename:

```
file := '..\fileList.txt' asFilename.
fileStrm := file appendStream.
fileStrm nextPut: Character cr; nextPutAll: 'Some additional text'.
fileStrm commit.
```

Note that to ensure writing the buffered content out to the OS, a commit message is sent to the stream. When closing the stream with a close message, the commit is performed before closing the stream.

Also note that, when opening a write stream on a file, if the file already exists, its contents is overwritten. So, for example:

```
file := '..\fileList.txt' asFilename.
fileStrm := file writeStream.
```

immediately overwrites the file, rendering it zero length. If this is not your intention, then use either a more appropriate stream creation message, or create the stream on a new file.

To ensure that a new stream is created, you can send a newReadWriteStream or newReadAppendStream message to the filename.

By default, external streams are created in text mode. To set to binary mode for working with binary files, send a binary message to the stream:

```
file := '..\bin\win\visual.exe' asFilename.
fileStrm := file readStream binary.
```

Reading and Writing External Data

The ability to read from or write to a stream depends on the kind of stream. Read streams only allow reading, write streams only allow writing, and read-write streams allow both. For external streams, there are also append and read-append streams, which write only to the end of the stream data.

To read the next object on a read or read-write stream, send a next message, which returns the object, if any, and moves the pointer ahead. To read a number of successive objects, send a next: message with the number of objects to return. The objects are returned in a Collection, and the pointer is advanced to the first position past the last object.

```
| rStrm |
rStrm := '..\fileList.txt' asFilename readStream.
```

```
Transcript cr;
   show: rStrm next printString; cr;
   show: (rStrm next: 3) printString
```

Similarly, to write an object to the next position, overwriting any object currently in that place, send a nextPut: message to the write or read-write stream. To write a number of successive objects, send a nextPutAll: message with the object to write as a collection.

```
| wStrm |
wStrm := '..\newFile.tmp' asFilename writeStream.
#(eliot dave sam bruce vassili tamara bob) do:
   [ :name |
      wStrm nextPutAll: name printString;
         nextPut: Character cr ].
wStrm close
```

Buffered Reading and Writing

External I/O is mediated by buffers within Smalltalk, which allow reading and writing larger blocks (by default, 4K bytes) of data rather than, for instance, individual bytes. This adds efficiency to the operations, involving fewer accesses to the external resource.

BufferedExternalStream is an intermediate abstract class, between ExternalStream and the concrete external stream classes, that provides the buffering behavior.

You can have multiple read and/or write streams on a resource. This can be useful, but can also cause problems because of the buffering and interaction with the OS. For example, the following looks like it just replaces characters in a file with the uppercase versions.

```
file := '..\fileList.txt' asFilename.
readStrm := file readStream.
writeStrm := file writeStream.
   [ readStrm atEnd ] whileFalse: [
      | char |
      char := readStrm next.
      writeStrm nextPut: char asUppercase ].
^file
```

In fact, however, because the write buffer flushes when full (at 4K characters), the read stream suddenly finds itself at the end, and quits, leaving a much smaller file than expected. In a case like this, a better solution would be to write to a new file, then delete the old file and rename the new file to the original file’s name.

Reading and Writing Past the End of Data

When reading through an external data source with next, you eventually reach the end. As illustrated above, you can test whether the position is at the end by sending an atEnd message to the stream. The message returns true if the position is at the end of data, and false otherwise.

Note that if you read past the end of the data, the returned value is nil. Because nil might be a legitimate data value, testing for nil is not a reliable way to detect the end of data, though it may be a positive indicator.

When writing past the current end of a data source, the additional data is simply appended.

Positioning

Since positioning in a stream requires the ability to read from the stream, the positioning messages are only supported by read and read-write streams. So, if you need to position for write operations, use a read-write stream.

The messages for reading and writing are described under Basic Protocol.

Positioning is maintained for an external resource, and buffers are updated as necessary. For example,

```
rStrm := '..\fileList.txt' asFilename  readStream.
rStrm position: 5000.
```

reads the second 4KB worth of data into the buffer. Repositioning to a location in the earlier part of the file, reloads that earlier 4KB into the buffer:

```
rStrm position: 9.
```

Encoded Streams

When data is communicated between computers in digital form, as a stream of bits, it is often represented using octets. An octet, which consists of 8 bits, represents a value between 0 and 255, inclusive. Different encoding conventions have been established for how these values represent data. In this section we are specifically concerned with character data.

In VisualWorks, a string composed of Latin characters is generally represented using instances of class ByteString, while strings containing the most common Unicode characters are generally represented as instances of TwoByteString. For any character in a string, you can always get its code-point value by sending asInteger to the Character object.

To properly interpret character data, its encoding must of course be identified and properly handled. When streaming, VisualWorks does this by wrapping the data stream in an EncodedStream instance.

This section discusses both line-end and character encoding issues.

Line-end Conventions

Text streams mark the end of a line in some conventional way, noting the end of a record, or line of text.

Within Smalltalk there is only one line-end character, CR. Data coming from or going to external data sources, however, may need to conform to any number of conventions. For example, on Windows platforms the standard line-end is a CR-LF combination and on UNIX, Linux, and Mac OS X it is LF. Line-end conversion replaces the platform line-end with the internal Smalltalk representation for the data as it is represented within Smalltalk.

When working in a homogeneous environment, such as a network of only MS Windows systems, the default line-end convention is adequate; VisualWorks assigns a line-end convention based on the platform on which it is running. So, if accessing a local file, no line-end convention needs to be specified:

```
'..\fileList.txt' asFilename readStream.
```

Because this example is run under Windows, the platform default is used to replace the Windows CRLF with the Smalltalk CR line-end for the internal representation of the file data.

In a heterogeneous environment, however, it is best to specify a line-end handling strategy. In general, it is best to let VisualWorks handle the line-end conversions itself, by sending lineEndAuto to the stream, e.g.:

```
aStream := '\\LinuxBox\bboyer\vw7.4\fileList.txt' asFilename readStream.
aStream lineEndAuto.
```

This example illustrates a cross-platform environment, in which we are reading a file on a Linux filesystem from an MS Windows machine. Here, VisualWorks correctly identifies the line-end convention of the source, LF, and replaces it with an internal CR.

Conversion is performed both on reads and writes. So, when writing out data that has been accumulated in VisualWorks using the CR representation, VisualWorks converts that to the appropriate platform representation. Note, however, that lineEndAuto cannot be used when creating a new file, because it takes its convention from existing data. In this case, you need to know the target filesystem and its conventions.

If you do not want VisualWorks to convert the line-end representation, but retain the platform representation, send lineEndTransparent to the stream:

```
aStream := '\\LinuxBox\bboyer\vw7.4\fileList.txt' asFilename readStream.
aStream lineEndTransparent.
```

In this case, the source line-end representation, LF, is retained in the internal representation, and is not changed upon writing.

In circumstances where the line-end convention must be set explicitly, such as creating a new file in a cross-platform environment, the following messages are available:

lineEndCR, lineEndCRLF, lineEndLF

Sets the line-end conversion to the indicated convention.

For example,

```
| wStream |
wStream := '\\LinuxBox\bboyer\vw7.4\testFile.tmp' asFilename
   writeStream lineEndLF.
wStream nextPutAll: 'This is a test';
   nextPut: Character cr;
   nextPutAll: 'with a linefeed.'.
wStream close.
```

creates a new file on the Linux system from a Windows machine, with the correct platform line-end representation.

Encodings

Encodings provide a mapping between byte data and representations that are useful to an application. Encodings are used for many purposes. In VisualWorks they are used primarily for text data, to identify character set representations for data.

Instances of class EncodedStream are used as a wrapper for streams, and provides this functionality. As illustrated in the examples below, you typically create an instance of EncodedStream on a stream and specify its encoding. EncodedStream provides line-end conversion behavior just like external streams. StreamEncoder is an abstract superclass for classes that define stream encoders.

For many years, ASCII was the standard character encoding for English. In fact, this is a 7-bit encoding, with only 128 defined characters. The ISO-8859 family of encodings improved upon ASCII, using one full byte for each character to allow a range of 256 code points.

The first half (128 characters) of the ISO-8859 encodings are the ASCII character set, while the second half contains a different set of characters depending on which region the encoding is aimed at, i.e. 8859-1 is western, 8859-2 central/eastern europe, etc.

Sometimes referred to as ISO Latin 1, ISO-8859-1 is the default encoding for documents transferred via HTTP with a MIME type of ‘text/’, and each character is represented internally using a single octet.

As of 2004, the ISO-8859 character sets are no longer under development, with current standardization efforts being focused on the Universal Character Set (UCS) and Unicode (UTF-8 and UTF-16). Unicode can represent nearly all known international characters using a variable number of bytes for each.

In UTF-8, for example, a single character can require anywhere from 1 to 4 bytes when encoded. As the most compact and compatible encoding for Latin scripts, UTF-8 is becoming a de facto standard.

UCS-2 is a fixed two-byte encoding, that can represent 65536 code points. Unicode UTF-16 is an extension of UCS-2, in which each character can take either 2 bytes or 4 bytes per character.

VisualWorks provides support for the ISO-8859 family of encodings, UCS and Unicode. The available encodings are identified in the EncoderDirectory class variable defined in class StreamEncoder. For additional discussion of encoding, refer to the Internationalization Guide.

Encoding a Stream

There are two equivalent methods for creating an encoded stream.

One is to assign an encoding to the external connection, by sending a withEncoding: message to the data source, and then open a stream on that. For example:

```
('..\fileList.txt' asFilename withEncoding: #utf8) readStream
```

The withEncoding: message actually returns an encoded stream constructor of some type (a subclass of EncodedStreamConstructor), which then determines the kind of stream to create.

Filenames, ByteArrays, and ExternalConnections all support the withEncoding: message to create a stream constructor, which in turn responds to some subset of readStream, writeStream, appendStream etc.

An alternative is to explicitly create an EncodedStream instance by sending an on:encodedBy: instance creation message. The arguments are the stream and the encoding. For example:

```
EncodedStream
   on: ('..\fileList.txt' asFilename readStream)
   encodedBy: (StreamEncoder new: #utf8)
```

Both approaches return the same thing: an encoded stream on the data source with the specified encoding, but you should use on:encodedBy: only when you're trying to wrap a pre-existing stream object.

Reading and Writing

An encoded stream is generally used for character data, so data is written and read as characters (unless the stream is set to binary mode). The read and write protocol is very simple:

next

Return the next character on the stream.

nextPut: *aCharacter*

Write *aCharacter* to the next position on the stream.

For example:

```
stream := (ByteArray new withEncoding: #ascii) readWriteStream.
stream nextPut: $A.
stream nextPut: (Character value: 66).
stream position: 0.
stream next.          "$A"
```

Positioning on an Encoded Stream

Positioning an encoded stream with position: works as usual. Positioning using skip:, however, is restricted in a couple of ways.

- Skipping backward more than one character is prohibited, and
- Once having read past the end of an encoded stream, skipping back is no longer allowed.

In either case, an exception is raised.

The prohibition against skipping backwards more than one character applies to successive sends of skip: -1, as in:

```
rStream
   skip: -1;   "ok"
   skip: -1    "error"
```

as well as to sending with a smaller step:

```
rStream skip: -2
```

To capture an attempt to read past the end, and so to protect the ability to skip back one character, test for EndOfStreamNotification:

```
rStream := ('..\fileList.txt' asFilename withEncoding: #ascii) readStream .
[rStream atEnd] whileFalse: [rStream next].
[rStream next]
   on: EndOfStreamNotification
   do: [:x | Transcript cr; show: 'Attempt to read past end.']
rStream skip: -1.
```

Encoding and Decoding String Data

While the next, next:, and nextPut: messages all work with characters and strings, in fact the underlying data in an EncodedStream is held as bytes, not characters. In other words, it's always bytes at the bottom and characters at the top. This enables you to use encoded streams for simple encoding/decoding of string data.

For example, to encode a string into bytes using UTF-8 encoding, you can write it into an EncodedStream that wraps a WriteStream on a ByteArray. To fetch the UTF-8 encoded bytes, use encodedContents:

```
wStream := (ByteArray new withEncoding: #utf8) writeStream.
wStream nextPutAll: 'à voir et à revoir encore'.
wStream encodedContents.
```

The message encodedContents returns the contents of the underlying stream. As a shortcut, you may send asByteArrayEncoding: to a String, e.g.:

```
'à voir et à revoir encore' asByteArrayEncoding: #utf8.
```

To decode bytes, wrap them in a ReadStream, construct an EncodedStream on top of it, and then read from that. For example:

```
incomingBytes := #[195 160 32 118 111 105 114].
rStream := (incomingBytes withEncoding: #utf8) readStream.
rStream contents.
```

Alternately, you can use asStringEncoding: as a shortcut:

```
#[195 160 32 118 111 105 114] asStringEncoding: #utf8.
```

Stream Exceptions

Only a few exception classes are defined for streams. The are organized as follows:

```
Notification
   EndOfStreamNotification
Error
   StreamError
      IncompleteNextCountError
      PositionOutOfBoundsError
```

EndOfStreamNotification is raised on any attempt to read past the end of the stream. As illustrated under Positioning on an Encoded Stream, this can be used to protect the ability to skip backwards on an encoded stream. Generally, it is useful for specifying actions to take once the end of stream has been reached. As a test for the end of the stream, sending an atEnd test message is generally the better approach.

StreamError is a general exception raised for errors occurring in stream access. By default, stream errors are resumable.

IncompleteNextCountError is raised if a read operation requests more elements from the stream than are available. The exception parameter instance variable holds the number of elements that were read, which may be useful for determining subsequent processing.

```
coll := #( $a $b $c $d ).
rstrm := coll readStream.
[ rstrm next: 5 ]
   on: IncompleteNextCountError
   do: [ :x | Transcript cr; show: 'Not enough elements;',
      x parameter printString, ' read.' ]
```

PositionOutOfBoundsError is raised on an attempt to position beyond the bounds of the stream. The exception parameter instance variable holds the attempted position.

```
coll := #( $a $b $c $d ).
rstrm := coll readStream.
1 to: 5 do: [ :n | [ rstrm skip: 1. Transcript cr; show: 'ok' ]
   on: PositionOutOfBoundsError
   do: [ :x | Transcript cr; show: 'Out of bounds:',
      x parameter printString, '.' ]]
```

Random Numbers

A pseudo-random number can be generated by an instance of Random. This object is a kind of stream, so the next message gets the next number in the sequence.

Class Random is an abstract superclass for random number generators that provides a uniform interface for accessing random numbers, and also makes it simple to add further generators. The VisualWorks base includes three sample generators: FastRandom and MinimumStandardRandom (a subclass of ParkMillerRandom), and LaggedFibonacciRandom. For applications depending on good security, DSSRandom can be loaded and used (refer to the Security Guide for an explanation of this generator and its use).

A random stream returns a Double as value, generally between 0 and 1 but dependent on the seed value.

```
| randomStream x |
randomStream := Random new.
x := randomStream next.
^x
```

The seed: message changes the seed value, allowing you to force a specific sequence. This message is sent to an instance of Random, and restarts the sequence:

```
| randomStream x |
randomStream := Random new seed: 4.
x := randomStream next.
^x
```

The new message invokes the default random generator, which is set to LaggedFibonacciRandom. FastRandom and ParkMillerRandom are available for backward compatibility. You can easily subclass Random to implement your own generator, and make it the default if you wish.

3

Numbers

VisualWorks includes a variety of classes defining several types of numerical and related objects.

- Standard numeric types (integers, floating point, etc.) are implemented as subclasses of the Magnitude class.
- Complex numbers involve the “imaginary” number, i.
- Metanumbers allow dealing infinite and infinitary numbers, as well as determining whether an arbitrary object is a number.

Numeric Types

VisualWorks provides several number types, each defined in its own class. The basic types are:

Integer

The Integer class is an abstract superclass with two subclasses: SmallInteger and LargeInteger. LargeInteger further has subclasses LargePositiveInteger and LargeNegativeInteger. A SmallInteger is any integer in the range 229-1 (536,870,911) to -229, inclusive. Large integers are limited only by available memory. The system coerces integers into the proper subclass transparently, so you rarely need to pay attention to this issue.

Floating Point

The Float class creates instances of single-precision floating point numbers between plus and minus 1038, with eight or nine digits of precision. The Double class creates double-precision floating point numbers between plus and minus 10307, with 14 to 15 digits of precision. A floating-point number has a decimal point, at least one digit before the decimal, and at least one digit after the decimal.

Because of the imprecise way floating point numbers are represented in computer memory, mathematically equivalent representations of floating point numbers may not turn out to be equivalent in comparisons. So, for comparing numbers, avoid Float, and consider using instances of Fraction or FixedPoint instead.

Fraction

An instance of Fraction is a number with an integral numerator and denominator, separated by a division slash, as in 3/4. Fractions are always reduced to lowest terms.

Fixed Point

A fixed-point number (an instance of FixedPoint) is useful for business applications in which a fixed number of decimal places is required. Their literal representation appends the character $s to the number (e.g., 5.2s).

Three related classes, Random, Date, and Time, are described later in this chapter.

Numeric Constants

There are three numeric constants defined in VisualWorks: zero, unity, and pi. All three are returned by class methods for various numeric classes.

Zero

The zero message is defined for all numeric classes, and returns the appropriate value to ensure additive identity. The type of the zero value varies; for example, Float returns 0.0 and Integer returns 0.

To get a zero of the same class as an existing number, first get the class of that number by sending a class message to it and then send zero to the resulting object.

```
| x y z |
x := Float zero.
y := Integer zero.
z := x class zero.
^x + y + z
```

Unity

The unity message is defined for all numeric classes, and returns the appropriate value to ensure multiplicative identity. The type of one returned varies; for example, Float returns 1.0 and Integer returns 1.

To get a one of the same class as an existing number, first get the class of that number and then send unity to the resulting object.

```
| x y z |
x := Float unity.
y := Integer unity.
z := x class unity.
^x + y + z
```

Pi

The pi message is defined for Float or Double. Float returns a single-precision version while Double returns a double-precision version.

To get a pi of the same class as an existing number, first get the class of that number and then send pi to the resulting object.

```
| x y z |
x := Float pi.
y := Double pi.
z := x class pi.
^x + y + z
```

Complex Numbers

An instance of class Complex has two components, a real number such as a Float, and an imaginary number (a multiple of i, which represents the square root of -1). A Complex number is represented in the following format: (5.5 + 3 i)—white space inside the parentheses is ignored.

Support for complex numbers is an optionally loaded component. Load the **AT MetaNumerics** parcel to add this support.

An instance can be created by using the literal form shown above, or via the real:imaginary: method, as in Complex real: 5.5 imaginary: 3. When the real component is zero, sending the message i to an integer is sufficient, as in 3 i. When the imaginary component is zero, the shorter fro-mReal: method can be used. In summary, the expressions in the left column generate the Complex numbers in the right column below:

3 i	(0 + 3 i)
5.5 + 3 i	(5.5 + 3 i)
Complex fromReal: 5.5	(5.5 + 0 i)
Complex real: 5.5 imaginary: 3	(5.5 + 3 i)

Complex numbers support the usual numeric operations, including accessing, arithmetic, mathematical functions, coercion, comparison, conversion, testing, and generality. Nonequal comparison, truncation, and rounding are not valid with complex numbers. Additional methods include:

Accessing

r	Same as abs, which returns an absolute magnitude. For example, (5.5 + 3 i) r returns 6.26498.
theta	Return the angle between the receiver and the positive real axis, in radians

Arithmetic

conjugated	Reverse the sign of the imaginary component.

Converting

asPoint	Return a Point with the real component as the x value and the imaginary component as the y value.
i	Multiply the receiver by (-1 sqrt). This message is also understood by Number after MetaNum.st is filed in.

Metanumbers

The MetaNumeric class is an abstract superclass with four subclasses, as follows:

```
MetaNumeric
   Infinity
   Infinitesimal
   NotANumber
   SomeNumber
```

Support for metanumbers is an optionally loaded component. Load the **AT MetaNumerics** parcel to add this support.

Infinity and Infinitesimal are the best examples of metanumbers, providing mathematically useful objects. NotANumber and SomeNumber provide support for inquiring about the numberhood of an object.

The MetaNumeric class provides coercion and conversion support for its subclasses. Must of this support comes in the form of double dispatching methods, which bring coercion into play when two unlike numbers fail in some arithmetic or comparison operation.

For example, suppose you execute the following expression:

```
2.3 + (Infinity positive)
```

The Float method for addition doesn't know how to add infinity to a floating point number directly, so it asks the Infinity object to perform the addition. It does so by evaluating:

```
(Infinity positive) sumFromFloat: self
```

The sumFromFloat: method is implemented by MetaNumeric, the abstract superclass of Infinity. After coercing the floating point number into meta form (making it an instance of SomeNumber), the superclass hands off to Infinity to perform the specific addition. All metanumbers need to have non-metanumbers coerced to meta form, so this behavior is performed by their common superclass, MetaNumeric.

Infinity Class

Infinity represents a number too large to be represented in any other form. We will use the terms +infinity and -infinity to denote the positive and negative forms of this number.

It is defined to mean that for any real number x, the following is true:

```
-infinity < x < +infinity
```

The expression Infinity positive creates a positive instance of Infinity, and Infinity negative creates a negative instance.

The usual numeric operations are supported by Infinity, according to the following rules (where x is any real number):

```
x + +infinity = +infinity
x - +infinity = -infinity
x * +infinity = +infinity when x > 0
x * -infinity = -infinity when x > 0
0 * +infinity = 0
+infinity + +infinity = +infinity
-infinity - +infinity = -infinity
```

```
+infinity * (+/-)infinity = (+/-)infinity
-infinity * (+/-)infinity = (-/+)infinity
+infinity - +infinity = undefined value, and an error occurs
```

Because +infinity is not a single value, but a set of all real numbers that are unrepresentably large, it makes no sense to ask whether +infinity = +infinity. Doing this will cause an error.

Infinitesimal Class

infinitesimal is a number so close to zero it cannot be represented as a conventional number—it can be thought of as the reciprocal of Infinity.

Creating an instance of Infinitesimal is done exactly as with Infinity, by executing an expression such as:

```
Infinitesimal positive
Infinitesimal negative
Infinitesimal negative: aBoolean
```

We will use the terms +tiny and -tiny to denote the positive and negative forms of this number.

The usual numeric operations are supported, according to the following rules (where x is any real number unless otherwise specified):

```
x + +tiny = x when x ~= 0.
0 + +tiny = +tiny
x * +tiny = +tiny when x > 0
x * -tiny = -tiny when x > 0
0 * +tiny = 0
+tiny + +tiny = +tiny
-tiny - +tiny = -tiny
+tiny * (+/-)tiny = (+/-)tiny
-tiny * (+/-)tiny = (-/+)tiny
+tiny - +tiny = undefined value, and an error occurs
x / +infinity = +tiny when x > 0
x / +tiny = +infinity when x > 0
+tiny * +infinity = undefined value, and an error occurs
```

Loosely speaking, +tiny is not a single value, but a set of all real numbers that are unrepresentably small. As with infinity, it makes no sense to ask whether +tiny = +tiny.

NotANumber Class

An instance of NotANumber can be used as a placeholder for the result of an illegal mathematical expression, such as 8 arcSin. Since the behavior of NotANumber consists of various kinds of error signals of the form “You can’t do such-and-such with a NaN,” the result is substituting one kind of error for another. In theory, NotANumber error signals could be trapped by a signal handler at a high level in your application, which could then decide, for example, to continue with some time-consuming computation, noting the error in a log, rather than abort because of the error. NotANumber was created for the sake of completeness—along with Infinity and Infinitesimal, it is defined by IEEE in the set of floating point numbers.

To create an instance, execute NotANumber new.

NotANumber implements the common arithmetic and comparison methods, raising an error signal for each.

The printable form of an instance is “NaN” so error strings use that term, as in:

```
'Can't perform arithmetic functions on NaN'
```

SomeNumber Class

SomeNumber represents a conventional scalar number coerced into metanumeric form so it can be used in both conventional and metanumeric computations. Such a number responds to numeric operations as usual, but has the same generality as other metanumbers and can be used in metanumeric computations. It is essentially a support class for the other metanumeric classes, so it has little potential for reusability.

Converting Numeric Type

A number of type conversion messages are available. Refer to the method definitions for details of their behavior.

asFixedPoint:	returns a fixed point number with the specified number of decimal places
asFloat	returns a floating point number
asDouble	returns a double-precision floating point number
asRational	returns an integer or a fraction
asCharacter	returns the character represented by the number
printString	returns a String representation of the number
printStringRadix:	returns a String representation of the number with the specified radix (base)

Operations on Numbers

Creating a Number

Numbers are created either by a literal numerical expression or by an arithmetic operation. The kind (or class) of a number resulting from an arithmetical operation depends on the numbers involved and the operation.

The following are literal expressions for numbers:

100	integer (appropriate Integer subclass)
5.3	floating point (Float)
5.5d	double-precision floating point (Double)
3/5	fraction
99.95s	fixed point (“s” for “scale”, giving the precision)
99.95s4	fixed point, giving precision explicitly
1.555e3	exponential notation
3.955d2	double precision exponential notation. VisualWorks accepts q in place of d for compatibility with other Smalltalk systems
16r1A	radix notation: base, followed by “r”, followed by the number expressed in the base notation.

The following are arithmetical expressions for numbers:

^3 + 8	integer
3 * 100.2	floating point

As shown above, fractions are a real class of object. An alternative method for creating a fraction is to explicitly declare its numerator and denominator:

```
y := Fraction
   numerator: 3
   denominator: 4.
```

Arithmetic Operations

Arithmetic operators are defined as messages for each class of number, but each number class defines the standard operations and many more. Use the system browser to examine the messages in the arithmetic protocol for each number class for details:

+	addition
–	subtraction
*	multiplication
/	division
//	division, discarding any remainder for an integer result
\\	division, returning only the remainder
sqrt	square root
**	raise to a power (x ** 3) or taking the root (x**(1/3))
abs	absolute value
reciprocal	reciprocal value

Rounding and Truncating

There are several methods for rounding or truncating numbers. These are implemented in different numeric classes, as required.

rounded	Answer the integer nearest the receiver.
roundTo:	Answer the integer that is a multiple of the argument, aNumber, that is nearest the receiver.
truncated	Answer a SmallInteger equal to the value of the receiver without its fractional part.
truncateTo:	Answer the next multiple of the argument, aNumber, that is nearest the receiver toward zero.

Comparing Numbers

Numeric comparison operators are defined as messages for each class of number, but each number class defines the standard operations and many more. These tests all return a Boolean value:

=	equality
==	identity. Identity works only for SmallInteger, so in general test for equality instead.
~=	inequality
~~	non-identity
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
min:	returns the smaller of two numbers
max:	returns the larger of two numbers

Note that, when comparing floating point numbers (class Float), certain comparisons may give incorrect results. For example, equality and identity (= and ==) may fail between two representations that are mathematically equal. This is due to the way floating points are represented by computers, and has nothing specific to do with Smalltak or VisualWorks. For such comparisons, consider representing these numbers as Fraction or FixedPoint numbers instead.

Testing Numbers for Properties

Because variables have no declared type in VisualWorks, it is sometimes necessary to test a variable that is expected to hold a number. If it does hold a number, you can safely send arithmetic and other number messages to it.

To test whether a variable holds a number, send it a respondsToArithmetic message. If the object is a number, it responds true.

```
| x |
x := 55.
^x respondsToArithmetic
```

More specific tests are also available, such as isInteger and isReal.

A large variety of messages are available for testing for specific properties of numbers:

isInteger	tests for integers
isReal	tests, in effect, for members of subclasses of Number
even	tests for even numbers
odd	tests for odd numbers
isZero	tests for zero
positive	tests for zero or greater
strictlyPositive	test for greater than zero
negative	tests for less than zero

Mathematical Functions

VisualWorks number classes support a large number of advanced mathematical functions. Browse the number classes for details about available functions.

Factoring

Three messages are defined for Integer, providing factoring operations:

gcd:	greatest common denominator
lcm:	least common multiple
factorial	factorial

Trigonometric Functions

Trigonmetrical functions are defined to either operate on or return the value for an angle expressed in *radians*.

To convert an angle expressed in degrees to radians, send the degreesToRadians message to the number:

```
| x |
x := 45 degreesToRadians.
^x sin
```

Conversely, to convert a result angle expressed in radians to degrees, send the radiansToDegrees message:

```
| x y |
x := 45 degreesToRadians sin.
y := x arcSin radiansToDegrees.
^y
```

The functions supported are:

sin	sine
cos	cosine
tan	tangent
arcSin	ArcSine
arcCos	ArcCosine
arcTan	ArcTangent

Logarithmic Functions

Send the following unary messages to a number to perform logarithmic functions:

log	Return the base 10 logarithm
log: base	Return the logarithm for the specified base
ln	Return the natural logarithm (lowercase l)
exp	Return the exponential

4

Chronology

A timestamp represents a specific temporal point. A duration represents an interval of time. With these notions, VisualWorks implements a variety of chrononolgy classes.

Times are commonly specified relative to a time zone, which is a shift from Coordinated Universal Time (UTC) to local time. A TimeZone represents a mapping between UTC and local time.

Dates

A Date represents the date part (year, month, day) of a Timestamp.

Creating a Date

There are a variety of messages for creating a date. Browse the class instance creation methods of Date for the complete list. We will describe a few methods here.

To create a date for today's date, send a today message to the Date class.

```
| date |
date := Date today.
^date
```

It is often useful to create a date from a string, which can be done by sending a readFromString: message to Date. The argument is a string containing the month, day, and year in any of several formats. The year is always last. The month can be either a number (1 through 12) or the unique first letters of the name (case is irrelevant). The month, day, and year can be separated by a space, comma, hyphen, slash, period, or nothing:

```
Date readFromString: 'January 31, 1994'
Date readFromString: '31 January 1994'
Date readFromString: '1/31/94'
Date readFromString: '1.31.1994'
Date readFromString: '1-31-1994'
Date readFromString: '31JAN94'
```

You can create a date by specifying the day, month and year. To specify each by a number, send a newDay:monthNumber:year: message to the Date class. Alternatively, specify the month by name, send a newDay:month:year: message to Date. The month argument is the unique first letters of a month name expressed as a Symbol:

```
| date1 date2 |
date1 := Date
   newDay: 31
   monthNumber: 1
   year: 1994.
date2 := Date
   newDay: 31
   month: #Jan
   year: 1994.
^date1 = date2
```

Note that if a two-digit year is specified, the year is given in the current century, so

```
Date newDay: 2 month: 'jan' year: 52
```

Returns 1952 before the year 2000, and 2057 after 2000. To create a Date for a year prior to 1000, use newDay:year:, for example:

```
Date newDay: 136 year: 52
```

in which the number of days is specified from the start of the year.

Getting Information about a Day

Several messages retrieve information about a date. Browse the Date class for a complete set of messages:

weekday	returns the name of the week day as a Symbol, such as #Friday
dayOfMonth	returns the day number within the month
day	returns the day number within the year
asDays	returns the day number since January 1, 1901
monthName	returns the month name as a Symbol, as in #January
monthIndex	returns the number of the month
daysInMonth	returns the number of days in the month
year	returns the year number
daysInYear	returns the number of days in the year

Adding and Subtracting with Dates

Doing arithmetic with dates is supported by a number of messages.

To add a number of days to a date, send an addDays: message to the date. The argument can be a negative number:

```
| date daysToAdd |
date := Date today.
daysToAdd := 60.
^date addDays: daysToAdd
```

Similarly, you can send a subtractDays: message to the date.

To get the number of days between to dates, send a subtractDate: message to a date with the date to be subtracted as argument:

```
| date1 date2 |
date1 := Date today.
date2 := Date readFromString: '31 December 1999'.
^date2 subtractDate: date1
```

Comparing Dates

The usual numerical comparison operations can be performed on dates:

=	equality
~=	inequality
<	earlier than
<=	earlier than or equal to
>	later than
>=	later than or equal to

Formatting a Date

A date can describe itself in a string having a variety of formats. The printFormat: message takes as its argument an array containing six elements. The six elements are interpreted as follows:

- Day's position in the string (1, 2, or 3)
- Month's position in the string (1, 2, or 3)
- Year's position in the string (1, 2, or 3)
- The separator character
- Month's format: 1 (numeric), 2 (abbreviation), or 3 (full name)
- Year's format: 1 (with century) or 2 (without century)

To format a date string, send a printFormat: message to the date with a six-element array as argument specifying the formats:

```
| date |
date := Date today.
^date printFormat: #(2 1 3 $- 3 1)
```

Times

A Time represents the time part (hours, minutes, seconds) of a Timestamp.

Creating a Time

There are several methods for creating instances of Time. Browse the class methods in the Time instance creation protocol for details and the complete set.

To create a time to represent the current time, send a now message to the Time class:

```
| time |
time := Time now.
^time
```

You can create a time from a string representation by sending a readFromString: message to Time. The argument is a string containing the hours, minutes, and seconds, separated by colons. The minutes and/or seconds can be omitted. The “am/pm” designation can be omitted (“am” is the default) and can be in upper- or lowercase.

```
| times |
times := OrderedCollection new.
times
   add: (Time readFromString: '3:47:26 pm');
   add: (Time readFromString: '03:47');
   add: (Time readFromString: '::26 PM').
^times
```

In computations involving times on different dates, it is sometimes useful to represent each time as a number of seconds since midnight. At the end of the computation, you can convert the number of seconds back into an instance of Time. To convert seconds back to a time, send a fromSeconds: message to Time. The argument is the number of seconds that have elapsed since midnight:

```
| time |
time := Time fromSeconds: (60 * 60 * 4).
^time
```

Getting the Seconds, Minutes, and Hours

Time includes protocol for retrieving its number of seconds, minutes, and hours individually. Send a seconds message to the time.

```
| time scnds mins hrs|
time := Time now.
scnds := time seconds.
mins := time minutes.
hrs := time hours
```

Adding and Subtracting Times

Times can be added and subtracted.

To add times, send an addTime: message to a time. To subtract times, send a subtractTime: message to the time. The argument is either a time or a date:

```
| time1 time2 |
time1 := Time readFromString: '5'.
time2 := Time readFromString: '8:51:39 am'.
^time1 addTime: time2
```

Timestamp

Timestamp represents a moment to millisecond accuracy. It contains numeric representations of year, month, day, hour, minute, second and millisecond.

Creating Timestamp

The Timestamp class has two methods to return a new timestamp representing the current moment:

now

Returns a Timestamp representing the current local time and date.

nowUTC

Returns a Timestamp representing the current UTC time and date.

Timestamp interacts with TimeZone to provide local time.

Other instance creation methods provide creating a Timestamp from specific information, such as these (browse the instance creation method category for the full list):

fromDate: *aDate* **andTime:** *aTime*

Returns a Timestamp representing *aDate* and *aTime*.

fromMilliseconds: *milliseconds*

Returns a Timestamp from the number of milliseconds since 12:00 am, January 1, 1901.

Other classes can also return a time stamp. For example, the from a creation date of a file:

```
'visual.im' asFilename creationTimestamp
```

Comparing Timestamps

Several methods are available for comparing two time stamps. A few are shown here. Browse Timestamp instance methods for more.

< *aChronologicalValue*

Return a Boolean indicating whether the receiver is less than *aChronologicalValue* (a Date, Time, or Timestamp).

= *aChronologicalValue*

Return a Boolean indicating whether the receiver is equal to *aChronologicalValue*.

differenceFromDate: *aDate*

Returns a Duration containing the difference between the receiver and the start of *aDate*.

lessFromDate: *aDate*

Returns a Boolean indicating whether aDate precedes the date of the receiver.

TimeZone

The virtual machine microsecond clock reports time in UTC (coordinated universal time, like Greenwich Mean Time, GMT, but with leap seconds) on all platforms. The Time class converts UTC to local time with the aid of another class, TimeZone.

The primary tasks performed by a TimeZone are converting a Timestamp between local and universal time, and between counts of seconds since the beginning of Smalltalk epoch (1/1/1900).

By default, the TimeZone is set to SystemTimeZone, which accesses operating system resources to perform the conversion services. Accordingly, the time zone reflects the operating system's time zone configuration.

To invoke the current TimeZone, send a default message. That instance can then be requested to do a conversion, for example:

```
TimeZone default localToUniversal: Timestamp now
```

Browse other conversion methods in the api protocol.

As an alternative to the default SystemTimeZone, you can create a TimeZone object that stores an offset from UTC for local time, including settings for daylight savings time. You define a TimeZone instance to set these to appropriate values for your location, and then set it as the default TimeZone. Two instance creation messages are provided. The more general form is:

timeDifference: *hours* **DST:** *amount* **start:** *startHour* **end:** *endHour* **from:** *startDate* **to:** *endDate* **startDay:** *startDaySymbol*

where:

hours is the difference from UTC (e.g., -5 for Eastern time).

amount is the amount of time change for Daylight Savings Time (usually one hour).

startHour is the hour at which the change takes effect.

endHour is the hour at which the change ends.

startDate is the integer number of the latest day DST starts.

endDate is the integer number of the latest day DST ends.

startDaySymbol is the name of the day, as a Symbol, of the week when the change takes effect, prior to startDate and endDate.

To set the time zone in VisualWorks, send a setDefaultTimeZone: message to the TimeZone class, with a TimeZone instance:

```
TimeZone setDefaultTimeZone:
   (TimeZone timeDifference: -5
      DST: 1
      start: 2
      end: 2
      from: 97 "on April 7"
      to: 304 "until October 31"
      startDay: #Sunday).
```

CompositeTimeZone represents a collection of TimeZone instances for timezones in which the policy changes from year to year. The policy and conversions can then be carried out as appropriate for the year.

Regardless of whether the current TimeZone is a SystemTimeZone or a TimeZone, you access it the same way, by sending a default message, and the API methods are the same.

There are, however, differences in the behavior of the API methods depends on whether the time zone is a SystemTimeZone or a TimeZone. This is due to SystemTimeZone using operating system services, which are based on UTC with leap seconds, while TimeZone is based on GMT without leap seconds. The conversion algorithms are different, so the results will differ.

When a Timestamp is outside the range of the operating system timezone facilities, an Error is raised when attempting to do a conversion. This only occurs for SystemTimeZone, and does not happen when using TimeZone or CompositeTimeZone. Note that the ranges differ between operating systems.

Within the range of the operating system timezone, SystemTimeZone is more accurate, and so is to be the preferred implementation. TimeZone is to be preferred only:

- when you need to do conversions with Timezone that is different from the OS timezone
- when you need to convert Timestamps outside of the SystemTimeZone range

For backwards compatibility, TimeZone keeps both a default time zone and a reference time zone in the class variables DefaultTimeZone and ReferenceTimeZone, respectively. There is no longer a distinction between these.

Duration

Instances of Duration represent an interval of time. It supports resolution to the nanosecond.

The implementation is based on the ANSII Smalltalk specification, but refer to the class comments for differences.

The usual methods for creating a Duration are with unary methods implemented in Number:

```
4.2 seconds
6800 milliseconds
450 ms
800 microseconds
8 days
9 minutes
-12 hours
```

There are instance creation methods (class methods), as well, though these are seldom used.

Because Duration is a subclass of Number, durations can be added and subtracted:

```
8 seconds + 25 seconds
2 days + 4 hours
```

They can also be compared, queried for sign, have their sign changed, and be queried for the component fields (days, hours, minutes, etc).

Durations are useful for system services which need to specify time elapsed, such as Delay and profiling. For example, you can cause a wait state by sending a wait message to a Duration:

```
8 seconds wait
```

(This is shorthand for: 8 seconds aDelay wait).

A BlockClosure can be timed by sending a timeToRun message:

```
[1000 factorial] timeToRun
```

The returned value is a Duration object.

Timer

Timer performs an action every period after an initial wait period.

The initial period can be a Timestamp (absolute timer) or a Duration (relative timer).

startAfter: *initialPeriod*

Start the timer to fire after the *initialPeriod* duration.

startAt: *aTimestamp*

Start the timer to fire at *aTimestamp*.

A Timer action can fork blocks, resume suspended processes, or signal semaphores.

block: *aBlock*

Configure the timer to run *aBlock* in a new process when it fires.

process: *aProcess*

Configure the timer to resume *aProcess* when it fires. Hold on to *aProcess* so it does not garbage collect while it is waiting.

semaphore: *aSemaphore*

Configure the timer to signal *aSemaphore* when it fires. Hold on to *aSemaphore* so it does not garbage collect while it is waiting.

If given a repeat period, a timer will keep firing indefinitely after the initial wait period. If a timer does not repeat, it stops itself after the first iteration.

period: *aDuration*

Set *aDuration* between repetitions.

For example

```
timer := Timer newblock: [Transcript cr; show: 'Repeating myself'] ;
period: 5 seconds;startAt: Timestamp now
```

Convenience protocol for setting up timers is provided on the class side (and illustrated below).

Note that active (scheduled) timers can get garbage collected if not held strongly. In this case they will be de-scheduled when they finalize. However, traditionally, active Delays were held strongly. To allow mimicking the same behavior, Timers that are initialized with semaphores and processes are automatically registered to prevent their garbage collection as well. Therefore the following timer will not be collected until the action is performed:

```
Timer
   after: 10 seconds
   resume: [ Transcript cr; show: 'Time is up!' ] newProcess
```

However, this is not the case for the more general, block based actions. This allows to take advantage of automatic reclamation when desired.

```
| timer |
timer := Timer every: 0.2 seconds do: [ Transcript nextPut: $.; flush ].
3 seconds wait.
timer := nil
```

Timers can be realized using either the “classic” VM facilities or native OS facilities, where OS facilities are available. These facilities have different strengths and limitations (more detailed discussion of these can be found in the class comment of TimerSystem).

Both kinds of timers can be used simultaneously. The default choice can be configured on the TimerSystem. Specific choice for a given Timer can be forced *before* the timer is activated, using either useNativeInterface or useClassicInterface.

5

Graphical Images

An Image is a graphic object composed of a rectangular array of pixels. It is similar to a Pixmap and a Mask in many respects, the main differences being:

- An Image is stored in Smalltalk memory, so it is saved with the Smalltalk image. For that reason, a graphical image can be used as a storage device for Pixmaps and Masks.
- An Image is not a display surface, so you can't display other graphic objects on it as a means of assembling the desired picture.
- An Image can be either color-based or coverage-based, depending on its palette.

Common uses of images in an application are for cursors and icons, and increasingly as decoration for an application GUI.

VisualWorks includes support for BMP, JPEG, GIF, and XBM formats in the base. Support for PNG can be loaded from the PNGImageReader parcel.

Color Depth and Images

Class Image is an abstract class providing the general protocol for images. Its concrete subclasses provide specific representations for images of different color depths (or bits per pixel) of 1, 2, 4, 8, 16, 24, or 32.

For each pixel, an Image stores the value of the picture at that position, which is either the color value or the coverage value of the pixel.

An Image's palette can be either color-based or coverage-based (see Colors and Patterns). The type of palette determines what kind of display surface the image can be displayed on and copied to. A coverage-based Image can be displayed on any surface a Mask can, while a color-based Image can be displayed on a Window or a Pixmap. When copying a region from an Image to a display surface, however, the two objects must have similar palettes.

To create a display surface bearing an Image's contents, send asRetainedMedium to the Image. A Pixmap is returned when the Image has a color-based palette, and a Mask is returned when the palette is coverage-based. This operation is equivalent to creating a new Pixmap or Mask and then displaying the Image on it.

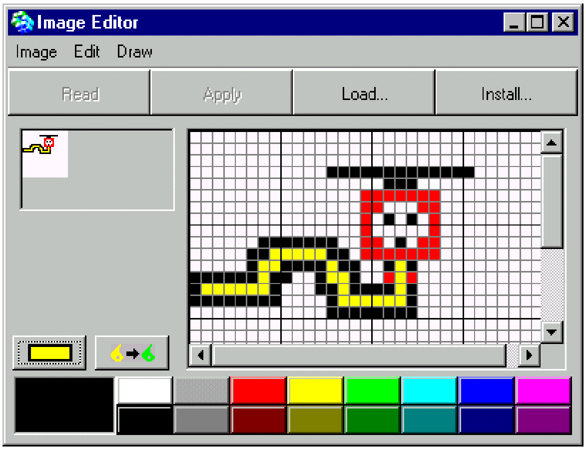
Creating a Graphic Image

A graphic image is a rectangular painting made up of colored pixels arranged in rows. Complex graphics that involve non-geometric elements are typically graphic images.

Using the Image Editor

VisualWorks includes an Image Editor that you can use to paint an image pixel by pixel, and then store it in a compilable resource method. Because of the size of the encoded image, the Image Editor is best suited for producing small images, such as for cursor shapes or icons.

To open an Image Editor, choose **Tools•Image Editor** from the VisualWorks main window.



Paint the desired image in the scrollable pixel grid. The controls are pretty standard for simple paint programs.

To make the graphic available to your application, click the **Install** button, then specify your application class as the class into which to install the graphic, and a method name for the graphic. This installs the graphic as a resource, which you can access in the resources browser. The method is installed as a class method in a resource protocol of a selected class.

Reading an Image from a File

To creating an Image from an external source, such as a file, send a fromFile: message to the ImageReader class, with the name of the file as a String. The result is an instance of the ImageReader subclass appropriate for the image format, such as GIFImageReader. To get the image from the image reader, send an image message to it. For example:

```
image := ( ImageReader fromFile: '..\bin\win\herald.bmp' ) image
```

This returns an Image instance.

It is often useful to store the image in a resource method. To do so, send an imageFromFile:toClass:selector: message, with the file name, the target class name, and the resource selector name as arguments:

```
ImageReader
   imageFromFile: 'herald.bmp'
   toClass: DummyTree
   selector: #herald
```

Capturing an Image from the Screen

You can also capture a graphic image from the screen, whether the image is in a VisualWorks window or another program's window.

The Image Editor allows you to select a relatively small area of the screen. To use its capability, open an Image Editor and choose the **Image•Capture** command. The cursor changes to a cross-hair. Move the cursor to the top left of the selection area, press the mouse button, drag the cursor to the lower-right corner, and release the mouse button. You can then edit or install the resulting image.

To capture a larger area, or to invoke the screen capture capability from your application, send a fromUser message to the Image class. The cursor changes to a cross-hair, and you can select the area as above. You will need to capture the image in a variable and process it as needed. This example simply displays it in a scratch window:

```
| gc capturedImage |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
capturedImage := Image fromUser.
capturedImage displayOn: gc.
```

Creating a Bitmap Manually

You can create an Image manually by directly editing its bits. Except for very simple graphics, this is seldom done directly. In general, you would create a tool to do this, as is provided by the Image Editor.

An Image is stored in rows that have been padded to multiples of 32 bits, called packed rows.

To manually edit an Image, you can create an intermediate ByteArray containing one byte for each pixel. In intensive applications, this wastefulness can become noticeably slow.

An alternate set of bitmap accessors operate on the packed row format directly:

packedRowAt: rowIndex
packedRowAt: rowIndex **into:** anArray
packedRowAt: rowIndex **into:** anArray **startingAt:** destinationIndex
packedRowAt: rowIndex **putAll:** anArray
packedRowAt: rowIndex **putAll:** anArray **startingAt:** sourceIndex

Use these accessors to manipulate the bit values of one packed row at a time.

Displaying an Image

As with other visual objects, an image can display itself on a graphics context. The image's palette must match that of the graphics context: coverage-based to display a Mask, color-based to display on a Window or Pixmap.

To display an image positioned at the origin (0@0), send a displayOn: message to the image with the graphics context as argument. To specify a display position other than the default 0@0, send a displayOn:at: message to the image with a Point as the second argument:

```
| gc logo |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
logo := LogoExample logo.
logo convertForGraphicsDevice: Screen default.
logo displayOn: gc.
logo displayOn: gc at: 50@50.
```

The convertForGraphicsDevice: message is necessary to ensure that the image displays properly, by making sure that the color depth and bits per pixel are correct. While it is not always required, it is strongly recommended, especially for images that are read from files.

Creating a Display Surface Bearing an Image

A common situation requires creating a hidden display surface (a Mask or Pixmap) of the same size as an image, and then displaying the image on it. The asRetainedMedium message returns a Pixmap if the image has a color-based palette, and a Mask if the image has a coverage-based palette:

```
| image pixmap |
image := LogoExample logo.
pixmap := image asRetainedMedium.
^pixmap
```

Caching an Image

A display surface such as a Pixmap or Mask, because it uses resources from the operating system, usually can be displayed on another display surface (such as a window) more quickly than an equivalent Image. However, an Image has greater longevity because it does not require a resource from the operating system, so it can be saved with the image to survive when you quit and restart VisualWorks.

A CachedImage combines the longevity of an Image with the displaying speed of a display surface. Whenever its display surface is unavailable, as when it has been destroyed by a save-and-restart operation, it is recreated from the image automatically. This relieves your application from having to recreate such display surfaces manually.

A CachedImage must be treated like a display surface, not an image. For example, you cannot rotate a CachedImage.

Create a CachedImage by sending an on: message to the CachedImage class, with the image as argument:

```
| gc logo |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
logo := CachedImage on: LogoExample logo.
logo displayOn: gc.
```

Coloring Pixels in an Image

Individual pixel colors can be changed by changing the color value at a point. The colors that you substitute, however, must exist in the image's palette.

Changing Color by Color Value

To get the current color of a pixel, send a valueAtPoint: message to the image, with a Point as argument indicating the coordinates of the pixel in the image. To set the color of a pixel, send a valueAtPoint:put: message to the image. The first argument is the location of the pixel, and the second is a color that exists in the image's palette.

```
| gc logo oldColor newColor white black |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
logo := LogoExample logo.
white := ColorValue white.
black := ColorValue black.
"Change each black pixel to white, and vice versa."
0 to: logo height -1 do: [ :y |
   0 to: logo width - 1 do: [ :x |
      oldColor := logo valueAtPoint: x@y.
      oldColor = white
         ifTrue: [newColor := black]
         ifFalse: [newColor := white].
      logo valueAtPoint: x@y put: newColor]].
logo displayOn: gc
```

Changing Color by Numeric Value

To get the current color number of a pixel, send an atPoint: message to the image. The argument is a Point indicating the coordinates of the pixel in the image. The number that identifies the pixel color in the image's palette is returned.

To change the color of a pixel, send an atPoint:put: message to the image. The first argument is the location of the pixel and the second argument is a color number that exists in the image's palette.

```
| gc logo oldColor newColor |
gc := (Examples.ExamplesBrowser
    prepareScratchWindow) graphicsContext.
logo := LogoExample logo.
"Change each black pixel to white, and vice versa."
0 to: logo height -1 do: [ :y |
    0 to: logo width - 1 do: [ :x |
        oldColor := logo atPoint: x@y.
        oldColor = 1
            ifTrue: [newColor := 0]
            ifFalse: [newColor := 1].
        logo atPoint: x@y put: newColor]].
logo displayOn: gc
```

Masking an Image

Sometimes an image contains extraneous material that needs to be removed. In the simplest case, you can mask off a rectangular area. For more complex shapes, a Mask graphical object is used.

A Mask is a DisplaySurface, and so is not saved with the Smalltalk image, so on startup has a nil value. To preserve a Mask, store it as a CachedImage with color depth 1.

Creating a Mask

The simplest way to create a mask is using the Image Editor. Select **Image • Store B&W Mask**, so this selection is checked. Then draw the mask shape and install it as a resource in your application. The areas you draw in black will allow the image to show through, and the areas in white will be transparent, allowing the background to show through.

For regular geometric shapes, you can create a mask by sending messages to the Mask class. Send an extent: message to the Mask class, with a Point as argument specifying the size of the mask. You can display the desired shape or shapes on the Mask as with a window or other display surface. In the example, a solid oval is drawn. The shapes on the mask define the visible regions of the image:

```
| ovalMask |
ovalMask := Mask extent: 66@66.
ovalMask graphicsContext
    displayWedgeBoundedBy: ovalMask bounds
    startAngle: 0
    sweepAngle: 360.
^ ovalMask
```

You can also create a mask from an image by changing the palette of the image to a coverage palette. Send a convertToCoverageWithOpaquePixel: message to the image. The argument is an integer specifying the position in the image palette of the color to make opaque, to allow the image to show through.

Masking a Rectangular Area

For masking an image to a rectangular area, you do not need to create a mask. Instead, you can simply specify the rectangle in a completeContentsOfArea: message that you send to the display surface.

1. Create a display surface (Pixmap) containing the image by sending an asRetainedMedium message to the image.
2. Send a completeContentsOfArea: message to the display surface, with a rectangle as argument.

   The copied portion is returned as an image, which can then be displayed on the graphics context.

```
| gc logo subImage pixmap copyRect |
gc := (Examples.ExamplesBrowser
    prepareScratchWindow) graphicsContext.
logo := LogoExample logo magnifiedBy: 2@2.
pixmap := logo asRetainedMedium.
copyRect := 0@0 extent:
    (logo width @ logo height / 2) rounded.
subImage := pixmap completeContentsOfArea: copyRect.
subImage displayOn: gc at: 10@10.
```

Notice the limitation to this approach, however, that part of the graphic that you might expect to be treated as background is not. This is exhibited in the example browser if the background is not white.

Masking a Nonrectangular Area

When the desired portion of an image is not rectangular, you can either create a Mask of the desired geometric shape, or specify a mask resource. The mask is then used as a stencil through which the image is displayed.

1. Create a display surface (Pixmap) for the image by sending asRetainedMedium to the image.

2. Create the desired mask, if necessary.

   The mask may be created in a resource method built by the Image Editor, in another method, or on the fly in the displaying message.

3. Send a copyArea:from:sourceOffset:destinationOffset: message to the graphics context of the destination display surface.

   The copyArea argument is the mask. The from argument is the graphics context of the source display surface. The sourceOffset argument is a Point indicating the origin of the mask when placed over the source display surface. The destinationOffset argument is the origin of the subimage when displayed on the destination display surface.

```
| gc logo pixmap ovalMask |
gc := (Examples.ExamplesBrowser
    prepareScratchWindow) graphicsContext.
logo := LogoExample logo magnifiedBy: 2@2.
pixmap := logo asRetainedMedium.
ovalMask := Mask extent: 66@66.
ovalMask graphicsContext
    displayWedgeBoundedBy: ovalMask bounds
    startAngle: 0
    sweepAngle: 360.
gc copyArea: ovalMask
    from: pixmap graphicsContext
    sourceOffset: 0@0
    destinationOffset: 10@10.
```

Modifying an Image

There are a variety of modifications you can make to images using facilities provided in VisualWorks, such as rotating and expanding.

Expanding or Shrinking an Image

You can get a copy of an image that has been magnified or shrunken in either the x dimension, the y dimension, or both.

To get an expanded copy of an image, send a magnifiedBy: message to the image. The argument is a Point whose x value is multiplied by the width of the image to derive the width of the expanded version; similarly, the y value controls the height of the expanded version.

To shrink an image, send a shrunkenBy: message to the image. The argument is a point that is used as a divisor to reduce the width and height in the shrunken version.

```
| gc logo bigLogo tinyLogo |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
logo := LogoExample logo.
bigLogo := logo magnifiedBy: 1@2.
tinyLogo := logo shrunkenBy: 1@2.
logo displayOn: gc.
bigLogo displayOn: gc at: logo extent.
tinyLogo displayOn: gc at: logo extent + bigLogo extent.
```

Flopping an Image

Sometimes you need a mirror copy of an image. The basic steps show how to get a reflected copy in which the imaginary mirror is aligned with the x axis, the y axis, or both. This process of rotating an image about the x axis or the y axis is known as flopping an image, from the photographic process in which a negative is flopped onto its backside to produce a mirror image.

To flop an image about the x axis, send a reflectedInX message to the image. To flop an image about the y axis, send a reflectedInY message. To flop an image about both axes, send a reflectedInX message followed by a reflectedInY message.

```
| gc helpImage |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
helpImage := ToolbarIconLibrary help20x20 image.
helpImage
   displayOn: gc at: 10@10.
helpImage reflectedInX
   displayOn: gc at: 60@10.
helpImage reflectedInY
   displayOn: gc at: 10@60.
helpImage reflectedInX reflectedInY
   displayOn: gc at: 60@60.
```

Rotating an Image

You can rotate an image about the z axis in 90-degree increments by sending a rotatedByQuadrants: message to the image. The argument is an integer indicating how many 90-degree rotations you want. A rotated copy of the image is returned.

```
| gc helpImage rotatedImage |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
helpImage := VisualLauncher helpIcon image.
rotatedImage := helpImage rotatedByQuadrants: 1.
helpImage
   displayOn: gc at: 10@10.
rotatedImage
   displayOn: gc at: 60@10.
```

Each rotated copy uses time and memory resources. For a series of rotations, you can reduce the resources required by reusing the same scratch image for each subsequent copy, as shown in the variant. The scratch image must be of the same size as the unrotated image, so this technique works only when all images in the series are the same size.

Create a scratch image the same size as the image that is to be rotated by sending a copyEmpty message to the original image. Then send a rotateByQuadrants:to: message to the image to be copied. The first argument is the number of quadrants to rotate the image. The second argument is the scratch image.

```
| gc helpImage scratchImage |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
helpImage := ToolbarIconLibrary help20x20 image.
scratchImage := helpImage copyEmpty.
1 to: 4 do: [ :quads |
   helpImage rotateByQuadrants: quads to: scratchImage.
   scratchImage displayOn: gc at: (60 * quads) @ 10]
```

Overlaying Images

You can achieve a variety of layering effects by combining two images and applying a filtering algorithm to the overlapping portions. VisualWorks provides 16 built-in algorithms, called combination rules. The rules are numbered 0 through 15, and the more commonly used rules have names. Thus, sending an erase message to the RasterOp class returns the combination rule for erasing shared pixels from the combined image. Combining two images involves copying a region from one image (the source) onto the other image (the destination), applying the combination rule.

Raster operations work correctly only on monochrome screens that have the most commonly used polarity characteristics. On color screens and on monochrome screens of the opposite polarity, the effects are unpredictable. Because of this, only the RasterOp over rule is portable across screen types.

To preserve the destination image in its unchanged state, first make a copy on which to merge the source image, by sending a copy message to the image (in the example, triangle).

Next, send a copy:from:in:rule: message to the copy. The copy argument is a rectangle identifying the region in the destination image to be merged with the source image (the lower part of the triangle). The from argument is the origin of the rectangle within the source image (the origin of the circle, because we want to copy the entire circle). The in argument is the source image. The rule argument is an integer identifying a combination rule (which can be derived by sending and, over, erase, reverse, under, or reverseUnder to the RasterOp class).

```
| gc triangle circle scratch |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
triangle := Pixmap extent: 50@100.
triangle graphicsContext
   displayPolygon: (Array
      with: 0@0
      with: 0@50
      with: 50@50).
triangle := triangle asImage.
circle := Pixmap extent: 50@50.
circle graphicsContext
   displayDotOfDiameter: 50
   at: 25@25.
circle := circle asImage.
0 to: 15 do: [ :rule |
   scratch := triangle copy.
   scratch
      copy: (0@20 extent: 50@50)
      from: 0@0
      in: circle
      rule: rule.
   scratch displayOn: gc at: (50 * rule \\ 400) @ (50 * rule // 400 *
   100)]
```

6

Working with Geometric Objects

VisualWorks implements several types of geometric objects, in subclasses of Geometric.

- A LineSegment connects two points, named start and end.
- A Polyline connects three or more points (its collection of vertices) as a series of line segments, and is closed between the start and end points. A polygon is a Polyline that is filled rather than stroked.
- A Rectangle represents a rectangular region whose axes are aligned with the x and y axes. Rectangles are frequently used to describe areas of a screen, but can also be used as a geometric shape.
- An ElipticalArc is a curved line defined by three parameters:
  - The smallest rectangle that can contain the ellipse of which the arc is a segment (adjusted for line width).
  - The angle at which the arc begins, measured in degrees clockwise from the 3 o'clock position (or counterclockwise for negative values).
  - The angle traversed by the arc, known as the sweep angle. The sweep angle is measured from the starting angle and proceeds clockwise for positive values and counterclockwise for negative values.
- A Bezier is a curve between two endpoints, with a control point for each endpoint determining the angle of the curve at that endpoint.

- A Circle is a circle, specified by a center and radius.
- A Spline is a curve interpolated through a series of points.

Geometric Objects

This section introduces the classes of geometric objects, all defined as subclasses of Geometric. Many of the same operations are defined for each class, and are described together later. This section will include operations specific to the classes, if any.

Rectangles

Rectangles are used in a variety of graphic operations, from setting the size of a window to specifying the bounding box of an ellipse, as well as simply to create a rectangular graphic. Accordingly, rectangles figure prominently in the discussion of the VisualWorks graphics framework in the Application Developer's Guide. In this section we focus on rectangles simply as geometric objects.

Creating a Rectangle

There are several ways to create a Rectangle, accommodating a variety of contexts.

One of the most common methods are to send an extent: or corner: message to an origin (top left) Point. Both of the following expressions create a rectangle 100 pixels wide, 250 pixels high, with its origin at 50@50:

```
50@50 extent: 100@250
50@50 corner: 150@300
```

The extent: message specifies the rectangle by its size, setting the x and y distance from the starting point. The corner: message, on the other hand, specifies the absolute corner position.

Most instance creation methods are defined on the Rectangle class itself. Similar to the above are the origin:extent: and origin:corner: messages which work the same way:

```
Rectangle origin: 50@50 extent: 100@250
Rectangle origin: 50@50 corner: 150@300
```

Instead of specifying the top left and bottom right as points, you can specify the x- and y-values of the four sides:

```
Rectangle left: 50 right: 300 top: 50 bottom: 150
```

And if you prefer not to distinguish between the origin and the corner point, you can let Rectangle do the comparison and create an instance:

```
Rectangle vertex: 300@150 vertex: 50@50
```

These are only a few of the instance creation methods available. Browse the Rectangle instance creation methods to see the whole set.

There are also a number of messages that return a new Rectangle based on a model Rectangle.

align: *aPoint1* **with:** *aPoint2*

Answer a new Rectangle with the same dimensions as the receiver, but translated by *aPoint2* - *aPoint1*.

expandedBy: *aScalarPointOrRectangle*

Answer a Rectangle that is outset from the receiver by the argument, which is a Rectangle, Point, or scalar.

insetBy: *aScalarPointOrRectangle*

Answer a Rectangle that is inset from the receiver by the argumetn, which is a Rectangle, Point, or scalar.

insetOriginBy: *origin* **cornerBy:** *corner*

Answer a new Rectangle that is inset from the receiver by the amounts in *origin* and *corner*.

merge: *aRectangle*

Answer a new Rectangle that contains both the receiver and the *aRectangle*.

translatedBy: *aScalarOrPoint*

Answer a new Rectangle translated by *aScalarOrPoint*.

A common use for these is to create the model Rectangle with the desired dimensions, then create a new Rectangle positioned more appropriately, and use the new Rectangle discarding the model. For example, to create a rectangle aligned with another rectangle:

```
| gc rect1 rect2 modelRect |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
rect1 := Rectangle origin: 10@10 corner: 50@50.
modelRect := Rectangle origin: 0@0 extent: 75@100.
rect2 := modelRect align: modelRect topLeft with: rect1 bottomLeft.
rect1 displayStrokedOn: gc.
rect2 displayStrokedOn: gc
```

Getting and Setting a Rectangle's Dimensions

Once created, a Rectangle can tell you a number of things about its dimensions and its contents.

Internally, a Rectangle is defined by its origin and corner points, held in its origin and corner instance variables.

origin

Answer the origin point.

corner

Answer the corner point.

You can change the size and position of the Rectangle with these corresponding messages:

origin: *aPoint*

Set the origin point to *aPoint*.

corner: *aPoint*

Set the corner point to *aPoint*.

origin: *aPoint* **corner:** *anotherPoint*

Set the origin point to *aPoint* and the corner point to *anotherPoint*.

A variety of other messages are available to getting and setting the Rectangle dimensions. For example, the size can be changed by setting the positions of the sides of the Rectangle.

left: *xDimension*

Set the position of the left side to *xDimension*.

top: *yDimension*

Set the position of the top side to *yDimension*.

right: *xDimension*

Set the position of the right side to *xDimension*.

bottom: *yDimension*

Set the position of the bottom side to *yDimension*.

Browse the accessing method category for additional messages.

Other useful information about a Rectangle can be accessed with these messages.

area

Answers the receiver's area, the product of its width and height.

height

Answer the height of the receiver.

width

Answer the width of the receiver.

The height and width can also be set, and the size of the Rectangle is adjusted relative to the origin.

Moving a Rectangle

In addition to being able to create a new rectangle that conforms to specified conditions, it is often useful to be able to move an existing rectangle. This ability is provided by two messages:

moveBy: *aPoint*

Change the corner positions of the receiver so that its area translates by the amount defined by *aPoint*.

moveTo: *aPoint*

Change the corners of the receiver so that its top left position is *aPoint*.

Testing Rectangle Relations

It is often necessary or useful to know whether a rectangle contains a point or an area (another Rectangle). These messages provide this information.

areasOutside: *aRectangle*

Answer a Collection of Rectangles comprising the parts of the receiver that do not lie within *aRectangle*.

contains: *aRectangle*

Answer true if the receiver is equal to or entirely contains *aRectangle*, and false otherwise.

containsPoint: *aPoint*

Answers true if *aPoint* is within the receiver, inclusive of the Rectangle itself, and false otherwise.

intersect: *aRectangle*

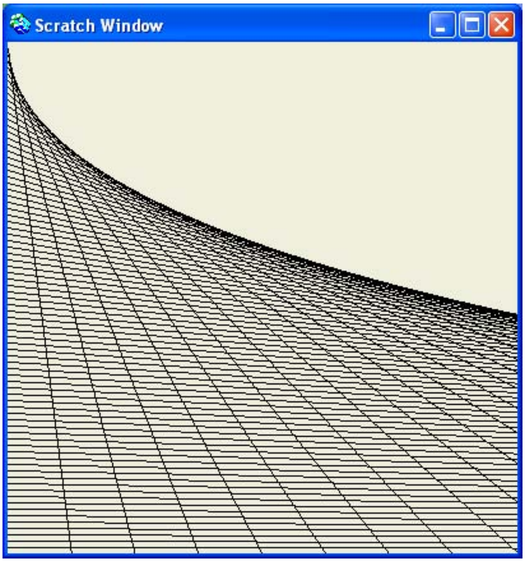
Answer a Rectangle that is the area in which the receiver overlaps with *aRectangle*. Note, if the receiver and the argument do not intersect, then the resulting rectangle will have negative width or height.

intersects: *aRectangle*

Answers true if *aRectangle* intersects the receiver at any point, and false otherwise.

Lines

A straight line is represented by an instance of LineSegment, which is simply a straight line between two points.

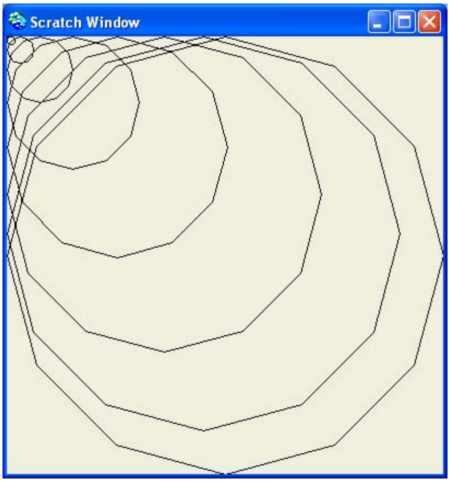


To create a line segment, send a from:to: message to the LineSegment class. The first argument is the starting point of the line and the second argument is the endpoint.

```
| gc line scaleFactor |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
scaleFactor := 10@1.
5 to: 400 by: 5 do: [ :i |
   line := LineSegment from: 0@i to: i@400.
   line := line scaledBy: scaleFactor.
   line displayStrokedOn: gc].
```

Polylines and Polygons

A jointed line, or polyline, is created as an instance of Polyline. A polygon is a filled PolyLine.



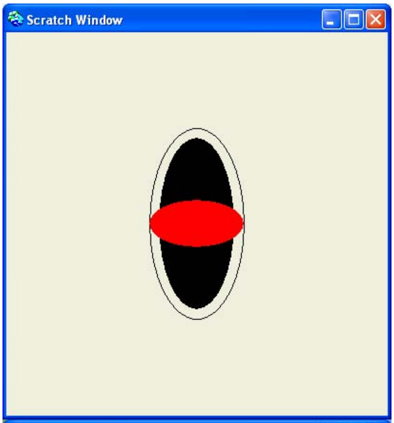
To create and display a polyline object, create a Polyline by sending a vertices: message to the Polyline class, with a collection of points (vertices) as the argument. Then wrap the polyline in a stroking wrapper and display it on the graphics context by sending displayStrokedOn:.

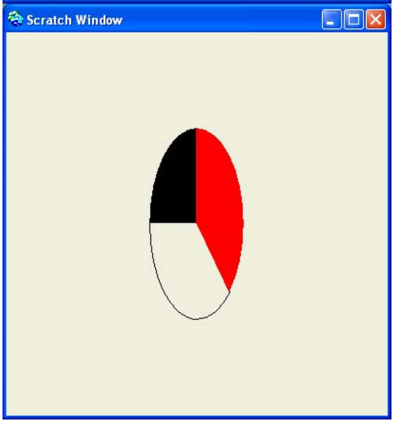
```
| gc points x y radians polyline |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
points := OrderedCollection new.
0 to: 360 by: 30 do: [ :angle |
   radians := angle degreesToRadians.
   x := 200 - (200 * radians cos).
   y := 200 - (200 * radians sin).
```

```
    points add: x@y].
polyline := Polyline vertices: points.
polyline displayStrokedOn: gc.
0.9 to: 0.1 by: -0.1 do: [ :scale |
    polyline := polyline scaledBy: scale.
    polyline displayStrokedOn: gc].
```

To fill the polyline, make the Polyline, then wrap it in a filling wrapper and display it by sending displayFilledOn: to the wrapper with the graphics context as argument.

Arcs and Ellipses





An arc is a curved line defined by three elements of information:

- The smallest rectangle that can contain the ellipse of which the arc is a segment (adjusted for line width).

- The angle at which the arc begins, measured in degrees clockwise from the 3 o'clock position (or counterclockwise for negative values).
- The angle traversed by the arc, known as the sweep angle. The sweep angle is measured from the starting angle (not necessarily the 3 o'clock position) and proceeds clockwise for positive values and counterclockwise for negative values.

An ellipse is an arc with a sweep angle of 360 degrees. An ellipse with a square bounding box describes a circle.

If the arc does not describe a closed ellipse, the ends of the arc are connected to the center of the ellipse to define the filling region, forming a wedge.

To create either an arc or an ellipse, create an instance of EllipticalArc by sending a boundingBox:startAngle:sweepAngle: message to the class, specifying the rectangle that encloses it, the beginning angle, and the number of degrees traversed (the sweep angle) from that starting angle.

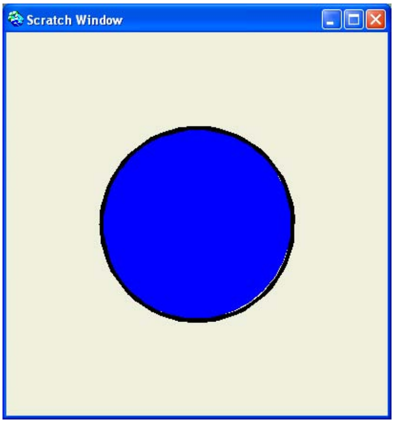
```
| gc arc box |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
box := 150@100 extent: 100@200.
"Black stroked arc"
arc := EllipticalArc boundingBox: box
   startAngle: 45
   sweepAngle: 135.
arc displayStrokedOn: gc.
"Black filled arc"
arc := EllipticalArc boundingBox: box
   startAngle: 180
   sweepAngle: 90.
arc displayFilledOn: gc.
"Red arc"arc := EllipticalArc boundingBox: box
   startAngle: 270
   sweepAngle: 135.
arc displayFilledOn: (gc paint: ColorValue red)
```

For a complete ellipse, the angle is 360, regardless of the start angle.

```
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
"Black stroked ellipse"
ellipse := EllipticalArc boundingBox: (150@100 extent: 100@200)
   startAngle: 0
   sweepAngle: 360.
```

```
ellipse displayStrokedOn: gc.
"Black filled ellipse"
ellipse := EllipticalArc boundingBox: (160@110 extent: 80@180)
   startAngle: -45
   sweepAngle: 360.
ellipse displayFilledOn: gc.
"Red ellipse"
ellipse := EllipticalArc boundingBox: (150@175 extent: 100@50)
   startAngle: 45
   sweepAngle: 360.
ellipse displayFilledOn: (gc paint: ColorValue red)
```

Circles and Dots



A circle is created by specifying its center point and radius.

```
| gc circle |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
"Blue filled circle"
circle := Circle center: 200@200 radius: 100.
circle displayFilledOn: (gc paint: ColorValue blue).
"Black stroked circle"
gc paint: ColorValue black; lineWidth: 2.
circle displayStrokedOn: gc.
```

Graphics contexts understand a displayDotOfDiameter:at: message, which displays a filled circle with the specified diameter and center point. This can be used, for example, to display points, which are not otherwise displayable objects:

```
| gc random points |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
random := Random new.
points := OrderedCollection new.
"Create 10000 random points in a 100-pixel square."
10000 timesRepeat: [
   points add: ((random next * 100) @ (random next * 100))].
"Display each random point."
points do: [ :pt |
   gc displayDotOfDiameter: 2 at:  pt * 4]
```

Curved Lines

Besides circular and elliptical arcs, VisualWorks provides two kinds of smooth curve: Spline and Bezier.

A Spline is similar to a polyline in that it connects a collection of vertices, except that it smooths the corners.

```
| gc points spline random x y |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
points := OrderedCollection new.
random := Random new.
"Collect 10 random points."
10 timesRepeat: [
   x := random next * 400.
   y := random next * 400.
   points add: x@y.
   gc displayDotOfDiameter: 8 at: points last].
spline := Spline controlPoints: points.
spline displayStrokedOn: gc.
```

A Bezier curve is similar to a line segment, in that it has a start and an end point, but it also has two control points that determine the curve angle. Each control point causes the line to curve toward it, as if exerting gravity on the line.

```
| gc points bezier random x y |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
points := OrderedCollection new.
random := Random new.
"Collect 10 random points."
4 timesRepeat: [
   x := random next * 400.
   y := random next * 400.
   points add: x@y.
```

```
    gc displayDotOfDiameter: 8 at: points last].
bezier := Bezier
    start: (points at: 1)
    end: (points at: 2)
    controlPoint1: (points at: 3)
    controlPoint2: (points at: 4).
bezier asStroker displayOn: gc.
```

Splines and Bezier curves support comparison, intersection testing, scaling, and transforming. A Spline can also be asked whether it folds back on itself (isCyclic).

When drawing either a Spline or a Bezier, the curve is actually approximated as a Polyline of some number of short line segments. To some extent you can specify the number of segments, by setting the value of the flatness instance variable for a Spline, or the scaledFlatness instance variable for a Bezier. The are used as divisors on the number of control points, so a larger number reduces the number of segments, increasing the degree of “flatness.” See the demoFlatness class methods in each class for examples.

Drawing a Geometric Object

As illustrated in the examples of the individual geometric objects, geometrics can be drawn either as line drawings, or “stroked,” or as solid objects or “filled.” By themselves, the geometrics do not know whether they are line drawing or solids, although some, such as lines, can only be line drawings.

There are two ways of specifying the drawing style for geometrics: either using the display message specifying that style, or explicitly wrapping the geometric in a StrokingWrapper or FillingWrapper. The approach you select depends on whether the shape will be displayed once, without needing to do any other operations on it, or whether the shape needs to be operated on and displayed or refreshed repeatedly.

Geometric objects support a pair of messages for directly displaying themselves on display surfaces:

displayFilledOn: *aGraphicsContext*

Displays the geometric on *aGraphicsContext* as a solid. Not all geometrics implement this (LineSegment, Bezier, and Spline)

displayStrokedOn: *aGraphicsContext*

Displays the geometric on *aGraphicsContext* as a line drawing.

These messages provide a convenient method for displaying graphical objects.

These messages have been demonstrated in the previous sections.

Using a Drawing Style Wrapper

A more flexible mechanism is to display such objects using a wrapper object, either in a StrokingWrapper or in a FillingWrapper object, to determine the drawing style. Both wrapper objects use a single message to display themselves: displayOn:. Using the wrapper technique allows VisualWorks to provide a uniform display interface for all geometric objects.

The choice of Wrapping object depends on whether the drawing should be a line drawing (StrokingWrapper), or should be filled with a color or pattern (FillingWrapper). Some objects, such as lines, cannot be wrapped in a filling wrapper since that would clearly be inappropriate.

To display any geometric object, create the object and perform any needed transformations on it. Then create a wrapper for the geometric object by sending it one of these message:

asStroker

Wrap the geometric for display as a line drawing.

asFiller

Wrap the geometric for display as a solid.

To display the wrapper object, send the displayOn: message to it, with the target graphic context as its argument.

For example, the following expression creates a line, performs some operations on it, wraps the line, and displays it in an examples browser:

```
| gc line scaleFactor |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
scaleFactor := 10@1.
5 to: 400 by: 5 do: [ :i |
   line := LineSegment from: 0@i to: i@400.
```

```
    line := line scaledBy: scaleFactor.
    line asStroker displayOn: gc].
```

When displaying a filled object, you must also specify the color for the filler:

```
| gc rect1 rect2 border |
gc := (Examples.ExamplesBrowser
    prepareScratchWindow) graphicsContext.
"Black rectangle"
rect1 := 100@100 extent: 200@200.
rect1 asFiller displayOn: gc.
"Gray rectangle"
border := 3.
rect2 := (rect1 origin + border) corner: (rect1 corner - border).
rect2 asFiller displayOn: (gc paint: ColorValue green).
```

Drawing Transient Shapes

If you do not want to create the geometric object itself at all, but simply to draw a shape, GraphicsContext recognizes a number of messages to do this. The arguments for the messages are recognizable from the usual creation methods for the relevant geometric. The following is a sampling. Browse the GraphicsContext displaying method category for additional messages.

displayArcBoundedBy: *aRectangle* **startAngle:** *start* **sweepAngle:** *sweep*

Draws a stroked EllipticalArc on the graphics context.

displayWedgeBoundedBy: *aRectangle* **startAngle:** *start* **sweepAngle:** *sweep*

Draws a filled EllipticalArc on the graphics context.

displayLineFrom: *start* **to:** *end*

Draws a LineSegment (stroked).

No geometric object is actually created by these messages, so no transformations or other operations can be performed.

```
| gc |
gc := (Examples.ExamplesBrowser
     prepareScratchWindow) graphicsContext.
5 to: 400 by: 5 do: [ :i |
    gc displayLineFrom: 0@i to: i@400].
```

Transformations on Geometrics

Geometrics respond to two standard transformations: scaling and translation. Rectangles respond to many more transformations, as described under Rectangles. The two common messages are:

scaledBy: *aScalarOrPoint*

Answer a new Geometric scaled by the argument amount, which can be a Point or a scalar value.

translatedBy: *aScalarOrPoint*

Answer a Geometric translated within the graphics context by *aScalarOrPoint*, which can be a Point or a scalar value.

Note that these return new geometric objects of the same type as the receiver, rather than transform the receiver itself.

Storing Graphic Attributes

The graphics context holds general display properties, such as line width and paint policies. These attributes provide the default properties for any object rendered on that graphics context.

However, frequently the attributes need to be different for individual graphical objects, for instance to draw lines of different width. When the attribute properly belongs to the object rather than the context, it is desirable to store it with the object. This ability is provided by wrapper classes for the various objects, which allows encapsulating graphical attributes with the graphical object.

More general and for graphical objects other than geometrics, but applicable to geometrics as well, is the GraphicsAttributesWrapper.

It is frequently necessary to store color information with the graphic object. To do this, wrap the geometric object in a GraphicsAttributes-Wrapper.

1. Wrap the geometric object in a stroking or filling wrapper by sending asStroker or asFiller to it.
2. Wrap the stroking or filling wrapper in a GraphicsAttributes-Wrapper by sending an on: message to that class, with the wrapper from the basic step as the argument.

3. Create a new GraphicsAttributes and send a paint: message to it. The argument is a color or pattern.
4. Install the graphics attributes in the GraphicsAttributesWrapper by sending an attributes: message with the attributes as the argument.
5. Display the graphics attributes wrapper by sending a displayOn:at: message to it. The first argument is the graphics context of the display surface. The second argument is the origin point at which the geometric object is to be displayed.

```
| gc circle wrapper1 wrapper2 random pt attributes1 attributes2 |
gc := (Examples.ExamplesBrowser
    prepareScratchWindow) graphicsContext.
circle := Circle center: 0@0 radius: 50.
wrapper1 := GraphicsAttributesWrapper on: circle asFiller.
attributes1 := GraphicsAttributes new paint: ColorValue red.
wrapper1 attributes: attributes1.
wrapper2 := GraphicsAttributesWrapper on: circle asFiller.
attributes2 := GraphicsAttributes new paint: ColorValue blue.
wrapper2 attributes: attributes2.
random := Random new.
100 timesRepeat: [
    pt := random next * 300 + 50 @ (random next * 300 + 50).
    wrapper1 displayOn: gc at: pt.
    pt := random next * 300 + 50 @ (random next * 300 + 50).
    wrapper2 displayOn: gc at: pt]
```

7

Working with Text

Characters and strings are primarily managed by the two classes Character and String. This chapter discusses operations at the character level first, followed by string operations.

The final section places Character and String in the context of their abstract superclasses and, in the case of String its concrete subclasses.

As a collection of characters, a string responds to the messages described in Collections The more pertinent behavior is reviewed in this chapter.

Characters

Character objects are instances of the class Character. As with all objects in Smalltalk, and unlike many languages, characters are full blooded objects, not primitive data types.

Note: Any application that manipulates characters should be prepared to encounter any character value from 0 to 65535.

Creating Characters

Many characters can be represented by printable keyboard characters. Instances of these characters can be created by preceding the desired character with a dollar sign:

```
char := $C
```

Certain characters cannot be created as keyboard literals, such as <Delete> and <Return>. Smalltalk provides class messages for creating many of these characters. Send one of the following messages to the Character class to create the corresponding character: backspace, cr, del, esc, leftArrow, lf, newPage, space, tab. For example:

```
char := Character cr
```

A character can also be created from its Unicode numeric equivalent. Send a value: message to the Character class, with the numeric Unicode representation for the character:

```
char := Character value: 67
```

The numeric value is displayed in a character's print string.

A composed character is a character consisting of base character plus a diacritical mark. To create a composed character, send a composeDiacritical: message to a character. The argument is a diacritical character, which can be obtained by sending diacriticalNamed: to the Character class. The argument is a symbol naming a diacritical character. A list of valid diacritical character names is included in the method comment.

```
| baseChar diacrit composedChar |
baseChar := $a.
diacrit := Character diacriticalNamed: #grave.
composedChar := baseChar composeDiacritical: diacrit
```

Testing Character Types

Because the extended character set contains so many subsets, Character provides a variety of tests to help you characterize an instance:

Method	Returns true if the character is...
isLowercase	a-z or a lowercase special character
isUppercase	A-Z or an uppercase special character
isAlphabetic	a-z, A-Z, or a special character
isVowel	in the set: AEIOUaeiou (with or without diacritical marks)
isDigit	0-9
isAlphaNumeric	a-z, A-Z, 0-9, or a special character
isSeparator	space, cr, tab, line feed, form feed, or null
isDiacritical	a diacritical mark (has a value in the range 16rC1 to 16rCF)
isComposed	composed of base and diacritical parts (has a value of 16rF100 or higher)
isLetter	English alphabet or extended character

Comparing Characters

Characters can be compared using the usual binary comparison operators defined for numbers: =, ==, <, >, ~=, and so on. Comparison is performed based on the integer values of the characters, so

```
$C < $D
```

evaluates as True, but

```
$c < $D
```

evaluates as False.

Strings

A String in Smalltalk is a collection, or more specifically an array, of characters. While protocol is defined at the level of the String class, a string is actually represented as a platform-specific subclass of String.

Strings are the foundation of all text operations in VisualWorks, including the text formatting and display operations described later in this chapter.

Creating a String

Most frequently a string is created by enclosing the desired characters in single quotes:

```
| string |
string := 'This is a string.'.
^string
```

You can create an empty string by sending a new message to the String class. This is equivalent to enclosing nothing between single quotes.

```
| emptyString |
emptyString := String new.
^emptyString
```

Although you can, in effect, grow as a string to accommodate added characters, this is accomplished in a copy. If you know you will do this, it is more memory efficient to create a string of the appropriate size and then change its characters. To do this, send a new: message to the String class with the length specified:

```
| newString |
newString := String new: 15.
^newString
```

By default, the string is filled with null characters, but you can specify the default character by using the new:withAll: message. The first argument is the number of characters, and the second argument is the character to fill the string:

```
| filledString |
filledString := String new: 10 withAll: $x.
^filledString
```

It is frequently necessary to represent a character by a string. There are several ways to create such a string, using obvious variations of the methods already shown. Another way is to send a with: message to the String class, with the character that is to be the sole element of the string as the argument. This is especially useful, and often necessary, when the character is a non-printing or white-space character:

```
| oneCharString |
oneCharString := String with: Character tab.
^oneCharString
```

Getting a String's Length and Width

A String is a kind of Collection with characters as its elements. Counting the characters in a string is accomplished by sending a size message to the string:

```
| string |
string := '123456789'.
^string size
```

The width of a string changes depending on the font and point size that is used to display it. Because the font choice is controlled by the graphics context of the display surface, that object can compute the width of a string in pixels. Send a widthOfString: message to the graphics context of the display surface on which the string will be displayed. The argument is the string. The width in pixels is returned.

```
| window string width |
window := ScheduledWindow new.
string := 'Hello, world'.
width := window graphicsContext
    widthOfString: string.
^width
```

Combining Strings

There are a variety of ways in which two or more strings can be combined to form longer strings, or to perform replacements within a string.

The simplest operation is concatenation, which is performed by putting a comma between the two string expressions, for example:

```
| firstName lastName fullName space |
firstName := 'Bill'.
lastName := 'Clinton'.
space := String with: Character space.
fullName := firstName, space, lastName.
^fullName
salutation := 'Dear ', fullName.
```

The result is a new string, without changing either of the original strings.

Another useful approach, especially for constructing strings of dynamically generated data, like reports, is to use a WriteStream. Create a stream by sending an on: message to the WriteStream class. The argument is typically an empty string, but it could be any string, such as a preassembled report heading. Then append each string in the series to the stream by sending a nextPutAll: message to the stream, with the string as argument. Get the stream contents in the form of a string by sending a contents message to the stream.

```
| classNames formalList |
classNames := Smalltalk classNames.
formalList := WriteStream on: String new.
classNames do: [ :name |
   formalList nextPutAll: 'Class: ';
      nextPutAll: name;
      cr].
^formalList contents
```

Modifying String Contents

Class String is implemented as a subclass of CharacterArray. Accordingly, the contents of a string consists of characters. To change the content of a String, you change the characters in it.

Changing Characters in Place

You can change a character in a String at a specific location by sending an at:put: message to the String, with the position and new character as arguments:

```
| aString |
aString := String new: 5.
aString at: 1 put: $a;
   at: 1 put: $b;
   at: 1 put: $c;
   at: 1 put: $d;
   at: 1 put: $e.
```

Note, however, that because a literal String is immutable, this fails:

```
| aString |
aString := 'abcde'.
aString at: 2 put: $x.    "ERROR"
```

Instead, if you need to do this kind of substitution, create a copy of the original String:

```
| aString |
aString := 'abcde'.
aString copy at: 2 put: $x. "SUCCESS"
```

Changing the Case in a String

Applications that manipulate strings sometimes need to convert one or more lowercase letters to uppercase, or vice versa. You can change the case of an entire string or of a selected letter.

Note: Do not use case-changing protocol with strings whose characters are caseless (for example, Japanese Katakana characters).

To convert a string to all lowercase letters, send an asLowercase message to the string. Similarly, send asUppercase to convert the entire string to uppercase letters:

```
| string |
string := 'North American Fertilizer Company'.
^string asUppercase
```

To change the case of individual characters in a string, you identify the character by its index (place in the string), use the asUppercase or asLowercase message to the character, then put the converted character back in the string at the same location. The following example uses the keysAndValuesDo: message to cycle through the string, and set all characters to lowercase except the first and those preceded by a separator character:

```
| string prevCharIsSeparator newChar |
string := 'NORTH AMERICAN FERTILIZER COMPANY' copy.
prevCharIsSeparator := true.
string keysAndValuesDo: [ :index :char |
   prevCharIsSeparator
      ifTrue: [newChar := char asUppercase]
      ifFalse: [newChar := char asLowercase].
   string at: index put: newChar.
   prevCharIsSeparator := char isSeparator].
^string
```

Some character sets contain single lowercase characters that become multiple characters in their uppercase form. If you are working with such a character set, your code should handle the results of asUppercase accordingly.

Inserting Line-End Characters

In Smalltalk methods, certain conventions of indentation and line wrapping make the code more readable. Sometimes a string disrupts the readability of the code because it contains embedded carriage returns.

Rather than embed returns in a string, you can substitute a backslash character (\). Then, when you print the string, send a withCRs message to the string to convert the backslashes back to carriage returns.

```
Dialog
   request: 'This string\has 3 lines\when displayed.' withCRs
   initialAnswer: 'No response needed'.
```

Note: This technique is not recommended for cross-cultural applications, because it interferes with text lookup in message catalogs. Instead, use separate strings and recombine them with literal line-end characters.

Abbreviating a String

Abbreviations are rarely as comprehensible as the full form of a string, and automatically derived abbreviations tend to be even less readable. In some situations, however, an abbreviation is useful, and VisualWorks provides a few useful abbreviation messages. Here are two methods. Browse the String class and its superclasses for others.

Send a contractTo: message to the string. The argument is the number of characters in the abbreviation, including three for the ellipsis. Half of the abbreviation will be taken from the beginning of the string and the other half from the end.

```
| string contractedString |
string := 'North American Free Trade Agreement'.
contractedString := string contractTo: 15.
^contractedString
```

Send a dropFinalVowels message to the string. An abbreviated string is returned in which only the leading vowel (if any) remains.

```
| string noVowelString |
string := 'North American Free Trade Agreement'.
noVowelString := string dropFinalVowels.
^noVowelString
```

String Substitution Parameters

Strings can include formal parameters, enclosed in the angle brackets < >. The parameters are expanded by sending a version of the expandMacros: message to the string.

Simple parameters are <n> and <t>, which specify substitution of CR and Tab, respectively. For example, the String 'This is a <n><t>test' can be expanded:

```
'This is a <n><t>test.' expandMacros
```

to print:

```
This is a
   test.
```

Positional substitution parameters are also allowed. Immediately following the opening bracket there may be an integer that specifies which of the expansion arguments to substitute for this parameter. This allows for substitutions in the string to appear in a different order than that in which the arguments are passed in, and for the same argument to be substituted more than once.

Following this parameter index is a character that identifies how the substitution is to be performed. The characters and the substitution they indicate are:

p

Substitute the printString value of the argument. For example:

```
'This is a <1p> test.' expandMacrosWith: 'substitution'
```

expands to

```
'This is a ''substitution'' test.'
```

s

Substitute the argument itself, which must be a CharacterArray. For example:

```
'This is a <1s> test.' expandMacrosWith: 'substitution'
```

expands to

```
'This is a substitution test.'
```

?

Requires two arguments in the parameter, and a Boolean expression argument. The first is substituted if the expression argument is true

```
'One is greater than <1?zero:two>.' expandMacrosWith: true
```

, and the second if the argument is false. For example:expands to

```
'One is greater than zero.'
```

#

Requires two arguments in the parameter, and a numeric expression argument. The first is substituted if the expression argument is equal to 1; otherwise, the second is substituted. For example:

```
'The book "<1#War:Peace> and <2#War:Peace>" is a must read.'
    expandMacrosWith: 1 with: 2.
```

expands to

```
'The book "War and Peace" is a must read.'
```

The versions of expandMacros in these examples take one and two positional substitution arguments. For up to four arguments, there are also expandMacrosWith:with:with:, and expandMacrosWith:with:with:with:. For more than four arguments, use exapndMacrosWithArguments:, with an Array of arguments. All of these expand <n> and <t> as well.

The character $% acts as the escape character. Unless otherwise specified, any character following the escape character is itself, and is not treated specially. For example, '%<' becomes '<', which, because it is preceded by the escape character, is not treated as the beginning of a formal parameter. So,

```
'This is %<1s%> test' expandMacrosWith: 'a'
```

expands to

```
'This is <1s> test'
```

Substring Operations

When a string contains two or more parts, getting the parts as separate strings is a common requirement. For example, you might need to extract the first and last names from a string containing a full name. You can copy a portion of a string, using the starting and stopping character locations.

In certain situations, the only part of a string that you need is a prefix that ends at a specific character. You can copy the characters that precede a specific endpoint character.

Copying a Substring

Send a copyFrom:to: message to the string. The first argument is the starting index and the second argument is the ending index of the desired substring.

```
| fullName firstName lastName spaceIndex |
fullName := 'Mahatma Gandhi'.
spaceIndex := fullName indexOf: Character space.
firstName := fullName
    copyFrom: 1
    to: spaceIndex - 1.
lastName := fullName
    copyFrom: spaceIndex + 1
    to: fullName size.
^Array with: firstName with: lastName
```

Copying a Prefix

Send a copyUpTo: message to the string. The argument is the character that marks the end of the prefix (but is not included in it).

```
| fullName firstName |
fullName := 'Boris Yeltsin'.
firstName := fullName copyUpTo: Character space.
^firstName
```

Removing or Replacing a Substring

A string can be quite long and complicated, representing an entire report or the contents of a lengthy text file. In long strings especially, replacing a portion of the string with a new substring is frequently useful.

Removing characters is accomplished by creating a copy in which the unwanted characters have been replaced by an empty string.

When a string contains multiple occurrences of a substring, you can replace all occurrences.

Replacing a Substring

To replace characters in a string with another string, send a copyReplaceFrom:to:with: message to the string. This returns a copy of the original string with the replacement made. The first and second arguments are the index locations of the starting and stopping characters in the substring that is to be replaced. The with: argument is the substitution string.

You can also use this method to insert a substring without removing any characters in the existing string, by making the ending index one less than the starting index.

To remove characters, replace them with an empty string.

For details of the operation of this method, refer to the method comment.

```
| aString anotherString newString |
aString := 'abcd'.
anotherString := 'efgh'.
" Replacement, returns 'aefghd' "
newString := aString copyReplaceFrom: 2 to: 3 with: anotherString.
" Insertion, returns 'aefghbcd' "
newString := aString copyReplaceFrom: 2 to: 1with: anotherString.
" Prefixing, returns 'efghabcd' "
newString := aString copyReplaceFrom: 1 to: 0 with: anotherString.
^newString
```

For replacements with a return size greater than 1024, use changeFrom:to:with: instead.

Replacing All Occurrences of a Substring

Send a copyReplaceAll:with: message to the string. The first argument is the substring that is to be replaced. The second argument is the replacement substring.

```
| colorNames |
colorNames := String new.
ColorValue constantNames do: [ :name |
   colorNames := colorNames, name asString, ' '].
colorNames := colorNames"Variant Step"
   copyReplaceAll: 'Gray'
   with: 'Grey'.
^colorNames
```

Tokenizing Substrings

It can be useful to convert a String to a collection of elements in it. Send a tokensBasedOn: message to a String to return a collection of substrings separated by the argument. For example:

```
'brave new world' tokensBasedOn: Character space
```

returns a collection of three strings.

Notice that this method is implemented several classes above String in the hierarchy. Browse these superclasses for methods that might provide results that you need.

Searching

The ability to find a specific character or substring is essential in applications that parse strings. Often a special character or series of characters identifies a field within a string, especially when the string represents the contents of a structured text file.

By default, searching is case-sensitive, but there are methods which ignore case during a search.

Searches can also use wildcard characters. A pound sign (#) takes the place of any single character, and an asterisk (*) takes the place of zero or more characters.

Get the Index of a Character in a String

To get the index of a character, send an indexOf: message to the string. The argument is the search character. If it is not found, zero is returned.

To find the starting index of a substring, send a - findString:startingAt:ifAbsent: message to the string. The first argument is the substring to be found. The second argument is the character position at which the search is to begin. The third argument is a block containing actions to be taken if the substring is not found (often an empty block, to avoid the default error).

```
| classComment searchChar searchString index1 index2 |
classComment := String comment.
searchChar := $<.
searchString := 'Class Variables:'.
index1 := classComment indexOf: searchChar.
index2 := classComment
    findString: searchString
    startingAt: 1
    ifAbsent: [ ].
^Array with: index1 with: index2
```

Ignoring Case in a Search

Send a findString:ignoreCase:useWildcards: message to the string. The findString argument is the substring to be found. The ignoreCase argument is true when case difference is to be ignored.

The useWildcards argument is true when the pound sign and asterisk are to be interpreted as wildcard characters rather than literal characters. Because the presence of an asterisk wildcard affects the endpoint of the found string, this variant returns an Interval identifying the index range of the found string. A zero interval is returned when the search string is not found.

```
| classComment searchString interval |
classComment := String comment.
searchString := 'Var*:'.
interval := classComment
    findString: searchString
    startingAt: 1
    ignoreCase: true
    useWildcards: true.
^classComment
    copyFrom: interval first
    to: interval last
```

Comparing Strings

Unlike characters, strings are not compared by numerical value of their characters. When comparing strings, case is ignored and alphabetical order is used, unless the two strings have exactly the same letters in the same order. In this latter case, numerical values are used to differentiate uppercase and lowercase letters.

Testing for Equality and Identity

Two strings are equal when both have the same number of characters, and both have the same characters in the same order.

To test for equality or inequality, send an = or ~= (not equal) message to one string with another string as argument:

```
| str1 str2 |
str1 := 'abc'.
str2 := 'ABC'.
^str1 = str2
```

To compare based on identity, send an == or ~~ (not identical) message to the object. Two different strings cannot be identical, though two variables that refer to the same string are identical.

```
| str1 str2 str3 |
str1 := 'Excellent'.
str2 := 'Excellent'.
```

```
str3 := str1.
^Array
   with: (str1 == str2)
   with: (str1 == str3)
```

The sameAs: message compares the equality of strings while ignoring case:

```
| str1 str2 str3 |
str1 := 'north'.
str2 := 'North'.
str3 := 'northwest'.
^Array
   with: (str1 sameAs: str2)
   with: (str1 sameAs: str3)
   with: (str2 sameAs: str3)
```

Comparing by Sorting Order

The usual comparison operators, in addition to equality and identity, can be used to compare strings:

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

Comparison is by alphabetical order in most cases, rather than numerical value of the characters. So,

```
'BCD' < 'bcde'
```

evaluates to true.

If two strings have exactly the same letters in the same order, the integer values of the characters is used to differentiate them. So,

```
'bcD' > 'bcd'
```

evaluates to false, because uppercase letters have lower integer values than lowercase letters.

Rating the Similarity of Two Strings

Two messages return a similarity rating of strings.

A sameCharacters: message returns an integer indicating how many characters are the same (including case) up to the first mismatch. So,

```
'bcDe' sameCharacters: 'bcde'
```

returns 2.

A spellAgainst: returns an integer from 1 (entirely different) through 100 (equal) is returned, giving a percentage of match to mismatch. So,

```
'bcDe' spellAgainst: 'bcde'
```

returns 75.

8

Colors and Patterns

VisualWorks uses Colors and Patterns to draw lines and fill shapes.

Colors as represented as instances of ColorValue. VisualWorks stores colors as red, green, and blue (RGB) components, but allows colors to be specified by constant names, by RGB values, or by hue, saturation, and brightness (HSB) values.

A Pattern is an arrangement of pixels created by replicating a tile throughout a painted region. For example, the gray background used by many window managers is created by employing a four-pixel tile. The tile can be an Image, a Pixmap, or a Mask.

Pixel Coverage

A CoverageValue identifies the fraction of a pixel that is covered. Since a pixel, by its nature, must be displayed in its entirety, only the values 0 and 1 are typically used. Fractional coverages can be specified, however, as explained in the discussion of coverage palettes on Image Color Palettes.

CoverageValue is the paint basis for Masks. An Image can also be coverage-based, typically when it is used as a storage medium for a Mask, which does not survive after the system is shut down.

A CoverageValue can be created by name or by value:

```
CoverageValue transparent
CoverageValue coverage: 0
CoverageValue opaque
CoverageValue coverage: 1
```

Creating a Color

ColorValue class methods provide simple protocol for creating instances by either color constant name, RGB values, or HSB values.

Create by Color Name

Several color constants are defined by class method selectors for each color name. To create a color, send the appropriate color message to ColorValue class. For example, to create an instance of cyan, send the cyan message to the ColorValue class:

```
| gc color |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
color := ColorValue cyan.
gc paint: color.
gc displayDotOfDiameter: 400 at: 200@200.
```

The following example displays all the predefined colors in a ray chart.

```
| gc endPoint colors |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
gc lineWidth: 7.
endPoint := 350@0.
colors := ColorValue constantNames.
colors do: [ :c |
   endPoint := endPoint + (-10@12).
   gc paint: (ColorValue perform: c).
   gc displayLineFrom: 0@0 to: endPoint.
   gc paint: ColorValue black.
   c asString displayOn: gc at: endPoint + (0@8)]
```

Create by Red, Green, and Blue Values

Send a red:green:blue: message to the ColorValue class. All arguments are numbers between zero and one, representing the intensity of their respective colors. In the example, the intensity of green is varied while the red and blue intensities remain at zero.

```
| gc origin |
gc := (Examples.ExamplesBrowser
     prepareScratchWindow) graphicsContext.
origin := 0@0.
1 to: 0 by: -0.01 do: [ :grn |
    gc paint: (ColorValue red: 0.0 green: grn blue: 0.0).
    origin := origin + 4.
    gc displayRectangle: (origin extent: 400 - origin)]
```

Create by Hue, Saturation, and Brightness Values

Send a hue:saturation:brightness: message to the ColorValue class. The hue argument is a number from 0 to 1, where 0 is red, 0.333 is green, 0.667 is blue, and 1 is red again. The saturation argument is a number from 0 to 1, representing minimum vividness (white) to full color; a more saturated color makes an object appear closer to the viewer. The brightness argument is a number from 0 to 1, representing minimum brightness (black) to full color; varying the brightness is useful for representing shadows.

```
| gc r x y |
gc := (Examples.ExamplesBrowser
    prepareScratchWindow) graphicsContext.
r := 50.
gc lineWidth: 2.
gc translation: 150@150.
0 to: 1 by: 0.005 do: [ :i |
    x := (i * Float pi) cos * r.
    y := (i * Float pi) sin * r / 2.
    gc paint: (ColorValue hue: 0.0 saturation: 0.5 brightness: i).
    gc displayLineFrom: x@y to: 0@-100 ].
gc translation: 200@200.
0 to: 1 by: 0.005 do: [ :i |
    x := (i * Float pi) cos * r.
    y := (i * Float pi) sin * r / 2.
    gc paint: (ColorValue hue: 0.0 saturation: 0.75 brightness: i).
    gc displayLineFrom: x@y to: 0@-100 ].
gc translation: 250@250.
0 to: 1 by: 0.005 do: [ :i |
    x := (i * Float pi) cos * r.
    y := (i * Float pi) sin * r / 2.
    gc paint: (ColorValue hue: 0.0 saturation: 1.0 brightness: i).
    gc displayLineFrom: x@y to: 0@-100 ]
```

Coloring a Graphical Object

By default, a color-based display surface (ApplicationWindow or Pixmap) displays geometric objects in black. To change the color of an object, set the color for the graphic context before drawing the object. To set the color, send a paint: message to the graphics context of the display surface with the color as argument:

```
| gc circle colors |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
circle := Circle center: 200@200 radius: 200.
colors := ColorValue constantNames.
colors do: [ :colorName |
   gc paint: (ColorValue perform: colorName).
   circle := circle scaledBy: 0.9.
   circle asFiller displayOn: gc]
```

Creating a Pattern

A Pattern is created by filling a space with a single graphic image that is repeated in tiles. A Pattern can be used in any situation that you can use a solid color.

To create a pattern, send an asPattern message to the graphic image to serve as the tile:

```
| gc tile |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
tile := Image cincomSmalltalkLogo shrunkenBy: 4@4.
tile := tile asPattern.
gc paint: tile.
gc displayRectangle: (50@50 extent: 300@300).
```

The graphic image is typically an Image subclass instance, but can also be a window, Pixmap, or Mask.

Applying a Pattern

Patterns are applied in the same way as colors. Send a paint: message to the graphics context of the display surface on which the object is to be displayed. The argument is a pattern, or in the case of a Mask, a coverage.

```
| gc tile |
tile := Pixmap extent: 10@10.
gc := tile graphicsContext.
"Tile background"
gc paint: ColorValue chartreuse.
gc displayRectangle: (0@0 extent: 10@10).
"Tile foreground"
gc paint: ColorValue red.
gc displayDotOfDiameter: 10 at: 4@4.
"Patterned circle"
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
gc paint: tile asPattern.
gc displayDotOfDiameter: 400 at: 200@200.
```

Adjusting a Pattern’s Tile Phase

For some patterns, the placement of that first tile can be critical to the pattern. By default, the first tile is placed with its upper left corner at the origin of the display surface’s GraphicsContext.

To adjust the start location, send a tilePhase: message to the graphics context of the display surface on which the patterned object is to be displayed. The argument is a point that defines the origin of the first tile in the pattern.

In the example, the tile phase is the same as the origin of the painted object, which aligns the tiles with the top and left edges of the object.

```
| gc tile |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
tile := Image cincomSmalltalkLogo shrunkenBy: 4@4.
tile := tile asPattern.
gc paint: tile.
gc tilePhase: 50@50.
gc displayRectangle: (50@50 extent: 300@300).
```

Image Color Palettes

A Palette represents the collection of colors available for coloring pixels. For colored objects, such as images, the color of each pixel is stored as a numeric value. A palette is needed to translate those numeric values to instances of ColorValue or Cover-ageValue.

Coverage Palettes

A CoveragePalette is used by Masks and masking images, to specify levels of transparency. It has a maxPixelValue, which determines the number of levels of transparency. Usually, maxPixelValue is set to 1, because a pixel can only be fully transparent (pixel value 0) or fully opaque (1).

However, you may want to allow for intermediate levels of translucence. By specifying the maxPixelValue, you can create an image having any number of coverage levels (currently, masks are restricted to two levels).

Color Palettes

A color palette can be either fixed or mapped.

A FixedPalette breaks a pixel value into red, green, and blue fields, each of which controls the intensity of that primary color.

A MappedPalette stores a table of colors, so each numeric pixel value can be associated with an arbitrary color. A MonoMappedPalette is a MappedPalette that contains only black and white.

Mapped palettes are most appropriate for images on color-mapped display screens and for images that use a small number of colors. Fixed palettes support true-color display screens that do not use a hardware color map. Such true-color screens typically support a large number of colors. A mapped palette for a typical true-color screen, which has a depth of 24, requires a color mapping table with more than 16 million elements.

To create a mapped palette, send a withColors: message to MappedPalette, specifying an array of colors used to initialize the palette.

A fixed palette uses RGB values. Depending on the depth of the image, one set of RGB values might occupy 8 bits, 24 bits, or 32 bits (or even something in between). When you create a fixed palette, you must provide it with the means to locate the red bits, the green bits, and the blue bits. You do so by indicating the number of the bit that begins each RGB component as well as the maximum value for that component. In the creation message, the starting bit is called the shift value and the maximum value is called the mask value.

Fixed palettes for 8-bit pixel values are structured in which the high three bits specify the red component, the next three bits the green component, and the low two bits the blue component.

Image Display Performance

The composition of an image's palette greatly affects the amount of time required to display the image. An image can be displayed quickly in either of two circumstances:

- Its palette is the same as that of the display surface
- Its palette contains only two colors, which can be rendered without halftoning.

Otherwise, displaying the image requires creating a temporary image, which can take a substantial amount of time. To avoid generating a temporary image, convert the image to the native palette and then display the converted image. For example, to convert an image to the color palette of the default screen (and therefore also of all windows and pixmaps on the default screen), perform:

```
anImage convertToPalette: Screen default colorPalette
```

By default, the convertToPalette: operation employs a NearestPaint renderer.

Device Color Map

The window manager's color map is not accessible from within Smalltalk. The screen's colorPalette is assembled based on that color map, as indicated in the following table. In the Comment column, "Fully populated" means the VisualWorks palette is the same as the device color map. "Partially populated" means VisualWorks uses only a portion of the color map, leaving enough unused cells so neighboring applications will have a chance to allocate their colors, too. When the platform provides a hint as to the default set of colors to be shared by applications, we use that set.

Screen depth	Window system	Palette type	Comment
1	All	Mapped	Fully populated
2	All	Mapped	Fully populated
4	All	Mapped	Fully populated
8	X	Mapped*	Partially populated
8	MS-Windows	Mapped	Partially populated
8	Macintosh	Mapped	Fully populated
15	MS-Windows	Fixed	RGB values
16	All	Fixed	RGB values
24	All	Fixed	RGB values
32	All	Fixed	RGB values

* Using X, an 8-bit color map can be made fixed instead of mapped.

Applying a Palette to an Image

In a graphic image, each pixel is associated with a color in the image's palette of colors. You can effectively change one or more colors in an image by creating a new palette with the desired colors at the old colors positions, and then install the new palette. The new palette must have the same number of color entries as the old palette.

1 Create an array representing the old color palette.

   To create the array, send a palette message to the image, and then send a colors message to the resulting palette.

2 Modify the palette by replacing colors in the array as desired.

3 Create a new palette by sending a withColors: message to the MappedPalette class, with the new array as argument.

4 Install the new palette by sending a palette: message to the image, with the new palette as argument.

In this example, every white pixel is converted to yellow.

```
| gc palette image colors whiteIndex |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
image := InputFieldSpec paletteIcon asImage.
colors := image palette colors.
```

```
whiteIndex := colors indexOf: ColorValue white.
colors at: whiteIndex put: ColorValue yellow.
palette := MappedPalette withColors: colors.
image := image palette: palette.
image displayOn: gc at: 10@10.
```

Converting an Image to Use the Default Palette

When a color palette differs from the palette used by the display surface, a temporary image is created so VisualWorks can simulate the desired colors when necessary. This step can take a significant amount of time. To display an image quickly, convert it to use the default palette that is used by display surfaces.

To convert the palette for an image, send a convertToPalette: message to the image. The argument is the default color palette, which can be accessed by sending a default message to the Screen class and then sending a colorPalette message to the resulting screen.

```
| gc image |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
image := Image cincomSmalltalkLogo magnifiedBy: 2@2.
image := image convertToPalette: Screen default colorPalette.
image displayOn: gc at: 10@10.
```

For a coverage-based image, send a coveragePalette message instead of colorPalette.

Color Rendering Policies

When an image makes liberal use of the color turquoise, what should a black-and-white window do when asked to display that alien color? How about a color window that doesn't happen to have just the right shade of turquoise in its palette?

VisualWorks provides three common techniques for rendering unknown colors, represented by the classes: NearestPaint, Ordered-Dither and ErrorDiffusion.

Any of the three can be used to render an image, but only NearestPaint and Ordered-Dither are appropriate for rendering paints. A PaintPolicy object holds both a paintRenderer and an imageRenderer, which may be the same.

The default renderers are determined as follows:

NearestPaint	Used by Pixmaps and Windows on color systems
OrderedDither	Used by Masks on all types of screens
OrderedDither	Used by Pixmaps and Windows on monochrome or gray-scale systems

NearestPaint

NearestPaint simply chooses the nearest available paint from the screen's palette. On color screens, NearestPaint usually produces satisfactory results and always gives the best performance of the three renderers.

On a limited palette, such as on a monochrome screen, the results can be disappointing. For example, a magenta image on a chartreuse background will result in a white rectangle, because both colors are luminous enough to be converted to white by NearestPaint.

OrderedDither

OrderedDither employs a threshold array to synthesize unrecognized colors by blending neighboring colors from the screen's palette. This has the effect of smoothing the transition from one palette color to the next in a continuous tone. While the result is often more pleasing than with NearestPaint, you pay a price in performance.

ErrorDiffusion

An ErrorDiffusion uses a more sophisticated blending algorithm. When it makes a choice from the screen's palette, it keeps track of how far off that choice was from the requested color. When this error accumulates sufficiently, the renderer uses the color on the other side of the threshold.

For example, suppose that a region of the image uses a red-brown color, but the screen's palette has only red and brown. An ErrorDiffusion may supply red at first, but keeps track of the numeric difference between red and the red-brown. When that remainder accumulates to a breakpoint, a brown pixel is displayed instead. In this way, red and brown pixels are blended to give a red-brown effect.

Applying a Renderer to an Image

If an image is to be displayed repeatedly, there is a performance advantage to converting it to use the screen's renderer, rather than leaving it to the display surface to perform the conversion each time the original image is displayed on it.

To convert an image, send a convertForGraphicsDevice:renderedBy: message to the image. The first argument is typically Screen default. The second argument is the renderer to use.

```
| gc r g b im |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
im := Image
   extent: 60@60
   depth: 15
   palette: (FixedPalette
      redShift: 10 redMask: 31
      greenShift: 5 greenMask: 31
      blueShift: 0 blueMask: 31).
0 to: 59 do: [:x |
   0 to: 59 do: [:y |
      r := 1 - ((x@y - (10@10)) r / 30) max: 0.
      g := 1 - ((x@y - (20@50)) r / 30) max: 0.
      b := 1 - ((x@y - (50@30)) r / 30) max: 0.
      im atPoint: x@y put: (im palette
         indexOfPaintNearest:(ColorValue red: r green:g blue: b))]].
(im convertForGraphicsDevice: Screen default
   renderedBy: NearestPaint new)
      displayOn: gc at: 10@10.
(im convertForGraphicsDevice: Screen default
   renderedBy: OrderedDither new)
      displayOn: gc at: 80@10.
(im convertForGraphicsDevice: Screen default
   renderedBy: ErrorDiffusion new)
      displayOn: gc at: 150@10.
```

Converting an Image to a Specific Palette

The image can be converted to a palette other than the screen's palette. This is useful for showing what the image would look like on a screen that has a limited palette.

Send a convertToPalette:renderedBy: message to the image, where the first argument is the desired palette (in the example, a monochrome palette), and the second argument is the desired renderer.

```
| gc r g b im |
gc := (Examples.ExamplesBrowser
   prepareScratchWindow) graphicsContext.
im := Image
   extent: 60@60
   depth: 15 palette: (FixedPalette
      redShift: 10 redMask: 31
      greenShift: 5 greenMask: 31
      blueShift: 0 blueMask: 31).
0 to: 59 do: [:x |
   0 to: 59 do: [:y |
      r := 1 - ((x@y - (10@10)) r / 30) max: 0.
      g := 1 - ((x@y - (20@50)) r / 30) max: 0.
      b := 1 - ((x@y - (50@30)) r / 30) max: 0.
      im atPoint: x@y put: (im palette
         indexOfPaintNearest:
            (ColorValue brightness: 1-((1-r)*(1-g)*(1-b))))]].
(im convertToPalette: MappedPalette whiteBlack
   renderedBy: NearestPaint new)
      displayOn: gc at: 10@10.
(im convertToPalette: MappedPalette whiteBlack
   renderedBy: OrderedDither new)
      displayOn: gc at: 80@10.
(im convertToPalette: MappedPalette whiteBlack
   renderedBy: ErrorDiffusion new)
      displayOn: gc at: 150@10.
```

Setting the Rendering Policy for Nonimage Graphics

Graphic objects other than images do not have their own color, so the rendering is performed by the graphics context of the display surface. To change the renderer, you install the desired renderer in the graphics context.

Install a paint policy in the graphics context of the display surface by sending a paintPolicy: message to the graphics context. The argument is a PaintPolicy, typically a new instance. Then, set the rendering algorithm by sending a paintRenderer: message to the paint policy with either a NearestPaint or an OrderedDither as argument. (ErrorDiffusion is only used with images).

```
| gc |
gc := (Examples.ExamplesBrowser
    prepareScratchWindow) graphicsContext.
gc paintPolicy: (PaintPolicy new imageRenderer: OrderedDither new).
gc paintPolicy paintRenderer: NearestPaint new.
0 to: 60 by: 4 do: [:i |
    0 to: 60 by: 4 do: [:j |
        gc paint: (ColorValue red: i/60 green: j/60 blue: 0).
        gc displayRectangle: (i@j+(10@10) extent: 4@4)]].
gc paintPolicy paintRenderer: (OrderedDither order: 1).
0 to: 60 by: 4 do: [:i |
    0 to: 60 by: 4 do: [:j |
        gc paint: (ColorValue red: i/60 green: j/60 blue: 0).
        gc displayRectangle: (i@j+(80@10) extent: 4@4)]].
gc paintPolicy paintRenderer: (OrderedDither order: 6).
0 to: 60 by: 4 do: [:i |
    0 to: 60 by: 4 do: [:j |
        gc paint: (ColorValue red: i/60 green: j/60 blue: 0).
        gc displayRectangle: (i@j+(150@10) extent: 4@4)]].
```

By default, a new OrderedDither has an order of 6, which means it synthesizes 65 (2 to the sixth, plus 1) intermediate color values between each pair of neighboring colors in the palette. You can set the order by sending an order: message to the OrderedDither class to create an instance; the argument is the desired order number.

9

Socket Programming

Sockets provide the basic communication structure for all internet programming in VisualWorks. The VisualWorks socket implementation is a thin Smalltalk API to BSD (UNIX) sockets.

VisualWorks supports all BSD socket types:

- SOCK_STREAM
- SOCK_DGRAM
- SOCK_SEQPACKET
- SOCK_RAW
- SOCK_RDM

Stream sockets are the most common for internet communications, so this chapter focuses on that type.

While all socket protocols can be used, using the "raw" protocol, only TCP and UDP protocols are explicitly supported by VisualWorks. Not all socket options are supported, or supported well, at this time.

Socket communication is a peer-to-peer conversation; both "client" and "server" sockets are identical kinds of things. They are configured differently, however, so that a "server" socket listens for connection requests from "clients." VisualWorks allows you to implement both.

In the abstract, sockets are very simple. However, the intricacies of making a socket-based application robust across multiple platforms requires perseverance and practice. Those complications are (mostly) beyond the scope of this document.

VisualWorks Implementation Classes

Socket support is provided primarily in these classes:

SocketAccessor

Corresponds to the BSD notion of a socket, and provides the creation and connection protocol.

SocketAddress

Corresponds to the BSD sockaddr C-structure. For internet purposes, the IPSocketAddress subclass is the most important, since it provides for identifying an address by host name or IP address, and port number.

Socket Basics

Creating a socket is a simple matter of sending the appropriate socket creation message to class SocketAccessor. The basic procedure is essentially the same in VisualWorks as it is in other programming environments that implement BSD sockets.

For simplicity, convenience methods for creating server and client TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) sockets are provided. We'll create a simple TCP client and server using the convenience methods to illustrate a simple pattern for implementing both.

Creating a socket

The bare socket creation protocol identifies the address family, the socket type, and possibly the protocol family:

family: *addrFamily* **type:** *sockType*

Creates a socket for family *addrFamily* of socket type *sockType*. The protocol family defaults to PF_UNSPEC.

family: *addrFamily* **type:** *sockType* **protocol:** *protoFamily*

Creates a socket for family *addrFamily* of socket type *sockType*, and protocol family *protoFamily*.

The arguments are integer identifiers, but can be supplied by SocketAccessor class methods that return the appropriate identifier. For example, to create a TCP socket for transferring streams of ASCII data, you can define the socket like this:

```
sockAccessor := SocketAccessor family: SocketAccessor AF_INET
   type: SocketAccessor SOCK_STREAM
```

Browse the SocketAccessor class method categories constants-address families, constants-socket types, and constants-protocol families for the complete set of defined identifiers. The method names are the BSD API family and type names.

Making a client or server socket

Whether a socket is a service provider (“server”) or user (“client”), it is the same kind of object. The difference is how it connects to other sockets: a client “connects” to a server socket at an IP address, and a server “listens” on (or “binds” to) an IP address (a local address) on which it is to provide services.

The IP address is represented by an instance of IPSocketAddress. The address can be defined by host name or IP address, and either with or without a port number. The available creation messages are:

hostAddress: *ipAddress*

Creates a new IPSocketAddress for the host specified by *ipAddress*, with the port unspecified. *ipAddress* is specified as an array of integers (see below).

hostAddress: *ipAddress* **port:** *portNo*

Creates a new IPSocketAddress for the host specified by *ipAddress* on port *portNo*.

hostName: *ipName*

Creates a new IPSocketAddress for the host specified by *ipName*, with the port unspecified.

hostName: *ipName* **port:** *portNo*

Creates a new IPSocketAddress for the host specified by *ipName* on port *portNo*.

thisHostAnyPort

Creates a new IPSocketAddress for the local machine's IP address, with a system-assigned port number. (Send getName to the to socket to get the assigned IPSocketAddress.)

For example, these both create new IPSocketAddress instances:

```
sockAddr := IPSocketAddress hostAddress: #[ 128 16 16 101 ]
sockAddr := IPSocketAddress hostName: 'bob.myco.com' port: 10559
```

The port number is typically specified by the service you are accessing, which assigns the port that the server is "listening" on. Many common services have reserved, well-known, port numbers. For example, port 80 is reserved for HTTP (web) servers, and port 21 is reserved for the FTP control channel. (Refer to Port numbers for more information.)

A client socket typically makes a connection to a server socket. To do this, send a connectTo: message to the socket with an IPSocketAddress for the server as the argument:

```
sockAddr := IPSocketAddress hostName: 'bob.myco.com' port: 10559.
sockAccessor := SocketAccessor family: SocketAccessor AF_INET
   type: SocketAccessor SOCK_STREAM.
sockAccessor connectTo: sockAddr.
```

This is essentially what newTCPclientToHost:port: does in a single message. At this point the client can read and write data on the socket.

.A server, on the other hand, binds to the IP address on which it offers services, i.e., on which it is willing to accept connections. To do this, send a bindTo: message to the socket with the IPSocketAddress as the argument:

```
sockAddr := IPSocketAddress hostName: 'bob.myco.com' port: 10559.
sockAccessor := SocketAccessor family: SocketAccessor AF_INET
   type: SocketAccessor SOCK_STREAM.
sockAccessor bindTo: sockAddr.
```

The server then listens on the socket for incoming connection requests. To begin listening, send a listenFor: message to the socket with an integer argument specifying the maximum number of connection requests the OS will queue up at one time:

```
sockAccessor listenFor: 5.
```

Multiple connection requests may come in all at once, and you don't want to lose them, or at least not all of them. The queue size specifies how many will be held in the backlog for pending connection. Setting the queue size to 5 means 6 connection requests can be handled at once; one being processed and 5 in the backlog. This is also a typical system maximum. Setting it to 0 allows handling only one connection request at a time.

When a connection request does come in, the socket needs to accept the request. To do this, send an `accept` message to the socket:

```
sockAccessor accept.
```

This creates a new socket on which the server handles communication with the client, and clears the listening socket to handle the next incoming connection request. Your application program will need to loop on the `accept` message so more than one connection will be accepted. This is illustrated below.

The original socket continues listening for connection requests.

Closing a socket

When you are finished using a `SocketAccessor`, you need to close the connection. Two methods are available:

close

Informs the OS that the accessor's handle should be released. Also, removes registry references. (Defined in `BlockableIOAccessor`.)

shutdown: *anInteger*

Inform the `SocketAccessor` that no more IO will be performed: 0 -- read channel 1 -- write channel 2 -- both

`shutdown: 2` is more dramatic (and faster) than `close` because it discards any pending data anywhere along the network path. Buffered data on the receiving end may also be lost.

When a `Stream` is opened on a `SocketAccessor`, sending `close` to the `Stream` also closes the socket.

Note that closing a socket involves network traffic, if the network is down an error will result.

Port numbers

Some services are provided by a server only on a specific port number.

You can get the port number for many standard services in either of two ways. You can send the servicePortByName: message to IPSocketAddress, with the name of the service as a String:

```
IPSocketAddress servicePortByName: 'ftp'
```

This retrieves the port number from a file, usually called services, on your system (e.g., /etc/services, or c:\windows\services).

Alternatively, a number of service ports are returned by class methods in SocketAccessor, in the constants-well known ports method category:

```
SocketAccessor IPPORT_FTP
```

Non-standard services typically use large (four digits or more) numbers to avoid conflicts, just as we're using 9009 in the example.

Even many four-digit numbers are reserved. For a list of "reserved" port numbers, see http://www.graffiti.com/services.

"Well-known" ports are controlled by the IANA. An up-to-date list is available from http://www.iana.org/assignments/port-numbers.

Building a TCP socket client

TCP socket clients are the most common clients for internet communications. To simplify creating and connecting a TCP client socket, VisualWorks provides the message newTCPclientToHost:port:. Send this message to SocketAccessor, as follows:

```
sockAccessor := SocketAccessor newTCPclientToHost: 'hostname'
   port: 9009
```

This one line both creates the socket and connects it to the specified host and port, reducing three lines to one.

The socket is now ready to read and write data, but we need to decide how to do that. For the moment we'll use Streams as a simple read/write interface to our socket.

To attach a read/write stream to the socket, send the readAppendStream message to the SocketAccessor:

```
stream := sockAccessor readAppendStream
```

The Stream could also have been created as a readStream or a writeStream, but for most purposes you need a read/write stream.

Once you have the socket open and a stream attached to it, you are ready to begin sending and/or receiving data using the usual Stream protocols. (See Stream Style Communication for more information.)

For example, the following is a simple-minded log-in and close exchange with an FTP server that simply dumps the server responses to the Transcript. Stream protocol is used for sending the login commands and receiving the responses.

```
| sockAccessor stream |
"connect a stream socket"
   sockAccessor := SocketAccessor
      newTCPclientToHost: 'ftp.parcplace.com' port: 21.
   stream := sockAccessor readAppendStream.
"Set the FTP line-end convention"
   stream lineEndCRLF.
"Read the server connection response before logging in"
   Transcript nextPutAll: (stream upTo: Character cr) ; cr ; flush .
"Log in, writing responses to Transcript"
   stream nextPutAll: 'USER anonymous'; cr ; commit.
   Transcript nextPutAll: (stream upTo: Character cr) ; cr ; flush.
   stream nextPutAll: 'PASS test@parcplace.com' ; cr ; commit.
   Transcript nextPutAll: (stream upTo: Character cr) ; cr ; flush.
"close everything down"
   stream close.     "closes both the stream and the socket"
```

Obviously, there is a lot more work to do to make an interesting session.

Building a TCP socket server

A socket server has more responsibility than does a client, and so is a little more complex. Instead of connecting to a port, the server has to listen for a connection request on a port.

Analogous to the client, we create a TCP server socket by sending newTCPServerAtPort: to SocketAccessor:

```
sockAccessor := SocketAccessor newTCPServerAtPort: 9009
```

This line creates the AF_INET socket and binds it to the host at the specified port, on which it will accept connection requests. We’ll use a good, high port number that isn’t currently reserved for anything.

Next, the server needs to start listening for connection requests. To do this, send a listenFor: message to the socket, with the maximum number of pending connection requests the OS should queue up before rejecting requests (not the number of connections):

```
sockAccessor listenFor: 5
```

At this point the socket is set up and listening for connection requests. To accept a request, send the accept message to the socket. When a connection request is received, accept returns a new SocketAccessor connected to that client:

```
newSocket := sockAccessor accept.
```

This message blocks, and will wait indefinitely for a connection request.

When a request comes in, the message returns a new SocketAccessor over which the server can communicate with the client. The original server socket itself returns to listening for and accepting further connection requests.

Clearly, to accept multiple connections, the server must loop on the accept message, and the new connections will need to be handled. One simple way is to fork a process for each new connection. Another is to use the non-blocking version of accept, acceptNonBlocking, and use the equivalent of the BSD select() command. The latter approach will be described later.

Having accepted the connection, we attach a Stream to the socket, to provide a simple read/write protocol:

```
newStream := newSocket readAppendStream
```

Here's a simple way to fork the new socket process that can be run in a workspace. All the server does in this example is return anything the client sends to it:

```
| server |
server := SocketAccessor newTCPserverAtPort: 9009.
server listenFor: 5.
[ | acceptedSocket |
   "wait for a new connection"
   acceptedSocket := server accept.
   "fork off processing of the new stream socket"
   [   | stream char |
      stream := acceptedSocket readAppendStream.
      stream lineEndTransparent.
      [ (char := stream next) isNil ] whileFalse: [
         stream nextPut: char; commit ].
```

```
        stream close. "close the stream when client disconnects"
    ] forkAt: Processor userSchedulingPriority -1.
] repeat.            "end loop"
```

You can use your favorite telnet client to connect to port 9009 of your machine ('localhost') to test this server.

Building UDP socket clients and servers

The UDP protocol is the protocol for transferring datagrams. It is referred to as a "connectionless" protocol, meaning that it usually does not hold open a connection the way a TCP connection does.

The UDP protocol does not give any guarantee that datagram packets are received in any particular order, or that they are received at all, as TCP does. Packets may also get duplicated. Responsibility for acknowledging receipt of a packet and reassembling the packets in order is the responsibility of your application. We do not cover these details here.

To create a UDP socket, two instance creation convenience methods are provided:

newUDP

Creates a UDP socket with the local machine as host on a system assigned port.

newUDPserverAtPort: *portNo*

Creates a UDP socket with the local machine as host on *portNo*.

Both of these methods create the socket and *bind* it to an IPSocketAddress. newUDP is most appropriate for clients, since the port number isn't usually important, and the server gets the address when it receives a message (see below). You can use newUDP for a server as well, but you will have to then make the system-assigned port number known somehow; typically you want to specify the port number.

The long version of the server creation sequence, specifying port 9009, is:

```
| sockAccessor sockAddr |
sockAccessor := SocketAccessor family: SocketAccessor AF_INET
    type: SocketAccessor SOCK_DSOCK.
sockAddr := IPSocketAddress hostAddress: IPSocketAddress thisHost
    port: 9009.
sockAccessor bindTo: sockAddr.
```

For a system-assigned port, the port can be specified as 0, the anonymous port. The system then assigns the port.

Since UDP is a connectionless protocol, there is no equivalent to the connectTo: or accept operations (though there is a "connected mode" which we'll describe below). Instead, communication is performed using the sendTo:/receiveFrom: idiom, (see SendTo:/ReceiveFrom: style communication).

A very simple UDP server that receives a packet from any client, and does nothing with it except record the fact by writing "Received" to the Transcript, might look like this:

```
| server peerAddr |
server := SocketAccessor newUDPserverAtPort: 9009.
peerAddr := IPSocketAddress new.
buffer := ByteArray new: 1024.
[ server readWait.
server receiveFrom: peerAddr buffer: buffer.
Transcript show: 'Received' ; cr ; flush. ] repeat.
```

After receiving a message, the server knows the address of the client, which is now held in peerAddr. This address can be used for sending back an acknowledgement, or stored in a collection for broadcasting messages back to all clients (see the chat server example in SocketAccessor).

An equally simple-minded client that only sends a message to the server would be the following:

```
| client serverAddr buffer |
client := SocketAccessor newUDP.
serverAddr := IPSocketAddress hostName: 'bruce-linux' port: 9009.
buffer := 'This is a udp test packet'
client sendTo: serverAddr buffer: buffer.
client close.
```

The client doesn't care what its port number is, so newUDP is appropriate as the creation message. It does need the server address and port number, which is uses for the sendTo: message. To receive an acknowledgement, it would have to do a receiveFrom:.

Connected UDP

UDP sockets are usually connectionless, as shown above. There is a use for connected UDP sockets, however, namely that using a connected UDP socket is the only way that the client can receive ICMP error messages back from the server.

When in connectionless mode, the socket must use sendTo: and receiveFrom: messages. If the client sends a connectTo: message, however, the IPSocketAddress to which it connects becomes the default address, and it can now send and receive using the read/write idiom.

There is still no equivalent to accept on the server side; it continues to operate connectionless. The differences are entirely on the client side. Notably, when connected, only datagram packets received from the connected peer are returned; all others are ignored.

Reading from and Writing to a Socket

Once you have a socket created and connected, you have your choice of methods for communicating over that socket.

The simplest defines a read and/or write stream on the socket, and then communicates using the usual Stream protocol.

Slightly more complicated, but familiar to users of sockets especially in Unix environments, are the “read/write” and “sendto/recvfrom” idioms.

Stream Style Communication

Creating a Stream on a socket provides a simple method of communicating over a socket. The Stream protocol handles a number of issues that can complicate communication, such as coordinating reading and writing.

Using Stream protocols has been illustrated in the examples above. This is a simple mechanism familiar to Smalltalk programmers, and is quite straight-forward, except to note that you do need to send a commit message to clear the Stream, to ensure that the entire contents of the buffer is written.

An important sampling of the protocol for reading and writing a socket stream is (assuming String and Character data):

next

Read and return the next Character on the Stream.

upTo: *anObject*

Read and return a subcollection from the current stream position up to, but excluding, the first occurrence of the specified Character.

throughAll: *aCollection*

Read and returns the Stream from the current position up to, and including, the first occurrence of *aCollection*, typically a String.

nextPut: *anObject*

Write the specified Character onto the Stream.

nextPutAll: *aCollection*

Write the specified String onto the Stream.

flush

Write any unwritten data in the buffer.

commit

Writes any buffered data to the OS.

Browse the accessing method category in the Stream class for other important messages.

Note that a socket stream does not usually have an end-of-file (EOF) until the socket is closed (when read(2) returns 0). Accordingly the upToEnd message blocks until the socket has closed, and so must be used with care.

In various examples in the chapter we use both flush and commit messages. We use flush on internal stream writing to the Transcript, just so the data goes somewhere. And, we send commit, on the other hand, to the stream on a socket, which is an external stream, to ensure that the data is written out to the OS. The commit message, in this context, is equivalent to flush() on UNIX systems. Sending a flush message generally works as well.

Positioning on a Stream

A Stream on a socket is not positionable, or at least not reliably so. For positioning on a socket stream, limit yourself to the following messages:

peek

Answer what would be returned with a self next, without changing position. If the receiver is at the end, answer nil.

peekFor: *anObject*

Answer false and do not move the position, if the next object is not *anObject*, or if the receiver is at the end. Answer true and increment the position if the next object is *anObject*.

skipToAll: *aCollection*

Skip forward to the next occurrence (if any) of *aCollection*. If found, leave the stream positioned before the occurrence, and answer the receiver; if not found, answer nil, and leave the stream positioned at the end.

throughAll: *aCollection*

Answer a subcollection from the current position through the occurrence (if any, inclusive) of *aCollection*, and leave the stream positioned after the occurrence. If no occurrence is found, answer the entire remaining stream contents, and leave the stream positioned at the end.

upToAll: *aCollection*

Answer a subcollection from the current position up to the occurrence (if any, not inclusive) of *aCollection*, and leave the stream positioned before the occurrence. If no occurrence is found, answer the entire remaining stream contents, and leave the stream positioned at the end.

skipUpTo: *anObject*

Skip forward to the occurrence (if any, not inclusive) of *anObject*. If not there, answer nil. Leaves positioned before *anObject*.

Line-end conversion

Different operating systems and different protocols use different line-end conventions, to indicate the end of a line in a text (ASCII) file. For example, DOS/Windows CR-LF (carriage-return/line-feed), UNIX uses LF, and Macintosh uses CR. Also, the FTP specification (RFC 959) gives 8-bit ASCII as the default data format, with CR-LF as the line-end convention. Accordingly, converting line-end characters is necessary in some Stream transactions, such as file transfers.

VisualWorks internally uses CR as the line-end character, and, based on the operating system platform, assumes what a file's line-end convention is and converts accordingly. So, if reading a file on a Windows system, it assumes the line-end is CR-LF, and converts it to CR upon reading. Similarly, when writing, it converts its internal CR to a CR-LF, so the file is stored properly according to the platform.

With data and stream formats coming over a socket connection, it is not as obvious what convention to follow for reading and writing streams. VisualWorks provides the following simple protocol for specifying the proper conversion:

lineEndCR

Converts between VisualWorks' line-end and CR.

lineEndLF

Converts between VisualWorks' line-end and LF.

lineEndCRLF

Converts between VisualWorks' line-end and CRLF.

lineEndTransparent

Does no line-end conversion.

For example, with a read/write `Stream` on a socket doing FTP transfers, you can specify `lineEndCRLF` on the Stream. Then, on a read CRLF is converted to VisualWorks' internal CR representation, and on a write the VisualWorks is converted to CRLF for FTP conventions. The conversion is handled automatically.

We did this in the simple FTP example earlier in this chapter. It began like this:

```
sockAccessor := SocketAccessor
   newTCPclientToHost: 'ftp.parcplace.com' port: 21.
stream := sockAccessor readAppendStream.
"Set the FTP line-end convention"
stream lineEndCRLF.
```

By converting to CR within VisualWorks, searching up to an end-of-line is simplified. Instead of having to know whether you are reading a stream up to CR, LF, or CRLF to get a line, you can simply do:

```
line := stream upTo: Character cr.
```

Similarly, you can terminate a line simply with cr:

```
stream nextPutAll: buffer ; cr ; flush.
```

The conversion is handled by the specified line end convention for the stream.

In some cases, however, you do not want the line-ends converted at all. In this case, specify lineEndTransparent, and VisualWorks does no conversion.

Waiting for data

For process synchronization, you should tell the read and write processes to wait until data is available. This is handled “under the covers” by the streaming mechanism. For the read/write and send/receive idioms described below, however, you need to use readWait and writeWait messages to wait for data. readWait and writeWait employ semaphores for signaling when data is ready.

You may be able to get by for some limited testing without using these, but in the long run you will need them. So, get used to using these messages, as illustrated in the following sections.

readWait

Blocks indefinitely, until there is data on the socket to read, then signals to proceed.

readWaitWithTimeoutMs: *anInteger*

Blocks until there is data on the socket to be read, then signals to proceed, or times out after the specified number of milliseconds. Returns true if a time-out occurred, or false otherwise.

writeWait

Blocks indefinitely, until there is data in the buffer to write, then signals to proceed.

writeWaitWithTimeoutMs: *anInteger*

Blocks until there is data in the buffer to be written, then signals to proceed, or times out after the specified number of milliseconds. Returns true if a time-out occurred, or false otherwise.

Read/Write Style Communication

The read/write idiom uses the protocol defined for a general, buffered I/O in class IOAccessor. In general, this provides support for the read(2) and write(2) buffered I/O defined on UNIX (but see the note below). The read/write idiom only makes sense for "connected" socket protocols, such as TCP.

The read/write messages should only be used with streaming sockets (type SOCK_STREAM). For "connectionless" protocols, like UDP, and other socket types, use the send/receive commands.

The readInto:* and writeFrom:* commands map to read(2) and write(2) on all platforms except Windows, where they map to recvfrom(2) and sendto(2). Because the error messages returned are different, the readInto:* and writeFrom:* messages are not cross-platform compatible.

Reading and writing is done through a buffer, which is either a ByteArray or a String. The buffer itself is under application control, and must be managed appropriately. It is also the responsibility of the application to read from or write to the socket from the buffer precisely the intended amount of data.

The basic protocol is as follows. Browse the IOAccessor class for additional methods.

readInto: *aBuffer*

Attempts to read bytes into *aBuffer*, until either the buffer is filled or data is exhausted. Returns the number of bytes actually read, as a SmallInteger.

readInto: *aBuffer* **startingAt:** *index* **for:** *count*

Attempts to read up to *count* bytes into *aBuffer*, starting at *index* in *aBuffer*. Returns the number of bytes actually read, as a SmallInteger.

readInto: *aBuffer* **untilFalse:** *aBlock*

Attempts to read bytes into *aBuffer* until *aBlock* evaluates to false or the buffer is filled. *aBlock* is a one-arg block that is sent the count thus far. While it evaluates to true, reads are repeated, up to the size of the buffer. Returns the number of bytes actually read, as a SmallInteger.

writeAll: *aBuffer*

Attempts to write all data from the buffer onto the socket. Ensures that the number of bytes written is the same as the buffer size, unless an error occurs. Returns the number of bytes actually written, as a SmallInteger.

writeFrom: *aBuffer*

Attempts to write all data from the buffer onto the socket. Returns the number of bytes actually written, as a SmallInteger.

writeFrom: *aBuffer* **startingAt:** *index* **for:** *count*

Attempts to write *count* bytes onto the socket, starting at *index* in *aBuffer*. Returns the number of bytes actually written, as a SmallInteger.

writeFrom: *aBuffer* **startingAt:** *index* **forSure:** *count*

Attempts to write *count* bytes onto the socket, starting at *index* in *aBuffer*. Ensures that *count* bytes are written, unless an error occurs. Returns the number of bytes actually written, as a SmallInteger.

writeFrom: *aBuffer* **startingAt:** *index* **for:** *count* **untilFalse:** *aBlock*

Attempts to write bytes onto the socket, starting at *index* in *aBuffer*, until either *aBlock* answers false or *count* bytes are written. *aBlock* is a one-arg block which is sent the number of bytes written thus far. Returns the number of bytes actually written, as a SmallInteger.

The following simple example collects text from a dialog, writes the text out on a socket and reads the reply, displaying it in the Transcript. Used with the server example above, it receives what it sent.

It maintains two buffers, one for reading and one for writing. The write buffer is filled with text received from the dialog using Stream messages, and the buffer contents are then written. Similarly, the read buffer is filled from the socket, and the buffer contents is then written to the Transcript using a Stream. The streams themselves, however, are not involved in the socket communication.

```
sockAccessor := SocketAccessor
   newTCPclientToHost: 'bruce-linux' port: 6001.
buffer1 := ByteArray new: 100.
buffer2 := ByteArray new: 100.
outProc := [ [(inputString := Dialog request: 'Say something') isEmpty]
```

```
    whileFalse: [
        sockAccessor writeWait.
        outStream := ( buffer1 withEncoding: #UTF_8 ) writeStream.
        outStream nextPutAll: inputString ; cr.
        sockAccessor
            writeFrom: buffer1 startingAt: 1 for: inputString size + 1. ]]
forkAt: Processor activePriority -1.
inProc := [ [ | size |
    ( sockAccessor readWaitWithTimeoutMs: 10000 )
        ifTrue: [ sockAccessor close. ^nil].
    size := sockAccessor readInto: buffer2.
    inStream := ( buffer2 withEncoding: #UTF_8) readStream.
    1 to: size do: [ :x |
    "next line should use ForkedUI"
    Transcript nextPut: inStream next ; flush ]] repeat
    ] forkAt: Processor activePriority -1
```

Note that putting data to the Transcript, near the end, is performing a UI operation in a forked process. This may cause VisualWorks to crash since the UI is not thread safe. This code should use ForkedUI, as described in the *Application Developer's Guide*.

Since each successive input string can be of different length, and we only want to write out the current string, not the entire buffer with any old data or filler zeros, we use the writeFrom:startingAt:for: message, so just the right part of the buffer is written.

For the read, on the other hand, we want to accept all the data that the socket has to offer, so we use simply the readInto: message. But, since we only want to process the new data received, we capture the number of bytes read and use it when writing to the Transcript.

Notice also the use of writeWait and readWaitWithMs:. Waiting for data is in general a good practice and will prevent some errors. Using the time-out version on waiting for read data isn't necessary, but does allow us to give up and close a connection, as shown. Here, if there is no data to read for 10 seconds, we give up and close the socket.

SendTo:/ReceiveFrom: style communication

The send/receive idiom provides a socket-specific interface, similar to that provided under UNIX by sendto(2), recvfrom(2) and select(2). This idiom gives you access to all socket behavior, regardless of the socket type or protocol, except for socket type SOCK_RAW.

The send/receive idiom is very similar to the read/write idiom described above, except that send/receive messages specify the socket accessor, and they *require* synchronization using readWait and writeWait. These are described in Waiting for data.

The send and receive protocol is as follows:

receiveFrom: *aSocketAddress* **buffer:** *aBuffer*

Attempts to read bytes from host *aSocketAddress* into *aBuffer*. Returns the number of bytes actually read. As a side effect, *aSocketAddress* is set to the sender's IPSocketAddress.

receiveFrom: *aSocketAddress* **buffer:** *aBuffer* **start:** *index* **for:** *count*

Attempts to read *count* bytes from host *aSocketAddress* into *aBuffer*, starting at *index*. Returns the number of bytes actually read. As a side effect, *aSocketAddress* is set to the sender's IPSocketAddress.

receiveFrom: *aSocketAddress* **buffer:** *aBuffer* **start:** *index* **for:** *count* **flags:** *flags*

Attempts to read *count* bytes from host *aSocketAddress* into *aBuffer*, starting at *index*. *flags* is a SmallInteger specifying any special requirements. Returns the number of bytes actually read. As a side effect, *aSocketAddress* is set to the sender's IPSocketAddress.

sendTo: *aSocketAddress* **buffer:** *aBuffer*

Attempts to write bytes from *aBuffer* to the host *aSocketAddress*. Returns the number of bytes actually written.

sendTo: *aSocketAddress* **buffer:** *aBuffer* **start:** *index* **for:** *count*

Attempts to write *count* bytes from *aBuffer*, starting at *index*, to host *aSocketAddress*. Returns the number of bytes actually written.

sendTo: *aSocketAddress* **buffer:** *aBuffer* **start:** *index* **for:** *count* **flags:** *flags*

Attempts to write *count* bytes from *aBuffer*, starting at *index*, to host *aSocketAddress* into. *flags* is a SmallInteger specifying any special requirements. Returns the number of bytes actually written.

Modifying our previous example slightly, we get the same effect:

```
sockAccessor := SocketAccessor
    family: SocketAccessor AF_INET
    type: SocketAccessor SOCK_STREAM.
sockAddr := IPSocketAddress hostName: 'bruce-linux' port: 6001.
sockAccessor connectTo: sockAddr.
buffer1 := ByteArray new: 100.
buffer2 := ByteArray new: 100.
outProc := [ [(inputString := Dialog request: 'Say something') isEmpty]
    whileFalse: [
        sockAccessor writeWait.
        outStream := ( buffer1 withEncoding: #UTF_8 ) writeStream.
        outStream nextPutAll: inputString ; cr.
        sockAccessor sendTo: sockAddr buffer: buffer1 start: 1
            for: inputString size + 1. ]
    ] forkAt: Processor activePriority -1.
inProc := [ [ | size |
    sockAccessor readWait.
    size := sockAccessor receiveFrom: sockAddr buffer: buffer2.
    inStream := ( buffer2 withEncoding: #UTF_8) readStream.
    1 to: size do: [ :x |
    "next line should use ForkedUI"
    Transcript nextPut: inStream next ; flush ]] repeat
    ] forkAt: Processor activePriority -1
```

Creating a socket is exactly the same, though in the example above we show a variant, defining the socket and then connecting to the host as a separate operation. We do that here because we have to hold a SocketAddress to the host anyway, for use by the send and receive commands.

Send/Receive Flags

The argument to the flags: keyword in the receiveFrom:buffer:start:for:flags: and sendTo:buffer:start:for:flags: messages specify special handling, as required. The argument is a SmallInteger, but is provided by using defined constants. Three flags are implemented:

MSG_OOB

Permits processing out-of-bounds data. (Caution: this doesn’t work properly at this time.)

MSG_PEEK

Causes the receiver to return data from the beginning of the receive buffer, without removing the data from the buffer.

MSG_DONTROUTE

Sends data without using routing tables.

These constants are implemented as class messages, so the value is accessed, for example, by:

```
optionFlags := SocketAccessor MSG_OOB
sockAccessor receiveFrom: host buffer: buffer start:1 for: 10 flags:
optionFlags
```

To use multiple flags, you can use bitOr: :

```
optionFlags := SocketAccessor MSG_OOB bitOr: SocketAccessor
MSG_PEEK
```

Socket Error Handling

Sockets are an operating system provided feature, so socket communication errors are caught as subclasses of OSError.

OSError Subclass	OS Errors Covered
OsIllegalOperation	EAFNOSUPPORT, EISCONN, EISDIR, ENOTCONN, ENOTDIR, ENOTSOCK, EOPNOTSUPP
OsInaccessibleError	EACCES, EADDRINUSE, ENOENT, EPERM, EROFS
OsInvalidArgumentError	EBADF, EFAULT, EINVAL, ELOOP, EMSGSIZE, ENAMETOOLONG
OsNeedRetryError	EAGAIN, EALREADY, EINTR, EWOULDBLOCK
OsNoResourcesError	ENOBUFS, ENOMEM
OsNotification	EINPROGRESS
OsTransferFaultError	ECONNREFUSED, EIO, ENETUNREACH, ENOSPC, EPIPE, ETIMEDOUT

As the above table indicates, the error classes alone provide only a very coarse-grained view of the errors that might be returned by a socket. Accordingly, error trapping code such as:

```
[ code that can fail with EACCES ]
    on: OsInaccessibleError
    do: [ :ex | ex someAction ]
```

will respond to errors EACCES, EADDRINUSE, ENOENT, EPERM, or EROFS. This is usually too coarse.

For a finer granularity, most OS errors are represented by instances of Signal. (Error handling in VisualWorks was done using instances of Signal before the ANSI-compliant, class-based exception system was introduced in 3.0).

OsError subclass	OSErrorHolder signal	OS socket error
OsIllegalOperation	inappropriateOperationSignal	EISDIR, ENOTDIR, ENOTSOCK, EOPNOTSUPP
	unpreparedOperationSignal	EISCONN, ENOTCONN
	unsupportedOperationSignal	EAFNOSUPPORT
OsInaccessibleError	existingReferentSignal	EADDRINUSE
	nonexistentSignal	ENOENT
	noPermissionsSignal	EACCES, EPERM, EROFS
OsInvalidArgumentsError	(None)	EFAULT, EINVAL
	badAccessorSignal	EBADF
	rangeErrorSignal	ELOOP, EMSGSIZE, ENAMETOOLONG
OsNeedRetryError	notReadySignal	EAGAIN, EALREADY, EWOULDBLOCK
	transientErrorSignal	EINTR
OsNoResourcesError	noMemorySignal	ENOBUFS, ENOMEM
OsNotification	operationStartedSignal	EINPROGRESS
OsTransferFaultError	(None)	EIO
	peerFaultSignal	ECONNREFUSED, ENETUNREACH, EPIPE, ETIMEDOUT
	volumeFullSignal	ENOSPC

While not as fine-grained as the socket errors coming from the OS, it is considerably better, and adequate for most purposes.

The signals are returned by messages sent to the OSErrorHolder class, where the message name is the same as the signal shown above. To use the signals, we slightly modify the schematic example above:

```
[ code that can fail with EACCES ]
   on: OSErrorHolder noPermissionsSignal
   do: [ :ex | ex someAction ]
```

The more specific exception, or Signal, is referenced by the expression OSErrorHolder noPermissionsSignal. The exception handler now responds to EACCES, EPERM, and EROFS, but not to EADDRINUSE or ENOENT, a small improvement. It is, however, a much larger improvement than appears, because there is a large number of other OS errors that would trigger OsInaccessibleError, but not OSErrorHolder noPermissionsSignal.

Note that EFAULT, EINVAL, and EIO do not have a Signal, but map directly to an exception class (OsInvalidArgumentsError for EFAULT and EINVAL, and for OsTransferFaultError EIO).

Also, be aware that not all operating systems return the same error code for a given error; there is some variation. This can be important for applications that are portable between operating systems, where you may have to trap more than one error to catch an exception condition.

If your application needs to distinguish specific conditions for which a Signal is not provided, you can add it. Browse the initialization class methods in OSErrorHolder for examples. The system looks up the error by its error code (using reportOn: and similar messages), some of which are shown below.

Below are the error names, numbers, and brief descriptions that may be returned by the principle socket commands. On MS Windows, the names are prefixed with “WSA” and the error codes are 10000 higher than on UNIX systems.

Error Name	Description	Code	Win Code
EPERM	Operation not permitted	1	10001
ENOENT	No such file or directory	2	10002
EINTR	Interrupted system call	4	10004
EIO	I/O error	5	10005
EBADF	Bad file number	9	10009
EAGAIN	Try again	11	10011
ENOMEM	Out of memory	12	10012
EACCES	Permission denied	13	10013
EFAULT	Bad address	14	10014
ENOTDIR	Not a directory	20	10020
EISDIR	Is a directory	21	10021
EINVAL	Invalid argument	22	10022
ENOSPC	No space left on device	28	10028
EROFS	Read-only file system	30	10030
EPIPE	Broken pipe	32	10032
ENAMETOOLONG	File name too long	36	10036
ELOOP	Too many symbolic links encountered	40	10040
ENOTSOCK	Socket operation on non-socket	88	10088
EMSGSIZE	Message too long	90	10090
EAFNOSUPPORT	Address family not supported by protocol	97	10097
EADDRINUSE	Address already in use	98	10098
ENETUNREACH	Network is unreachable	101	10101
ENOBUFS	No buffer space available	105	10105
EISCONN	Transport endpoint is already connected	106	10106
ENOTCONN	Transport endpoint is not connected	107	10107
ETIMEDOUT	Connection timed out	110	10110
ECONNREFUSED	Connection refused	111	10111

Error Name	Description	Code	Win Code
EALREADY	Operation already in progress	114	10114
EINPROGRESS	Operation now in progress	115	10115
EWOULDBLOCK	Operation would block	11	10011

Trapping socket and protocol errors

The various protocols discussed in the following chapters have their own error classes. Sometimes you need to trap both protocol-specific and general socket errors. You do this by nesting on:do: statements.

For example, attempting to establish an HTTP connection may fail because there is no internet connection at all. This produces a socket error rather than an HTTP error. However, if the socket succeeds, an HTTP error may still occur. To trap both, use a construct such as:

```
client := HttpClient new.
req := HttpRequest get: 'http://www.some.net/page.html'.
[[ resp := client executeRequest: req ]
    on: OS.OsInaccessibleError
    do: [ :y | y inspect. "dialog - no connection" ]
    ] on: HttpException do: [ :ex | ex proceed]
```

Option level control

The socket level options controls the operation of sockets. Options may exist at multiple protocol levels, and all are available at the socket level.

Protocols and options are identified by integer values

To get and set an protocol option, send these messages:

setOptionsLevel: *protoInt* **name:** *optInt* **value:** *value*

Sets the option *optInt* for protocol *protoInt* to *value*.

getOptionsLevel: *protoInt* **name:** *optInt*

Returns the value of option *optInt* for protocol *protoInt*.

Protocol levels and options are identified by integer values. For convenience, many of these are represented by class methods, whose selectors are the protocol and option names, that return the integer values.

For example, the protocol level for sockets is named SO_SOCKET, and the socket level option that specifies the size of the receive buffer is named SO_RCVBUF. To retrieve that constant, send the message to SocketAccessor:

```
SocketAccessor SO_RCVBUF
```

So, to set the value of this option, send this message to the socket:

```
sockAccessor setOptionsLevel: SocketAccessor SOL_SOCKET
   name: SocketAccessor SO_RCVBUF
   value: 8192.
```

To retrieve the current value of this option for the socket level, send this message to the socket:

```
sockAccessor getOptionsLevel: SocketAccessor SOL_SOCKET
   name SocketAccessor SO_RCVBUF
```

The returned value is a ByteArray representing a 32-bit (4-byte array) or 64-bit (8-byte array) signed integer. To interpret it, do a conversion such as:

```
retVal := sockAccessor getOptionsLevel: SocketAccessor SOL_SOCKET
   name: SocketAccessor SO_RCVBUF.
retVal changeClassTo: UninterpretedBytes.
retVal := retVal longAt: 1. "use longLongAt: for an 8-byte array"
```

Now, instead of something like #[0 32 0 0], the value is a more meaningful 8192, or whatever the value happens to be.

Currently, the only protocol level defined is SOL_SOCKET, the socket level. Others can be defined using the same pattern. Browse the definition in the constants-socket option levels class method category in SocketAccessor.

Options for a few protocols are defined in other class method categories. See, for example, constants-socket options, constants-tcp options, and constants-ip options. Method comments describe their usage.

Solving Common Socket Problems

How do I avoid the 'Address in use' error?

During development and testing, you frequently open a socket on an address, close it, and then want to use it again for a repeat test. Since addresses aren't released immediately, you frequently get this error message.

To avoid this message, set the socket option SO_REUSEADDR. Send a `soReuseaddr:` message to your `SocketAccessor`, as follows:

```
sockAccessor soReuseaddr: true
```

10

XML Framework

XML (eXtensible Markup Language) has become an accepted standard for representing structured data between applications. XML is used internally to VisualWorks as a portable source code representation.

The XML framework supports working with XML documents using either the DOM (Document Object Model) or SAX (Simple API for XML) APIs.

Schema support as documented in this chapter remains in preview, but can be loaded from the preview/parcels/ subdirectory of your VisualWorks installation.

The discussion in this section assumes you already understand the essentials of XML and its components. For more basic information, there are a lot of resources available. See http://www.xml.org as a beginning resource.

Working with XML Documents

XML presents data as a structured document. The XML DOM (Document Object Model) is an programming interface for accessing that data as a tree structure. Using the DOM, you can build documents, navigate their structure, and add, modify, or delete elements and content.

The DOM represents an XML document as a hierarchy of objects. Being an object model, it is a natural way for VisualWorks to operate on XML documents.

Parsing an XML Document

Frequently you receive an XML document as a resource on the internet. Or, you may have it stored as a file. In any case, a standard way to work with it is to first represent it in memory. In VisualWorks, you do this by representing it as a XML.Document, which you do using XMLParser. (Note that XML.Document is a different class than Graphics.Document.)

The basic procedure is to generate an instance of XMLParser, and send it a parse: message with the XML resource to be parsed.

```
| parser |
parser := XMLParser new.
parser parse: 'mydocument.xml' asFilename.
```

In this example, the resource is given as a Filename, but it could be an URI or a ReadStream. For an URI, send asURI to a String describing the protocol, host, and path (refer to the Internet Client Developer's Guide for information on URI support):

```
| parser |
parser := XMLParser new.
parser parse: 'http://www.w3.org/XML' asURI.
```

By default, the parser is validating, so the document must include a document type declaration (DTD). If the document is only well-formed, you need to turn off validation by sending a validate: message to the parser with false as argument. For example:

```
| parser |
parser := XMLParser new.
parser validate: false.
parser parse:
  '<?xml version="1.0"?><doc><para>Hello, world!</para></doc>'
    readStream.
```

To summarize this protocol:

parse: *aDataSource*

Selects aDataSource, which may be an URI, a Filename, or a ReadStream. If successful, returns a XML.Document.

validate: *aBoolean*

Sets the parser's validation flag, determining whether the parser will validate the document against its document type definition. By default, this is set to true.

Validating Against a Schema

An XML Schema provides an alternative, and more powerful, document structure specification than does a DTD document. As an alternative to a DTD, you can validate a document against a schema. This is done by first parsing the schema, then parsing the document, and finally by validating the document against the schema. For example:

```
schemaURI := 'http://...' asURI.
docURI := 'http://...' asURI.
schema := SchemaHandler new parse: schemaURI.
doc := XMLParser new
    validate: false;
    parse: docURI.
schema validate: doc
```

A Schema is returned.

Schema support is provided as a preview at this time. Load the XSchema parcel, preview/parcels/XSchema.pcl.

Selecting a XMLParser Driver

By default, the parser represents the XML document according to the Domain Object Model (DOM), and the parser returns an XML.Document that supports the DOM API. There are occasions, however, when other processing is necessary.

The parser operates by handling SAX (Simple API for XML) events as specified by a SAX driver. The default driver is DOM_SAXDriver. There are a few other drivers provided, and you can build your own (see Building a SAX Driver).

To specify another driver, send a handlers: message to the parser.

handlers: *aSAXDriver*

Assigns aSAXDriver as the parser's current SAX driver.

In general, you only want to assign an alternate driver when you have built one for your own application. One driver that might be of some use, however, is the NullSAXDriver. This driver does simple syntax checking of a XML document without further processing. So, to substitute this driver to check the file, send a handlers: message with a new instance of the driver:

```
| parser |
parser := XMLParser new.
parser handlers: NullSAXDriver new.
parser parse: 'http://www.w3.org/XML' asURI.
```

This example does its work and returns nil, unless errors occur. For syntax checking, you still need to provide handlers for syntax errors, as described in XML Error Handling.

Browse the SAXDriver hierarchy to see what drivers are available. In general the classes provide superclasses for your own drivers.

For advanced users, it is possible to specify handlers for different aspects of a document. Browse the XMLParser contentHandler:, dtdHandler:, entityResolver: and errorHandler: methods for this option.

Accessing XML Document Elements

In the DOM, a document is represented as a tree structure of nodes. The main node is the document itself. In VisualWorks the DOM is implemented as a collection of classes, all subclasses of Node.

The following classes give a high-level view of the parts of a Document:

```
Node
   Attribute
   Comment
   Document
      DocumentFragment
   Element
   Entity
   Notation
   PI
   Text
```

To work with the document, a large number of messages are provided for accessing these various parts of a document.

root

Sent to a Document, returns the root element of the document.

document

For any element, returns the enclosing Document object.

children

Returns an OrderedCollection of all nodes immediately in the receiver, or an empty collection if there are none.

parent

Returns the node immediately containing the receiver.

elementNamed: *aNodeTag*

Returns the unique child element named *aNodeTag* in the receiver. An error is raised if there is not exactly one.

elementsNamed: *aNodeTag*

Returns an OrderedCollection of child elements named *aNodeTag*.

anyElementNamed: *aNodeTag*

Same as elementNamed:, except that the search is recursive from the receiver, so the receiver, its children, grandchildren, etc., are included in the search. An error is raised if there is not exactly one.

anyElementsNamed: *aNodeTag*

Same as elementsNamed:, except that the search is recursive from the receiver, so the receiver, its children, grandchildren, etc., are included in the search.

attributes

Returns a OrderedCollection of Attribute objects in the receiving Element.

selectNodes: *aBlock*

Returns an OrderedCollection of Node objects satisfying the selection criteria specified in *aBlock*.

The following sections will use these messages to explore a Document.

Get Document Root Element

The Document object may have many elements besides the root element, such as various comments or processing instructions. For example, parsing a help file yields a document with two elements: a processing instruction and the root element. To verify this, evaluate the following in a workspace:

```
parser := XMLParser new.
parser validate: false.
pdoc := parser parse: '..\help\01-xml-language\01-language.xml'
   asFilename.
pdoc children inspect.
```

(The filename in the above example is not portable, so will have to be written differently on non-Windows platforms.)

To extract only the root element, which contains the whole DOM tree structure, send a root message to the parsed document:

```
docRoot := pdoc root.
```

This is an Element object to which you can send other messages, and so traverse the document structure.

Selecting Elements

An XML document is structured as a hierarchy of elements with a single root element. Depending on the individual document, the structure may be very shallow, as in the case of a well-formed but unstructured document, or quite deep. To make use of the XML document involves traversing and digging through this element hierarchy.

The children message returns an OrderedCollection of elements contained immediately in the receiving element.

```
parser := XMLParser new.
pdoc := parser parse:
   'http://www.w3.org/XML/1999/XML-in-10-points' asURI.
elementCollection := pdoc root children.
```

The contents of the resulting collection may not all be elements as such. For example, elementCollection in the above code contains (at the time of this writing) some XML.Text nodes as well as Element nodes. This can be important when working down through the hierarchy because a Text does not respond to children.

The isElement message returns a Boolean indicating whether the receiver is an Element or not. You can use it to collect just those nodes that are elements, for example:

```
elementCollection := pdoc root children select: [ :el | el isElement ]
```

The elements of this collection now all respond to children, and you can continue digging into the hierarchy.

It is also frequently desirable to select only those elements with a particular tag, or name. For example, when dealing with a specific node, you may want to deal only with elements tagged “partNum”. To collect all these elements in a node (aNode), send a elementsNamed: message with a NodeTag or String argument:

```
partNumElements := aNode elementsNamed: 'partNum'.
```

The String format shown here only works if the element is in the default XML namespace; otherwise the argument must be an instance of NodeTag. You may retrieve a NodeTag from an element by sending a tag message to the Element, and then use that tag to identify other elements with the same tag. This can be useful for retrieving all other elements with the same tag as one you already have:

```
subjTag := someElement tag.
tagGroup := newDoc root elementsNamed: subjTag.
```

Alternatively, you can create a NodeTag by sending a qualifier:ns:type: message to a new instance:

```
subjTag := NodeTag new
   qualifier: '' ns: 'http://www.w3.org/1999/xhtml' type: 'a' .
tagGroup := newDoc root anyElementsNamed: subjTag.
```

There are variants of the elementsNamed: message, such as elementNamed:, which returns the unique element, if there is one, or an error otherwise. The messages anyElementNamed: and anyElementsNamed: (used above) are similar, but are recursive from the receiver element, and so include the receiver node and all children nodes, and all their children, etc., in the search. So, the above example returns all elements tagged “a” in the document.

Selecting Attributes

Elements often have attributes, specifying special features of the element. The attributes message, sent to an Element (anElement), returns an OrderedCollection of an element’s attributes.

```
attrs := anElement attributes.
```

Attributes are essentially key/value pairs, where the key is the attribute name, and the value is a String. The messages for accessing these are:

tag

Returns the Attribute name, as a NodeTag.

value

Returns the Attribute value.

To make use of an Attribute, you will need to search through the collection of attributes until you find one you are interested in, and then get its value. For example, if you need to process an “href” attribute for an element, you will search for that attribute and return the value. For example:

```
( attrs detect: [ :attr | attr tag type = 'href' ]  ifNone: [ ] ) value.
```

Since attributes are already key/value pairs, it may be worth setting them into a Dictionary, especially for repeated access:

```
attrDict := Dictionary withAll: (aCollection collect:
   [ :each | Association key: each tag type value: each value ]) .
```

Building a Document

Besides handling XML documents that your application receives, for conducting web-based commerce it is also necessary to build XML documents. You can do this simply by assembling a long string and transmitting that over the transport, but this places all of the responsibility for building proper XML on your application.

VisualWorks provides facilities for building an XML DOM tree that alleviates some of the responsibility for building a syntactically correct XML document.

Not all aspects of a document are supported, however, so you may need to provide some other mechanism for adding these aspects to the document. For example, the XML prolog and DTD declarations are not supported by the XML framework. If required in your application, these need to be written onto output stream before any document elements, and so are not handled as part of the document itself. (See Writing the XML Document).

This section describes how to build an XML document using the VisualWorks XML support, and noting where methods not included in the XML framework are required. The general procedure is to create an XML.Document instance and add nodes.

Create a Basic Document

The basic document is built simply by creating an instance of XML.Document:

```
newDoc := XML.Document new.
```

This is too basic to be useful, but this is the object to which you add nodes to build the document.

Node Ordering

The most straight-forward method for adding nodes is by sending addNode: to an existing node, with the new node as argument. This is the method we will use in the following discussion.

However, addNode: adds the new node to the end of the receiver's collection of nodes. Accordingly, you need to be careful to add nodes in order, from the start of the XML document to the end.

If you must insert a node someplace other than at the end, realize that you can add it using OrderedCollection messages. This may be useful, for example, to ensure that processing instructions are added early in the document, prior to the root element.

Add Element Nodes

Most of the document content is in elements, which are represented as instances of XML.Element. An Element is really just an envelope for other nodes.

An Element must have a name, called its tag, an instance of NodeTag, which is used to begin and end the element in the XML output. The Element may also have attributes and/or entities. To create an Element with only a tag, send a tag: instance creation message to the class:

```
XML.Element tag: 'XML'
```

This simple creation method builds a simple NodeTag for the element, consisting only of the tag name.

If you employ XML namespaces, things become a little more complicated. Refer to Using XML Namespaces for further information.

Add a Root Element

An XML document has a single root element. If the document has a DTD, the root element tag must match the declared root in the DTD. To add a root, send an addNode: message to the Document with the Element as argument:

```
newDoc := XML.Document new.
newDoc addNode: ( XML.PI name: 'xml' text: 'version="1.0" ' ) .
newDoc addNode: ( XML.Element tag: ‘XML’ ).
```

A document can have only one root node. All further elements are added to the root node or further subnodes. To access the root, send a root message to the document:

```
newDoc root
```

Add Nested Elements

Adding other elements is similar; the only difference is the receiver node of the addNode: message. For example, to create a document hierarchy like:

```
XML
   heading1
      heading2
         body
```

send messages like this:

```
newDoc := XML.Document new.
newDoc addNode: ( XML.PI name: 'xml' text: 'version="1.0" ' ).
newDoc addNode: ( XML.Element tag: 'XML' ).
newDoc root addNode: ( ( XML.Element tag: 'heading1' )
   addNode: ( ( XML.Element tag: 'heading2' )
      addNode: ( XML.Element tag: 'body' ) ) ).
```

The nodes can, of course, be constructed individually and added to the containing node in other ways.

The PI element defines a processing instruction. Refer to Add Processing Instructions for more information.

If you assemble a collection of nodes, you can add them as subnodes as a group when creating their parent, using the tag:elements: instance creation method. For example, to add a node structure to newDoc, do:

```
nodeGroup := Array with: (XML.Element tag: 'body')
   with: ( (XML.Element tag: 'heading2')
      addNode: (XML.Element tag: 'body' ) ).
newDoc root addNode: ( XML.Element tag: 'heading1' elements:
nodeGroup ).
```

Adding Element Attributes

An element may have attributes, which are additional labels identifying the contents of an element. For example, an image element may include alignment and source information:

```
<IMG ALIGN="left" SRC="http://www.w3.org/Icons/WWW/w3c_home">
```

Attributes are instances of XML.Attribute, which is a subclass of Node. To add attributes, create the Attribute instances and add them as a collection by sending a tag:attributes:elements: instance creation message to Element, sending an addNode: message to the containing element as usual. The argument to the elements: keyword can be provided as a collection of elements or as nil.

```
attrGroup := Array
   with: (XML.Attribute name: 'ALIGN' value: 'left' )
   with: (XML.Attribute name: 'SRC' value:
      'http://www.w3.org/Icons/WWW/w3c_home'.)
newDoc root addNode:
   ( XML.Element tag: 'IMG' attributes: attrGroup elements: nil ).
```

Again, the element tag in this example is simple. To include a namespace qualifier or declare a namespace, the specified tag must be an instance of NodeTag.

Adding Text

Many elements have a text content. Text is added as another node, as an instance of XML.Text. The instance creation method is simply text:, which takes a String argument.

```
XML.Text text: 'Hello, World!'
```

The text node is added using the usual addNode: message.

```
newDoc := XML.Document new.
newDoc addNode: ( XML.PI name: 'xml' text: 'version="1.0" ' ).
newDoc addNode: ( XML.Element tag: 'XML' ).
newDoc root addNode: ( ( XML.Element tag: 'body' )
   addNode: ( XML.Text text: 'Hello, World!' ) ).
```

Add Processing Instructions

Processing instructions contain special instructions to the application that will process the XML.

A processing instruction is represented by an instance of XML.PI. Its instance creation method, name:text:, specifies the target application and the specific instruction, both as Strings. To create the initial instruction, send:

```
XML.PI name: 'target' text: 'instruction'
```

Note that the text contains all instructions for this processing instruction, including any attributes and values for the instruction (see the next example).

For example, the processing instruction that occurs at the beginning of a VisualWorks help file is:

```
<?xml-stylesheet href="01-language.css" type="text/css" title="Smalltalk Language" charset="UTF-8"?>:
```

To create the processing instruction in VisualWorks write:

```
XML.PI name: 'xml-stylesheet' text: 'href="01-language.css" type="text/css" title="Smalltalk Language" charset="UTF-8" '
```

To add this to the document, we send addNode: with the new processing instruction as the argument:

```
newDoc := XML.Document new.
newDoc addNode: (XML.PI name: 'xml-stylesheet'
   text: 'href="01-language.css" type="text/css"
      title="Smalltalk Language" charset="UTF-8" ').
```

Evaluate and inspect the above code in a workspace to see that it produces what we want.

Note that the XML prologue line,

```
<?XML version='1.0'?>
```

while it looks like a processing instruction, technically is not. It, together with DTD declarations, is part of the prologue rather than part of the XML data itself. No support for these items is included in the XML framework at this time, and so they must be written separately, before the XML data. Refer to Writing the XML Document for a suggested approach.

Writing the XML Document

Once you have built a DOM tree, you can write it out as XML on a Stream. The stream can be on a file or a communication channel.

Remember that the XML framework does not support all aspects of an XML document, such as the prolog and any document type definition information. We can handle this, however, by writing this information on the write stream before the document itself.

To write the document on the stream, send a saxDo: message to the Document with a SAXWriter instance as argument. The SAXWriter has its output set to the output stream.

Suppose the goal is this document:

```
<?xml version="1.0"?>
<!DOCTYPE  XML SYSTEM "vwhelp.dtd">
<?xml-stylesheet href="01-language.css" type="text/css"
   title="Smalltalk Language" charset="UTF-8" ?>
<xml>Hello, world!</xml>
```

We create the XML.Document, which has the document content:

```
newDoc := XML.Document new.
newDoc addNode: (XML.PI name: 'xml-stylesheet'
   text: 'href="01-language.css" type="text/css"
   title="Smalltalk Language" charset="UTF-8" ').
newDoc addNode: (XML.Element tag: 'xml').
newDoc root addNode: (XML.Text text: 'Hello, world!').
```

Create an output stream:

```
str := 'c:\xmlTest\doc2.xml' asFilename writeStream.
```

Next, write the prolog and any DTD information:

```
str nextPutAll: '<?xml version="1.0"?>'; cr.
str nextPutAll: '<!DOCTYPE  XML SYSTEM "vwhelp.dtd">'; cr.
```

Finally, we create a SAXWriter, write the document, and close the stream:

```
writer := SAXWriter new output: str.
[newDoc saxDo: writer] ensure: [str close].
```

Examine the resulting file to see that it is what we expected.

Using XML Namespaces

XML namespaces allow documents to employ multiple markup vocabularies without collision. For example, different parts of a document might need to refer to different elements both named “employee”. XML namespaces provide a mechanism for differentiating these references by associating each with a URI.

Declare Namespaces

A Document can specify one or more namespaces for resolving element or attribute names within the document. A root element often specifies a namespace, such as this, from http://www.w3.org/xml:

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

A document can also have multiple namespaces, one of which may have no prefix, as in the above. All additional namespaces must have a prefix. To specify two XML namespaces, one without and the other with a prefix, the XML is specified like this:

```
< html xmlns="http://www.w3.org/1999/xhtml"
xmlns:foo="http://www.noplace/foo" >
```

To declare these namespace specifications in an XML Document in VisualWorks, create a Dictionary containing these namespaces, and then add the Dictionary to the document root element by sending it a namespaces: message. The Dictionary contains associations between a prefix string and the URI string. A namespace without a prefix is associated with an empty prefix.

```
nsDict := Dictionary new.
nsDict at: '' put: 'http://www.w3.org/1999/xhtml';
   at: 'foo' put: 'http://www.noplace/foo'.
newDoc := XML.Document new.
newDoc addNode: ( XML.Element tag: 'XML' ).
newDoc root namespaces: nsDict.
```

Evaluate the above in a workspace and inspect newDoc to see that the root element specifies the namespaces as intended.

There is one problem with the above example, however. If you inspect the newDoc tag variable, which contains a NodeTag, there is no namespace specified. This is a problem if you need to extract data from the DOM, or pass it to a processor such as XSchema, XSLT, or XPath. To include the namespace information in the document tag, modify the above to:

```
nsDict := Dictionary new.
nsDict at: '' put: 'http://www.w3.org/1999/xhtml';
   at: 'foo' put: 'http://www.noplace/foo'.
newDoc := XML.Document new.
newDoc addNode: ( XML.Element tag:
   (NodeTag new qualifier: '' ns:  (nsDict at: '') type: 'XML') ).
newDoc root namespaces: nsDict.
```

Note that holding the namespace declarations dictionary in a temporary variable, as above, is not necessary (the dictionary could be defined inline), but simplifies referring to the namespace, as it is in the tag definition shown here.

Applying a Namespace to an Element

If you use namespaces, you should use them consistently, and include the namespace in specifying the element tag. Do this by creating a NodeTag, and specify the qualifier (namespace prefix name), namespace, and type (tag name). This is the same as specifying the NodeTag for the root element shown above. For example:

```
nsDict := Dictionary new.
nsDict at: '' put: 'http://www.w3.org/1999/xhtml';
   at: 'foo' put: 'http://www.noplace/foo'.
newDoc := XML.Document new.
newDoc addNode:
   ( XML.Element tag:
      ( NodeTag new qualifier: '' ns: ( nsDict at: '' ) type: 'XML' ) ).
newDoc root namespaces: nsDict.
newDoc root addNode:
   ( ( XML.Element tag:
      ( NodeTag new
         qualifier: 'foo' ns: (nsDict at: 'foo') type: 'heading1' ) )
      addNode: ((( XML.Element tag:
         (NodeTag new
            qualifier: 'foo' ns:  (nsDict at: 'foo') type: 'heading2'))
         addNode: ( XML.Element tag:
            (NodeTag new qualifier: '' ns:  (nsDict at: '') type: 'body') ) ) ) ).
```

which produces the XML:

```
<XML xmlns:foo="http://www.noplace/foo"
   xmlns="http://www.w3.org/1999/xhtml">
   <foo:heading1>
      <foo:heading2>
         <body/>
      </foo:heading2>
   </foo:heading1>
</XML>
```

Elements can also declare additional namespaces for use within their scope. To do this, send a namespaces: message to the element, after it has been created. In this example, while both the heading1 and heading2 elements specify the foo namespace qualifier, heading2 is in a different namespace than heading 1 due to the new declaration:

```
nsDict1 := Dictionary new.
nsDict1 at: '' put: 'http://www.w3.org/1999/xhtml';
   at: 'foo' put: 'http://www.noplace/foo'.
nsDict2 := Dictionary new.
nsDict2 at: 'foo' put: 'http://www.noplace/bar'.
newDoc := XML.Document new.
newDoc addNode:
   ( XML.Element tag:
      (NodeTag new qualifier: '' ns:  (nsDict1 at: '') type: 'XML') ).
newDoc root namespaces: nsDict1.
newDoc root addNode:
   ( ( XML.Element tag:
      (NodeTag new
         qualifier: 'foo' ns: ( nsDict1 at: 'foo' ) type: 'heading1' ) )
      addNode: ( ( XML.Element tag:
         ( NodeTag new
            qualifier: 'foo' ns: ( nsDict2 at: 'foo' ) type: 'heading2' ) )
         namespaces: nsDict2 ;
         addNode: ( XML.Element tag:
            (NodeTag new qualifier: '' ns:  (nsDict1 at: '') type: 'body') ) ) ).
```

The resulting XML is:

```
<XML xmlns:foo="http://www.noplace/foo" xmlns="http://www.w3.org/
1999/xhtml">
   <foo:heading1>
      <foo:heading2 xmlns:foo="http://www.noplace/bar">
         <body/>
      </foo:heading2>
   </foo:heading1>
</XML>
```

Assigning a Namespace to an Attribute

Attribute names can be assigned a namespace to, as for elements. Again, instead of a simple String for the name, you define and assign a NodeTag. So, expanding the example used earlier for attributes, you can assign a namespace as follows:

```
nsDict := Dictionary new.
nsDict at: '' put: 'http://www.w3.org/1999/xhtml';
   at: 'foo' put: 'http://www.noplace/foo'.
attrGroup := Array
```

```
    with: (XML.Attribute
        name:
            (NodeTag new qualifier: 'foo' ns: (nsDict1 at: 'foo') type: 'ALIGN')
        value: 'left' )
    with: (XML.Attribute
        name: (NodeTag new qualifier: '' ns: (nsDict1 at: '') type: 'SRC')
        value: 'http://www.w3.org/Icons/WWW/w3c_home').
newDoc := XML.Document new.
newDoc addNode:
    ( XML.Element tag:
        (NodeTag new qualifier: '' ns:  (nsDict at: '') type: 'XML') ).
newDoc root namespaces: nsDict.
newDoc root addNode:
    ( XML.Element
        tag: (NodeTag new qualifier: '' ns: (nsDict at: '') type: 'IMG')
        attributes: attrGroup elements: nil ).
```

Namespace declarations are not allowed in attribute specifications.

Building a SAX Driver

SAX (Simple API for XML) is an event-driven interface for accessing XML documents without having to model the whole document in memory. Using SAX is often preferred, such as when the application needs to construct its own data structure from the XML document. In such a case, modeling the entire node tree first only to discard it is inefficient.

A SAX parser breaks a document into a linear set of events. For example, the XML document:

```
<?xml version="1.0"?>
<doc>
<para>Hello, world!</para>
</doc>
```

is rendered as this series of events:

```
start document
start element: doc
start element: para
characters: Hello, world!
end element: para
end element: doc
end document
```

The application specifies how to process each event in its event handlers.

Handling SAX Events

To create a SAX application, define a custom SAX driver as a subclass of SAXDriver or one of its subclasses. Your driver class defines handler methods for each of the SAX parsing events, specifying the action to take for each element or attribute of interest.

The default action for events, defined in SAXDriver, is to do nothing. Your driver overrides these with more appropriate handling. The following are the basic events to handle. For additional events provided for special purposes, browse the **content handler** method category in SAXDriver, and read the method comments.

startDocument

Triggered once at the start of the document.

endDocument

Triggered once at the end of the document.

startElement: *namespaceURI* **localName:** *localName* **qName:** *name* **attributes:** *attrList*

Triggered by an element start tag. *namespaceURI* is the namespace URI, or nil if there is none. *localName* is the name of the element, without prefix. name is the literal name of the element, or nil if processing namespaces. *attrList* is a SequenceableCollection of Attribute instances.

endElement: *namespaceURI* **localName:** *localName* **qName:** *name*

Triggered by an element end tag. Parameters are as described for startElement:localName:qName:attributes:.

startPrefixMapping: *prefix* **uri:** *anURI*

Triggered by an element with a namespace declaration. *prefix* is a String, if a prefix is specified in the declaration. *anURI* is the namespace URI, as a String.

endPrefixMapping: *prefix*

Triggered by the closing tag for an element that declared the namespace. *prefix*, if any, is the declared namespace prefix as a String.

characters: *aString*

Triggered by character data (CDATA). *aString* contains the character data.

skippedEntity: *name*

Triggered by a skipped entity. *name* is the name of the skipped entity. Parameter entity names start with '%'. If the entity is an external DTD subset, name is '[dtd]'.

processingInstruction: *name* **data:** *dataString*

Triggered by a processing instruction. *name* is the instruction name, and *dataString* is the instruction data.

ignorableWhitespace: *aString*

Triggered by ignorable whitespace in the document. *aString* contains the whitespace characters.

For example, to handle the simple document above, a driver should handle start and end document, start and end element, and character events. These five methods could be implemented, say in MySAXDriver, to simply write information to the Transcript:

```
characters: aString
   Transcript show: 'cdata: ', aString; cr.
startDocument
   Transcript show: 'Start of Document'; cr.
endDocument
   Transcript show: 'End of Doc';cr.
startElement: nsURI localName: lName qName: name attributes: attrList
   Transcript show: 'start: ', name; cr.
endElement: namespaceURI localName: localName qName: name
   Transcript show: 'end: ', name; cr.
```

To exercise this driver on the example document above, evaluate this in a workspace:

```
| doc p |
doc := '<?xml version="1.0"?><doc><para>Hello, world!</para></doc>'
   readStream.
p := XMLParser new.
p handlers: MySAXDriver new.
p validate: false.
p parse: doc.
```

Configuring SAX Features and Properties

VisualWorks supports the standard SAX2 interface for querying and setting the parser’s feature and property set, to control the parser’s behavior. Features and properties are identified by a URI with which is associated a Boolean value.

The general messages to set and get parser features and properties are:

atFeature: *featureURI*

Returns the Boolean value of *featureURI*, if recognized; otherwise raises a SAXNotRecognizedException exception.

atFeature: *featureURI* **put:** *aBoolean*

Sets the value of *featureURI* to *aBoolean*, if recognized; otherwise raises a SAXNotRecognizedException exception.

atProperty: *propertyURI*

Returns the Boolean value of *propertyURI*, if recognized. No properties are recognized, by default, so returns SAXNotRecognizedException.

atProperty: *propertyURI* **put:** *aBoolean*

Sets the value of *propertyURI* to *aBoolean*, if recognized. No properties are recognized, by default, so returns SAXNotRecognizedException.

Several common features are represented by shared variables defined in the XML.SAX namespace. Each shared variable holds a default URI for the feature, which is set in the variable's initialization string. Note that only the SAX namespace, namespace-prefixes, and validating features are currently supported by the VisualWorks XML framework, though you may add support for additional features and properties.

SAXExternalGeneralEntities

Not currently supported. Attempting to set or get the value raises a SAXNotSupportedException. Would be set false to ignore external general entities in the document.

SAXExternalParameterEntities

Not currently supported. Attempting to set or get the value raises a SAXNotSupportedException. Would be set false to ignore external parameter entities in the DTD.

SAXNamespace

Default true. Set to true if the parser should process namespaces, or false if the parser should ignore xmlns attributes.

SAXNamespacePrefixes

Default false. Set to true if xmlns attributes should appear in the attribute list of an element, or false if they should be filtered out.

SAXValidate

Default true. Set to true if the parser should do full validation, or false to suppress validation.

For accessing the values of the SAXNamespace, SAXNamespacePrefixes, and SAXValidate features, send these messages to the parser:

isValidating

Returns the Boolean value of the validation feature (SAXValidate).

validate: *aBoolean*

Sets the Boolean value of the validation feature (SAXValidate).

processNamespaces

Returns the Boolean value of the namespaces feature (SAXNamespace).

processNamespaces: *aBoolean*

Sets the Boolean value of the namespace feature (SAXNamespace).

showNamespaceDeclarations

Returns the Boolean value of the namespace-prefixes feature (SAXNamespacePrefixes).

showNamespaceDeclarations: *aBoolean*

Sets the Boolean value of the namespace-prefixes feature (SAXNamespacePrefixes).

Setting the validating feature using the validate: message was illustrated above, to parse a document without a DTD (see Parsing an XML Document). Using the more general messages, turning off validation can be done like this:

```
parser := XMLParser new.
parser atFeature: SAXValidate put: false.
```

The feature can also be identified by an URI, in which case the above could be:

```
parser := XMLParser new.
parser atFeature: 'http://xml.org/sax/features/validation' put: false.
```

For setting or getting SAX feature and property values, you should trap SAXNotRecognizedException and SAXNotSupportedException.

```
parser := XMLParser new.
featureStr := 'http://xml.org/sax/features/validating' .
[ [ parser atFeature: featureStr ]
    on: SAXNotRecognizedException
    do: [ :e | Dialog warn: 'Feature ', featureStr, ' is not recognized.' ] ]
        on: SAXNotSupportedException
        do: [ :e | Dialog warn: 'Feature ', featureStr, ' is not supported.' ]
```

Document Fragments

When using XML to exchange data, it is frequently inconvenient, or inefficient, to have to parse an entire document up to the element that one is actually interested in. For example, if you are only interested in one chapter (e.g., chapter 23), or one paragraph, of a book, it would be inefficient to have to parse all of the book up to that element.

Document fragments provide a way to represent a part of a document. The challenge for using fragments is to have enough context to be able to parse the fragment correctly.

The VisualWorks XML framework supports document fragments in the XML.DocumentFragment class. The main difference between a Document and a DocumentFragment is that a DocumentFragment does not require a single top-level element, but may have a sequence of elements at its top level. It may also have character data outside of an element. So, for example, a document fragment could include:

```
<body>Some introductory text.</body>
<heading2>
    Some heading
    <body>Discussion of this topic</body>
</heading2>
<heading2>
    Some other heading
    <body>Discussion of this topic</body>
</heading2>
```

This is understood as being parsed within a larger XML context that provides the missing information.

Building a Fragment

To build the above fragment, send the appropriate addNode: messages to an instance of DocumentFragment.

```
docFrag := XML.DocumentFragment new.
docFrag addNode: ( ( XML.Element tag: 'body' )
   addNode: ( XML.Text text: 'Some introductory text.' ) ).
docFrag addNode: ( ( ( XML.Element tag: 'heading2' )
   addNode: (XML.Text text: 'Some heading' ) )
      addNode: ( (XML.Element tag: 'body')
         addNode: ( XML.Text text: 'Discussion of this topic.') ) ).
docFrag addNode: ( ( ( XML.Element tag: 'heading2' )
   addNode: (XML.Text text: 'Some heading' ) )
      addNode: ( (XML.Element tag: 'body')
         addNode: ( XML.Text text: 'Discussion of this topic.') ) ).
```

Attempting this construct with a Document instance would result in errors due to the multiple top-level nodes, but it is acceptable as a DocumentFragment.

Parsing a Fragment

If an XML document references a fragment as an entity, you can parse the entire document as usual. The fragment is simply included in the document as if it were physically present within the XML.

A fragment-aware application, however, will want to deal with fragments it may receive from a data source. The application will have to be able to provide the context necessary for including the fragment in a document. In the case of the above fragment, the context may be simply:

```
<document>
   <heading1>
      Title
   </heading1>
         ***insert fragment here***
   </heading1>
</document>
```

The XML framework provides no specific support for providing this context. Recommendations are available from the World Wide Web Consortium (see http://www.w3.org/TR/xml-fragment), but it is the responsibility of your application to implement a strategy.

XSL Stylesheet Processing

VisualWorks supports applying an XSL stylesheet to an XML file to transform the XML file into another representation.

Most XSL Transformation elements are supported, and are implemented as subclasses of XSLCommand.

Loading XSL Support

XSL support is an add-in component to VisualWorks. To use XSL facilities, load the XSL parcel (xsl.pcl).

XSL support classes are in the XSL namespace. Your application may need to import this namespace into its own namespace or into relevant classes.

Applying a Stylesheet to a Document

XMLParser does not automatically apply a stylesheet to an XML document, even if the stylesheet is specified in the document. Instead, you generate an XSL rule database from the stylesheet and apply it to the parsed XML document.

For example (borrowed from *The XML Bible*, second edition, by Elliotte Rusty Harold), suppose we have an XML document representing the periodic table (periodictable.xml):

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xml" href="table.xsl"?>
<PERIODIC_TABLE>
  <ATOM STATE="GAS">
    <NAME>Hydrogen</NAME>
    <SYMBOL>H</SYMBOL>
    <ATOMIC_NUMBER>1</ATOMIC_NUMBER>
    <ATOMIC_WEIGHT>1.00794</ATOMIC_WEIGHT>
    <BOILING_POINT UNITS="Kelvin">20.28</BOILING_POINT>
    <MELTING_POINT UNITS="Kelvin">13.81</MELTING_POINT>
    <DENSITY UNITS="grams/cubic centimeter">
      <!-- At 300K, 1 atm -->
      0.0000899
    </DENSITY>
  </ATOM>
  <ATOM STATE="GAS">
    <NAME>Helium</NAME>
    <SYMBOL>He</SYMBOL>
    <ATOMIC_NUMBER>2</ATOMIC_NUMBER>
    <ATOMIC_WEIGHT>4.0026</ATOMIC_WEIGHT>
```

```
            <BOILING_POINT UNITS="Kelvin">4.216</BOILING_POINT>
            <MELTING_POINT UNITS="Kelvin">0.95</MELTING_POINT>
            <DENSITY UNITS="grams/cubic centimeter"><!-- At 300K -->
               0.0001785
            </DENSITY>
        </ATOM>
    </PERIODIC_TABLE>
```

and an XSL document (table.xsl) to transform the document into HTML:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="PERIODIC_TABLE">
        <html>
            <xsl:apply-templates/>
        </html>
    </xsl:template>
    <xsl:template match="ATOM">
        <P>
            <xsl:apply-templates/>
        </P>
    </xsl:template>
</xsl:stylesheet>
```

First generate the rules database, then parse the document and apply the rules as follows:

```
xslRules := ( XSL.RuleDatabase new ) readFileNamed:
'c:\xmlTest\table.xsl'.
parser := XMLParser new validate: false.
doc := parser parse: 'c:\xmlTest\periodicTable.xml' asFilename
readStream.
transDoc := xslRules process: testDoc.
```

The result is a new document, actually a DocumentFragment, that has been transformed according to the rules in the stylesheet:

```
<html>
   <P>
      Hydrogen
       H
       1
       1.00794
       20.28
       13.81
      0.0000899</P>
   <P>
       Helium
```

```
            He
            2
            4.0026
            4.216
            0.95
           0.0001785
        </P>
    </html>
```

Note that the XSL namespace declared in the stylesheet must be http://www.w3.org/1999/XSL/Transform. If it is not, the resulting document contains the stylesheet itself and not a transformed document.

There are several examples of applying an XSL transformation in class methods of RuleDatabase which you can examine and execute.

Using XPath

XPath is a language for addressing parts of an XML document. XPath models a document as a tree structure, allowing elements to be accessed by specifying a path to those elements, like a filesystem path.

Creating a Path Expression

An XPath expression is a string specifying selection criteria for a collection of nodes in a document. The XPath specification (http://www.w3.org/TR/xpath) provides the full, abstract syntax for XPath expressions. A few examples are:

Expression	Selection
/AAA	The root node AAA
/AAA/BBB/CCC	All elements tagged CCC that are children of BBB that are children of root AAA.
//CCC	All elements CCC in the document
//BBB/CCC	All elements CCC that are children of BBB
//BBB/*	All elements that are children of BBB
/*/*/*	All elements with two ancestors
//BBB/CCC[2]	Each second instance of element BBB that is a child of BBB
//CCC[text()]	All text elements in any CCC
//BBB | //CCC	All elements BBB and CCC
/AAA/BBB/descendant::*i	All elements that are descendents of /AAA/BBB
//@id	All attributes id
//BBB[@id]	Al elements BBB with an id attribute

The return value of an XPath expression can be either a Number, String, Boolean, or XPathNodeContext. The above expressions return an XPathNodeContext, which is a collection of XML nodes.

To use an XPath expression in VisualWorks, it must be parsed, using XPathParser. For example:

```
exprString := '//CCC[text()]'.
expr := XML.XPathParser new
   parse: exprString as: #expression.
```

In this example, exprString holds is assigned some legal (per the XPath specification) XPath expression, as a String. The XPathParser returns an instance of XPathRoot, which can then be applied to an XML node to retrieve the desired information.

If the XML uses namespaces, you must also provide the parser with an XML node that gives the context in which to resolve the namespace qualifiers. For example, if the expression includes a namespace qualifier “foo”, a node defining the qualifier must be provided:

```
exprString := '//foo:CCC[text()]'.
expr := XML.XPathParser new
   xmlNode: ( myDoc root) ;
   parse: exprString as: #expression.
```

The element’s sole purpose is to map "foo" to an URL, but could be, for example, the document root node, as done above. If you don't use namespaces in the path, the XML Element is optional.

Applying an XPath Expression

You apply the expression to an XML node by sending a xpathValueFor:variables: message to the XPathRoot instance, the result of parsing the expression string.

```
result := expr
   xpathValueFor: otherXmlNode
   variables: Dictionary new.
```

The XML document, or node, to be searched is the first argument value. The Dictionary passed as the second argument maps variable names to values, and is only important if the XPath expression uses variable references.

The return value of an XPath expression can be either a Number, String, Boolean, or XPathNodeContext, which is a collection of nodes. Usually the programmer will know, based on the syntax of the expression string, what type of value will be returned. These four return types can be converted amongst themselves using xpathAsBoolean, xpathAsString, and xpathAsNumber. These messages use the XPath conversion rules.

Selecting Nodes with an XPath

For expressions that return a collection of nodes, you can now use the XPathRoot to select nodes. First, get the XPathNodeContext by applying the expression to a node. To do this, send an xpathValueFor:variables: message to the XPathRoot instance. You can then retrieve the nodes as a sorted or unsorted collection, by sending sortedNodes or unsortedNodes message:

```
nodeSet := expr
   xpathValueFor: otherXmlNode
   variables: Dictionary new.
nodeSet xpathIsNodeSet
   ifTrue: [nodeSet := nodeSet unsortedNodes].
```

XML Error Handling

The VisualWorks XML engine is a SAX engine, so all error handling is provided by SAXException subclasses:

```
Error
   SAXException
      SAXNotRecognizedException
      SAXNotSupportedException
      SAXParseException
         InvalidSignal
         MalformedSignal
            BadCharacterSignal
         WarningSignal
```

Most of these exceptions are raised only during parsing, the exceptions being SAXNotRecognizedException and SAXNotSupportedException, which are raised when querying or setting a SAX parser's features or properties. Catching these exceptions is shown under Building a SAX Driver.)

The argument passed into the handler block is an instance of the specific error class, which you can use for further handling.

```
parser := XMLParser new.
[ pdoc := parser parse:
   '..\help\01-xml-language\01-language.xml' asFilename ]
   on: SAXException do: [ :e | Transcript show: e printString ; cr ]
```

11

Parser Compiler

The standard VisualWorks parser/compiler classes, Scanner, Parser, and Compiler, are in the base VisualWorks class library, and are used for the usual code compiling processes.

For cases creating special parsers and compilers, additional classes are provided as in an optionally loadable component, in the **AT Parser Compiler** parcel.

Standard Parser-Compiler

Although not often used directly, there are three classes that parse and compile Smalltalk programs: Scanner, Parser and Compiler. The Scanner parses a string into a sequence of tokens (numbers, names, punctuation, etc.) according to the lexical rules of the Smalltalk language. The Parser parses a string into a complete expression or method definition. The Compiler compiles a string into a method.

Scanner

To create an instance of Scanner, use new. To convert a string to a sequence of tokens, use scanTokens: as in the expression:

```
tokenArray := Scanner new scanTokens: aTextOrString
```

The string is interpreted approximately as though it were an Array, each word being converted to the equivalent literal (number, symbol, etc.) and installed as an element. However, the pound sign (#) that introduces a literal array is incorrectly treated like a binary operator, and the words “nil”, “true”, and “false” are not treated specially. For example, the following expression is true:

```
(Scanner new scanTokens: '3.5 is: GPA') = #(3.5 #is: #GPA)
```

Parser

To create an instance of Parser, use new.

To extract the selector from the source string of a method:

```
selector := Parser new parseSelector: aString
```

For example, the following expression is true:

```
(Parser new
   parseSelector: 'from: here to: eternity
      ^eternity - here')
= #from:to:
```

To parse an entire method or a **doIt** (just like a method without the initial pattern), use an expression such as the following:

```
methodNode := Parser new
   parse: sourceStream
   class: aClass
   noPattern: noPattern
   context: nil
   notifying: anEditor
   ifFail: aBlock
```

The noPattern argument is true for a doIt, false for a method. If the source was constructed by a program, or a TextEditor for interactive use, anEditor should be nil. If the source is not syntactically legal, this expression returns the result of evaluating aBlock; otherwise, it returns an instance of MethodNode.

Compiler

Compiler's class methods provide the most interesting public behavior, so there is usually no need to create an instance.

To evaluate a string as a Smalltalk expression:

```
Compiler
   evaluate: aString
   for: anObject
   notifying: anEditor
   logged: logFlag
```

The string will be evaluated as though it were the body of a method that had been invoked with anObject as the receiver. If logFlag is true, the string is written on the changes log. For example, the following expression returns 7:

```
Compiler
   evaluate: 'x + y'
   for: 3 @ 4
   logged: false
```

As for parsing, anEditor should be nil for noninteractive use, or a TextEditor for interactive use. To compile a source method into a CompiledMethod object, use an expression of the following form:

```
aMethod := Compiler
   compileClass: aClass
   selector: aSymbol
   source: aString
```

This message will rarely be useful, however. More useful methods in Behavior (such as compile:notifying:) perform the compilation and also install the result in the method dictionary of a class.

Advanced Parser-Compiler

The parser compiler classes make it easier to write compilers in Smalltalk. SQL classes (in the **AT Parser Example** parcel) provide an example of an SQL compiler written using the parser compiler facilities.

A typical compiler handles four functions:

- Scanning—breaking the source code into tokens (words, numbers, operators, etc.).
- Parsing—combining tokens into larger structured units.
- Semantic analysis—verifying that variables have been declared, performing type checking, etc.
- Code generation—producing a program in machine code or other final form. This may occur in several phases if optimization or more than one representation of the output code is involved.

The parser compiler classes provide the following support for these activities:

- Scanning—the Smalltalk Scanner, slightly modified.
- Parsing—This phase is the primary focus of the Parser Compiler, providing an efficient language for writing your parser.

- Semantic analysis—the Parser Compiler makes it fairly easy to mix in semantics during parsing. This helps to generate an error message that points at the right place in the source code.
- Code generation— you're on your own. The Parser Compiler itself demonstrates one style of code generation: It generates Smalltalk source code during parsing. The complexity of most languages prevents being able to combine code generation with parsing.

Scanning Source Code

The ParserCompiler class defines seven standard types of token:

- word—a variable or unary message selector
- number—integer or floating point
- character
- string
- binary—infix operators such as + and >=
- keyword—a word followed by a colon (see below)
- signedNumber—a number optionally preceded by a minus sign, with no intervening delimiters

There is an eighth standard token type, keywords, for one or more keywords in succession with no intervening delimiters. This produces a single token. Keywords are only recognized specially if your grammar uses the word keyword or keywords, or if your grammar includes any literal keywords. (This is for the benefit of grammars that don't use keywords, but use the colon for other purposes.)

In addition, the scanner makes assumptions about delimiters (blank, tab, end-of-line, and new-page), which separate tokens but aren't tokens themselves. It also assumes that the following characters are tokens on their own: # ( ) | [ ] . : = ^ and ;. To change any of these assumptions requires an understanding of the Scanner's mechanics—you have to write your own initScanner method that calls super initScanner and then substitutes the appropriate entries in the typeTable.

Parsing

For the parsing phase, begin by making your parser a subclass of ExternalLanguageParser—SQLCompiler has been provided as an example. If your source language is method-oriented and you want the output of the parser to be executable CompiledMethods, make your parser a subclass of GeneralParser instead.

This gives your class basic parsing functionality. The parser scans source code one character at a time and one token at a time. You must then write production rules describing the various parts of your language. These rules define parsing algorithms, which your parser will use to recognize constructs such as functions and clauses in the source code. The syntax of production rules will be described in a moment.

Each clause or other construct found in the source code must be instantiated as a node in a parse tree. For example, when an SQL clause is recognized in the source code by SQLCompiler, an instance of SQLClause is created. Classes such as SQLClause typically are subclassed from a more general class such as SQLNode.

As an example of this node-creation mechanism, the production rule implemented by SQLCompiler for recognizing an SQL commit statement creates an instance of SQLStatement as follows:

```
EmulationBorderDecorationPolicy unInstallcommitStatement =
   #COMMIT #WORK
         [statement: 'COMMIT WORK']
```

In this example, the word COMMIT followed by WORK in the source causes execution of the block. A statement: message is sent to SQLCompiler, and that method sends an instance creation message to SQLStatement with the 'COMMIT WORK' string as the statement name.

The ultimate output of the parser is an array containing objects such as SQLFunction, which themselves are often composites of smaller language constructs such as SQLClause. This array represents a parse tree that you can use to generate code.

As the parse tree is being assembled, it is stored in an OrderedCollection called stack, held by GeneralParser. This stack responds to collection protocol such as removeLast, and stack operations are frequently embedded in blocks within the production rules. For example, the SQLCompiler>>queryTerm rule contains the following assignment into a temporary variable:

```
tableExp := stack removeLast.
```

A Rule has a Name and a Definition

A production rule describes a semantic unit of the language in terms of other semantic units combined with literal tokens. It introduces the name of the semantic unit, followed by =, followed by the definition, which may include references to other production rules or to literal keywords that are expected at various points in the source-code.

As an example, the following production rule is taken from SQLCompiler:

```
assignment =
   column #= ( scalarExp | #NULL )
```

When a production rule is invoked, its definition is used as a template for the current source code. If the template fits, the rule returns true, triggering creation of the appropriate node in the parse tree. If the definition doesn't match, either the rule returns false, or an error notification occurs.

Rules are Similar to Methods

It is no accident that a production rule looks like a Smalltalk method. It is created just as a Smalltalk method is, by adding it to the instance protocol for your compiler class (SQLCompiler, in this case). You can use the System Browser to do so, or you can file it in. This is possible because the ParserCompiler's responsibility is to take production rules and translate them into equivalent Smalltalk code, which is then translated into an executable method. Each production rule is translated into a method whose selector is the name of the production rule. As a result:

- You can browse production rules in the same way you browse Smalltalk methods.
- Production rules can call Smalltalk code, and vice versa.

Temporary Variables Can be Used

A production rule can have temporary variables. These are defined the same way as in Smalltalk, by enclosing the list of names between two vertical bars.

A production rule begins with a method pattern consisting of the name of the rule, plus names for any arguments. Except for the terminating equal sign (=), the syntax is identical to that of a Smalltalk method, allowing for unary, binary, and keyword patterns.

A Rule Definition is a Series of Alternatives

The body of a production rule, called its definition, is a series of alternatives, separated by vertical bars (|). The parser tries to match the current source code to each alternative in turn. If a given alternative succeeds, the definition succeeds and returns `true`. If an alternative fails, the next alternative is tried.

The final alternative in a series can be left empty to return `true` immediately. If the series is enclosed in parentheses, the empty alternative is indicated by a vertical bar preceding the closing parenthesis. If the series is the body of the definition, the empty alternative is indicated by making a vertical bar the last element of the definition.

For example:

(a | b) c	The next tokens must match either 'a' or 'b', followed by 'c'
(a |) c	The next token or tokens must match either 'a' followed by 'c', or 'c' alone

An Alternative is a Series of Terms

An alternative is a series of terms, each alternative optionally preceded by an at sign (@). Each term is evaluated sequentially against the source code. If a term succeeds, the parser proceeds to the next term; otherwise it fails. If the last term in the alternative succeeds, the alternative returns `true`. If the alternative fails, behavior depends on several factors:

- If the at sign is present, the source code stream is rolled back to the state it was in when the alternative was started, and `false` is returned.
- If the term that failed was the first in the alternative, `false` is returned.
- Otherwise, an error notification is returned.

Two examples follow:

```
a b c
```

Expect to find an `a`, followed by `b` and `c`. If `a` is not found, proceed to the next alternative or return `false`. If `b` or `c` is not found, print an error message.

```
@ a b c
```

Expect to find an a, followed by b and c. If a, b, and c are not found when expected, proceed to the next alternative or return false.

Suppose the parser matches a, but fails to match b. For accurate error detection, the ParserCompiler will not automatically back up on failure, so in this case a message would appear saying b expected. However, it is possible that if the source stream were backed up, we might be able to match c d rather than a b. Therefore, in this case, it is appropriate to write the rule as:

```
@ a b | c d
```

Then, if a succeeds but b fails, the parser will back up and try to match c followed by d.

Another way to think about it is: When the first term in an alternative is matched, the parser assumes it has found the correct alternative. If a later term fails to match, the parser reports an error based on its assumption that the correct template was applied unsuccessfully. The at sign removes the assumption so that, instead of generating an error in this situation, the compiler proceeds to the next alternative.

A Term is an Action or a Unit-Plus-Qualifier

A term can be an action, or it can be a unit followed by one of the following symbols:

```
*   * ! + +!\  \!  !*
```

We will discuss the more common type of term first: units and their quantifying modifiers.

A Unit is a Word, Terminal, or Parenthesized Definition

A unit can be a word, a terminal, or a definition wrapped in parentheses. If it is a word, that word is assumed to be the name of another production rule. Some examples:

Word and associated production rule

foo	Evaluate the production rule foo on the current source code. If it returns false, fail the current alternative, else continue.
word=#ABC	If the next token in the source is ABC, push it on the stack and scan another token, else fail the alternative.
keyword=#ABC:	If the next token in the source is ABC:, push it on the stack and scan another token, else fail the alternative.

$(	If the next token is the open parenthesis character, scan another token, else fail the alternative. The stack is unaffected.
#ABC	If the next token in the source is ABC, scan another token, else fail the alternative. The stack is unaffected.
#ABC:[keyword type]	If the next token in the source is ABC:, scan another token, else fail the alternative. The stack is unaffected.
#~=	If the next token in the source is ~=, scan another token, else fail the alternative. The stack is unaffected.
#’<<=’	If the next token in the source is <<=, scan another token, else fail the alternative. The stack is unaffected.
(...)	When parentheses are encountered, the enclosed part of the rule is parsed according to the rules for alternatives.

The following examples illustrate the use of the seven quantifying symbols with units. In these examples, `foo` pushes a `FooNode` onto the stack, while `foo2` does not affect the stack.

Quantifying symbols

foo *	Expect zero or more repetitions of foo. The top value on the stack will be an Array of FooNodes.
foo *!	Expect zero or more repetitions of foo. The top N values on the stack will be FooNodes, where N is the number of repetitions.
foo +	Expect one or more repetitions of foo. The top value on the stack will be an Array of FooNodes.
foo +!	Expect one or more repetitions of foo. The top N values on the stack will be FooNodes.
foo \ foo2	Expect one or more repetitions of foo, separated by foo2. The top value on the stack will be an Array of FooNodes.
foo \! foo2	Expect one or more repetitions of foo, separated by foo2. The top N values on the stack will be FooNodes.
foo !*	Expect one occurrence of foo. Assume that foo leaves an Array on the stack. Pop the Array off the stack and push each of its elements onto the stack.

A Terminal is a Single Token

A terminal is a single token in the language, such as a number, a string, a variable name, or a keyword. In the ParserCompiler, the following terminals are recognized:

- A dollar sign ($) followed by a single character, representing a literal character in the source.
- A number sign (#) followed by:
  - A string (any sequence of characters enclosed in single quotes)
  - A word (an alphabetic character followed by alphabetic characters or digits)
  - A keyword (a word followed by a colon)
  - A binary symbol (anything that represents a legal binary operator in Smalltalk, such as //, \\, *, ~~, and ~=)
- The sequence word=#someWord, where someWord is a word as defined above.
- The sequence keyword=#someKeyword, where someKeyword is a keyword as defined above.

The difference between #someWord and word=#someWord is that in the former case someWord becomes a reserved word in the language and is always treated specially. In the latter case, someWord does not become a reserved word and is treated specially only when it is preceded by word=.

An Action is a Block or a Special Symbol

An action can be either a Smalltalk block or one of the following special symbols:

Action symbols

Symbol	Description
<	Saves the source position in a local variable (specifically, the temps instance variable in ParserCompiler). Note that only one source position per production rule is saved, so if you overwrite it, the old value is lost.
>	Assumes that the source position was previously saved via <, and that the top value on the stack is a parse node. The parse node is sent a sourcePosition:to: message, with the saved position as the first argument and the current position as the second argument. This implies that your node classes must implement a sourcePosition:to: message when you use this symbol in a production rule.
<<	Pushes the source position onto the stack.
>>	Assumes that the top value on the stack is a parse node, and that the next value is a source position saved by <<. The parse node is sent a sourcePosition: message, with an interval from the saved position to the current position as the argument. The source position is removed from the stack, and the parse node remains the top element.
?	Pops the top value off the stack. If it is true, proceed, otherwise fail the current alternative.
.	Pops the top value off the stack and proceed.

The first four operations are for matching source code positions to parse nodes. The last two are for use with Smalltalk blocks. When a Smalltalk block appears in a production rule, the block is evaluated and the result is pushed onto the stack. If you are interested in the effect of the block but not the returned value, follow the block with a period to get rid of the unwanted value. To decide whether to continue parsing after a block has been evaluated, follow the block with a question mark to cause the current alternative to proceed or abort depending on the returned value.

Two Types of Block Syntax are Allowed

Two distinct syntaxes are accepted for Smalltalk blocks. One form of syntax is identical to that of normal Smalltalk blocks having zero arguments. The second form is nonstandard and requires further explanation—it has the advantage of very concise coding, with the disadvantage of very restricted syntax.

Like a normal block, this special block is enclosed in square brackets. It consists of exactly one message—the message can be either a binary or keyword message, but not a unary message. The receiver is specially coded:

- If there is no receiver, the message is sent to the parser itself.
- If the message selector is preceded by a colon (:), the top value is popped off the stack and used as the receiver.

Each of the arguments is likewise specially coded:

- If there is no argument, or if the argument is a colon (:), the top value is popped off the stack and used as the argument.
- If the argument is a normal Smalltalk literal (Symbol, String, Number, Array, ByteArray, Character, or nil, true or false), it is used in the ordinary way.
- If the argument is a temporary variable, instance variable, class variable, or global variable, it is used in the ordinary way.

For example, the following block sends a copyWith: message to the top value on the stack, with the second value on the stack as argument:

```
[ :copyWith: ]
```

Note that no argument can be the result of a message send.

Summary of Grammar for Parsing Methods

Here is a simplified version of the grammar for parsing methods, written in itself:

```
method = pattern #= temporaries definition
pattern = word | (keyword word)+
temporaries = $| word* $| |d
definition = alternative ($| alternative)*
alternative = ($@ | ) term*
term = unit
        ( (#* | #*!)
        | (#+ | #+!)
        | (#\ | #\!) unit | )
unit = word | character
  | $# (word | keyword | binary | string)
  | $( definition $)
```

Creating your Own Compiler

In preparation for writing programs in your new language, first define a compiler class MyLanguageCompiler, then define a dummy class MyLanguage. Define the following class method for MyLanguage:

```
compilerClass
   ^MyLanguageCompiler
```

Then any methods defined in class MyLanguage or any of its subclasses will compile with MyLanguageCompiler rather than the standard Smalltalk compiler. The example methods in the SQL class are compiled by SQLCompiler in just this way.

The typical instance creation protocol for a parser takes either a Stream or a String as input, as well as the name of the top-level production rule to be applied. For example:

```
CParser parse: aStream as: #cFile
```

The final step in code generation is done by the message generate:. This message is defined in GeneralParser on the assumption that the output of your compiler (i.e., the single element left on the stack at the end of recognizing a method) is a string that is actually a Smalltalk source method, which then gets handed to the Smalltalk compiler.

However, you can override this method in your own compiler to do something different. It should return a selector if the code generation succeeds, or nil if it fails. In the case of the SQL example, the final object is an Array containing a parse tree in the form of a hierarchy of nodes. Try the examples on the instance side of the SQL class, inspecting the results recursively to see the structure of the parse tree.

This object responds to Smalltalk messages and can thus be manipulated to suit the next phase of compilation.

Index

R

S